
В. К. ГУЛАКОВ, А. О. ТРУБАКОВ, Е. О. ТРУБАКОВ



СТРУКТУРЫ И АЛГОРИТМЫ ОБРАБОТКИ МНОГОМЕРНЫХ ДАННЫХ

Монография

Издание второе, стереотипное



ЛАНЬ



• САНКТ-ПЕТЕРБУРГ •
• МОСКВА •
• КРАСНОДАР •
2021

УДК 004.657
ББК 32.81я73

Г 94 **Гулаков В. К.** Структуры и алгоритмы обработки многомерных данных : монография / В. К. Гулаков , А. О. Трубаков, Е. О. Трубаков. — 2-е изд. стер. — Санкт-Петербург : Лань, 2021. — 356 с. : ил. — Текст : непосредственный.

ISBN 978-5-8114-7965-8

Книга посвящена описанию структур и алгоритмов для индексирования и обработки многомерных данных. В ней систематизированы наиболее важные подходы, описаны их математические и логические принципы построения, проанализированы достоинства и недостатки. Содержится большое число примеров листинга, позволяющее более детально разобраться в представленных подходах. На различных примерах рассматриваются особенности проектирования и разработки приложений, обрабатывающих многомерные и многоатрибутные данные.

Монография предназначена для бакалавров и магистров, обучающихся по направлениям «Информатика и вычислительная техника», «Программная инженерия», «Математическое обеспечение и администрирование информационных систем», а также по близким направлениям. Также она будет полезна научным работникам, преподавателям, специалистам, аспирантам, связанным с прикладной математикой и разработкой программного обеспечения. Можно использовать специалистам, занимающимся хранением данных, поиском информации и другими смежными проблемами.

УДК 004.657
ББК 32.81я73

Рецензенты:

В. И. АВЕРЧЕНКОВ — доктор технических наук, профессор, заслуженный деятель науки РФ, почетный работник высшего профессионального образования;

В. В. МИРОШНИКОВ — доктор технических наук, профессор, действительный член Академии проблем качества, почетный работник высшего профессионального образования РФ.



Обложка
Е. А. ВЛАСОВА

© Издательство «Лань», 2021
© Коллектив авторов, 2021
© Издательство «Лань»,
художественное оформление, 2021

ОГЛАВЛЕНИЕ

ПРЕДИСЛОВИЕ.....	5
ВВЕДЕНИЕ.....	8
ГЛАВА 1. ОСНОВНЫЕ ПОЛОЖЕНИЯ.....	10
1.1. Понятие многомерного объекта.....	10
1.2. Виды запросов к многомерным хранилищам данных.....	14
1.3. Расстояние между многомерными объектами.....	18
1.4. Минимальный описывающий регион (MBR).....	21
1.5. Области использования многомерных структур.....	25
Резюме.....	31
ГЛАВА 2. ТОЧЕЧНЫЕ МЕТОДЫ ДОСТУПА.....	32
2.1. Классификация точечных методов доступа.....	33
2.2. Иерархические методы доступа.....	36
2.2.1. K-D-дерево.....	36
2.2.2. K-D-B-дерево.....	56
2.2.3. LSD-дерево.....	81
2.2.4. Quad-дерево.....	103
2.3. Многомерное хеширование.....	120
2.3.1. Файл-решетка.....	122
2.3.2. Хеширование EXCELL.....	152
2.3.3. Многомерное линейное хеширование с частичным расширением (MOLHPE).....	158
2.3.4. Многомерное линейное хеширование с сохранением порядка - PLOP.....	179
2.4. Кривые, заполняющие пространство.....	189
2.4.1. Упорядочивание по ключам.....	191
2.4.2. Кривая z-порядка.....	195
2.4.3. Кривая Гильберта.....	198
2.4.4. Кривая, основанная на кодах Грея.....	201
Резюме.....	202
ГЛАВА 3. ПРОСТРАНСТВЕННЫЕ МЕТОДЫ ДОСТУПА.....	203
3.1. Методы преобразования пространственных объектов.....	205
3.1.1. Преобразование в пространство большей размерности.....	205
3.1.2. Кривые, заполняющие пространство для объемных объектов.....	210
3.2. Структуры с перекрытием областей.....	214

3.2.1. R-дерево	215
3.2.2. R-дерево Грина	250
3.2.3. R*-дерево	253
3.2.4. SS-дерево	263
3.2.5. SR-дерево	274
3.2.6. TV-дерево	283
3.3. Структуры с разделением объектов	296
3.3.1. BSP-дерево	298
3.3.2. R+-дерево	307
3.4. Многослойные структуры	320
3.4.1. Многослойный файл-решетка	321
Резюме	329
ГЛАВА 4. МНОГОМЕРНАЯ ПИРАМИДА	330
Резюме	346
ЗАКЛЮЧЕНИЕ	347
СПИСОК ЛИТЕРАТУРЫ	348



ПРЕДИСЛОВИЕ

Монография предназначена для научных работников, преподавателей, специалистов, аспирантов и студентов, связанных с прикладной математикой и разработкой программного обеспечения. Наиболее полезно ее использование при обучении бакалавров и магистров по направлениям 09.03.01 «Информатика и вычислительная техника», 09.03.04 «Программная инженерия», 02.03.03 «Математическое обеспечение и администрирование информационных систем», а также по близким направлениям.

Для понимания материала требуются минимальные знания по дискретной математике и основ программирования на алгоритмическом языке. Учитывая новизну рассматриваемого направления и отсутствие соответствующих материалов в отечественной литературе, авторы в книге выделили первую главу, которая посвящена основным понятиям и определениям предметной области. В частности, в ней рассматриваются понятие многомерного объекта, виды запросов в многомерном пространстве и функции расстояния между объектами. В ней же дается краткое описание некоторых приложений многомерных структур, что свидетельствует об актуальности данного направления.

В монографии не рассматриваются теоретические и математические аспекты построения, обработки и оценки эффективности различных многомерных структур. В соответствующих параграфах дается достаточно ссылок на работы этого плана. Основное внимание уделяется системному подходу к описанию структур и их практическому приложению.

В качестве ключевых моментов классификации многомерных структур выделены объект доступа и специфика запроса. С этих позиций все рассматриваемые структуры разделяются на две большие группы.

Первой группе многомерных структур, оперирующих точечными данными, посвящена вторая глава. В ней подробно рассматриваются три подгруппы структур: структуры, ориентированные на иерархические методы доступа, многомерное хеширование и кривые, заполняющие пространство.

Иерархические методы позволяют получить гарантированную логарифмическую сложность алгоритмов обработки. Для точечных объектов основополагающей структурой этой группы является *K-D-дерево*. В главе подробно описаны его возможности и дальнейшая

его эволюция в виде различных модификаций. Так, *K-D-B*-дерево является синтезом идей *K-D*-дерева и *B*-дерева – одного из самых эффективных алгоритмов индексирования одномерных данных, расположенных на жестком диске.

Альтернативой иерархическим структурам данных является хеширование. При правильном выборе алгоритма хеширование может стать самым эффективным методом доступа к многомерным данным. Однако гарантировать эффективность не всегда возможно. В работе дается подробный анализ этих методов и областей их применения.

Также представляют определенный интерес структуры, оперирующие кривыми заполнения пространства. Исторически именно эта группа алгоритмов была первой попыткой ускорить операции обработки многомерных данных. Однако назвать эти алгоритмы многомерными в полной мере нельзя. Основная суть этих методов сводится к преобразованию многомерной задачи в одномерную.

Особый интерес представляют структуры и алгоритмы для пространственных объектов, которые рассмотрены в третьей главе. Самым простым подходом работы с подобными объектами является преобразование пространственной структуры объекта в точечную и применение к ней методов, рассмотренных во второй главе. Данная концепция выглядит достаточно элегантно и просто. Однако у нее есть ряд ограничений и недостатков, подробно описанных в начале третьей главы.

Индексирование пространственных объектов связано с проблемой, связанной с невозможностью однозначного деления объектов на непересекающиеся группы, что было несвойственно точечным структурам. Частично эту проблему можно решить, разрешив различным узлам структуры перекрывать друг друга, т.е. соответствовать одной и той же части пространства. После применения такого допущения появляется возможность без каких-либо проблем размещать в узлах объекты, имеющие некоторые пространственные размеры. При этом нет необходимости разбивать объект на более мелкие части. Структуры, обрабатывающие объекты по этому принципу, описаны во втором подразделе третьей главы. Практически все они ведут свое существование от структуры, получившей название *R*-дерева. В главе приводится описание оригинального алгоритма построения *R*-дерева, а также ряд его модификаций, улучшающих те или иные свойства.

При более тщательном рассмотрении можно заметить ряд серьезных проблем, которые могут свести к минимуму преимущества от использования подобных индексов в некоторых практических применениях. Достаточно глубокий анализ позволил выработать некоторые рекомендации для различных ситуаций. В соответствующих разделах заостряется внимание на этих проблемах и даются

рекомендации к полному устранению или уменьшению негативного эффекта при практической реализации структур.

Большой интерес представляют многослойные структуры. В монографии дается небольшое их описание, но полноценные результаты по этим структурам еще предстоят. Также в стороне остались темпоральные и метрические структуры данных, но это отдельная тема.

Книгой можно пользоваться как справочником. Большинство разделов можно изучать независимо. Для желающих проработать углубленно тот или иной вопрос или сомневающийся в тех или иных положениях, даются ссылки на первоисточники.



ВВЕДЕНИЕ



В ближайшем будущем мощь любого государства будет определяться не уровнем развития промышленности, новизной и эффективностью ее технической базы, а уровнем информатизации общества. Стратегический потенциал общества составят не вещество и энергия, а информация и научные знания [8, 9]. Ученые утверждают, что в недалеком будущем реально защищенным в социальном плане будет только широко образованный человек, способный гибко перестраивать направление и содержание своей деятельности в связи со сменой технологий или требований рынка. Владение информационными технологиями ставится сегодня в один ряд с такими качествами, как умение читать и писать.

Однако процесс перехода к информационному обществу породил целый ряд новых проблем. К числу важнейших исследователи относят следующие:

1. Наличие огромных и не всегда упорядоченных объемов информации, не позволяющих человеку эффективно ориентироваться в растущих информационных потоках.
2. Возникновение научных проблем, связанных с производством, накоплением, передачей и потреблением информации и знаний. Способность человека находить эквивалентность или аналогию между разными представлениями в одинаковых ситуациях.
3. Способность человека интерпретировать новую информацию в уже имеющихся понятиях формализованного знания.

Важность этого направления развития цивилизации очевидна. Практически экспоненциальный рост объемов информации в электронной форме ставит более остро проблему поиска и обработки нужной информации. Здесь успех зависит от ряда причин и в наибольшей степени от структур хранения данных и алгоритмов их обработки. В некоторых ситуациях, например, системы реального времени, время доступа и обработки информации является критичным.

В этих и других проблемах есть две наиболее важные задачи:

1. Фильтрация информации для определенных групп пользователей.
2. Структуризация информации.

Под структуризацией понимаются либо тематические базы данных или хранилища данных, либо индексы поисковых систем.

Другими словами, при структуризации необходимо использовать структуры для хранения и обработки информации, позволяющие быстро получать необходимую информацию по различным запросам.

Трудности усугубляются тем, что разная информация имеет разные атрибуты или измерения. Графическая информация имеет параметры – цвет, размеры, положение и т.д., текст и аудиоинформация имеют другие параметры.

С учетом специфики информации и запросов пользователей необходимы эффективные инструменты для построения стратегии поиска информации.

Значительно чаще мы выбираем объект по нескольким параметрам, а это значит, что мы ведем многомерный поиск. Для того чтобы он был эффективен, необходимо использовать многомерные структуры данных и соответствующие алгоритмы.

Другая проблема поиска – степень релевантности и полноты найденной информации.

Все перечисленные проблемы нельзя оставлять только пользователю. Необходима автоматизация этих процессов. Но для создания автоматизированных систем необходимы теоретическая база и соответствующие инструменты.

Предлагаемая книга посвящена использованию многомерных структур данных в программировании. Основное внимание уделяется не теоретическим исследованиям этих структур, а их характеристикам и практическому применению.

В зависимости от особенностей хранимой и обрабатываемой информации возможны различные структуры ее представления и, соответственно, ее обработки.

Многомерные структуры данных достаточно популярны среди разработчиков информационных систем и прикладного программного обеспечения. Об этом свидетельствуют и большое количество публикаций по этому направлению, и многообразие многомерных структур и алгоритмов. К сожалению, отечественная информация об этих достижениях весьма скупа, хотя успехи очевидны, а зарубежная информация либо слишком поздно доходит, либо существует только на языке разработчика. Полноценных же изданий, отражающих современное состояние рассматриваемой темы, практически нет.

Здесь, с одной стороны, делается попытка собрать и в некоторой степени обобщить и систематизировать весь материал по многомерным структурам, имеющимся в различных источниках, с другой стороны, дать возможность желающим углубить свои знания по рассматриваемой теме и, наконец, посмотрев на сегодняшние успехи в этой области, осознать проблемы и наметить пути их решения.

ГЛАВА 1. ОСНОВНЫЕ ПОЛОЖЕНИЯ



В настоящее время в отечественной литературе многомерным структурам уделяется крайне мало внимания. В зарубежной литературе эти структуры часто описаны в разрозненных статьях. Поэтому важно определиться с терминологией и базовыми понятиями многомерных структур и алгоритмов их обработки, что и делается в этой главе. Вводится понятие многомерного объекта и его минимальной ограничивающей фигуры, с помощью которых строится большинство современных методов обработки и индексирования данных. Дано определение метрики и представлены наиболее популярные функции расстояний, которые позволяют упорядочивать многомерные объекты в пространстве.

Также в этой главе описываются некоторые области применения многомерных структур, в которых использование обычных одномерных схем затруднено или нецелесообразно.

1.1. Понятие многомерного объекта

Чтобы понять основные требования к многомерным методам доступа и суть основных структур, применяемых для анализа пространственных данных, необходимо четко представлять, что же такое многомерный объект.

Самое важное свойство многомерного объекта и одновременно принципиальное отличие от обычных, простых данных – это комплексность, сложность самой структуры. Многомерный объект может представлять собой некоторую точку в пространстве или сложный, состоящий из тысяч полигонов или множества атрибутов объект. Но так или иначе его нельзя представить в виде одной числовой величины.

В простейшем случае многомерный объект рассматривается как некоторая точка с нулевыми размерами (размеры объекта на самом деле очень малы или являются несущественными для разрабатываемого приложения). В этом случае математически его можно описать следующим образом: имеется множество записей, каждая из которых характеризуется n ключами K_0, K_1, \dots, K_{n-1} . Рассмотрим множество

значений любого ключа как некоторое измерение. В этом случае получаем данные в n измерениях, или многомерные данные. При этом сама запись является точкой в n -мерном пространстве.

Для примера рассмотрим систему поиска изображений по гистограммному признаку (сходства цветов). Каждое имеющееся изображение можно по цветовому исполнению преобразовать в некоторую запись:

$$\{R, G, B\},$$

где R – число точек в изображении с преобладанием красной компоненты; G – зеленой; B – синей.

Приняв $K_0=R$, $K_1=G$, $K_2=B$, мы получаем запись для каждого изображения, состоящую из трех ключей, а само изображение – точку в многомерном пространстве (рис. 1.1). При поиске изображений, подобных заданному, необходимо просто выполнить процедуру поиска ближайших соседей для заданной многомерной точки.

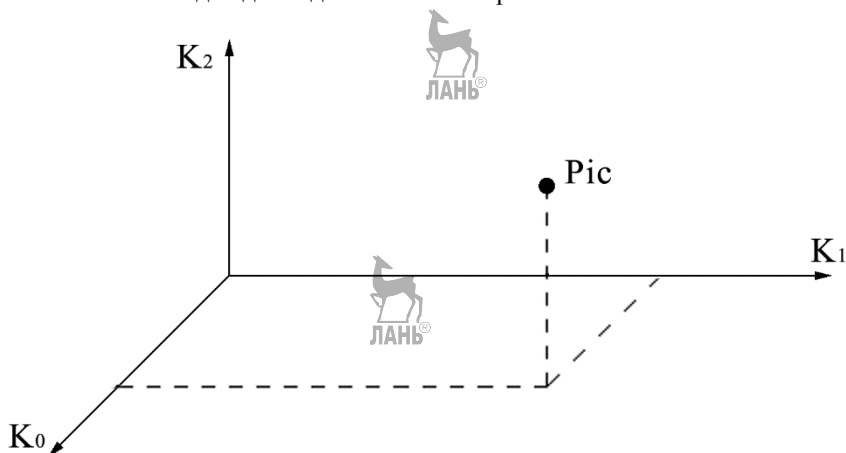


Рис. 1.1. Пример точки в трехмерном пространстве

Разработано множество алгоритмов для работы с многокритериальными объектами как с некоторыми точками в многомерном пространстве. Такие алгоритмы можно условно отнести к классу точечных методов доступа (ТМД, или в английской интерпретации *Point Access Methods – PAM*). Наиболее популярные из этих методов рассмотрены во второй главе данного издания.

Однако в практическом применении наиболее часто приходится иметь дело с объектами, размеры которых существенны, и пренебрегать ими нельзя. Это и геоинформационные системы (ГИС), и системы навигации, различные CAD-системы и др. Алгоритмы, рассматривающие объект не как точку, а как некоторый элемент с собственными размерами в пространстве, относятся к классу пространственных методов доступа

(ПМД, или в английском варианте *Spatial Access Methods – SAM*). Этот класс структур является самым большим и наиболее востребованным в практических задачах.

Значительный интерес представляют многомерные структуры, предназначенные для работы с объектами, изменяющими свое состояние или местоположение с течением времени. Особенностью таких структур является то, что в качестве одной из осей многомерного пространства выступает время. При этом изменение состояния объекта во времени будет представлять собой некоторую кривую по времени. Данный класс алгоритмов и структур получил название пространственно-временных методов доступа (ПВМД, или *Spatio-Temporal Access Methods – STAM*). Иногда в литературе также можно встретить название темпоральные базы данных – СУБД, поддерживающие работу с объектами, изменяющимися во времени.

Еще одним классом структур, приобретающим популярность в последнее время, являются структуры, рассматривающие объект не в обычном пространстве, а в некотором обобщенном метрическом. Такие структуры называют метрическими методами доступа (ММД, или *Metric Access Methods – MAM*).

К какой бы группе ни относилась та или иная многомерная структура, существует ряд свойств, присущих ей. Выделим эти основные свойства:

1. *Комплексность*. Структура, лежащая в основе практически любого алгоритма обработки многомерной информации, является сложной, составной. В нее включается столько полей, сколько ключевых записей содержит элемент (или в скольких измерениях он рассматривается).

2. *Динамичность*. Для эффективного решения практических задач метод доступа должен поддерживать такие операции, как вставка/удаление элементов, т.е. возможность динамического изменения структуры в целом.

3. *Адаптация к большим размерам*. В основе алгоритмов обработки структур данных должна быть предусмотрена работа не только в оперативной памяти, но и во внешнем хранилище (чаще всего – объединение объектов в страницы и совместное их хранение в одном секторе внешнего накопителя). Это связано с тем, что практические задачи оперируют с большими объемами данных, которые просто неспособны целиком размещаться в оперативной памяти. Так, карта некоторой поверхности или модель ландшафта местности могут занимать несколько десятков или даже сотен гигабайт (в зависимости от сложности и детализации).

4. *Поддержка нестандартных операций*. Строго говоря, для многомерных данных нет четкого перечня операций, как, например, для одномерных данных. Операции, которые должен поддерживать тот или

иной метод, зависят от системы, в которой он применяется. Так, в практических приложениях бывают гораздо важнее операции определения пересечения многомерных объектов или поиска ближайших соседей, чем тривиальная операция поиска объекта в базе данных по точному совпадению.

5. *Независимость структуры индекса от последовательности построения.* Данное требование выполняется далеко не всегда. Его суть заключается в том, что структура многомерного индекса не должна зависеть от последовательности добавления/удаления объектов в него. Она должна быть оптимальной и опираться только на значение ключевых полей самих объектов. Однако в практических применениях зачастую оказывается, что при хаотическом добавлении элементов может получиться не совсем оптимальная или сбалансированная структура. Поэтому данное требование в ряде алгоритмов остается недостижимым идеалом, к которому нужно стремиться.

6. *Расширяемость.* Используемый метод доступа к многомерным данным должен быть расширяемым, т.е. легко адаптироваться к постоянному росту базы данных.

7. *Производительность.* Наиболее важная операция – это пространственный поиск. В используемых структурах подобные запросы должны выполняться как можно быстрее. Эталон, с которым стоит сравнивать быстродействие поиска, могут служить В-деревья для одномерного случая. В идеале поисковый запрос должен выполняться с логарифмической сложностью для самого худшего случая.

8. *Пространственная эффективность.* Применяемая структура должна быть как можно меньше в размере и не приводить к значительному увеличению объема данных. Но, помимо объема, отводимого под саму структуру, очень важным является еще и использование дискового пространства страницами данных. Почти все структуры, предназначенные для хранения данных во внешней памяти, оперируют понятием страниц – некоторого дискового пространства, в котором реально размещаются объекты. При этом страницы могут быть заполнены не полностью, т.е. приводить к потере дискового пространства. Поэтому алгоритмы, лежащие в основе самой структуры данных, должны минимизировать неиспользуемое пространство страниц с целью улучшения процента использования памяти.

1.2. Виды запросов к многомерным хранилищам данных

Рассмотрим наиболее важные виды запросов, встречающиеся в приложениях, оперирующих многомерными объектами. При этом для простоты будем предполагать, что эти объекты расположены в евклидовом двумерном пространстве.

В данном параграфе будут только перечислены эти типы запросов и приведено их формальное математическое описание. Реализация же их зависит от используемой структуры данных и будет описана в соответствующих параграфах.

При математическом описании запросов используются следующие обозначения:

- P – точечный объект;
- O – пространственный объект;
- $O.g$ – внутренняя область объекта.

Поиск по точному совпадению (*Exact Match Query*). Самый простой тип запроса. Он используется как для точечных объектов, так и для объектов, имеющих некоторый пространственный размер.

$$\begin{aligned} Search(P') &= \{ P \mid P' = P \} \\ Search(O') &= \{ O \mid O'.g = O.g \} \end{aligned}$$

Суть запроса заключается в следующем: имеется некоторый точечный объект P' или пространственный O' ; необходимо проверить, присутствует ли данный объект в системе.

Поиск объекта, содержащего точку (*Point Query*). Данный запрос используется только для объектов, имеющих определенный размер. Причем в практическом применении он используется гораздо чаще предыдущего типа запросов.

$$Search(P) = \{ O \mid P \cap O.g = P \}$$



Рис. 1.2. Поиск объекта

Имеется некоторая многомерная точка P . Найти многомерный объект, содержащий данную точку (рис. 1.2).

Поиск по области (Range Query). Дан многомерный регион поиска R . Необходимо найти все объекты, имеющие хотя бы одну точку в этом регионе. Данный запрос может производиться как для точечных объектов (рис. 1.3(а)), так и для объектов, имеющих объем (рис. 1.3(б)).

$$Search(R) = \{P \mid P \cap R \neq \emptyset\}$$

$$Search(R) = \{O \mid O.g \cap R \neq \emptyset\}$$

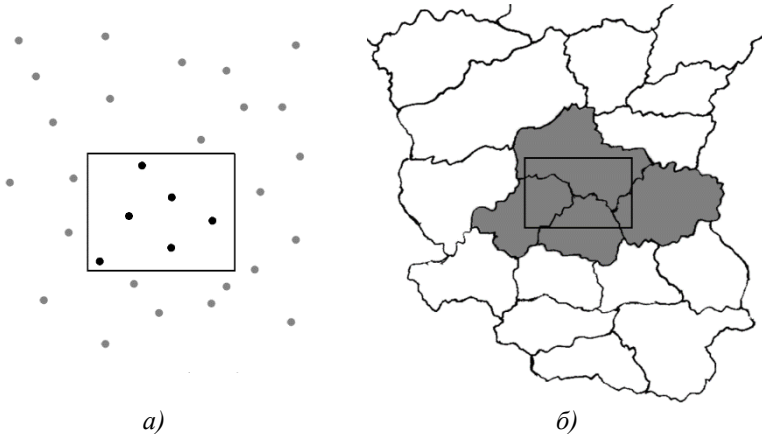


Рис. 1.3. Поиск по области: а – точечные объекты; б – пространственные объекты

Многомерный регион обычно задается как набор верхних и нижних границ для каждого измерения

$$R = [l_1, u_1] \times [l_2, u_2] \times \dots \times [l_n, u_n],$$

т.е. для каждого измерения указываются пределы, в которых может изменяться данная координата.

Поиск покрывающих объектов (Overlap Query). Дан некоторый многомерный объект O' . Найти все объекты, имеющие хоть какое-то пересечение с данным (рис. 1.4).

Данный запрос очень похож на предыдущий, однако если в прошлом запросе в качестве области поиска использовался многомерный прямоугольник, то в этом запросе может выступать многомерный объект произвольной формы. Причем поиск по области можно считать крайним случаем поиска покрывающих объектов (в том случае, если в качестве объекта O' выбран объект прямоугольной формы, то данный запрос станет полностью равносильным предыдущему).

$$Search(O') = \{O \mid O'.g \cap O.g \neq \emptyset\}$$



Рис. 1.4. Поиск покрывающих объектов

Поиск вмещающего объекта (*Enclosure Query*). Дан некоторый многомерный объект O' . Найти такой объект O (если он существует), который полностью содержит в себе объект O' (рис. 1.5).

В отличие от предыдущего запроса, в этом поиске необходимо найти не те объекты, которые имеют хоть какое-то пересечение с заданным, а те, которые его полностью вмещают.

$$\text{Search}(O') = \{ O \mid (O'.g \cap O.g) = O'.g \}$$

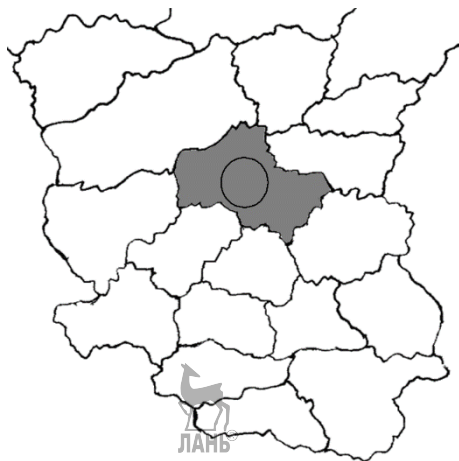


Рис. 1.5. Поиск вмещающего объекта

Поиск вхождений (Containment Query). Задан некоторый многомерный объект O' . Найти все объекты O , которые полностью входят в O' (рис. 1.6).

$$Search(O') = \{O \mid (O'.g \cap O.g) = O.g\}$$

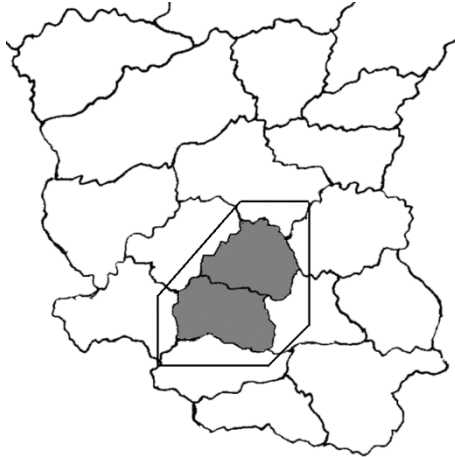


Рис. 1.6. Поиск вхождений

Поиск соседей (Adjacency Query). В этом запросе участвует реальный объект системы O' . Необходимо найти такие объекты O , которые имеют общие с ним границы, но не пересекают его (рис. 1.7).

$$Search(O') = \{O \mid (O'.g \cap O.g) \neq \emptyset \wedge (O'.g_{in} \cap O.g_{in}) = \emptyset\}$$

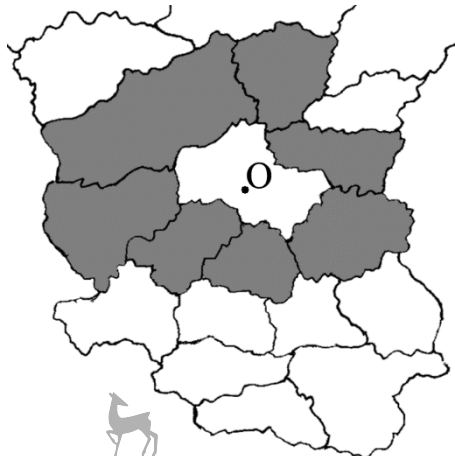


Рис. 1.7. Поиск соседей

Для математического описания данного запроса введем еще одно понятие. $O.g_{in}$ – внутренняя область объекта O . В отличие от принятого нами ранее обозначения $O.g$, $O.g_{in}$ содержит только ту часть объекта, что находится внутри его границ, не включает сами границы.

Поиск ближайшего соседа (*Nearest Neighbor Query*). Этот запрос можно считать чуть ли не одним из самых популярных. Были разработаны отдельные структуры данных, специально адаптированные для запросов ближайшего соседа. Такая популярность определяется тем, что очень много приложений сводят поиск необходимой информации к обычному запросу ближайшего соседа в многомерном пространстве.

Суть запроса проста: дан некоторый объект P . Необходимо найти такой объект в базе данных, который был бы расположен как можно ближе к данному.

Для точечных объектов данный запрос можно выразить следующей формулой:

$$Search(P') = \{ P \mid \forall P'' : Dist(P', P'') \geq Dist(P', P) \}.$$

Аналогично можно записать и для пространственных объектов:

$$Search(O') = \{ O \mid \forall O'' : Dist(O'.g, O''.g) \geq Dist(O'.g, O.g) \}.$$

При некоторых изменениях можно переписать данные требования для поиска k ближайших соседей, что также очень часто бывает необходимо в практических применениях.

Существует несколько вариантов вычисления функции расстояния ($Dist$) в многомерном пространстве. Более подробно эта функция будет описана в следующем параграфе.

Запрос всех пар элементов, удовлетворяющих условию (*Spatial Join*). Даны два множества объектов S и S' (в простейшем случае это может быть одно и то же множество). Найти все пары элементов $(O, O') \in S \times S'$, удовлетворяющих некоторому условию $Q(O.g, O'.g)$.

$$Search(S, S', Q) = \{ (O, O') \mid (O \in S) \wedge (O' \in S') \wedge Q(O.g, O'.g) \}.$$

Результат данного запроса зависит от того, что за условие устанавливает предикат Q . В частности, с помощью данного запроса можно найти все пары элементов, находящиеся в удалении друг от друга не более чем на заданную величину, или все пары объектов, имеющих общие границы.

1.3. Расстояние между многомерными объектами

Одним из аспектов алгоритмов над многомерными данными является способ вычисления расстояния между объектами. Очень большой класс приложений, таких как хранилища изображений, интеллектуальная обработка видеофайлов, бизнес-системы, в качестве одного из самых основных запросов поиска используют запрос на поиск

k ближайших соседей для заданной точки пространства. При этом без понятия расстояния обойтись просто нельзя. Разработаны даже отдельные структуры, которые не оперируют обычным понятием положения объекта, а сохраняют информацию положения объектов друг относительно друга (так называемые метрические структуры данных).

Интуитивно многие понимают, что расстояние должно отражать меру сходства, близости объектов между собой по всей совокупности используемых признаков. Расстоянием между объектами в пространстве признаков называется такая величина $Dist(P, P')$, которая удовлетворяет следующим условиям:

1. $Dist(P, P') > 0$ – неотрицательность;
2. $Dist(P, P') = Dist(P', P)$ – симметрия;
3. $Dist(P, P'') + Dist(P'', P') > Dist(P, P')$ – неравенство треугольника;
4. Если $Dist(P, P') \neq 0$, то $P \neq P'$ – различимость нетождественных объектов.

В многочисленных статьях, посвященных многомерному анализу и структурам обработки, описано более 50 различных способов вычисления расстояния между объектами. Кроме термина «расстояние», в литературе часто встречается и другой термин – «метрика», который подразумевает метод вычисления того или иного конкретного расстояния. Мы будем пользоваться как первым, так и вторым термином. Наиболее доступным для восприятия и понимания количественных признаков является так называемое «евклидово расстояние», или «евклидова метрика»:

$$Dist(P, P') = \sqrt[2]{\sum_{i=0}^n (K_i - K'_i)^2},$$

где $Dist(P, P')$ – расстояние между объектами P и P' ;

K_i – числовое значение i -й переменной для объекта P ;

K'_i – числовое значение i -й переменной для объекта P' ;

n – число переменных, которыми описываются объекты (или количество измерений).

Для двумерного пространства и использования в качестве осей обозначения Ox и Oy данная формула примет всем привычный вид:

$$Dist(P, P') = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}.$$

Весьма напоминает выражение для евклидова расстояния так называемое обобщенное степенное расстояние Минковского, в котором в степенях вместо двойки используется другая величина. В общем случае эта величина обозначается символом « L ».

$$Dist(P, P') = \sqrt[L]{\sum_{i=0}^n (K_i - K'_i)^L}.$$

Рассмотрим подробнее те значения L , которые наиболее часто применяются для вычисления расстояния в многомерных структурах данных.

Частным случаем расстояния Минковского является так называемое манхэттенское расстояние, или «расстояние городских кварталов» (*city-block*), соответствующее $L=1$. Для двухмерного случая пример вычисления данного расстояния показан на рис. 1.8.

$$Dist(P, P') = a + b.$$

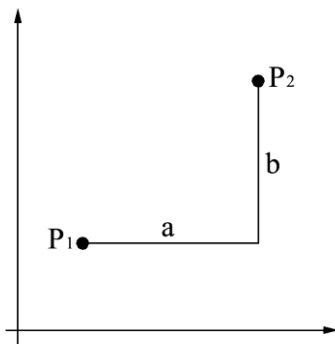


Рис. 1.8. Манхэттенское расстояние

При $L = 2$ мы получаем обычное евклидово расстояние, которое мы уже рассмотрели ранее (рис. 1.9)

$$Dist(P, P') = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}.$$

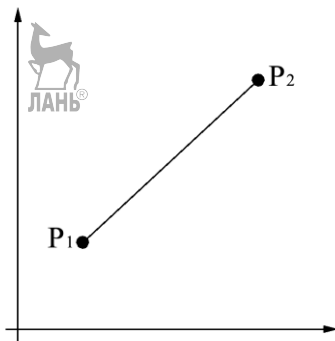


Рис. 1.9. Евклидово расстояние

Устремив L к бесконечности, мы получаем метрику «доминирования», или *Sup*-метрику:

$$Dist(P, P') = \sqrt[\infty]{(x_1 - x_2)^\infty + (y_1 - y_2)^\infty}.$$

По сути, эта метрика равна максимальной разнице координат:

$$Dist(P, P') = \max\{|x - x'|, |y - y'|\}.$$

Как можно легко увидеть, метрика Минковского фактически представляет собой большое семейство метрик, включающее и наиболее популярные способы измерения расстояния.

В данной работе примеры будут использовать вычисление расстояния между объектами по формуле евклидовой метрики, если не будет оговорено другого.

1.4. Минимальный описывающий регион (MBR)

Очень часто многомерные объекты имеют неправильный размер. Эффективных процедур для хранения и быстрого поиска по списку таких объектов в настоящее время не существует. Поэтому для увеличения производительности было решено подобные объекты описывать некоторыми геометрическими фигурами правильной формы, для которых разработаны эффективные алгоритмы обработки. Одной из таких фигур для двумерного пространства является прямоугольник (рис. 1.10).

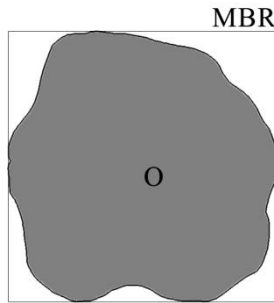


Рис. 1.10. Пример MBR

Как показано на рисунке, объект помещается в некоторую прямоугольную область, полностью вмещающую его, но не содержащую лишнего пространства. Такой подход был предложен для *R*-деревьев, одних из первых структур, предназначенных для хранения неточечных многомерных объектов. Этот подход оказался настолько успешным, что в

настоящее время большинство алгоритмов и структур используют именно его.

Прямоугольник, в который вписывался объект, был назван минимальным описывающим прямоугольником (*Minimum Bounding Rectangle – MBR*). Впоследствии были разработаны алгоритмы для других фигур, однако название *MBR* сохранилось и для них.

При обработке поисковых запросов происходит поиск тех минимальных описывающих прямоугольников, которые пересекаются с областью поиска. После определения таких *MBR* алгоритм рассматривает сам элемент, находящийся в нем, с целью уточнения, действительно ли он удовлетворяет условиям поиска.

Остается вопрос, каким образом размещать информацию о *MBR* в структуре. В настоящее время разработано несколько способов (применительно к двумерному случаю) хранения этой информации в структуре (рис. 1.11):

- сохраняются координаты двух угловых точек прямоугольника, расположенных на главной диагонали (чаще всего верхней левой и нижней правой);
- в структуре хранятся координаты верхней левой точки и информация о размере прямоугольника по всем измерениям (длина и ширина);
- в структуре размещается координата геометрического центра прямоугольника (точка пересечения диагоналей) и информация о размерах прямоугольника по всем измерениям, поделенная пополам.

Все эти способы равнозначны и легко преобразуются из одного в другой. Предпочтение стоит отдавать тому, который наиболее подходит для решения конкретной прикладной задачи.

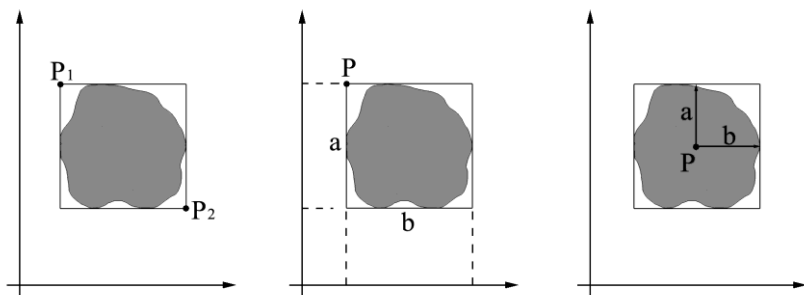


Рис. 1.11. Возможные варианты вычисления координат *MBR*

Иногда в один описывающий прямоугольник помещают не один объект, а целую группу близко расположенных друг к другу объектов

(рис. 1.12). Технология вычисления MBR для группы объектов принципиально ничем не отличается.

Однако для групп объектов разработан еще один способ образования MBR . Минимальную описывающую область формируют путем указания центра этой области и расстояния, на котором должны находиться объекты. Все объекты, попавшие в эту область, объединяются в один элемент (узел или поддерево). Для таких MBR специально разработаны алгоритмы, значительно увеличивающие скорость поиска ближайших соседей в многомерном пространстве. Среди структур, оперирующих такими MBR , особо выделяются SS - и SR -деревья.

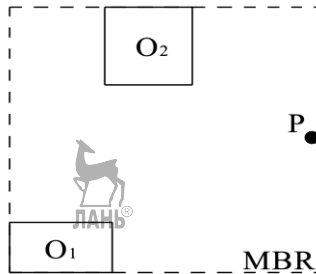


Рис. 1.12. Объединение нескольких объектов в один MBR

Для обработки таких MBR необходимо вводить понятие расстояния между объектами. Как было изложено в предыдущем параграфе, существует несколько функций расстояния. Применяя разные функции, мы можем получить MBR совершенно различных форм. Приведем примеры для основных видов расчета расстояния.

Манхэттенское расстояние (city-block). При использовании этой функции расстояния MBR примет форму ромба (рис. 1.13).

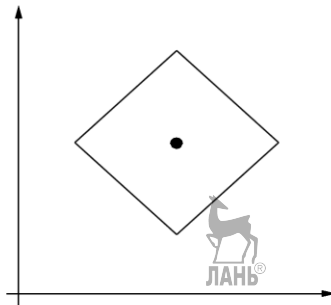


Рис. 1.13. Форма MBR при использовании манхэттенского расстояния

Евклидово расстояние. Наиболее часто используемая функция вычисления расстояния. При этом в качестве ограничивающей фигуры получаем окружность (рис. 1.14).

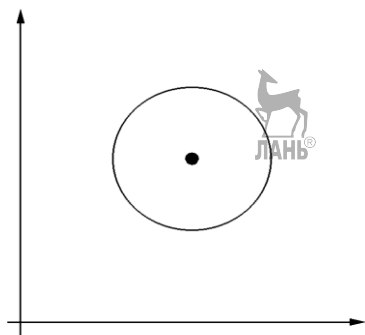


Рис. 1.14. Форма MBR при использовании расстояния Евклида

Sup-метрика. При использовании этой функции расстояния ограничивающая фигура принимает форму квадрата (рис. 1.15).

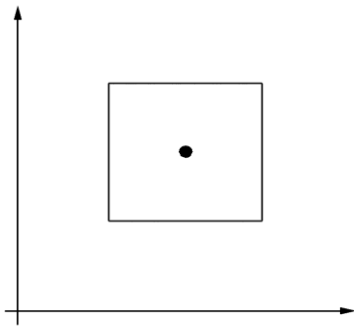


Рис. 1.15. Форма MBR при использовании sup-метрики

На первый взгляд данная метрика является ничем иным, как обычным MBR, представленным на рис. 1.10. Однако в этих фигурах есть одна большая разница. На рис. 1.10 в качестве ограничивающей фигуры используются прямоугольники, а на рис. 1.15 – квадрат.

Используя квадраты вместо прямоугольников, мы можем увеличить скорость поиска рядом лежащих точек, но при этом в ряде случаев увеличивается и перекрытие областей поиска, что может привести к снижению скорости остальных операций.

1.5. Области использования многомерных структур

Применение многомерных структур данных является очень разнообразным. Их используют в технике, экономике, политологии, биологии, медицине и других областях промышленности и техники. Такое положение вещей характеризуется тем, что большинство объектов реального мира является либо изначально многомерными, либо многокритериальными, что также преобразуется к многомерным величинам. Долгое время для обработки подобных объектов с помощью вычислительной техники приходилось преобразовывать их к одномерным, плоским величинам. При этом само преобразование накладывало ряд ограничений на процедуры обработки и анализа.

В последнее время очень бурно развивается область исследования многомерных структур данных, которые позволяют рассматривать объекты в их первоначальном многомерном состоянии без преобразования к линейному виду. Этому немало способствует развитие вычислительной техники. Постоянно увеличивающийся объем внешней памяти и растущая с каждым днем скорость обработки данных позволяют внедрять вычислительную технику в те области, где требуется нелинейная многомерная обработка. Подобный рост производительности вычислительных устройств и бурное развитие области структур и алгоритмов обработки многомерных данных позволяют программистам создавать новые приложения в сложных для проектирования областях. Далее в этом параграфе будет рассмотрено несколько областей науки и техники, в которых реализацию алгоритмов без многомерных структур сложно себе представить.

Геоинформационные системы (ГИС). Исторически так сложилось, что одной из первых областей применения многомерных структур были геоинформационные системы. И действительно, описывая Землю и объекты на ней, сложно обойтись без многомерных объектов. Даже если рассматривать одну из простейших задач ГИС – работу с плоскими картами, уже получаем необходимость применения структур хранения и обработки 2D-объектов [3]. Конечно, можно организовать хранение подобных объектов в обычных реляционных базах данных через простые типы данных, однако специфика запросов и операций поиска в ГИС такова, что реляционные СУБД без применения многомерных индексов просто не в состоянии обеспечить должную производительность. Простейшие пространственные запросы, такие как поиск объектов в некотором регионе или нахождение ближайшего города от текущего местоположения, приведут к практически полному перебору записей в таблице. А если учесть, что обычно объем информации в ГИС очень

большой, то становится очевидным необходимость использования специализированных структур индексирования данных.

Если же рассмотреть более сложные задачи ГИС, такие как геологоразведка, навигация в пространстве, диспетчерские службы в аэропортах и тому подобные применения, то в них число измерений намного больше двух, а количество данных значительно превосходит те объемы, которые можно обрабатывать линейными методами, без применения специальных структур индексирования.

Еще одним направлением развития ГИС является расширение их использования для систем с постоянным изменением характеристик или положения объекта [6]. Причем изменение свойств часто является непрерывным. Так, местонахождение мобильного телефона и его владельца постоянно изменяется с течением времени. Конечно, это должно быть отражено в средстве управления данными приложения. Постоянное перемещение объектов делает недопустимым использование традиционных СУБД в задачах такого характера, предназначенных в основном для работ со статическими данными. На роль средств, помогающих в решении этой проблемы, претендуют структуры данных подвижных объектов, которые становятся все более и более популярными с возрастающим количеством приложений такого рода. Такие структуры принято называть пространственно-временными, или темпоральными (от англ. *spatio-temporal access methods* [64]).

Изменение характеристик объекта может быть представлено с помощью изменяющихся (подвижных) параметров сущности и функции местоположения, которая определяет положение объекта в пространстве в любое заданное время. Скорость обновления данных в таких приложениях, а также скорость работы функции поиска объекта являются характеристиками, определяющими качество работы системы в целом. Различное число таких параметров и функций может быть согласовано с различными объектами, основываясь на типе движения их параметров.

Для ускорения работы данных запросов сущности, хранящие информацию об объектах, индексируются специальным образом. Алгоритмы индексирования пространственных объектов будут подробно описаны в следующих главах.

Компьютерная графика. Еще одним применением многомерных структур данных является компьютерная графика. В 70-х годах XX в. появилось новое направление в математике и информатике, названное вычислительной геометрией. В развитии этого направления большую роль сыграла игровая индустрия. Именно 3D-игры были тем фактором, который заставлял придумывать и модифицировать алгоритмы обработки и рисования пространственных объектов на дисплее монитора. Причем каждая сцена игры оперирует большим количеством объектов. Чем больше мелких деталей в сцене, тем сложнее их обрабатывать.

Современные игры содержат десятки и сотни тысяч пространственных примитивов в каждой сцене. Несмотря на то, что производительность видеокарт компьютера постоянно растет, они не способны справиться с таким объемом пространственной информации без применения различных индексов.

Одной из первых структур, адаптированных для использования в компьютерной графике, было *K-D-дерево* [19]. По его принципу была построена иерархическая структура данных, получившая название *BSP-дерево* [31]. Сегодня существует множество модификаций этой структуры, и большое количество алгоритмов в компьютерной графике используют ее как основу для хранения и обработки графических полигонов. Существуют как программные, так и аппаратные реализации данного алгоритма.

Еще одной структурой данных, часто применяющейся в компьютерной графике, является *Quad-дерево* [21] и его разновидности. Особенностью данной структуры является то, что на ее основе можно построить сцены с различной детализацией. В компьютерных 3D-играх очень часто используется эффект приближения камеры. Сначала камера может охватывать всю сцену, но по мере действий персонажа она перемещается, наезжает на отдельные элементы, показывая их более детально. Как уже описывалось выше, сцена содержит сотни тысяч полигонов и мелких объектов. При панорамной съемке сцены эти объекты практически не видны пользователю, однако видеопроцессор должен их обрабатывать и рисовать, так как они находятся на сцене. С другой стороны, если их удалить вообще со сцены, то при наезде камеры картинка потеряет точность и детализацию. Поэтому необходимо использовать механизмы, позволяющие вовлекать в рассмотрение разное число объектов в зависимости не только от того, в каком направлении повернута камера, но и от того, в каком масштабе в настоящее время рассматривается сцена. Такой структурой как раз и стала одна из разновидностей *Quad-дерева*.

Системы проектирования (CAD/CAM). Еще одним близким на первый взгляд направлением техники, связанным с многомерными структурами, являются системы проектирования. В них также необходимо моделировать сложные 3D-объекты с помощью простых примитивов. Однако в отличие от компьютерных игр, в системах проектирования основополагающим является не отображение сложной детали на экране, а ее просчет и моделирование, выполнение сложных математических операций над ее частями. В связи с этой особенностью выделяется ряд специфических требований и к структурам индексирования. Основным фактором в компьютерной графике было эффективное отсечение невидимых многомерных объектов и выбор только тех из них, которые попали в область камеры. В *CAD/CAM-системах* на первый план выходят математические и пространственные

расчеты, группировки, операции по пересечению и просчету физических характеристик как отдельных элементов, так и всего изделия в целом. Причем само изделие должно быть смоделировано с очень большой точностью, содержать миллионы примитивов (иначе погрешность расчета будет большой). Даже не самые сложные объекты моделирования обычно занимают в памяти гигабайты. Если использовать при этом стандартные линейные структуры хранения и обработки, то просчет будет длиться недели и месяцы, что в современном производственном цикле является абсолютно неприемлемым.

Чтобы данную ситуацию исправить, практически все современные системы проектирования используют структуры многомерной индексации объектов [82], которые позволяют не только визуализировать проектируемую модель, но и оптимизировать обращения к памяти и пространственные поиски при просчете ее характеристик.

Интеллектуальная обработка изображений. С развитием мультимедийного контента достаточно остро встает вопрос контекстного поиска нужной информации по изображениям. Человек уже не способен самостоятельно обработать большие объемы данных и нуждается в автоматических системах поиска. Часто используются системы текстового описания мультимедийного контента. Однако это недостаточно эффективно, так как требует дополнительных затрат и субъективно зависит от человека, составляющего описание. Более совершенным подходом можно считать контекстный поиск, ориентирующийся на внутренние характеристики изображения, и сопоставление их с заданными пользователем или соответствующие некоторому образцу (*CBIR – Content Based Image Retrieval*) [7, 13, 14].

Чтобы производить автоматический поиск подобных изображений, необходимо произвести сегментацию изображения и для каждого сегмента вычислить вектор характеристик. Отдельными компонентами таких векторов могут быть как некоторые цветовые показатели (например, гистограмма цветов), так и параметры текстуры и формы [48, 66].

Таким образом, после предобработки получаем некоторый вектор параметров, описывающий отдельные части изображения или его целиком. Задача поиска в этой постановке сводится к нахождению изображений с максимально похожими векторами характеристик. Для этого строится многомерный индекс на основе некоторых структур доступа (таких как *SR-дерево* или *X-дерево*), и в нем реализуются процедуры поиска ближайших соседей.

Данный алгоритм реализован во многих системах. Однако большинство современных продуктов ориентируются не на все перечисленные характеристики, а только на характеристики определенной группы. Так, программа *QBIC* основана на анализе цветовой составляющей, текстур и их положения на изображении;

система ZOMAX базируется на переходах по изменениям отражательных свойств поверхностей материала с учетом изменений их ориентации в наблюдаемой сцене, условий освещенности, затененности и зеркального отражения света от некоторых поверхностей в сцене; поиск в системе *VIR Image Engine* основан на таких характеристиках изображения, как насыщенность, цвет, тон.

Медицина. В данной области существует множество проблем, решение которых очень усложнено без использования многомерных структур данных. Наиболее исследованными из них являются нахождение отклонений в человеческом организме, хранение медицинских данных, создание трехмерных моделей органов человека. Необходимость эффективного и удобного решения данных вопросов появилась еще в середине XX века в Европе и Америке, где распространено лечение у личных врачей. Из-за того, что обследование пациент должен проходить периодически, приходилось часто менять медицинские карты и истории болезней. Поэтому появилась проблема легкого и быстрого обновления устаревших данных и анализа изменений. Таким образом, возникает необходимость наблюдения изменений характеристик во времени, поэтому становится целесообразным использование темпоральных структур даже для однокритериальных данных. В большинстве же случаев данные изначально имеют многокритериальный характер.

Приведем пример использования многокритериальных данных для анализа состояния пациента по составу крови. Для хранения статистики исследований можно использовать любую многомерную структуру. В качестве осей пространства в этом случае используют отдельные критерии, такие как количество эритроцитов, лейкоцитов, сахар и т.д. Каждая проба отмечается на осях и позиционируется в этом многомерном пространстве. Все пространство разбивается на области, каждая из которых помечается как область здоровых людей или некоторой категории риска. Таким образом, после анализа крови система автоматически может сделать выводы и дать некоторые рекомендации. Помимо таких простых операций, можно строить траектории изменения параметров отдельных пациентов со временем. Это позволит не только определять отклонения от нормы, но и предсказывать развитие болезней на основе статистики прошлых наблюдений.

Помимо описанных многокритериальных баз, можно выделить еще ряд популярных применений, таких как базы одномерных объектов (например, кардиограммы), двумерные изображения (например, рентгеновские снимки) и 3D-изображения (например, сканирование головы или других внутренних органов).

Биология. Еще одним применением многомерных структур стало использование их в качестве базы данных ДНК, которые содержат большое собрание строк из четырехбуквенного алфавита (А, Г, Ц, Т).

Дело в том, что исследование генетического кода сталкивается с задачей поиска подобных подстрок в базах ДНК. Данная задача осложняется тем, что размер подобных баз достигает гигантских размеров и обычный последовательный поиск становится просто невозможным. Применение же многомерных запросов ближайших соседей для многокритериального пространства позволяет сократить пространство поиска и привести к решению проблемы за приемлемое время. Конечно, полностью решить данную задачу пока не удалось, так как размерность пространства оказывается очень большой даже для небольших подстрок. Однако использование многомерных структур позволяет все же оптимизировать запросы поиска.

Экономические расчеты. В последнее время экономисты все чаще обращаются к различным системам, способным аккумулировать данные прошлых лет и проводить их анализ. Специфика этих данных такова, что они изначально являются многокритериальными и плохо преобразуются к линейным плоским величинам. Рассмотрим для примера потенциальную возможность использования многомерных структур в банковском секторе.

Сфера кредитования всегда была и остается областью большого риска. Выбор политики кредитования может привести либо к банкротству, либо к процветанию банка. Причем основополагающим здесь является именно оценка кредитоспособности потенциальных клиентов.

Чтобы оценить платежеспособность потенциального клиента, необходимо как можно полнее проанализировать всю доступную информацию о нем. Можно предложить несколько способов решения данной задачи. Одним из таких вариантов является использование многомерного индекса статистики прошлых лет. Для этого каждого клиента оценивают по ряду критериев, каждый из которых соответствует определенной оси пространства.

В этом случае каждый клиент банка представляет собой точку в этом пространстве. Его конкретное положение определяется характеристиками, полученными при анкетировании. Если окрашивать точки клиентов в разные цвета (например, белый цвет соответствует клиентам, которые исправно выплачивают ставку по взятому кредиту, а черный – ненадежным, постоянно задерживающим выплату людям), то можно получить некоторую картину зависимости распределения клиентов по критериям их кредитоспособности. Имея хорошую статистику прошлых кредитований, построить такую схему не представляет особых сложностей. А поиск по такой базе и сопоставление потенциального клиента с похожими на него людьми, кредитовавшимися в прошлом, может быть очень полезным в процессе определения платежеспособности.

Распознавание речи. Многие системы распознавания речи, распространенные сегодня, имеют довольно небольшие наборы фраз – порядка 100 слов. Так как размер словарей постоянно растет, то применение многомерных деревьев может сыграть важную роль в установлении значений «неизвестных высказываний», как например слов в словаре. Когда высказывание говорящего вводится в систему, оно раскладывается на составные части в фиксированное число «характеристик». Речь может быть пропущена через банк фильтров выражений, и изменение амплитуды во времени функций будет показывать изменение этих характеристик. Каждое слово (или «класс высказываний») в словаре представлено «шаблоном», который состоит из описания основных характеристик. Распознаватель должен найти, какому шаблону более тесно по значению подходит неизвестное высказывание, и сообщить, какое слово вероятнее всего было сказано говорящим.

Если шаблоны в словаре хранятся как записи в некотором многомерном дереве, с характеристиками, служащими в качестве атрибутов, то поиск шаблона наиболее вероятного слова может выглядеть как поиск ближайшего соседа в пространстве поиска. Использование процедур определения расстояния при этом поможет выбору подходящей меры сходства.

Резюме

На рассмотренных в главе понятиях многомерных объектов и их свойствах базируются классификация многомерных структур и виды соответствующих запросов. Для реализации этих запросов используется понятие метрики, или расстояния между объектами.

Рассмотрены некоторые варианты приложения многомерных структур, что говорит об актуальности этого направления и позволяет увидеть другие возможные приложения.

ГЛАВА 2. ТОЧЕЧНЫЕ МЕТОДЫ ДОСТУПА



В этой главе будут описаны несколько наиболее популярных точечных методов доступа (*PAM*), основанных на бинарных или сильноветвящихся деревьях. За исключением *BANG*-файлов и *Buddy*-деревьев, которые можно считать гибридными структурами, все методы являются иерархическими.

В большинстве методов предусмотрено объединение рядом находящихся точек в одну общую область. Точки такой области сохраняются в одной дисковой странице, на которую ссылается некоторый лист дерева (иногда такие узлы называют также узлами с данными). Внутренние узлы дерева (так называемые индексные узлы) предназначены только для разбиения пространства на подпространства и, соответственно, уменьшения области поиска. Причем вся область разбивается рекурсивно от корня к листу. Таким образом, процедура поиска, начавшаяся в корне и дошедшая до листа, проверяет на соответствие небольшой объем данных.

Первые структуры, разработанные для многомерного поиска, как раз относились к классу иерархических точечных методов доступа. Это были *K-D*-дерево (Bentley, 1975) и *Quad*-дерево (Finkel и Bentley, 1974). Данные структуры использовались прежде всего для оперативной памяти (*K-D*-дерево является бинарным, а узлы *Quad*-дерева имеют всего 4 потомка). В листовых узлах этих деревьев размещалась информация об одной многомерной точке, что является неэффективным для размещения на жестком диске, однако дает логарифмическую скорость поиска при размещении всей информации в оперативной памяти (конечно, при условии построения сбалансированного дерева, что является не всегда возможным, о чем будет рассказано в соответствующих параграфах).

В дальнейшем было разработано множество модификаций этих структур, в том числе и для жестких дисков (например, *K-D-B*-дерево, являющееся синтезом идей *K-D*-дерева и *B*-дерева – одного из самых эффективных алгоритмов индексирования одномерных данных, расположенных на жестком диске). В данной главе будут приведены алгоритмы для оригинальных структур, а также ряда их модификаций.

2.1. Классификация точечных методов доступа

Многомерная точка представляет собой набор чисел, каждое из которых является конкретным значением координаты по одной из осей пространства. Сами числа называют ключами записи. Будем саму запись, представляющую точку многомерного пространства, обозначать как P , а ее параметры – K_1, K_2, \dots, K_n , где n – размерность пространства. При поиске подобных записей указывается ряд ограничений, которым должны удовлетворять все ключи K_i . Подобный поиск принято называть ассоциативным.

Если в пространстве записей определена еще и функция расстояния между любыми двумя точками, то такое пространство называется метрическим и в нем можно производить дополнительные поиски, связанные с взаимным положением объектов относительно друг друга (например, поиск ближайшего соседа для определенной записи).

В настоящее время разработано большое число алгоритмов, которые можно отнести к классу точечных. Каждый из них имеет свои характеристики, свои особенности и область применения. Чтобы представлять особенности этих алгоритмов, обычно проводят их классификацию по некоторым признакам. В различных работах [55, 88] предложен ряд способов их классификации, однако общепризнанного способа на данный момент не существует. Это связано со сложностью выделения каких-либо существенных классификационных признаков, которые адекватно отражали бы ситуацию во всех применениях.

Некоторые исследователи делят все многообразие точечных методов доступа на две группы: структуры данных для оперативной памяти (к ним можно отнести K - D -дерево, BSP -дерево, $Quad$ -дерево, BD -дерево) и структуры данных, использующиеся для хранения элементов во внешней памяти (многомерное хеширование, K - D - B -дерево, LSD -дерево, hB -дерево).

Структуры первой категории изначально были разработаны для работы в оперативной памяти. В них минимизировано обращение к узлам дерева, но само дерево может иметь значительную высоту даже при незначительном объеме данных. Чаще всего это бинарные деревья. Для оперативной памяти это приемлемо, т.к. при такой структуре обеспечивается наиболее быстрый поиск элементов при небольшой нагрузке на процессор (логарифмическая скорость поиска). Однако если само дерево хранится на жестком диске или другом носителе, скорость работы с которым значительно ниже, чем с оперативной памятью, то структуры подобного класса демонстрируют не самые лучшие показатели по быстродействию. В этом случае более эффективными становятся структуры данных второй группы. К ним относятся сильноветвящиеся деревья и некоторые методы хеширования.

В подобных структурах в каждой вершине размещен не один многомерный объект, а целый набор таких объектов. Такие структуры стараются оптимизировать не только по отношению к времени работы *CPU*, но и по числу обращений к жесткому диску, причем именно обращение к жесткому диску считают приоритетной операцией для оптимизации.

В настоящее время широко используются оба класса структур. В целом ряде приложений можно загрузить весь индекс в память (при этом сами данные чаще всего все же остаются во внешнем хранилище данных). К таким приложениям относятся, например, построение трехмерных сцен в играх или бизнес-приложения с небольшой базой данных.

Однако есть целый ряд приложений, для которых недостаточно объемов оперативной памяти. К таким приложениям можно, например, отнести ГИС (геоинформационные системы). Размеры более или менее детальной карты даже небольшой области требуют таких объемов, которые пока недостижимы для оперативной памяти.

Еще один вариант классификации алгоритмов данного класса предложил Г. Самит [77]. Он разделил все многообразие алгоритмов на методы, имеющие иерархическую структуру (древовидные структуры), и методы, основанные на других принципах. К первой категории он отнес все виды деревьев (*K-D*-дерево, *Quad*-дерево), ко второй – методы, распространяющие идеи хеширования на многомерное пространство (файлы-решетки, EXCELL, MOLHPE, PLOP).

Другим интересным вариантом классификации *РАМ* можно считать классификацию, которую предложили Б. Сигир и Г. Крейгил [81]. Они разделили все алгоритмы точечного доступа в зависимости от некоторых свойств областей, соответствующих отдельным узлам структуры (табл. 2.1). Первое свойство, которое было ими выделено – это перекрытие областей. Под этим понимается возможность или невозможность взаимного пересечения областей отдельных наборов элементов. Если в третьей колонке табл. 2.1 стоит «–», то данная структура не допускает перекрытия областей, иначе – в структуре возможны пересечения узлов. Так, в случае древовидной структуры возможность перекрытия областей означает, что записи, находящиеся в поддеревьях определенного узла, могут пересекаться. Это дает некоторое упрощение алгоритмов обработки и редактирования, но приводит к снижению производительности.

Второе свойство данной классификации – это способ разбиения пространства в узлах дерева. Было выделено два варианта – интервальное разбиение (в качестве таковых могут выступать отрезки, прямоугольники, квадраты и т.д.) и неинтервальное (деление с помощью некоторой гиперплоскости или с помощью каких-либо других методов).

Третье свойство классификации – это покрытие структурой всего пространства или только той его части, в которой находятся данные. За данное свойство отвечает вторая колонка табл. 2.1. Плюс в этой колонке означает, что структура спроектирована таким образом, что индексирует все пространство целиком, независимо от того, в какой его части находятся данные.

Принимая во внимание эти три критерия и по-разному их комбинируя, можно выделить восемь групп алгоритмов (2^3). Однако не для всех комбинаций в настоящее время разработаны структуры. Все популярные алгоритмы можно отнести к одной из четырех комбинаций, представленных в табл. 2.1.

Было описано несколько вариантов классификации алгоритмов и структур точечного доступа (РАМ). Однако какой бы из этих вариантов ни был выбран, он не может считаться абсолютно верным, т. к. в последнее время появляются структуры, в которых происходит скрещивание всех известных методов. Такие гибридные алгоритмы невозможно отнести ни к одной из классификационных групп. К тому же методы индексирования многомерного пространства бурно развиваются в последние десятилетия. Появляются алгоритмы, основанные на совершенно новых принципах и имеющие свойства, нехарактерные для всех предыдущих структур [85]. Такие структуры и алгоритмы вообще могут не вписываться в общеизвестные классификации и правила.

Таблица 2.1

Свойства			Точечные методы доступа (РАМ)
Интервальное разбиение	Покрытие	Перекрытие областей	
+	+	–	<i>Quad</i> -дерево <i>K-D-B</i> -дерево <i>EXCELL</i> Файлы-решетки <i>MOLPHE</i> <i>PLOP</i> <i>LSD</i> -дерево
+	+	+	Двойные файлы-решетки
+	–	–	Многоуровневые файлы решетки <i>Buddy</i> -дерево
–	+	–	<i>BSP</i> -дерево <i>BD</i> -дерево <i>BANG</i> -файл <i>hB</i> -дерево

Далее в этой главе будут рассмотрены наиболее популярные структуры и алгоритмы, применяемые в большинстве приложений, требующих быстрой обработки точечных многомерных данных.

2.2. Иерархические методы доступа

В большинстве иерархических методов доступа предусмотрено объединение рядом находящихся точек в одну общую область. Точки такой области сохраняются в одной дисковой странице, на которую ссылается некоторый лист дерева (иногда такие узлы называют также узлами с данными). Внутренние узлы дерева (индексные узлы) предназначены только для разбиения пространства на подпространства и, соответственно, уменьшения области поиска. Причем вся область разбивается рекурсивно от корня к листу. Таким образом, процедура поиска, начавшаяся в корне и дошедшая до листа, проверяет на соответствие запросу всего лишь небольшой объем данных.

Первые структуры, разработанные для многомерного поиска, как раз относились к классу иерархических точечных методов доступа. Это были *K-D*-дерево (Bentley, 1975) [19] и *Quad*-дерево (Финкель и Бентли, 1974) [21]. Данные структуры предназначены прежде всего для оперативной памяти (*K-D*-дерево является бинарным, а узлы *Quad*-дерева имеют всего 4 потомка). В листовых узлах этих деревьев размещалась информация всего об одной многомерной точке, что является неэффективным для размещения на жестком диске, однако дает логарифмическую скорость поиска при размещении всей информации в оперативной памяти (конечно, при условии построения сбалансированного дерева, что является не всегда возможным).

В дальнейшем было разработано множество модификаций этих структур, в том числе и для хранения данных во внешней памяти (например, *K-D-B*-дерево является синтезом идей *K-D*-дерева и *B*-дерева – одного из самых эффективных алгоритмов индексирования одномерных данных, расположенных на жестком диске). Далее в данной главе будут приведены алгоритмы для оригинальных структур, а также ряда их модификаций.

2.2.1. *K-D*-дерево

K-D-дерево (*K-D-Tree*) – это индексная структура для доступа к точечным пространственным данным. Она была описана Д. Л. Бентли в 1975 году [19]. Впоследствии было предложено множество модификаций данной структуры, и семейство *K-D*-подобных деревьев стало поистине

самой популярной структурой для точечных данных в многомерном пространстве.

K-D-дерево было не самой первой структурой, разработанной для многомерных данных. Несколькоими годами ранее Д. Л. Бентли и Р. А. Финкель использовали структуру, которая получила название *Quadtree* [10, 18]. Данное дерево предназначалось только для двумерных данных. Каждый узел *Quad*-дерева разбивает двумерное пространство на четыре части, т. е. ветвление осуществляется сразу же по двум измерениям. Однако алгоритмы работы с такими деревьями (при индексировании данных большой размерности) чрезвычайно сложны, к тому же не всегда возможно построение сбалансированного *Quad*-дерева. Принимая во внимание сложность данных структур, Бентли в своих работах попытался упростить их представление, введя так называемые «*K-D-деревья*», которые имеют только двойное ветвление в каждом узле.

Термин *K-D-tree* расшифровывается как *K-Dimensional tree*, т. е. дерево в *K* измерениях. В зависимости от размерности представляемого пространства мы получаем разные виды деревьев. 1-*D-дерево* (дерево в одномерном пространстве) представляет собой обычное бинарное дерево поиска. 2-*D-дерево* (двухмерное дерево для представления объектов на плоскости) подобно ему, но ветвление в узлах происходит уже по двум направлениям (например, на нечетных уровнях ветвление происходит по координате *x*, а на четных – *y*).

В общем случае *K-D-дерево* имеет узлы с *K* координатами, и ветвление на каждом уровне базируется на сравнении одной из координат.

Структура *K-D-дерева*

Как было описано, *K-D-дерево* предназначено для организации точечных многомерных данных. В этом случае каждая запись характеризуется *k* ключами K_0, K_1, \dots, K_{k-1} . Так, в двумерном варианте запись представляет собой точку на плоскости, а в качестве ключей выступают ее координаты (K_0 – координата *x*, K_1 – координата *y*).

Бинарное дерево называется *многомерным деревом сортировки*, или *K-D-деревом сортировки*, если:

а) каждой вершине *V* сопоставлен дескриптор $Disc(V)$ так, что дескриптор корня равен нулю, дескрипторы его сыновей равны единице, дескриптор произвольной вершины в дереве равен

$$Disc(V) = (Level(V) \bmod k),$$

где $Level(V)$ – уровень вершины *V* в дереве,

k – число ключей вершины *V* (размерность пространства);

б) для каждой вершины *V* с дескриптором *j* выполнено условие: левый потомок $Left(V)$ имеет *j*-й ключ $K_j(Left(V))$ меньше *j*-го ключа $K_j(V)$ в вершине *V*, а правый потомок $Right(V)$ имеет *j*-й ключ $K_j(Right(V))$ больше *j*-го ключа в вершине *V*, т.е.

$$K_j(\text{Left}(V)) < K_j(V) < K_j(\text{Right}(V)).$$

При равенстве ключей можно поступать по-разному, но для большинства приложений полезным оказывается следующий прием. Определим для вершины V j -й суперключ $W_j(v)$, положив:

$$W_j(v) = K_j(v) K_{j+1}(v) \dots K_{k-1}(v) K_0(v) K_1(v) \dots K_{j-1}(v),$$

т.е. приняв в качестве суперключа циклическую конкатенацию ключей, начинающуюся с ключа K_j . Тогда условие (б) в определении многомерного дерева сортировки заменяется условием (б'):

б') для каждой вершины V с дескриптором j выполнено условие: левый потомок имеет j -й суперключ меньше j -го суперключа $W_j(V)$ в вершине V , а правый потомок имеет j -й суперключ больше j -го суперключа в вершине V .

Для иллюстрации изложенного рассмотрим пример K - D -дерева для двумерного евклидова пространства (рис. 2.1).

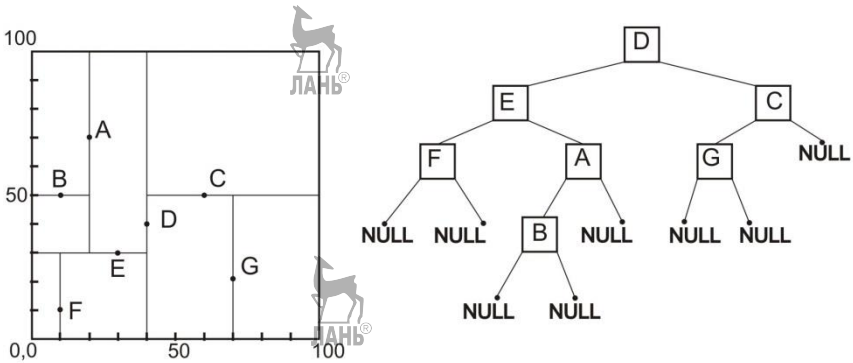


Рис. 2.1. Пример K - D -дерева

Если файл представлен K - D -деревом, то каждая запись в файле хранит узел дерева. Дополнительно к информационным полям записи у нее есть два указателя – на левого и правого сыновей. Эти указатели могут указывать либо на другие узлы дерева, либо быть равными $NULL$, что является признаком пустой вершины.

В зависимости от значения дескриптора в каждой вершине происходит деление пространства по одной из осей координат. Как следует из определения K - D -дерева, дескриптор корня равен 0, т. е. деление пространства в корневой вершине происходит по координате x . На рис. 2.1 в качестве корневой вершины выбрана точка D с координатами (40; 40). Она разбивает все пространство на две части: та часть, что находится левее линии $x = 40$ (т.е. левее вершины D), помещается в левое поддерево, а та что правее – в правое (т. е. точки A, B, E, F должны быть размещены в левом поддереве вершины D , а точки C и G – в правом).

Далее, каждая получившаяся подобласть разбивается аналогичным образом. Однако так как на этом шаге строится следующий уровень, то его дескриптор будет равен 1, а следовательно, разбиение производится по оси y . Для левого поддерева в качестве следующей вершины выбрана точка E . Она разбивает указанную подобласть линией $y = 30$ на две части – верхнюю (B и A) и нижнюю (точка F). Все точки нижней части помещаются в левое поддерево вершины E (так как они имеют координату по оси y меньше, чем у точки E), а точки верхней части – в правое поддерево.

Аналогичное разбиение продолжается и дальше, пока в каждом поддереве не останется больше точек. Если на каком-то шаге в одном из поддереьев не окажется ни одной точки, то соответствующий указатель просто обнуляется.

Как было описано ранее, каждая вершина дерева хранит в себе один элемент и дескриптор, который является признаком того, по какой из координат происходит разбиение пространства. Однако, используя формулу, указанную ранее для дескриптора, его можно рассчитывать динамически в процедурах работы с деревом. Это позволит уменьшить затраты используемой памяти в вершинах дерева.

Алгоритм поиска объекта в K-D-дереве

Процедура поиска заключается в нахождении множества записей, удовлетворяющих определенному поисковому запросу. Запрос может иметь различный вид, от простого, например $\{P \mid K_3(P) = 7\}$ (найти все вершины, у которых третий ключ равен 7), до сложного, например $\{P \mid (1 \leq K_2(P) \leq 5) \& (2 \leq K_5(P) \leq 4) \vee (K_1(P) = 8)\}$ (найти все вершины, у которых второй ключ находится в диапазоне от 1 до 5, а пятый – от 2 до 4, или те вершины, у которых первый ключ равен 8). Ниже рассматриваются три все более усложняющихся типа запросов, каждый из которых есть обобщение предыдущего.

Запрос по точному совпадению. Это простейший вид запроса. Он состоит в проверке, находится ли некоторая запись с ключами $(K_0, K_1, \dots, K_{k-1})$ в дереве. Обработка этого запроса очень проста и полностью определяется структурой и свойствами дерева. Процедура поиска объекта по точному совпадению показана в листинге 2.1.

Листинг 2.1

```
//=====
// Поиск объекта O в дереве по полному
// совпадению
// Параметры:
// O = (K0, K1, ..., Kk-1) - объект поиска
//=====
ПОИСК_ОБЪЕКТА(O)
    [1] V = корень дерева
```

```

j = 0 // дескриптор корня
[2] Если V = NULL, то
    Вернуть NULL
[3] Если V = O, то
    Вернуть V
[4] Если  $K_i(O) \geq K_i(V)$ , то
    V = Right(V)
Иначе
    V = Left(V)
j = (j+1) % K
Перейти к шагу 2
Конец ПОИСК_ОБЪЕКТА

```



Процедура ищет объект, заданный K_1, K_2, \dots, K_i ключами, в дереве поиска. Если такой объект не будет найден, то возвращается значение *NULL*, иначе – найденный объект.

На первом шаге алгоритма происходит инициализация переменных. В переменную *V* заносится текущая проверяемая вершина (изначально поиск начинается с корня), а в переменную *j* – ее дескриптор.

Если текущая проверяемая вершина отсутствует ($V = \text{NULL}$), то поиск заканчивается неудачей, в дереве нет объекта поиска и процедура просто возвращает значение *NULL* (или какой-либо другой признак неудачного поиска).

Если же текущая проверяемая вершина полностью совпадает по всем ключам с объектом поиска (для всех *i* от 0 до $(k-1)$ выполнено условие $K_i(O) = K_i(V)$), то поисковый объект найден и процедура возвращает в качестве результата вершину *V* (или какой-либо другой признак успешного поиска).

Если ни одно из описанных условий не выполнено, то необходимо перейти к следующему уровню в дереве (шаг 4). При этом сравнивается *j*-й ключ поискового объекта *O* и аналогичный ключ текущей проверяемой вершины *V*. Если ключ поискового объекта больше текущей проверяемой вершины, то переходим в правое поддерево ($V = \text{Right}(V)$), иначе – в левое ($V = \text{Left}(V)$). Затем процедура проверок повторяется заново (возврат к шагу 2).

Если в дереве предполагается выполнять только этот тип запроса, то использование *K-D*-дерева в качестве структуры данных ничем не оправдывается. В этом случае достаточно обойтись каким-либо другим типом дерева одномерной сортировки, используя в качестве ключа некоторый суперключ:

$$W = K_0, K_1, \dots, K_{k-1}.$$

K-D-дерево является многомерным деревом и дает ощутимое преимущество только при использовании многомерных запросов [3].

Запрос по множеству ключей. При этом виде запроса задаются набор ключей и их значения и требуется найти множество записей, у

которых специфицированные ключи принимают указанные значения. Пусть имеется t ключей, значения которых задано в запросе. Индексы этих ключей являются множеством $\{S_i\}$, а значения соответствующих ключей образуют другое множество $\{P_i\}$, где i принимает значения от 0 до t . Тогда искомым множеством записей будут такие вершины дерева V , которые удовлетворяют следующему условию:

$$K_{si}(V) = P_{si}, 1 \leq i \leq t.$$

Один из возможных вариантов процедуры подобного поиска представлен в листинге 2.2. При начальном вызове процедуры в нее передается множество индексов и множество значений соответствующих ключей. Также в процедуру передается текущая проверяемая вершина. Так как поиск начинается с корня дерева, то в качестве текущей вершины изначально передается корень.

Идея поиска состоит в следующем. Пусть на некотором шаге алгоритма мы попали в вершину V . Если вершина V удовлетворяет запросу, то она заносится в множество-ответ. В противном случае допустим, что дескриптор вершины V равен j . Тогда возможны две ситуации:

- $j \in \{S_i\}$ ($j = S_i$ для некоторого i , $1 \leq i \leq t$);
- $j \notin \{S_i\}$.

В первом случае продолжаем поиск в левом поддереве вершины V , если $P_{si} < K_j(V)$, и в правом, если $P_{si} > K_j(V)$. (Если существует равенство $P_{si} = K_j(V)$, то переход к правому или левому поддереву определяется принятыми соглашениями).

Если $j \notin \{S_i\}$, то мы должны продолжить поиск в обоих поддеревьях. При этом предполагается, что поддеревья не пусты.

Листинг 2.2

```
//=====
// Поиск объектов в дереве по множеству ключей
// Параметры:
// {S} - множество индексов ключей, которые заданы в
//      запросе
// {P} - множество соответствующих значений ключей
// {R} - множество-результат (в него помещаются объекты,
//      удовлетворяющие запросу)
// V - текущая вершина поиска
//=====
ПОИСК_ПО_МНОЖЕСТВУ_КЛЮЧЕЙ({S}, {P}, {R}, V)
[1] Если V = NULL, то
    Вернуться из процедуры поиска
[2] Если  $K_{S_i}(V) = P_{S_i}$  для всех  $i$  от 1 до  $t$ , то
    Добавить V в множество R
[3]  $j = \text{Disc}(V)$ 
    Если  $j = S_i$  для некоторого  $i$  от 1 до  $t$ , то
        Если  $P_j \geq K_j(V)$ , то
```

```

    ПОИСК_ПО_МНОЖЕСТВУ_КЛЮЧЕЙ({S}, {P}, {R}, Right(V))
Иначе
    ПОИСК_ПО_МНОЖЕСТВУ_КЛЮЧЕЙ({S}, {P}, {R}, Left(V))
Иначе
    ПОИСК_ПО_МНОЖЕСТВУ_КЛЮЧЕЙ({S}, {P}, {R}, Right(V))
    ПОИСК_ПО_МНОЖЕСТВУ_КЛЮЧЕЙ({S}, {P}, {R}, Left(V))
Конец ПОИСК_ПО_МНОЖЕСТВУ_КЛЮЧЕЙ

```

Для пояснения работы процедуры рассмотрим пример поиска в трехмерном евклидовом пространстве. Распределение объектов и соответствующее им дерево показаны на рис. 2.2.

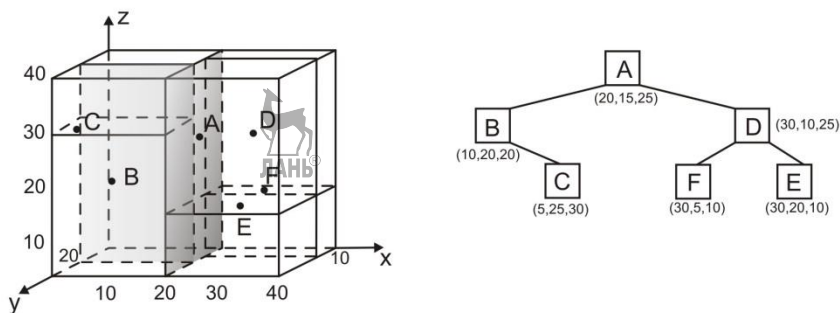


Рис. 2.2. K-D-дерево в трехмерном пространстве

Пусть задан запрос на частичное совпадение со следующими параметрами:

$$\{S\} = \{0, 2\}, \\ \{P\} \neq \{30, 10\}.$$

Это означает, что необходимо найти все вершины дерева, у которых заданы две координаты – $x = 30$ и $z = 10$. Координата y при этом может быть произвольной.

Для запуска процедуры поиска необходимо вызвать процедуру ПОИСК_ПО_МНОЖЕСТВУ_КЛЮЧЕЙ, передав ей в качестве параметров множество индексов ключей $\{S\}$, множество значений ключей $\{P\}$, пустое множество для результата $\{R\}$ и корень дерева в качестве текущей проверяемой вершины (в нашем случае это вершина A).

Алгоритм проверяет, не пустая ли вершина была передана в функцию (шаг 1), а также удовлетворяет ли текущая вершина условию поиска (шаг 2). Ни одно из этих условий не подходит для вершины A .

На третьем шаге вычисляется дескриптор просматриваемой вершины ($j = 0$) и проверяется наличие индекса с таким же номером в множестве $\{S\}$. В нашем случае множество $\{S\}$ действительно содержит индекс 0, поэтому необходимо сравнить нулевую координату текущей вершины (координату x точки A) с соответствующим значением из

множества значений ключей $\{P\}$. Так как $P_0 > K_0(A)$, то необходимо повторить указанный алгоритм только для правого поддерева (в котором находятся точки D , E и F). Для этого вызывается эта же процедура, но в качестве четвертого параметра передается правый потомок вершины A .

Вершина D также не удовлетворяет ни одному из условий шагов 1 и 2. Однако ее дескриптор ($j = 1$) не входит в множество $\{S\}$, поэтому на третьем шаге алгоритма процедуру необходимо вызывать и для левого поддерева (для вершины F), и для правого (для вершины E).

И вершина F , и вершина E удовлетворяют поисковому запросу полностью, поэтому они обе будут добавлены в множество результатов $\{R\}$. Однако так как у этих объектов нет дочерних вершин, то дальнейший рекурсивный поиск в глубину будет прекращен с помощью условия в шаге 1 приведенного алгоритма.

Таким образом, процедура поиска завершится, вернув в качестве результата множество $\{R\} = \{E, F\}$.

Для определения трудоемкости данного алгоритма необходимо узнать число вершин, посещаемых процедурой поиска. Для этого рассмотрим идеально сбалансированное дерево из N вершин (при отсутствии балансировки дерево может вырождаться в линейный список и любой поиск будет абсолютно неэффективен).

Алгоритм поиска начинается в корне и идет вниз по дереву, посещая либо одного из потомков текущей вершины (если дескриптор текущей вершины присутствует в множестве индексов $\{S\}$), или двух потомков (если дескриптора вершины нет в множестве $\{S\}$). Таким образом, темп прироста на любом уровне зависит от дескриптора этого уровня. Если в условии поиска заданы значения для всех ключей, то на каждом уровне необходимо будет посетить только одно из поддеревьев, просмотрев число вершин, равное глубине дерева (при идеально сбалансированном дереве – $\log(N)$).

Когда в условии поиска указано всего t ключей, сложность запроса, как показал Д. Л. Бенгли, оказывается равной $O(n^{(K-t)/K})$ [21, 22].

Нетрудно заметить, что описанный ранее запрос по точному совпадению ключей является частным случаем поиска по множеству ключей. Для этого просто необходимо задать в качестве условия поиска значения для всего множества ключей.

Запрос по области. Более общий тип запроса, поиск в ограниченной области, предполагает поиск объектов, попавших в некоторую область. При этом совсем не обязательно знать определение области, в которой ведется поиск, достаточно уметь вычислять две функции: В_ОБЛАСТИ (проверка нахождения вершины в некоторой области) и ВНЕ_ГРАНИЦЫ_ОБЛАСТИ (определение, имеет ли некоторая часть пространства пересечение с областью поиска). Алгоритм в этом случае примет вид, показанный в листинге 2.3. В нем используется такое понятие, как область поиска – *Res*. Это некоторая область, для которой

необходимо найти все объекты, пересекающиеся с ней. От ее определения зависит реализация функций В_ОБЛАСТИ и ВНЕ_ГРАНИЦЫ_ОБЛАСТИ. Ниже показан случай, когда область поиска представляет собой некоторый прямоугольник. В этом случае его можно определить в виде массива верхних и нижних значений для каждого измерения:

$$Rec = [Up_0, Down_0, Up_1, Down_1, \dots, Up_k, Down_k],$$

где Up_i и $Down_i$ – соответственно верхняя и нижняя границы по i -му измерению.

Аналогичным образом определяется и область пространства, соответствующая некоторому поддереву (*Bound*).

Листинг 2.3

```
//=====
// Проверка нахождения вершины в области
// Параметры:
//   V    - проверяемая вершина
//   Rec  - область поиска
//=====
В_ОБЛАСТИ(V, Rec)
    [1] // Проверка границ
        Цикл по i от 0 до (k-1)
            Если  $K_i(V) < Rec[2*i]$  или
                $K_i(V) > Rec[2*i+1]$ , то
                Вернуть ЛОЖЬ, вершина вне области
    [2] // Вершина находится внутри области
        Вернуть ИСТИНА
Конец В_ОБЛАСТИ

//=====
// Определение пересечения двух областей
// Параметры:
//   Rec  - область поиска
//   Bound - область, соответствующая некоторому поддереву
//=====
ВНЕ_ГРАНИЦЫ_ОБЛАСТИ(Rec, Bound)
    [1] // Сравнение границ
        Цикл по i от 0 до (k-1)
            Если  $Bound[2*i] < Rec[2*i+1]$  или
                $Bound[2*i+1] > Rec[2*i]$ , то
                Вернуть ИСТИНА, Bound не пересекается с Rec
    [2] Вернуть ЛОЖЬ, область Bound пересекается с Rec
Конец ВНЕ_ГРАНИЦЫ_ОБЛАСТИ

//=====
// Поиск объектов в дереве по области
// Параметры:
//   Rec  - область поиска
```



```

// Bound - область, соответствующая некоторому поддереву
// V      - текущая вершина поиска
// {R}    - множество-результат (в него помещаются
//          объекты, удовлетворяющие запросу)
//=====
ПОИСК_В_ОБЛАСТИ(Rec, Bound, V, {R})
[1] // Проверка, существует ли вершина V
    Если V = NULL, то
        Вернуться из процедуры поиска
[2] // Добавление вершины V в {R} при
    необходимости
    Если В_ОБЛАСТИ(V, Rec), то
        Добавить V в множество R
[3] j = Disc(V)
[4] // Создание области для обоих поддеревьев
    Цикл по i от 0 до (k-1)
        BoundL[i] = Bound[i]
        BoundR[i] = Bound[i]
        BoundL[2*j+1] = Kj(V)
        BoundR[2*j] = Kj(V)
[5] // Проверить левое поддерево
    Если !(ВНЕ_ГРАНИЦЫ_ОБЛАСТИ(Rec, BoundL)), то
        ПОИСК_В_ОБЛАСТИ(Rec, BoundL, Left(V), {R})
[6] // Проверить правое поддерево
    Если !(ВНЕ_ГРАНИЦЫ_ОБЛАСТИ(Rec, BoundR)), то
        ПОИСК_В_ОБЛАСТИ(Rec, BoundR, Right(V), {R})
Конец ПОИСК_В_ОБЛАСТИ

```

Следует отметить тот факт, что если в системе предусмотрен поиск только по прямоугольной области, то алгоритм может значительно упроститься. В частности, можно не использовать процедуру ВНЕ_ГРАНИЦЫ_ОБЛАСТИ, а соответствующие условия просто внести в основную процедуру. Данный вариант алгоритма показан в листинге 2.4.

Листинг 2.4

```

//=====
// Поиск объектов в дереве по области
// Параметры:
//   Rec - область поиска
//   V   - текущая вершина поиска
//   {R} - множество-результат (в него помещаются объекты,
//          удовлетворяющие запросу)
//=====
ПОИСК_В_ОБЛАСТИ(V, Rec, {R})
[1] // Проверка, существует ли вершина V
    Если V=NULL, то
        Вернуться из процедуры поиска

```

```

[2] // Добавление вершины V в {Res} при
    // необходимости
    Если В_ОБЛАСТИ(V, Rec), то
        Добавить V в множество R
[3] j = Disc(V)
[4] // Проверить левое поддерево
    Если  $K_j(V) \geq \text{Rec}[2*i]$ , то
        ПОИСК_В_ОБЛАСТИ(Left(V), Rec, {R})
[5] // Проверить правое поддерево
    Если  $K_j(V) \leq \text{Rec}[2*i+1]$ , то
        ПОИСК_В_ОБЛАСТИ(Right(V), Rec, {R})
Конец ПОИСК_В_ОБЛАСТИ

```

Рассмотрим работу данного алгоритма более подробно на примере. Пусть задано двумерное дерево, показанное на рис. 2.1. Допустим, что необходимо найти все точки, координаты которых удовлетворяют следующим ограничениям:

$$\begin{aligned} 50 &\leq x \leq 80, \\ 10 &\leq y \leq 60. \end{aligned}$$

В этом случае массив для области поиска будет иметь следующий вид:

$$\text{Rec} = [50, 80, 10, 60].$$

Для выполнения процедуры поиска необходимо вызвать процедуру из листинга 2.4, передав ей в качестве параметров вершину D (так как согласно рис. 2.1, вершина D является корнем дерева), массив границ поиска Rec и пустое множество для результата поиска.

Алгоритм на первом шаге проверяет, не пустая ли вершина была передана. Если вершина является не пустой, то проверяется попадание ее в область поиска (шаг 2). В рассматриваемом случае вершина D является не пустой и она не попадает в область поиска, поэтому соответствующие действия для данных шагов алгоритма просто пропускаются.

На третьем шаге определяется дескриптор проверяемой вершины ($j = \text{Disc}(D) = 0$), а затем сравнивается соответствующий ключ вершины D с границами поисковой области. Так как $K_0(D) < \text{Rec}[0]$ (координата x вершины D меньше левой границы области поиска, $40 < 50$), то левое поддерево проверять не нужно.

Однако условие для правой границы выполнено ($K_0(D) < \text{Rec}[1]$), поэтому эта же функция вызывается рекурсивно для правого поддерева (для вершины C). Выполняя проверки шагов 1 и 2, можно увидеть, что вершина C попадает в область поиска, поэтому она должна быть помещена в множество результатов $\{R\}$. Проверки шагов 3 и 4 для вершины C дадут положительный результат, поэтому процедуру поиска необходимо будет выполнить рекурсивно для обоих поддеревьев вершины C (в случае левого поддерева в множество результатов будет

добавлена еще одна вершина – G , в случае правого – поддереву окажется пустым и процедура просто завершит свою работу).

Поиск ближайшего соседа. Несколько особняком стоит запрос следующего вида. Для данной функции расстояния $Dist$, семейства точек $\{S\}$ в k -мерном пространстве и точки P (в том же пространстве) необходимо найти в $\{S\}$ ближайшего соседа точки P . Ближайший сосед – это такая точка Q , что

$$(\forall R \in \{S\}) (R \neq Q) \Rightarrow (Dist(R, P) \geq Dist(Q, P)).$$

Геометрически это означает, что необходимо найти точку, расположенную в пространстве ближе всех остальных к данной точке P . Аналогичный запрос можно сформулировать и для нахождения m ближайших соседей точки P .

На самом деле данный запрос является очень сложным. Первоначальный алгоритм, предложенный Д. Л. Бентли в оригинальной работе [21], труден для понимания и реализации. После этого были предложены другие варианты, более быстрые, понятные и простые. Многие из них предназначены для модификаций K - D -деревьев.

Листинг 2.5

```
//=====
// Поиск ближайшего соседа
// Параметры:
//   P      - точка, для которой производится поиск
//            ближайшего соседа
//   V      - текущая проверяемая вершина
//   BestV   - лучшее приближение на данном шаге
//=====
ПОИСК СОСЕДА (P, BestV, V)
[1] // Проверка на завершение
    Если V = NULL, то
        Вернуть BestV
[2] // Проверка текущей вершины
    Если Dist(P, V) < Dist(P, BestV), то
        BestV = V
        j = Disc(V)
[3] // Поиск в поддеревьях
    Если  $K_j(P) \geq K_j(V)$ , то
        BestV = ПОИСК СОСЕДА (P, Right(V), BestV)
    Если  $K_j(P) - K_j(V) < Dist(P, BestV)$ , то
        BestV = ПОИСК СОСЕДА (P, Left(V), BestV)
    Иначе
        BestV = ПОИСК СОСЕДА (P, Left(V), BestV)
    Если  $K_j(V) - K_j(P) < Dist(P, BestV)$ , то
        BestV = ПОИСК СОСЕДА (P, Right(V), BestV)
[4] Вернуть BestV
Конец ПОИСК СОСЕДА
```

Мы рассмотрим один из наиболее простых и понятных вариантов. Несмотря на свою простоту, он использует основные отсеечения и является эффективным для сбалансированных деревьев. Вариант для поиска m ближайших соседей из-за его сложности здесь представлен не будет.

Процедура просматривает поддерево, заданное третьим параметром, и возвращает вершину, которая находится ближе к точке P , чем переданная ей в качестве второго параметра ($BestV$). Если такой вершины найдено не будет, то возвращается значение $BestV$. Для выполнения поиска во всем дереве необходимо запустить процедуру со следующими параметрами:

ПОИСК_СОСЕДА (P , $Root$, $Root$).

Данный алгоритм начинается с проверки существования текущего поддерева (шаг 1). Если поддерево отсутствует ($V = NULL$), то та вершина, что была выбрана ранее как ближайший сосед и является лучшим кандидатом. Процедура ее и возвращает.

На втором шаге алгоритм сравнивает текущую вершину V и ту, которая считается лучшей на данный момент – $BestV$. Если текущая вершина ближе к точке P , то ее запоминают как лучшую.

Последнее, что необходимо сделать, это провести процедуру поиска ближайшего соседа в поддеревьях данной вершины (шаг 3). Сначала поиск ведется в ближайшем к точке P поддереве, а затем (если это необходимо) – в дальнем поддереве. Такой порядок поиска не является обязательным, однако он дает выигрыш в производительности процедуры.

Алгоритм добавления нового объекта в K -D-дерево

Включение новой вершины в K -D-дерево принципиально не отличается от аналогичной процедуры для одномерных бинарных деревьев. Построение дерева также начинается с создания вершины $Root$ – корня дерева, левому и правому его потомку которой присваивается значение $NULL$:

$Left(Root) = Right(Root) = NULL$.

Для включения новой записи в K -D-дерево проводится ее поиск. При удачном завершении процедуры поиска вставка не производится (данная вершина уже находится в дереве). Если же поиск заканчивается неудачей, то запись вставляется на место левого или правого потомка той вершины, на которой был закончен поиск. Процедура вставки представлена в листинге 2.6.

Листинг 2.6

```
//=====
// Вставка нового объекта в дерево
// Параметры:
```

```

// V - запись, которую необходимо вставить в дерево
//=====
ВСТАВКА(V)
[1] // Подготовка переменных
    Left(V) = Right(V) = NULL
    V' = Root // текущий объект поиска
    j = 0     // дескриптор корня
[2] // Проверка существования корня
    Если Root = NULL, то
        Root = V
        Выход из процедуры
[3] // Проверка на присутствие объекта в дереве
    Если V' = V, то
        Выход из процедуры, вершина есть в дереве
[4] Если  $K_i(V) \geq K_i(V')$ , то
    Если Right(V') = NULL, то
        Right(V') = V
        Выход из процедуры, вершина добавлена
    Иначе
        V' = Right(V')
Иначе
    Если Left(V') = NULL, то
        Left(V') = V
        Выход из процедуры, вершина добавлена
    Иначе
        V' = Left(V')
    j = (j+1) % K
    Перейти к шагу 3
Конец ВСТАВКА

```

Данная процедура практически полностью повторяет шаги той, что представлена в листинге 2.1. Единственным отличием являются завершающие этапы. Если объект в результате поиска не был найден в дереве, то добавляется новая запись (а не возврат из процедуры, как было представлено в листинге 2.1).

Построить *K-D*-дерево можно с помощью последовательных вставок объектов в пустое дерево. Нетрудно проверить, что *K-D*-дерево, представленное на рис. 2.1, может быть получено последовательной вставкой записей в порядке *D, E, F, A, B, C* и *G*.

Построение дерева с помощью последовательных вставок может привести к неоптимальной структуре дерева. Так, если те же вершины вставлять в другой последовательности (например, *C, G, D, A, B, E, F*), то вместо дерева получится простой список и все процедуры поиска будут неэффективны.

Бентли в своей работе доказал, что на реальных данных с помощью случайной последовательной вставки элементов в изначально пустое дерево получается структура, аналогичная по своим свойствам со

случайным одномерным бинарным деревом. А для одномерного случая выведена теорема, согласно которой поиск в случайно построенном дереве приблизительно равен $1,386 * \log_2 n$ [22].

Алгоритм удаления вершины из K-D-дерева

Удаление вершины из K-D-дерева можно рассматривать как удаление корня некоторого поддеревя. Если вершина V не имеет поддеревьев (листовой узел), то ее можно просто удалить. Если вершина V имеет потомков, то она может быть заменена одной из них, скажем вершиной Q , которая сохранит порядок, установленный вершиной V . Иными словами, все вершины из правого поддерева вершины V должны быть в правом поддереве вершины Q и то же самое должно быть выполнено относительно левого поддерева. Предположим, что дискриптор вершины V равен j . Тогда вершина Q должна быть j -м максимальным элементом (вершиной, в которой ключ K_j имеет наибольшее значение среди всех вершин поддерева) в левом поддереве вершины V (или j -минимальным в правом поддереве вершины V). Когда вершина Q найдена, она может служить новым корнем вместо вершины V , нужно только провести необходимую реорганизацию дерева для удаления вершины Q из прежней позиции в K-D-дереве (листинг 2.7).

Листинг 2.7

```
//=====
// Удаление объекта из K-D-дерева
// Параметры:
//   O - объект, который необходимо удалить
//=====
УДАЛЕНИЕ(O)
[1] // Поиск удаляемого объекта
    V = ПОИСК_ОБЪЕКТА(O)
    Если V = NULL, то
        Выйти из процедуры, удаляемого объекта нет
[2] // Удаление вершины
    УДАЛЕНИЕ_ВЕРШИНЫ(V)
Конец УДАЛЕНИЕ

//=====
// Удаление вершины K-D-дерева
// Параметры:
//   V - вершина, которую необходимо удалить
//=====
УДАЛЕНИЕ_ВЕРШИНЫ(V)
[1] // Удаление листа
    Если Left(V) = NULL и Right(V) = NULL, то
        Если Parent(V) ≠ NULL, то
            Если Left(Parent(V)) = V, то
                Left(Parent(V)) = NULL
```

```

        Иначе
            Right(Parent(V)) = NULL
        Удалить V
        Выйти из процедуры, объект удален
[2] // Поиск замены для вершины V
    j = Disc(V)
    Если Right(V) = NULL, то
        Q = максимальный узел из Left(V) по Kj
    Иначе
        Q = минимальный узел из Right(V) по Kj
[3] // Удаление вершины Q из дерева
    УДАЛЕНИЕ_ВЕРШИНЫ(Q)
[4] // Замещение удаляемой вершины найденной
    Left(Q) = Left(V)
    Right(Q) = Right(V)
    Если Parent(V) ≠ NULL, то
        Если Left(Parent(V)) = V, то
            Left(Parent(V)) = Q
        Иначе
            Right(Parent(V)) = Q
    Удалить V
Конец УДАЛЕНИЕ_ВЕРШИНЫ

```

Процедура удаления объекта из *K-D*-дерева (процедура УДАЛЕНИЕ) начинается с поиска этого объекта в дереве (шаг 1). Если объекта *O* в дереве нет, то процедуру удаления нужно завершить. Если же в дереве найдена вершина *V*, которая содержит удаляемый объект *O*, то вызывается процедура удаления вершины УДАЛЕНИЕ_ВЕРШИНЫ.

При удалении вершины из дерева могут возникнуть две ситуации: первая – удаляемая вершина является листовой вершиной. В этом случае вершина просто удаляется, а соответствующая ссылка у предка делается равной *NULL* (шаг 1).

Вторая возможная ситуация – удаление внутренней вершины дерева. В этом случае удаляемую вершину нужно рассматривать как корень поддерева (шаги 2–4).

Стратегия удаления внутренней вершины дерева заключается в следующем: производится поиск в поддеревьях удаляемой вершины такой вершины, замена на которую позволит не перестраивать структуру дерева. Такой вершиной может являться либо максимальная по *j*-му ключу вершина из левого поддерева, либо минимальная по *j*-му ключу вершина правого поддерева (где *j* – дескриптор удаляемой вершины).

После нахождения вершины *Q*, которой можно заменить удаляемую вершину *V*, вершина *Q* удаляется из прежнего места в дереве (шаг 3) и добавляется вместо вершины *V* (шаг 4).

Условный оператор на шаге 2[®] может быть источником неприятностей при множественном удалении, так как при

последовательном удалении вершин из дерева он все время производит удаление вершин из правого поддерева. Это продолжается до тех пор, пока не будет исчерпано все правое поддерево, что приведет к резко деформированному дереву. Чтобы избежать этого, можно использовать различные стратегии. Например, на втором шаге можно выбирать вершину из большего по размеру поддерева или поочередно из правого и левого поддеревьев, или выбирать поддерево с помощью датчиков случайных чисел.

Для оценки сложности алгоритма рассмотрим основные его части. Вначале идет поиск вершины дерева, содержащей удаляемый объект. Сложность данной процедуры для оптимального дерева является $O(\log_2 N)$. Следующим сложным элементом алгоритма является поиск нового кандидата для замены удаляемой вершины. Эта операция имеет такую же сложность, как и поиск по множеству ключей с заданными $(K-1)$ ключами ($O(N^{(K-1)/K})$). В заключение стоит отметить, что процедура удаления вершины дерева может привести к рекурсивному удалению вершин в его поддеревьях. Однако Д. Л. Бентли показал, что суммарная стоимость процедуры удаления все же остается равной $O(\log_2 N)$ [19, 21, 22].

Алгоритм построения оптимизированного K-D-дерева

В некоторых приложениях использовать K - D -деревья, получаемые последовательным добавлением объектов в произвольном порядке, невыгодно из-за возникающей конфигурации дерева (возможно, неудачной). К ним относятся случаи, когда впоследствии нужно будет выполнять большое число поисковых операций при малом числе добавлений и удалений (статичные данные), а также когда заранее известно, что объекты будут подаваться на вход алгоритма вставки в «разрушительном» порядке, при котором вместо древовидной структуры может получиться линейный список. В таких случаях можно осуществить процедуру оптимизации K - D -дерева, при которой получится такая структура, у которой все листья будут расположены на двух смежных уровнях (хотя эта операция и является довольно трудоемкой).

Ниже приведен алгоритм, который строит K - D -дерево таким образом, что число вершин в правом поддереве каждой внутренней вершины будет отличаться от числа вершин в ее левом поддереве не более чем на единицу. Чтобы построить оптимизированное K - D -дерево, используется процедура ПОСТРОИТЬ_ДЕРЕВО с параметрами $List$ и j , где $List$ – множество записей, заданное, например, в виде связанного списка, j – дескриптор, который может принимать значение от 0 до $(K-1)$. Процедура возвращает указатель на корень оптимизированного дерева, при этом корень имеет дескриптор $j = 0$ (листинг 2.8).


```

//=====
// Оптимизация K-D-дерева
// Параметры:
// List - множество записей для включения в дерево
// j - дескриптор
//=====
ПОСТРОИТЬ_ДЕРЕВО(List, j)
[1] // Проверка на пустое множество
    N = количество элементов в List
    Если N = 0, то
        Выйти из процедуры и вернуть NULL
[2] // Поиск медианы множества List
    СОРТИРОВАТЬ(List, j)
    M = [N/2]
    V = List[M]
[3] // Разбиение набора записей на два множества
    Переместить элементы с 0 до M из List в ListL
    Переместить элементы с (M+1) до N из List в ListR
[4] // Построение дерева
    j = (j+1) % K
    Left(V) = ПОСТРОИТЬ_ДЕРЕВО(ListL, j)
    Right(V) = ПОСТРОИТЬ_ДЕРЕВО(ListR, j)
[5] // Завершение процедура
    Вернуть V
Конец ПОСТРОИТЬ_ДЕРЕВО

```

Изначально в процедуру оптимизации передается все множество объектов (*List* содержит все записи) и дескриптор корня, равный 0.

На первом шаге алгоритм проверяет, не является ли множество объектов пустым. Так как алгоритм рекурсивный, то рано или поздно такая ситуация должна произойти. В этом случае производится выход из процедуры построения дерева, а в качестве результата возвращается *NULL*.

Если множество *List* не пустое, то на следующих шагах оно разбивается на два приблизительно равных по размеру подмножества, которые будут являться левым и правым поддеревом. Для этого сортируются элементы множества *List* по *j*-му ключу в порядке возрастания (выбор метода сортировки в процедуре СОРТИРОВАТЬ не является важным с функциональной точки зрения и влияет только на быстродействие процедуры построения в целом). После этого выбирается средний элемент множества в качестве текущего корня поддеревя, а все элементы, находящиеся левее выбранного в отсортированном списке, помещаются в множество для левого поддеревя (*ListL*), а находящиеся правее – в правое (*ListR*).

Для получившихся множеств вызывается рекурсивно процедура построения дерева, и ее результат принимается в качестве левого и правого потомков текущей вершины (шаг 6).

Трудоёмкость алгоритмов обработки K-D-дерева

Как показал Д. Л. Бентли [21], случайно растущее *K-D*-дерево имеет те же значения средней длины пути и то же распределение ветвей, что и обычное бинарное дерево поиска, потому что предположения, лежащие в основе их роста, те же, что и в одномерном случае. Поэтому для случайного *K-D*-дерева мы получаем практически те же значения быстродействия, что и в одномерном случае.

Если файл не изменяется динамически, можно сбалансировать любое *K-D*-дерево с N узлами так, чтобы его высота составляла примерно $\log_2 N$, выбрав среднее значение для ветвления в каждом узле (один из вариантов такой процедуры был показан выше). После этого можно быть уверенным в эффективности обработки запросов различных фундаментальных типов. Так, Д. Л. Бентли доказал, что можно найти все записи, имеющие t определенных координат (запрос по частичному совпадению), за $O(N^{(K-t)/K})$ шагов. Кроме того, можно найти все записи, лежащие в заданной прямоугольной области, не более чем за $O(N^{t-1/k} + q)$ шагов, если всего имеется q таких записей. В действительности, если данная область близка к кубической, и q мало, и если к тому же выбранные для ветвления координаты в каждом узле имеют наибольший разброс значений атрибутов, то, как показано в работе Фридмана, Бентли и Финкеля [35], среднее время обработки запроса в такой области будет составлять всего лишь $O(\log_2 N + q)$. Эта же формула применима и при поиске в подобном *K-D*-дереве ближайших соседей некоторой точки K -мерного пространства.

В заключение хотелось бы отметить трудоёмкость основных операций редактирования дерева. Так, добавление новой вершины в дерево имеет в среднем трудоёмкость, равную $O(\log_2 N)$. Удаление корневой вершины имеет трудоёмкость $O(N^{(k-1)/k})$, а удаление произвольной вершины – $O(\log_2 N)$. Трудоёмкость построения оптимизированного дерева составляет $O(N * \log_2 N)$, при этом оптимизированное дерево гарантирует логарифмическую трудоёмкость поиска.

Дальнейшее развитие идей, заложенных в K-D-деревьях

Одним из существенных недостатков *K-D*-деревьев является то, что структура дерева очень сильно зависит от того, в каком порядке вставлялись вершины при его построении (если использовался классический алгоритм построения с помощью вставок вершин, а не построение сбалансированного дерева). В последующие годы многие исследователи пытались придумать модификации алгоритмов,

позволяющие уменьшить влияние порядка вставки вершин на конечную структуру получающегося дерева.

Еще одним недостатком можно считать то, что многомерные объекты (точки) как бы «размазаны» по всему дереву. Для многих алгоритмов более эффективным и удобным является вариант, когда все объекты находятся в листьях дерева. В 1979 году Бентли и Фридман [22] попробовали избавиться от этих двух недостатков. Они предложили модификацию структуры, получившую название адаптивные *K-D-деревья* (*adaptive K-D-tree*). Суть изменений заключалась в следующем: предлагалось во внутренних вершинах дерева в качестве разделителя выбирать не вставляемый объект, как в оригинальных *K-D-деревьях*, а некоторое произвольное число, которое позволит разбить все множество объектов на два приблизительно равных набора. Процедура построения дерева при этом очень похожа на процедуру, описанную в листинге 2.8.

Адаптивные деревья являются почти статичной структурой. Они показывают очень хорошее быстроедействие на наборах данных, которые известны заранее. Если же в программе предусмотрена частая вставка новых и удаление существующих объектов, то их производительность резко снижается.

В 1984 году Тамминен [87] предложил еще одну модификацию деревьев, получившую название *Bintree*. В этой структуре предлагается рекурсивно разбивать все пространство *K*-размерными гиперпрямоугольниками одинакового размера до тех пор, пока в каждом из них не останется ровно один объект. Главным достоинством такого подхода является то, что в вершинах не нужно хранить координаты гиперплоскости разбиения. Их всегда можно вычислить, зная положение объекта в дереве (принимая во внимание тот факт, что гиперпрямоугольники разбиения имеют равный размер). Данная идея в дальнейшем применялась и к другим структурам.

Ломет и Сальдзберг в своей работе [63] обратили внимание на то, что одним из существенных недостатков *K-D-деревя* является тот факт, что часто невозможно в процессе построения выбрать гиперплоскость по правилам построения структуры, которая бы наиболее адекватно разбивала пространство.

Этот недостаток попытались устранить, позволив в каждой вершине разбивать пространство произвольной гиперплоскостью, даже не параллельной осям пространства. Такая структура впервые была предложена Финчем, Кедедом и Нэйлором в 1980 году и получила название *BSP-дерево* [35]. В дальнейшем она также была многократно модернизирована. Более подробно она обсуждается в следующих параграфах.

Также представляют интерес *LSD-дерево* [46] и *hB-дерево* [63]. В них несколько узлов объединяются в одну общую страницу, которая и

размещается во внешней памяти, что также позволяет оптимизировать *K-D*-дерево для внешней памяти. Более подробно эти структуры будут также рассмотрены далее.



2.2.2. *K-D-B*-дерево

Рассмотренное ранее *K-D*-дерево является одной из самых популярных структур индексирования точечных данных в оперативной памяти. Однако данная структура малопригодна для внешней памяти. При этом самым большим ее недостатком является одно из ее же достоинств: в сбалансированном дереве любая вершина может быть найдена не более чем за $\log_2(N)$ шагов, где N – общее число проиндексированных данных. Это очень хорошие показатели для поиска в оперативной памяти. Однако при размещении данных во внешней памяти для поиска объекта нам необходимо произвести $\log_2(N)$ доступов к жесткому диску. Каждое обращение к внешней памяти занимает время, равное сотням или даже тысячам операций в оперативной памяти. Поэтому такие показатели становятся неприемлемыми.

В свое время с подобной проблемой столкнулись исследователи, пытающиеся применить обычные бинарные деревья поиска для индексирования данных на жестком диске. В результате многочисленных исследований было предложено *B*-дерево (Р. Бэйер и Е. Маккрейт [92]), позволяющее многократно снизить число доступов к жесткому диску.

В 1981 году Д. Т. Робинсон предложил подобную технологию и для *K-D*-дерева [74]. По сути, он разработал алгоритмы, скрестившие идеи *K-D*-дерева и *B*-дерева. Этот подход позволил индексировать большие объемы точечных данных во внешней памяти. В настоящее время разработаны алгоритмы, обладающие большей эффективностью, чем предложенное *K-D-B*-дерево, однако в некоторых приложениях до сих пор еще применяются *K-D-B*-деревья или их модификации.

Структура *K-D-B*-дерева



По своей структуре *K-D-B*-дерево похоже на *B*-дерево. Оно так же, как и ее предшественник является сильноветвящейся иерархической структурой с фиксированным размером узла. От *B*-дерева рассматриваемая структура унаследовала и еще одно очень важное и полезное свойство – полную сбалансированность. В отличие от *K-D*-дерева, путь от корня к листовым узлам является одинаковым для всех листовых узлов дерева.

Так как *K-D-B*-дерево разрабатывалось исключительно для внешней памяти и предназначено для больших объемов информации, то каждый узел этого дерева (как внутренний, так и листовой) сохраняется в одной странице на жестком диске. Однако в отличие от *B*-деревьев, в данной структуре не гарантируется 50%-ное использование памяти.

Теоретически вполне возможен случай, когда полезные данные индексной структуры занимают намного менее 50% от всего объема занятой памяти. Это является следствием неэффективного рекурсивного расщепления при вставке объектов в дерево. Однако в практических задачах малая заполненность маловероятна. Многочисленные эксперименты показали, что использование памяти структурой находится в пределах 60% для двух- и трехмерного дерева. Это характерно при случайной вставке равномерно распределенных данных. Для деревьев более высокой размерности показатель заполненности сильно ухудшается даже для равномерных данных. Однако использование памяти можно увеличить, если использовать некоторую технологию реорганизации добавляемых данных.

В последнее время появились структуры, имеющие гораздо более высокий показатель заполненности (они будут рассмотрены позже).

По принципу хранения данных *K-D-B*-дерево практически полностью совпадает со вторым своим прародителем – *K-D*-деревом. В листовых узлах дерева находятся элементы, которые можно представить, как точки в некотором k -мерном пространстве, т. е. любая запись, индексируемая данной структурой, представляет собой кортеж k ключей K_0, K_1, \dots, K_{k-1} . Для случая с двумя ключами такую запись можно представить, как некоторую точку на плоскости, первый ключ которой K_0 соответствует координате x , а второй ключ K_1 – координате y . При делении пространства поиска, по аналогии с *K-D*-деревом, происходит его разбиение некоторой гиперплоскостью вдоль одной из осей пространства (для двумерных данных в качестве гиперплоскости выступает прямая, параллельная оси Ox или Oy). Однако, в отличие от *K-D*-дерева, в *K-D-B*-дереве нет привязки оси пространства, вдоль которой происходит разбиение пространства в некотором узле, к уровню этого узла в дереве. Вместо этого в каждом узле хранятся координаты ограничивающего гиперпрямоугольника подпространства, полученные после такого разбиения. Это вносит дополнительные накладные расходы и усложняет некоторые алгоритмы, однако позволяет объединить несколько разбиений пространства в одном узле дерева. Позднее были разработаны структуры, которые позволяют не отказываться от данного свойства *K-D*-дерева, однако алгоритмы работы с такими деревьями усложнились еще сильнее.

Рассмотрим структуру *K-D-B*-дерева и его свойства более подробно. Для примера будем ориентироваться на двухмерное дерево, представленное на рис. 2.3. Однако все изложенные особенности также относятся к дереву произвольной размерности.

Все узлы *K-D-B*-дерева можно разделить на два класса по типу хранящейся в них информации: внутренние (иногда называемые страницами областей) и листовые (называемые страницами данных).

Внутренние узлы дерева. Данные узлы не содержат непосредственно объектов индексирования и предназначены только для разбиения пространства на подобласти поиска. Они аналогичны внутренним узлам адаптивного *K-D-дерева* (*adaptive K-D-Tree*), в которых также не хранятся объекты данных. На рис. 2.3 такими узлами являются узлы *A*, *B*, *C* и *D*.

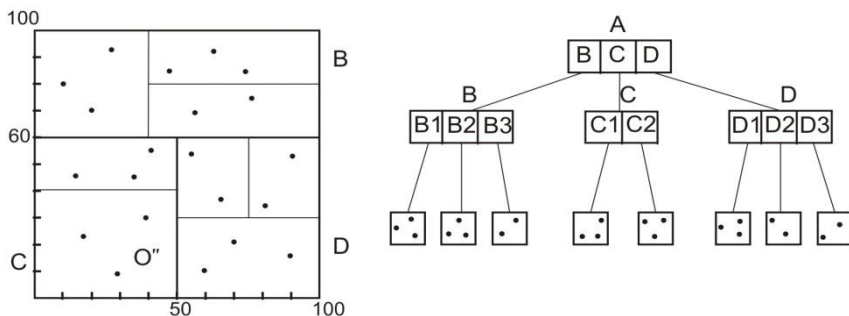


Рис. 2.3. Пример *K-D-B-дерева*

Каждый внутренний узел дерева представляет собой массив записей следующего типа:

$[BR, \text{ссылка_на_потомка}]$,

где *BR* – *bounding rectangle* – ограничивающий прямоугольник или область, способная разместить все элементы, находящиеся в дочерних узлах; *ссылка на потомка* – указатель на дочерний узел.

Так, на рис. 2.3 корневой узел представляет собой массив, состоящий из следующих трех элементов:

$[\{0,60\}-\{100,100\}, \text{ссылка_на_узел_B}]$,

$[\{0,0\}-\{50,60\}, \text{ссылка_на_узел_C}]$,

$[\{50,0\}-\{100,60\}, \text{ссылка_на_узел_D}]$.

Ограничивающие области в этом примере представляют собой координаты двух точек – нижнего левого угла и верхнего правого угла. В качестве некоторой координаты каких-либо областей разрешается выбирать значения бесконечности. Это означает, что в данном узле находятся объекты, расположенные в неограниченной подобласти. В практических реализациях в качестве бесконечности выбирают либо максимально-возможное (и минимально-возможное) число, вмещающиеся в разрядную сетку, либо максимальные и минимальные координаты объектов, предусмотренные по условию задачи. В нашем примере в качестве нижней границы по обоим осям было выбрано значение 0, а в качестве верхней – значение 100.

Листовые узлы дерева. Данные узлы не содержат координат ограничивающих областей. Они представляют собой массив координат

точек, соответствующих индексируемым объектам, находящимся в заданной части пространства. В более сложных системах, помимо координат точек в листовых узлах, могут содержаться ссылки на записи, хранящие дополнительную информацию об индексируемых объектах. В примере, представленном на рис. 2.3, листовыми узлами являются все узлы дерева, расположенные на третьем уровне.

Алгоритмы построения и работы с *K-D-B*-деревом были спроектированы так, чтобы полученная структура всегда удовлетворяла следующим требованиям.

1. Данная структура является сбалансированной, т.е. путь от корня до любого листового узла дерева имеет одну и ту же длину.

2. Индексируемые объекты находятся только в листовых узлах дерева (внутренние узлы дерева предназначены только для сокращения пространства поиска и не содержат объектов).

3. Каждый внутренний узел дерева имеет хотя бы одного потомка (внутренние узлы дерева не могут быть пустыми, однако в дереве допускается наличие пустых листовых узлов).

4. Если некоторый элемент внутреннего узла ссылается на листовую вершину дерева, то все объекты этой вершины находятся внутри области данного элемента.

5. Если некоторый элемент внутреннего узла ссылается на нелистовую вершину дерева, то область данного элемента равна объединению всех областей его дочерних узлов.

6. Области дочерних узлов некоторого внутреннего узла дерева имеют нулевое пересечение (свойство однозначности поиска, позволяющее однозначно идентифицировать единственную вершину расположения некоторого объекта).

7. Область корневого узла равна всему индексируемому пространству.

Алгоритм поиска объекта в K-D-B-дереве

Процедуры поиска в *K-D-B*-дереве похожи на аналогичные процедуры в *K-D*-дереве с той лишь разницей, что *K-D-B*-дерево является сильноветвящимся. Поэтому в каждой вершине необходимо выбирать путь продвижения по дереву не между левым и правым поддеревьями, а среди всех потомков данной вершины.

Для примера рассмотрим два вида поиска – поиск по точному совпадению ключей объекта и поиск по области (принадлежность ключей объекта некоторой области поиска).

Запрос по точному совпадению. Это самый простой вид поиска. Он заключается в нахождении объекта с ключами $(K_0, K_1, \dots, K_{k-1})$ в дереве. Данный алгоритм полностью определяется структурой *K-D-B*-дерева и теми свойствами, которым оно должно удовлетворять.

Функция поиска объекта по точному совпадению ключей представлена в листинге 2.9. Она возвращает листовой узел дерева, в котором находится найденный объект. В приведенном листинге алгоритм разбит на две функции – функцию поиска листовой вершины ПОИСК_ЛИСТА, в которой может находиться объект запроса, и непосредственно сам поиск этого объекта в листе (функция ПОИСК_ОБЪЕКТА). Это сделано специально, потому что функция ПОИСК_ЛИСТА будет использована нами в дальнейшем в алгоритмах удаления и вставки объектов.

Листинг 2.9

```
//=====
// Поиск объекта О в дереве по полному
// совпадению
// Параметры:
//   О = (K0, K1, ..., Kk-1) – объект поиска
//=====
ПОИСК_ОБЪЕКТА(О)
[1] // Ищем лист, в котором возможно будет О
    L = ПОИСК_ЛИСТА(О)
[2] // Если такого листа нет, то вернуть ЛОЖЬ
    Если L = NULL, то
        Выйти из процедуры и вернуть ЛОЖЬ
[3] // Проверяем, есть ли в этом узле О
    Для всех объектов О' вершины L проверить
        Если О' = О, то
            Выйти из процедуры и вернуть ИСТИНА
[4] // Объект О не найден, возвращаем ЛОЖЬ
    Вернуть ЛОЖЬ
Конец ПОИСК_ОБЪЕКТА

//=====
// Поиск листовой вершины, в которой может
// находиться объект О
// Параметры:
//   О = (K0, K1, ..., Kk-1) – объект поиска
//=====
ПОИСК_ЛИСТА(О)
[1] // Если дерево не существует,
    // то вернуть NULL
    Если Корень = NULL, то
        Выйти из процедуры и вернуть NULL
[2] // Начальная инициализация переменной V
    V = Корень дерева
[3] // Если дошли до листа, то вернуть его
    Если V является листовой вершиной, то
        Выйти из процедуры и вернуть V
[4] // Найти дочернюю вершину и перейти к ней
```


Для всех потомков V' вершины V проверить
 Если область V' может содержать объект O , то
 $V = V'$

Перейти к шагу [3]

Конец ПОИСК_ЛИСТА

Рассмотрим работу этих процедур на примере поиска объекта O'' с координатами (30;10), показанного на рис. 2.3.

Процедура ПОИСК_ОБЪЕКТА (O'') на первом шаге своей работы вызывает процедуру ПОИСК_ЛИСТА (O''), которая должна найти листовую вершину с возможным расположением объекта O'' . Процедура ПОИСК_ЛИСТА не проверяет наличие поискового объекта в листе, а просто ищет подходящий узел дерева, в котором по всем правилам расположения такой объект может находиться.

Процедура ПОИСК_ЛИСТА может завершиться неудачей, например, если дерево является пустым и в нем нет ни одного узла. В этом случае она вернет значение $NULL$. Поэтому на втором шаге алгоритма мы должны сравнить значение L с $NULL$. Если листовой узел найден не был, необходимо вернуть значение ЛОЖЬ, означающее отсутствие поискового объекта в дереве.

В рассматриваемой ситуации дерево является не пустым, и процедура ПОИСК_ЛИСТА вернет листовую вершину $C2$. Только в ней может находиться объект O'' , если он присутствует в дереве. Поэтому на третьем шаге алгоритма необходимо просто проверить в цикле все объекты найденного узла $C2$. Если среди них будет обнаружен объект с координатами O'' , то процедуру поиска можно считать успешной и вернуть в качестве результата значение ИСТИНА.

Рассмотрим теперь алгоритм работы процедуры ПОИСК_ЛИСТА, чтобы понять, каким образом она нашла необходимую листовую вершину $C2$.

На первом шаге алгоритма необходимо проверить существование дерева. Для этого сравнивается корень дерева со значением $NULL$. Если такое сравнение даст положительный результат, то это будет означать, что дерево является пустым и дальнейший поиск невозможен. В этом случае возвращается значение $NULL$, что является признаком неудачного завершения поиска.

В рассматриваемом примере корнем дерева является вершина A , поэтому происходит переход ко второму шагу алгоритма. На этом шаге инициализируется переменная V (в нее заносится значение корня). Переменная V выполняет роль текущей просматриваемой вершины (т. е. той вершины, в которой ищется объект O'' на данном шаге алгоритма).

На третьем шаге происходит определение, является ли вершина V листовой вершиной. Так как в нашем случае пока что в переменной V

находится корневая вершина A , данный шаг будет пропущен, и выполнение перейдет к шагу 4.

На четвертом шаге будут проверены все дочерние узлы вершины A (потомки B , C и D) с целью определения такого потомка, который может содержать в себе объект O'' . Такая проверка выполняется простым сравнением координат объекта O'' и ограничивающей области соответствующего узла. Из шестого свойства $K-D-B$ -дерева, рассмотренного ранее, можно сделать вывод, что такой узел будет всего одним. В нашем случае это вершина C . Определив эту вершину, мы заносим ее в переменную I и возвращаемся к шагу 3 нашего алгоритма.

На этом шаге снова проверяется условие, является ли текущая вершина листовой. Ответ снова отрицательный (вершина C является внутренней вершиной дерева), поэтому происходит переход к шагу 4.

На четвертом шаге среди всех потомков вершины C (узлы $C1$ и $C2$) опять выбирается тот узел, область которого может содержать объект O'' (потомок $C2$). Он заносится в переменную I , и происходит очередной возврат к шагу 3.

На этот раз проверка, является ли $C2$ листовым узлом дерева, даст положительный результат, поэтому процедура поиска будет завершена и в качестве своего результат вернет узел $C2$.

Процедура ПОИСК_ЛИСТА не определяет наличие объекта в дереве, а только ищет листовой узел, в котором такой объект может находиться. Это позволяет использовать ее как для поиска объекта (это было показано в листинге 2.9), так и для вставки новых объектов в дерево (для определения листового узла, в который нужно поместить новый объект).

Запрос по области. Запрос по области является более общим случаем поиска. задается некоторая область поиска, и необходимо найти все объекты, находящиеся внутри данной области. Процедура, выполняющая такой поиск, представлена в листинге 2.10.

Листинг 2.10

```
//=====
// Поиск объектов по области
// Параметры:
//   Rec - область поиска
//=====
ПОИСК_В_ОБЛАСТИ(Rec)
[1] // Создать пустое множество для результата
    {R} = пустое множество
[2] // Если дерево пустое, выйти из процедуры
    Если Корень = NULL, то
        Выйти из процедуры и вернуть {R}
[3] // Произвести рекурсивный поиск объектов
    ПОИСК_В_ОБЛАСТИ_РЕК(Rec, Корень, {R})
```

```

[4] // Вернуть полученное множество {R}
      Выйти из процедуры и вернуть {R}
Конец ПОИСК_В_ОБЛАСТИ

//=====
// Рекурсивный поиск объектов по области
// Параметры:
//   Rec - область поиска
//   V    - текущая вершина поиска (первоначально
//          передается корень)
//   {R}  - множество-результат (в него помещаются объекты,
//          удовлетворяющие запросу поиска)
//=====
ПОИСК_В_ОБЛАСТИ_РЕК(Rec, V, {R})
[1] // Проверка для листовых вершин дерева
      Если V - листовая вершина, то
        Для всех объектов O вершины V проверить
          Если  $O \in \text{Rec}$ , то
            Добавить O в множество {R}
        Выйти из процедуры
[2] // Проверка для внутренних узлов дерева
      Для всех дочерних узлов V' узла V проверить
        Если область V' пересекает Rec, то
          ПОИСК_В_ОБЛАСТИ_РЕК(Rec, V', {R})
Конец ПОИСК_В_ОБЛАСТИ_РЕК

```

Рассмотрим работу данного алгоритма более подробно. Для этого вернемся к рассмотренному ранее двумерному *K-D-B*-дереву. Допустим, что необходимо найти все объекты, находящиеся в дереве (точки плоскости), координаты которых удовлетворяют следующим ограничениям (рис. 2.4):

$$30 \leq x \leq 70, \quad 30 \leq y \leq 55.$$

В этом случае массив для области поиска будет иметь следующий вид:

$$\text{Rec} = [30, 70, 30, 55].$$

Для выполнения поиска необходимо вызвать процедуру из листинга 2.10, передав ей в качестве параметров массив границ поиска *Rec*.

Процедура ПОИСК_В_ОБЛАСТИ на первом шаге своего выполнения создает пустое множество объектов $\{R\}$. В него будут добавляться найденные объекты, удовлетворяющие условиям поиска. Процедура в качестве своего результата как раз и возвращает данное множество.

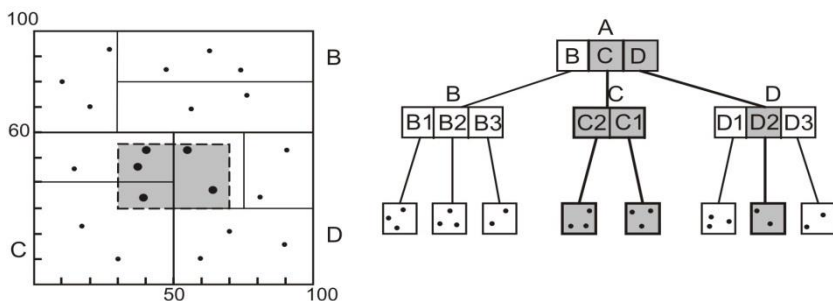


Рис. 2.4. Поиск по области

На шаге 2 алгоритма проверяется существование дерева. Если корень дерева еще не создавался, то объектов, удовлетворяющих данному запросу, нет, поэтому работа процедуры прерывается и в качестве своего результата она возвращает пустое множество $\{R\}$.

В рассматриваемом нами примере (рис. 2.4) корень дерева является не пустым, поэтому алгоритм переходит к шагу 3. На этом шаге вызывается рекурсивная процедура ПОИСК_В_ОБЛАСТИ_РЕК, которая выполняет все необходимые действия по поиску объектов заданной области. По сути, процедурой поиска объектов, ключи которых соответствуют некоторому диапазону, является процедура ПОИСК_В_ОБЛАСТИ_РЕК, а процедура ПОИСК_В_ОБЛАСТИ – это простая обертка, написанная для удобства использования.

Процедура ПОИСК_В_ОБЛАСТИ_РЕК в качестве своих параметров принимает область поиска (переменная *Res*), текущую вершину, в которой происходит поиск (поиск начинается с корня дерева, поэтому в качестве второго параметра первоначально передается корневая вершина) и множество для результата (множество, в которое необходимо поместить все найденные объекты, удовлетворяющие условию поиска).

Действия процедуры ПОИСК_В_ОБЛАСТИ_РЕК зависят от типа переданной ей в качестве второго параметра вершины. Если вершина *V* является листовой, то выполняется первый блок процедуры, иначе – второй. В рассматриваемом примере в качестве второго параметра была передана корневая вершина *A*. Для этой вершины происходит перебор в цикле всех ее дочерних узлов и поиск тех из них, области которых пересекаются с *Res*. У вершины *A* есть три дочерних узла (*B*, *C* и *D*), но с заданной областью пересекаются только *C* и *D*. Поэтому происходит два рекурсивных вызова, в которых в качестве второго параметра (текущей вершины) передаются сначала *C*, а затем *D*. Рассмотрим рекурсивный вызов для вершины *D* (вызов для вершины *C* будет аналогичным).

Вершина *D* является внутренней вершиной дерева, поэтому для нее будет выполняться второй блок процедуры ПОИСК_В_ОБЛАСТИ_РЕК.

Среди всех потомков вершины только $D2$ имеет пересечение с областью Rec , поэтому процедура будет вызвана еще раз рекурсивно для вершины $D2$.

Для вершины $D2$ будет работать первый блок процедуры ПОИСК_В_ОБЛАСТИ_РЕК (так как $D2$ является листовой вершиной). В этом блоке происходит перебор всех содержащихся в вершине объектов и поиск тех из них, которые находятся внутри области Rec . В листовой вершине $D2$ находится всего два объекта, и оба они попадают в область поиска. Поэтому оба эти объекта будут добавлены в множество результатов $\{R\}$.

Аналогичным образом будут просмотрены вершины $C1$ и $C2$. В результате всех проверок будет найдено пять объектов, удовлетворяющих запросу, при этом будет проверено три внутренних узла (A , C и D) и три листовых ($C1$, $C2$ и $D2$).

Алгоритм добавления нового объекта в К-D-B-дерево

Одним из важных алгоритмов для динамической структуры является алгоритм добавления новых объектов. В большинстве случаев не разрабатывается отдельного алгоритма для первоначального построения дерева, а такое построение происходит с использованием именно этого алгоритма. Именно поэтому от качества работы алгоритма добавления нового объекта зависит, насколько полученная структура будет оптимальной, а скорость поисковых запросов – быстрой и эффективной.

Рассмотрим простейший алгоритм добавления объекта в $K-D-B$ -дерево, предложенный создателем структуры Д. Т. Робинсоном [74] (листинг 2.11).

Листинг 2.11

```
//=====
// Вставка нового объекта в дерево
// Параметры:
//   О - объект, который необходимо вставить в дерево
//=====
ВСТАВКА(О)
[1] // Определяем лист для размещения О
    L = ПОИСК_ЛИСТА(О)
[2] // Если дерево пустое, добавляем
// объект в корень
    Если L = NULL, то
        L = новая пустая листовая вершина
        Добавить объект О в L
        Корень = L
        Выйти из процедуры вставки
[3] // Проверка, возможно объект уже в дереве
    Для всех объектов О' из листа L, проверить
        Если О' = О, то
```



Выйти из процедуры вставки
[4] // Добавление объекта O в листовую вершину L
Добавить объект O в L
Если количество объектов в $L > M$, то
РАСЩЕПЛЕНИЕ_УЗЛА (L)
Конец ВСТАВКА

Данный алгоритм добавляет объект O в $K-D-B$ -дерево. Перед тем, как рассмотреть его более подробно, отметим один очень важный факт. Каждый узел $K-D-B$ -дерева должен иметь не меньше одного потомка (свойство 3) и не больше M потомков. Показатель M является внутренней характеристикой дерева и называется арностью $K-D-B$ -дерева. Обычно M выбирают таким, чтобы все данные одного узла целиком размещались на одной странице жесткого диска.

Рассмотрим непосредственно сам алгоритм вставки. На первом шаге определяется листовая вершина L , в которую можно поместить объект O . Для этого используется написанная ранее процедура ПОИСК_ЛИСТА (листинг 2.9). Эта процедура вернет листовой узел или $NULL$, если дерево пустое и в нем нет ни одного узла. Поэтому на втором шаге алгоритма необходимо проверить переменную L на равенство ее $NULL$. Если такое равенство выполняется, то необходимо создать новый листовой узел и разместить в него объект O . Затем созданный узел устанавливается как корень дерева и процедура вставки объекта завершает свою работу.

Если же дерево оказалось не пустым, то необходимо проверить наличие объекта O в вершине L (шаг 3). Вполне возможно, объект O уже был добавлен в дерево ранее и вставку проводить не нужно. Для выполнения такой проверки происходит перебор в цикле всех объектов вершины L и сравнение их с O . Если будет найдено соответствие, то процедура вставки также прекращается.

Если дерево является не пустым, а объекта в нем нет, то необходимо выполнить добавление объекта в него. Для этого объект O вставляется в найденную ранее вершину L (шаг 4). Однако при такой операции может произойти переполнение вершины L (число объектов в ней станет больше показателя арности дерева M). В этом случае необходимо выполнить расщепление листового узла L на два новых узла, что приведет к восстановлению свойств дерева. Для выполнения подобной операции вызывается функция РАСЩЕПИТЬ. Сама по себе эта функция является не тривиальной. От ее работы зависит структура дерева и его показатель использования памяти. Один из простейших вариантов этой процедуры представлен в следующем пункте данной главы.

Алгоритм расщепления вершины K-D-B-дерева

Расщепление вершины вызвано жесткими ограничениями на максимально возможное число элементов в узле дерева. При превышении этого предела узел должен быть разделен на две части вдоль одной из осей пространства. Такая ситуация может произойти как в листовой вершине (при добавления конечных объектов в дерево), так и во внутренней вершине дерева (расщепления листовых вершин приводят к наполнению и переполнению внутренних вершин дерева). Причем процедуры расщепления как внутренних, так и листовых вершин очень похожи. Поэтому в листинге 2.12 приведена общая процедура РАСЩЕПЛЕНИЕ_УЗЛА, которая может быть вызвана как для листового, так и для внутреннего узла дерева. В качестве параметра в данную процедуру передается вершина, в которой произошло переполнение.

Листинг 2.12

```
//=====
// Расщепление узла дерева
// Параметры:
// V - переполненный узел дерева
//=====
РАСЩЕПЛЕНИЕ_УЗЛА (V)
[1] // Выбираем ось расщепления
    i = (Предыдущее_i_для_V + 1) % K
[2] // Выбираем координату расщепления
    Arr = массив из (K+1) элемента
    Для j от 0 до (K+1)
        Arr[j] = i-я координата Childj(V)
    Сортировать Arr
    Xi = Arr[K/2]
[3] // Проверка возможности расщепления
    Left = 0
    Right = 0
    Для j от 0 до (K+1)
        Если верхняя i-я координата
            Childj(V) > Xi
            Right++
        Если нижняя i-я координата
            Childj(V) ≤ Xi
            Left++
    Если Right>K или Left>K, то
        Вернуться к шагу 1
[4] // Разделение вершины на две
    V2 = новая пустая вершина
    РАСПРЕДЕЛИТЬ_ОБЪЕКТЫ(V, V2, i, Xi)
[5] // Если был расщеплен корень, создать новый
    Если V = корень дерева, то
        Создать новую вершину Vp
```

```

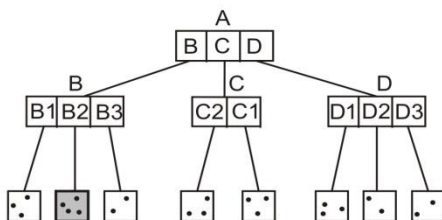
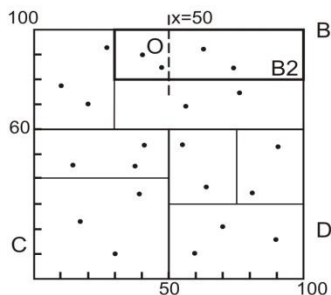
        Поместить в Vp вершины V и V2
        Сделать корнем вершину Vp
        Выйти из процедуры расщепления
[6] // Рекурсивное расщепление вверх (если нужно)
    Vp = Parent(V)
    Добавить V2 в Vp
    Если количество объектов в Vp > M, то
        РАСЩЕПЛЕНИЕ_УЗЛА(Vp)
Конец РАСЩЕПЛЕНИЕ_УЗЛА

//=====
// Распределение объектов вершины V1 между V1 и V2
// Параметры:
//   V - переполненная вершина
//   V2 - вторая вершина для распределения
//   i - ось, вдоль которой происходит распределение
//   Xi - координата, по которой происходит распределение
//=====
РАСПРЕДЕЛИТЬ_ОБЪЕКТЫ(V, V2, i, Xi)
[1] // Распределяем объекты по вершинам V и V2
    Для всех объектов V' вершины V выполняем
        Если V' находится левее Xi по оси i, то
            Оставить V' в V
        Если V' находится правее Xi по оси i, то
            Перенести V' в V2
        Если V' пересекается Xi по оси i, то
            V'' = новая пустая вершина
            РАСПРЕДЕЛИТЬ_ОБЪЕКТЫ(V', V'', i, Xi)
            Поместить V' в V2
[2] // Изменяем границы вершин V и V2
    Границы(V2) = Границы(V)
    Максимальная граница V по измерению i = Xi
    Минимальная граница V2 по измерению i = Xi
Конец РАСПРЕДЕЛИТЬ_ОБЪЕКТЫ

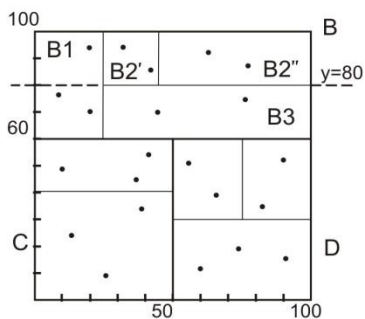
```

Разберем действие этих процедур подробнее. Первым делом процедура расщепления выбирает ось, вдоль которой будет произведено расщепление (шаг 1), и координату на этой оси, по которой будет проведена граница (шаг 2). На рис. 2.5(а) представлена ситуация переполнения узла B2. В качестве оси деления выбрана ось Ox , а в качестве координаты – $x = 50$. После выбора оси и координаты разбиения необходимо проверить возможность распределения объектов данной вершины на две группы по данным параметрам. На практике могут появиться ситуации, при которых будет невозможно разбить вершину по некоторой оси так, чтобы обе полученные вершины оказались не переполненными. В случае невозможности распределения объектов на две группы с сохранением условия максимального числа объектов в них, нужно перейти к первому шагу и выбрать другую ось и другую точку

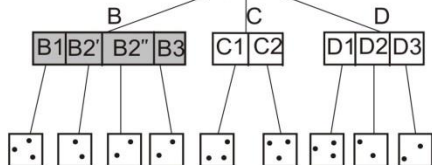
разбиения. Подробнее о принципах выбора оси и координаты разбиения, а также о случаях невозможности разбиения будет рассказано ниже.



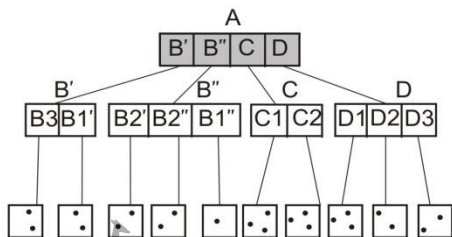
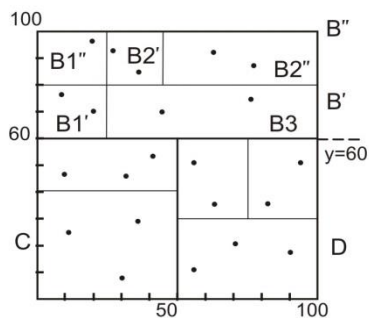
a)



ЛАНЬ®



б)



в)



ЛАНЬ®

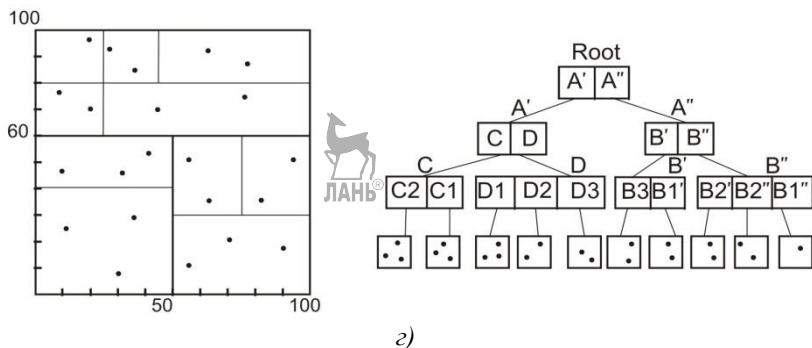


Рис. 2.5. Пример обработки переполнения: а – переполнение вершины B2; б – переполнение вершины B; в – переполнение вершины A; г – окончательное дерево после всех разбиений

Если возможно разбиение вершины, то распределяются все ее дочерние объекты на две группы (шаг 4). Для этого создается еще одна пустая вершина V_2 и вызывается процедура РАСПРЕДЕЛИТЬ_ОБЪЕКТЫ. Этой процедуре передается переполненная вершина V , пустая вершина V_2 (между этими вершинами распределяются объекты), а также ось и координата, по которым должно произойти распределение.

После выполнения этой процедуры необходимо добавить новую вершину V_2 в дерево. При этом возможны две ситуации. На шаге алгоритма 5 рассматривается вариант, когда переполненная вершина оказалась корнем. При этом необходимо создать новый корень дерева V_p и сделать вершины V и V_2 его потомками.

Если вершина V не является корнем дерева (шаг 6), то вновь созданную вершину V_2 просто добавляют как дочернюю вершину V_p (где V_p – родительская вершина для V). При этом необходимо учесть, что такое добавление может привести к переполнению самой вершины V_p . В этом случае нужно будет вызвать процедуру расщепления узла и для V_p (произвести рекурсивное расщепление вверх).

Еще одной процедурой, заслуживающей особого внимания, является процедура распределения объектов переполненной вершины между двумя вершинами дерева (процедура РАСПРЕДЕЛИТЬ_ОБЪЕКТЫ). Эта процедура также приведена в листинге 2.12. Ее работа состоит из двух шагов: распределение объектов на две группы, соответствующих вершинам V и V_2 (шаг 1), и корректировка ограничивающих прямоугольников этих вершин (шаг 2).

На первом шаге процедура в цикле проверяет положение всех дочерних объектов вдоль плоскости разбиения (плоскость разбиения

задается осью и координатой, переданными в третьем и четвертом параметрах). При этом возможны три варианта:

1. Дочерний узел целиком находится левее (или ниже) плоскости разбиения (например на рис. 2.5(б), при расщеплении вершины B линией $y = 80$, таким узлом является вершина $B3$). В этом случае ни сам узел, ни его положение изменениям не подвергаются. Все остается как и было ранее.

2. Дочерний узел целиком находится правее (или выше) плоскости разбиения (на рис. 2.5(б) это вершины $B2'$ и $B2''$). При этом данные объекты переносятся в новую вершину.

3. Дочерний узел пересекается плоскостью разбиения (на рис. 2.5(б) это вершина $B1$). При этом необходимо произвести рекурсивное разбиение данной вершины вниз.

Для конечных объектов считается, что вершина находится слева от плоскости разбиения, если ее i -я координата меньше или равна координате разбиения. Иначе он считается лежащим правее плоскости разбиения. Так как конечный объект является точкой многомерного пространства, то для него невозможен случай, когда он должен быть разбит на две части, т.е. для конечного объекта O с координатами $(x; y)$ для разбиения по оси Ox описанные условия примут следующий вид:

Если $O.x > X_i$, то

Перенести O в вершину $V2$

Иначе

Оставить O в вершине V

Для внутренних узлов дерева считается, что узел находится левее плоскости разбиения, если максимальная i -я координата его ограничивающего прямоугольника меньше или равна координате разбиения. Если его минимальная координата больше или равна координате разбиения, то считается, что данная вершина находится правее плоскости разбиения. Иначе вершину необходимо разбивать на две, так как она пересекается плоскостью. Для вершины V' , у которой ограничивающий прямоугольник имеет координаты $(x_{min}; y_{min}) - (x_{max}; y_{max})$, а разбиение происходит вдоль оси Ox , соответствующие проверки примут следующий вид:

Если $V'.x_{max} \leq X_i$, то

Оставить V' в вершине V

Иначе, Если $V'.x_{min} \geq X_i$, то

Перенести V' в вершину $V2$

Иначе

V'' = новая пустая вершина

РАСПРЕДЕЛИТЬ_ОБЪЕКТЫ(V' , V'' , Ox , X_i)

Поместить V'' в $V2$

Рассмотрим действие приведенного алгоритма на конкретном примере. Допустим, у нас есть дерево, представленное на рис. 2.3. Для этого дерева введено ограничение на максимальное число потомков вершины, равное 3. Пусть в это дерево вставлен еще один объект O с координатами (40; 90). Такой объект будет размещен в вершину $B2$, что приведет к ее переполнению (рис. 2.5(а)). Поэтому будет вызвана процедура РАСЩЕПЛЕНИЕ_УЗЛА($B2$).

Допустим, процедура расщепления выберет в качестве оси разбиения ось Ox , а в качестве координаты разбиения – 50. При этом вершина $B2$ должна быть разбита на две новые вершины – $B2'$ и $B2''$, которые вместе образуют область, ранее занимаемую вершиной $B2$ (рис. 2.5(б)). При этом две точки вершины $B2$ перейдут в вершину $B2'$ (те, которые раньше находились в $B2$ и располагались левее прямой Ox), а две других – в вершину $B2''$.

При расщеплении вершины $B2$ на две новые вершины $B2'$ и $B2''$ было восстановлено условие на максимальное заполнение листовых узлов (и вершина $B2'$, и $B2''$ содержат по две точки). Однако это привело к тому, что вершина B стала содержать четыре узла и в ней нарушаются условия заполнения. Поэтому необходимо провести рекурсивное разбиение родительской вершины B на B' и B'' .

Допустим, в качестве оси разбиения на этот раз была выбрана ось Oy , а в качестве точки разбиения – $y = 80$. При этом вершина $B3$ находится ниже прямой разбиения $y = 80$, поэтому она будет помещена в B' . Вершины $B2'$ и $B2''$ находятся выше данной прямой, поэтому они перейдут в вершину B'' .

Однако вершина $B1$ пересекается прямой $y = 80$, поэтому ее нельзя отнести ни к B' , ни к B'' . Согласно описанному алгоритму, необходимо произвести рекурсивное разбиение $B1$ по прямой $y = 80$ (шаг 1 процедуры РАСПРЕДЕЛИТЬ_ОБЪЕКТЫ). В результате получим дерево, показанное на рис. 2.5(в).

Это не окончательный вариант дерева, потому что вершина A (корень дерева) содержит четыре элемента, в то время как максимально возможное число в нашем примере – 3. Поэтому необходимо произвести еще и разбиение корня.

Допустим, для разбиения корневой вершины были выбраны ось Oy и значение разбиения, равное 60. При этом мы получим две вершины A' и A'' . В первой из них будут размещены вершины C и D , а во второй – B' и B'' .

Согласно шагу 4 процедуры РАСЩЕПЛЕНИЕ_УЗЛА, при делении корня необходимо создать новую корневую вершину, потомками которой будут две вершины, полученные в результате деления старого корня. Поэтому мы получим увеличение высоты дерева. Окончательный результат вставки объекта в дерево показан на рис. 2.5(г).

Как видно из описанного примера, в процессе вставки объекта в дерево может происходить рекурсивное разбиение вершин от листового узла до корня (рекурсивное распространение деления вверх по дереву). Причем на каждом шаге также может произойти и рекурсивное разбиение дочерних узлов (рекурсивное разбиение вниз по дереву). Таким образом добавление объекта в дерево может затронуть большое количество вершин этого дерева и привести к серьезной реорганизации структуры. Этот процесс может потребовать больших вычислительных и временных ресурсов.

Еще одним недостатком рекурсивного распространения деления вершин является факт уменьшения коэффициента заполнения. При рекурсивном распространении деления вниз происходит деление непереполненных вершин. Поэтому появляются сильно незаполненные вершины. Возможно даже появление пустых листовых узлов. Однако, если даже листовой узел пуст, его нельзя удалить, иначе произойдет нарушение свойств *K-D-B*-дерева.

Выбор измерения. В процессе описания процедур деления вершины не уточнялся механизм выбора измерения, вдоль которого будет произведено разбиение узлов. Такой выбор во многом зависит от конкретной задачи, и не существует единственного хорошего варианта выбора, пригодного абсолютно во всех случаях. В данном пункте мы опишем наиболее универсальные и часто используемые варианты.

Самым простым способом является чередование измерений разбиения в каждой вершине. Для этого во всех узлах дерева необходимо наличие еще одной переменной, в которой хранится номер измерения, по которому разбиение произошло в прошлый раз. Наличие этой переменной дает возможность чередовать разбиения. Для этого используется простая формула:

$$i = (\text{Предыдущее}_i + 1) \% K,$$

где K – число измерений;

Предыдущее_i – измерение предыдущего разбиения;

i – измерение следующего разбиения.

Так, при двухмерном пространстве разбиения будут происходить по следующей последовательности:

$$x, y, x, y, x, y, \dots$$

Данный способ подходит для большинства приложений и легко реализуется программно. Его можно еще сильнее упростить, введя одну единственную глобальную переменную для чередования измерений во всем дереве. Однако одна общая переменная может привести к плохой структуре дерева. Чередование измерений в пределах каждой отдельной вершины приводит к более сбалансированному виду. Хотя даже в этом случае не исключается неравномерная структура.

Если заранее известна некоторая информация о распределении или наиболее часто используемых запросах, то можно подобрать закон

чередования специально под данную конкретную задачу. Так, если известно, что вдоль некоторого измерения происходит в два раза больше запросов, чем вдоль другого, то можно модифицировать схему таким образом, чтобы последовательность осей разбиения приняла следующий вид:

$$x, x, y, x, x, y, x, x, y, \dots$$

Структура дерева будет выстраиваться таким образом, что деление пространства поиска вдоль оси Ox будет происходить в два раза чаще, что даст преимущество в запросах, ориентированных на это измерение.

В некоторых приложениях более удобным может быть другой способ выбора измерения. Так, в качестве измерения разбиения может быть выбрано измерение, имеющее наибольшую длину в данной вершине или наибольший разброс точек (для листовых вершин).

Выбор координаты разбиения. Еще одним моментом, оставшимся не раскрытым, является способ выбора координаты разбиения. После выбора измерения, по которому будет совершено разбиение, необходимо выбрать конкретную точку на нем, по которой пройдет плоскость разбиения. Эта задача также во многом зависит от конкретной решаемой задачи. Но есть один общий способ, подходящий для большинства приложений – разбиение по среднему объекту.

Для выбора координаты разбиения по этому способу происходит сортировка всех элементов (точек или дочерних узлов) по координате измерения разбиения. После этого выбирается объект в отсортированном массиве, находящийся в его середине, и в качестве точки разбиения используется его граничная координата. Именно этот способ был использован в листинге 2.12 (шаг 2 процедуры РАСЩЕПЛЕНИЕ_УЗЛА). Данный способ имеет два неоспоримых преимущества перед другими вариантами:

- легко реализуем на практике;
- гарантирует, что хотя бы один из дочерних узлов данной вершины не будет расщеплен рекурсивно вниз.

Есть и более качественные способы. Одним из них является поиск координаты, при которой произойдет как можно меньше рекурсивных разбиений вниз, т. е. среди всех возможных вариантов выбора точки разбиения выбирается тот, при котором плоскость разбиения пересечет как можно меньше дочерних объектов.

Существует и более простое решение – выбор в качестве точки разбиения середины отрезка вдоль выбранной оси. Однако данный способ практически неприменим на практике, так как он может привести к варианту, при котором все дочерние объекты будут рекурсивно разделены на две части, а это повлияет на то, что расщепление узла не избавит от переполнения.

Проверка корректности выбора оси и координаты разбиения.

Еще одной проблемой является проверка возможности разбиения по

выбранной оси и координате. В зависимости от способа выбора параметров разбиения может возникнуть ситуация, при которой разбиение окажется невозможным или не приводящим к положительному результату.

Одним из таких вариантов является ситуация (рис. 2.6(а)), когда все объекты листовой вершины имеют одну и ту же координату вдоль выбранной оси разбиения. После разбиения такой вершины один из полученных узлов останется переполненным, при этом второй узел будет пустым, т. е. показанное разбиение не принесет никакого результата (в отличие от *K-D*-дерева, в *K-D-B*-дерево не используются суперключи и при равенстве координат все объекты просто помещаются в один из потомков, предопределенный логикой программы).

Ситуация с невозможностью разбиения может произойти и во внутреннем узле дерева (рис. 2.6(б)). Наиболее часто это встречается, когда необходимо произвести множественное рекурсивное разбиение дочерних узлов. При этом образуются дополнительные вершины, которые могут привести к переполнению одного или даже обоих полученных узлов. Так, на рис. 2.6(б) представлена ситуация, когда любое разбиение вдоль оси Ox приводит к делению всех дочерних элементов узла и соответственно не имеет никакого смысла.

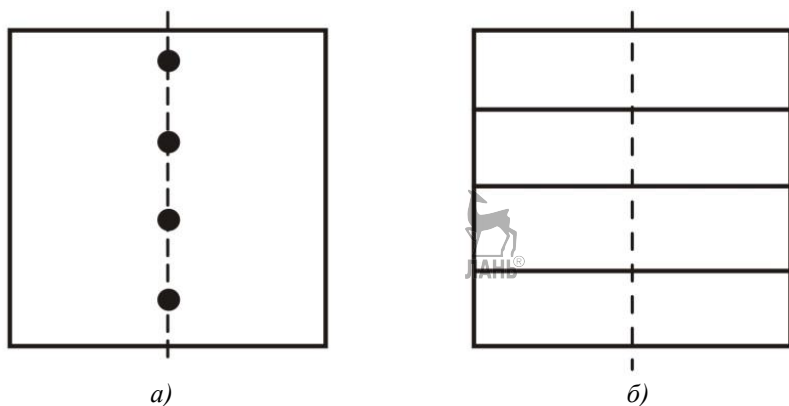


Рис. 2.6. Примеры неправильных осей и точек деления: а – листовая вершина; б – внутренняя вершина дерева

Несмотря на кажущееся многообразие случаев, при которых выбранное разбиение произвести нельзя, проверка измерения и координаты на возможность или невозможность довольно таки проста. Для этого необходимо просто посчитать количество дочерних узлов, имеющих свои левые границы – меньшие точки разбиения и правые

границы – большие точки разбиения. Если хотя бы один из таких счетчиков окажется больше арности дерева M , то выбранное разбиение невозможно и необходимо произвести повторный поиск оси и точки деления.

Именно такой алгоритм был использован в реализации процедуры РАСЩЕПЛЕНИЕ_УЗЛА листинга 2.12. Данная проверка в нем проводится сразу же после выбора оси и точки разбиения (шаг 3). В случае неудачи алгоритм возвращается к первому шагу и выбирается следующее измерение в качестве кандидата на разбиения узла.

Алгоритм удаления вершины из K-D-B-дерева

В K-D-B-дереве все объекты хранятся в листовых узлах. Внутренние вершины служат только для деления пространства поиска. Поэтому удаление объекта из дерева затрагивает в первую очередь листовые вершины. Однако в свойствах K-D-B-дерева нет ограничения на минимальное число объектов в листовых вершинах (вплоть до того, что листовые вершины могут быть пустыми). Поэтому процедура удаления может быть совсем тривиальной, как показано в листинге 2.13.

Листинг 2.13

```
//=====
// Удаление объекта из K-D-B-дерева
// Параметры:
//   O – объект, который необходимо удалить
//=====
УДАЛЕНИЕ (O)
[1] // Определяем вершину, в которой находится O
    L = ПОИСК_ЛИСТА (O)
[2] // Если дерево пустое, выходим
    Если L = NULL, то
        Выйти из процедуры удаления
[3] // Удаление объекта
    Для всех объектов O' из листа L, проверить
        Если O' = O, то
            Удалить O' из L
Конец УДАЛЕНИЕ
```

На первом шаге алгоритма происходит поиск листовой вершины, в которой может находиться удаляемый объект. Для этого используется описанная ранее процедура ПОИСК_ЛИСТА, которая может вернуть *NULL*, если дерево не существует. В этом случае необходимо просто завершить процедуру удаления (шаг 2).

Если поиск листовой вершины дал положительный результат, то необходимо проверить все его дочерние объекты O' . При нахождении совпадения с объектом O такие объекты удаляются из вершины. На этом процедура удаления заканчивает свою работу.

В простейшем случае нет необходимости в реорганизации дерева, так как свойства K - D - B -дерева не запрещают наличие пустых листовых вершин. Однако при частых удалениях объектов из дерева коэффициент заполнения пространства может значительно сокращаться и структура дерева будет неэффективной. Поэтому на практике применяют процедуру реорганизации некоторых вершин дерева сразу же после удаления объектов.

Алгоритм реорганизации вершин K - D - B -дерева

В K - D - B -дереве есть два алгоритма, которые плохо влияют на его структуру – алгоритм рекурсивного расщепления вершин вниз при вставке объектов и алгоритм удаления объектов. Оба эти алгоритма приводят к уменьшению коэффициента использования памяти структурой и, как следствие, увеличению времени поиска. Для улучшения показателей структуры используется перестроение (реорганизация) некоторых вершин дерева, которые оказались редко заполненными или даже пустыми (в случае листовых вершин).

Разработчиками структуры было предложено два механизма реорганизации дерева:

- слияние смежных вершин – техника, позволяющая объединить рядом расположенные узлы в один общий узел;
- перераспределение объектов – полное перестроение всех дочерних узлов некоторой вершины дерева.

Рассмотрим эти методологии более подробно.

Слияние смежных узлов. Данный способ заключается в поиске соседних узлов некоторой вершины дерева таких, что в случае их объединения будут выполняться следующие условия:

- полученный в результате объединения узел не будет переполненным (т.е. в нем будет меньше M дочерних объектов);
- полученный в результате объединения узел будет иметь ограничивающую область прямоугольной формы.

Стоит заметить, что далеко не всегда можно подобрать смежный объект для слияния, удовлетворяющий описанным правилам. На рис. 2.7 показан узел некоторого дерева с ограничением $M = 8$. В этом узле все дочерние вершины являются полупустыми (во всех трех вершинах находится всего десять объектов). Однако объединение с помощью слияния в этом случае сделать невозможно, так как при объединении вершин B и C мы получаем переполненный узел, а при объединении A и B , так же как и при объединении A и C , получается вершина, имеющая ограничивающий прямоугольник не прямоугольной формы.

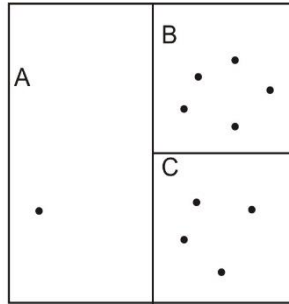


Рис. 2.7. Вершина K-D-B-дерева

Несмотря на описанный недостаток, слияние смежных вершин может эффективно применяться на практике. Алгоритм работы такой реорганизации представлен в листинге 2.14.

Листинг 2.14

```
//=====
// Слияние смежных вершин
// Параметры:
//   V - вершина, потомков которой необходимо объединить
//=====
РЕОРГАНИЗАЦИЯ(V)
[1] // Объединение потомков
    Для всех пар V' и V'' проверить
        n1 = количество элементов в V'
        n2 = количество элементов в V''
        Если n1+n2 < M, то
            BR - ограничивающий прямоугольник (V' ∩ V'')
            S - площадь BR
            S' = площадь V'
            S'' = площадь V''
            Если S = S' + S'', то
                Перенести все объекты из V'' в V'
                Удалить V''
            Ограничивающий прямоугольник V' = BR
[2] // Сокращение высоты дерева
    n = число элементов в V
    Если (V - корень дерева) И (n = 1), то
        V' = потомок вершины V
        Удалить V
        Корень дерева = V = V'
        Вернуться к шагу 2
Конец РЕОРГАНИЗАЦИЯ
```

Алгоритм данной процедуры является простейшим, но при этом не самым эффективным. Если необходимо преобразовать дочерние элементы некоторой вершины V , то происходит перебор всех возможных пар ее дочерних элементов, которые при объединении дадут непереполненную вершину. Для этого могут быть использованы два вложенных цикла, формирующих все возможные комбинации пар элементов. Однако на практике, при большом числе потомков, полный перебор может занимать значительное время. Поэтому можно просматривать только те пары элементов, в которых хотя бы один из элементов является пустым или редко заполненным.

После генерации таких пар необходимо проверить возможность их объединения. Для этого сравнивается площадь ограничивающего прямоугольника, который получится при объединении вершин, и сумма площадей самих этих вершин. Если в результате сравнения получится равенство, то можно утверждать, что данные вершины являются смежными и их можно объединить в одну общую вершину.

В результате многочисленных объединений может возникнуть ситуация, в которой у корня останется всего один потомок. В этом случае необходимо этого потомка сделать корнем дерева, а прежнюю корневую вершину удалить. Необходимость подобной операции может возникнуть несколько раз. Поэтому второй шаг алгоритма РЕОРГАНИЗАЦИЯ заиклен.

Перераспределение объектов. Перераспределение является вторым способом улучшения свойств дерева. Его суть заключается в следующем: если у некоторой вершины V потомки являются редко заполненными, то можно создать новую вершину, в которую перенести все элементы из старых потомков, а старые вершины просто удалить. Пример такой процедуры показан в листинге 2.15.

Листинг 2.15

```
//=====
// Реорганизация потомков некоторой вершины
// Параметры:
//   V - вершина, потомков которой необходимо объединить
//=====
РЕОРГАНИЗАЦИЯ(V)
[1] // Перенести все объекты в список
    List - пустой список
    Для всех потомков V' вершины V выполнить
        Перенести объекты из V' в List
        Удалить V'
[2] // Сделать для вершины V одного пустого потомка
    V' = новая пустая вершина
    Ограничивающий прямоугольник V' = V
    Сделать V' единственным потомком V
[3] // Перенос объектов из списка в новую вершину
```

```

Для всех элементов E1 списка List выполнить
  Добавить E1 в один из потомков V' вершины V
  Если V' переполнилась, то
    РАСЩЕПЛЕНИЕ_УЗЛА V'
[4] // Сокращение высоты дерева
  n = число элементов в V
  Если (V - корень дерева) И (V = 1), то
    V' = потомок вершины V
    Удалить V
    корень дерева = V = V'
  Вернуться к шагу 4
Конец РЕОРГАНИЗАЦИЯ

```



На первом шаге алгоритма происходит удаление всех потомков вершины V . При этом все элементы удаляемых вершин сохраняются в некотором временном списке *List*.

После очистки вершины V в ней создается всего один потомок, ограничивающая область которого равна все ограничивающей области родителя (шаг 2). Далее все элементы, временно размещенные в списке *List*, переносятся обратно в потомков вершины V . Если в процессе переноса происходит переполнение одной из вершин, то вызывается функция РАСЩЕПЛЕНИЕ_УЗЛА, рассмотренная ранее в листинге 2.12.

На последнем шаге алгоритма, как и в предыдущем методе, происходит сокращение высоты дерева (если это возможно).

Данный способ реорганизации возможен практически всегда, независимо от того, являются ли редко заполненные вершины смежными или нет. Однако он требует больших затрат и полностью перестраивает некоторый узел или даже целое поддерево.

Эффективность K-D-B-дерева

Разработчик данной структуры Д. Т. Робинсон провел ряд экспериментов, в которых доказал эффективность применения структуры для ряда приложений, использующих многомерные данные небольшой размерности [74]. Однако дальнейшие исследования [37] выявили значительное ухудшение свойств дерева при числе измерений больше 3, что значительно уменьшает область применения структуры на практике. Более того, даже для двумерных данных возможен случай, когда коэффициент использования памяти не превосходит 50%. При этом высота дерева становится значительной даже при небольшом числе записей в дереве. Это приводит к лишним обращениям к памяти в процессе поиска.

Данный недостаток связан с рекурсивным делением пространства вниз при вставке объектов. Такое деление может привести к пустым листовым вершинам в дереве, которые не могут быть удалены из него, так как одно из свойств K-D-B-дерева запрещает наличие пустых

внутренних вершин или отсутствие листовых вершин, не соответствующих какой-то части пространства.

В дальнейшем был разработан ряд структур, использующих память более эффективно и имеющих более высокие показатели по производительности процедур поиска. Такие структуры будут рассмотрены в данной главе далее.

2.2.3. LSD-дерево

LSD-дерево (*LSD-Tree*), как и описанное ранее *K-D-B-дерево*, является индексной структурой для доступа к многомерным точечным данным, размещенным во внешней памяти. Эту структуру предложили А. Генрих, Г. Сикс и П. Видмар в 1989 году [46]. В своей работе эти исследователи описали не только новые алгоритмы работы с точечными данными, принципы хранения и обработки, но и способ использования данной структуры для данных, имеющих не точечный характер (интервалы, полигоны и т. д.). Предложенный ими способ называется трансформацией объектов в пространство большей размерности. Он будет рассмотрен далее.

Термин *LSD-tree* расшифровывается как *Local Split Decision tree*. В нем отражена мысль, что пространство в каждой вершине дерева делится локально, независимо от остального дерева, причем деление это бинарное. В этом смысле *LSD-дерево* очень похоже на адаптивное *K-D-дерево*. Однако *LSD-дерево* предназначено для хранения и обработки гораздо больших объемов данных, которые не могут полностью находиться в оперативной памяти. Поэтому оно содержит в себе механизмы выгрузки части узлов во внешнюю память.

Структура *LSD-дерева*

LSD-дерево можно считать адаптивным *K-D-деревом*, приспособленным для внешней памяти. Для этого создателям пришлось изменить ряд оригинальных алгоритмов и свойств адаптивного *K-D-дерева*, добавить новые понятия и элементы. Здесь эти нововведения будут описаны постепенно, по мере их отличия от адаптивного *K-D-дерева*.

На рис. 2.8 представлена общая структура *LSD-дерева* для двумерного случая (на рисунке представлена структура без использования внешних блоков). Разберем некоторые отличия *LSD-дерева* от *K-D-дерева* на этом примере.

Как можно видеть, структура дерева не похожа на оригинальное *K-D-дерево*, описанное ранее. Внутренние узлы дерева не содержат объектов. Они предназначены только для деления пространства на две части – левее и правее некоторой плоскости, перпендикулярной одной из осей пространства. Поэтому во внутренних узлах дерева не содержатся

координаты какого-либо объекта или ссылки на него. В них находятся всего два числа – номер оси, перпендикулярно которой проведена плоскость разбиения, и координата на выбранной оси, по которой эта плоскость проходит.

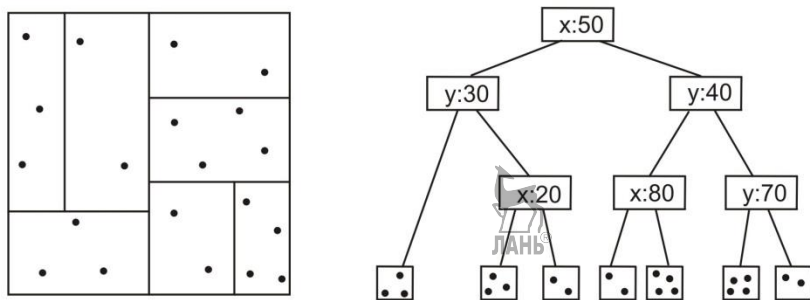


Рис. 2.8. *LSD-дерево*

Еще одним интересным наблюдением может оказаться тот факт, что в дереве нет жесткой последовательности плоскостей деления Ox , Oy , Ox , Oy , Ox ... У одной из вершин, которая делит пространство плоскостью, перпендикулярной оси Oy по координате 40, правый потомок также производит деление вдоль оси Oy . То, по какой из осей такое деление будет происходить, зависит только от принципов, заложенных в процедуру разбиения узла. Некоторые подходы к выбору оси деления будут изложены позже в данном параграфе.

Радикальным отличием *LSD*-дерева от *K-D*-дерева является то, что листовые вершины содержат не один индексируемый объект, а целый набор таких объектов, попавших в соответствующую часть пространства. В отличие от *K-D*-дерева, в *LSD*-дереве происходит деление пространства не до тех пор, пока в каждой из его частей останется один-единственный объект (который и сохраняется в листовой вершине), а до тех пор, пока в каждой части пространства останется не более N объектов. При этом величина N является одним из параметров дерева и зависит от конкретной реализации. Назовем эту величину *объемом листового узла дерева*. В практических реализациях величину N выбирают такой, чтобы весь листовой узел помещался в одном блоке внешней памяти. Это позволяет за одно обращение к внешней памяти получить все объекты листа для дальнейшей их проверки и обработки.

Листовые вершины *LSD*-дерева, а следовательно, и сами конечные объекты, хранятся во внешней памяти. В оперативной памяти находятся только внутренние вершины дерева, что позволяет индексировать большие объемы данных при наличии меньших ресурсов. Такой подход значительно улучшает ситуацию с большим объемом индексируемых данных. Однако внутренний индекс также может разрастись, и

оперативной памяти станет недостаточно. Тогда некоторую из его частей можно перенести во внешнюю память. Для этого разработчики *LSD*-дерева предусмотрели механизм объединения некоторой части дерева в один блок, который и будет выгружен во внешнюю память. В дальнейшем, при обращении к дереву (например при поиске объектов), недостающие поддеревья будут временно подгружены из внешней памяти так, чтобы не нарушить алгоритмов обработки.

На рис. 2.9 показан рассмотренный ранее пример, в котором введено жесткое ограничение: в оперативной памяти не должно находиться более трех вершин.

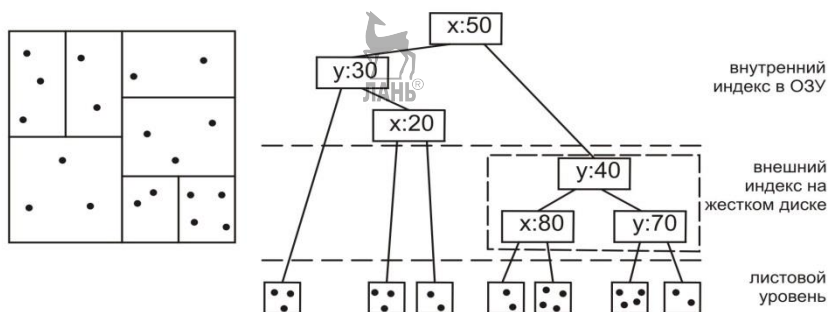


Рис. 2.9. *LSD*-дерево с внешними блоками

В представленном примере часть внутренних узлов (правое поддерево корневой вершины) было объединено в один общий блок и перемещено во внешнюю память. Назовем блоки, в которых хранятся некоторые поддеревья *LSD*-дерева, аналогичные некоторым поддеревьям адаптивного *K-D*-дерева и предназначенные для хранения во внешней памяти, *внешними блоками* дерева.

Число вершин, которые могут находиться в оперативной памяти, являются еще одним параметром *LSD*-дерева. Назовем этот параметр *объемом внутреннего индекса дерева* и обозначим как M (на рис. 2.9 M равен трем). По сути, M является тем параметром, который ограничивает затраты оперативной памяти. Чем меньше M , тем меньше оперативной памяти будет занято индексирующей структурой. Но с другой стороны показатель M очень сильно влияет на производительность. Если M выбрать таким, что все внутренние вершины будут размещены в оперативной памяти, то операции поиска не будут замедлены обращением к внешней памяти, что приведет к значительному росту скорости поиска. Поэтому в реальных системах желательно находить компромисс между минимальным и максимально возможным значением M . Внутренний индекс ни в коем случае не должен занимать всю оперативную память. При выполнении операции поиска в

оперативную память временно подгружаются некоторые внешние блоки и листовые вершины. Если для такой операции будет недостаточно ресурсов, то система не сможет корректно работать.

Рассмотрим еще один нюанс, связанный с внешними блоками – число вершин, которые помещаются в один внешний блок. Этот параметр также не может быть безграничным. Должно быть некоторое ограничение, которое позволит в дальнейшем без проблем целиком загружать любой внешний блок в память при выполнении операций над деревом. Такое ограничение может вводиться либо на максимальную высоту поддеревы во внешнем блоке, либо на максимальное число элементов в этом блоке. Назовем этот параметр *объемом внешнего блока* и обозначим как H (в дальнейших примерах в качестве H будет использовано количество узлов во внутреннем блоке, хотя изначально разработчики предполагали использовать высоту поддеревы).

Если в системе введено ограничение H , то рано или поздно после некоторых манипуляций с деревом произойдет переполнение внешнего блока (число вершин, сохраненных в нем, станет больше H). При этом внешний блок должен быть расщеплен некоторой вершиной на два других самостоятельных внешних блока. Поэтому в системе одновременно может существовать несколько поддеревы, выгруженных во внешнюю память.

Резюмируя все написанное, можно сделать вывод, что при конкретной реализации *LSD*-дерева необходимо определиться с тремя его параметрами:

N – объем листового узла дерева – максимальное число объектов, которые могут находиться в одной листовой вершине дерева;

M – объем внутреннего индекса дерева – максимальное число внутренних вершин дерева, которые могут находиться в оперативной памяти;

H – объем внешнего блока – максимальное число вершин, которые могут находиться в одном внешнем блоке.

Выбор этих параметров очень сильно влияет на производительность структуры в целом. Однако общих рекомендаций, подходящих абсолютно для всех случаев, не существует. Все зависит от количества оперативной памяти, размера объекта и минимального блока внешней памяти, к которому можно обращаться в данной системе.

Прежде чем переходить к описанию алгоритмов построения и обработки дерева, рассмотрим сильные и слабые стороны данной структуры по сравнению с описанными ранее структурами.

Одним из интересных свойств является снятие ограничений на выбор плоскостей разбиения во внешних узлах дерева. Это связано с тем, что, во-первых, гиперплоскости деления могут проходить через любую координату, а не через некоторый объект, как в оригинальном *K-D*-дереве, а во-вторых, отсутствует жесткая последовательность

чередования осей, перпендикулярно которым проходят деление пространства поиска. Данный факт позволяет произвольно выбирать фактор деления пространства, наиболее подходящий для данной конкретной задачи.

Такой подход похож на принцип, примененный в *K-D-B*-дереве, в котором так же нет жестких ограничений на последовательность плоскостей деления и их положение. Однако одним из минусов *K-D-B*-дерева является рекурсивное деление вниз вершин при вставке новых объектов. Данная процедура приводит к появлению полупустых или даже пустых листовых вершин и маленькому коэффициенту использования памяти. В *LSD*-дереве нет процедуры рекурсивного деления дочерних узлов, непосредственно не участвующих во вставке некоторого объекта. Поэтому данный недостаток для него не характерен.

Вместе с сильными сторонами у *LSD*-дерева есть и слабые места. Одним из них является несбалансированность структуры. Если не используется процедура полной повторной балансировки дерева, то теоретически может появиться ситуация, при которой дерево выродится в простой список и не будет предоставлять значительных преимуществ при поиске объектов. Этот недостаток достался *LSD*-дереву в наследство от *K-D*-деревьев, которые также являются несбалансированными структурами. Вторым недостатком *LSD*-дерева можно считать отсутствие идеальных процедур деления внешних блоков. При несбалансированном поддереве во внешнем блоке деление этого блока при переполнении даст плохие результаты. Один из образовавшихся блоков будет редко заполненным, в то время как второй блок останется практически переполненным.

Чтобы избавиться хотя бы частично от указанных недостатков, разработчики *LSD*-дерева ввели новое понятие – *блочная высота дерева*. При этом пришлось модернизировать алгоритмы вставки объектов для сбалансированности по этому показателю.

Под блочной высотой дерева разработчики понимают число внешних блоков, которое нужно пройти при поиске определенного объекта в дереве. По сути, эта высота равна числу обращений к внешней памяти при поиске объекта.

На рис. 2.9 блочная высота дерева для объектов, находящихся в правом верхнем углу, равна двум (для поиска этих объектов нужно будет один раз обратиться к внешней памяти для получения правого поддерева из внешнего блока и один раз – для получения объектов из листовой вершины). При этом блочная высота для объектов, расположенных в левом нижнем углу, в том же примере равна 1 (для доступа к этим объектам необходимо всего один раз обратиться к внешней памяти для загрузки листовой вершины).

Если учесть, что блочная высота является показателем числа обращений к внешней памяти, то можно сделать вывод, что именно этот

параметр и будет играть главную роль при работе с *LSD*-деревом. Поэтому разработчики структуры адаптировали свои алгоритмы таким образом, чтобы в любом *LSD*-дереве выполнялось следующее условие: число внешних блоков, через которое необходимо пройти от корня до листовой вершины, чтобы найти любой объект или сделать вывод о его отсутствии в дереве, может отличаться не более чем на 1 для всех возможных путей в дереве.

Фактически разработчики этим условием ввели понятие балансировки по внешним блокам и заявили, что показатель сбалансированности по этой величине в *LSD*-дереве является почти идеальным.

Алгоритм поиска объекта в LSD-дереве

Структура внутреннего узла *LSD*-дерева очень похожа на аналогичную структуру адаптивного *K-D*-дерева. В ней также содержится два параметра – номер оси пространства, перпендикулярно которой проведена гиперплоскость разбиения, и координата на этой оси, через которую проходит эта гиперплоскость. Однако, в отличие от *K-D*-дерева, конечные объекты находятся во внешней памяти, и некоторые внутренние вершины дерева также могут быть выгружены во внешний блок. Поэтому в каждом узле необходимо предусмотреть флаг, по которому можно будет определить, является ли данная ссылка указателем на внутренний узел в оперативной памяти, указателем на некоторое поддерево во внешнем блоке или ссылкой на листовой блок. Если учесть данный флаг, то процедура поиска становится очень похожей на процедуры, описанные ранее. Рассмотрим для примера одну из таких процедур – запрос по точному совпадению (см. листинг 2.16).

Листинг 2.16

```
//=====
// Поиск объекта О в дереве по полному совпадению
// Параметры:
//      О = (K0, K1, ..., Kk-1) – объект поиска
//=====
ПОИСК_ОБЪЕКТА(О)
[1] // Проверка дерева на пустоту
    V = корень дерева
    Если V = NULL, то
        Вернуть ЛОЖЬ
[2] // Поиск в листовых вершинах
    Если V – листовая вершина, то
        Для всех объектов О' вершины V проверить
            Если О = О', то
                Вернуть ИСТИНА
            Вернуть ЛОЖЬ
[3] // Продвижение по дереву (для внутренних вершин)
```

```

i = V.i
k = V.k
Если  $K_i(O) \geq k$ , то
    V = Right(V)
Иначе
    V = Left(V)
Перейти к шагу 2
Конец ПОИСК_ОБЪЕКТА

```

Процедура ищет объект, заданный k_1, k_2, \dots, k_i ключами, в дереве поиска. Если такой объект не будет найден, то возвращается значение ЛОЖЬ, иначе – значение ИСТИНА.

На первом шаге алгоритма происходит инициализация переменных и проверка существования дерева. В переменную V заносится текущая проверяемая вершина (изначально поиск начинается с корня дерева), и она проверяется на пустоту. Если корень дерева окажется равен $NULL$, то искомого объекта в дереве нет, поэтому можно просто вернуть значение ЛОЖЬ.

Второй шаг алгоритма выполняется только для листовых вершин. Для этого используется флаг текущей вершины, по которому можно определить ее тип. Если окажется, что данная вершина является листом дерева, то необходимо проверить все ее объекты на совпадение с поисковым объектом O . Если среди элементов листовой вершины подобных объектов найдено не будет, то поиск можно завершить с отрицательным результатом.

В листинге 2.16 с целью упрощения были опущены загрузка данных из внешней памяти. Все листовые вершины LSD -дерева находятся во внешней памяти. Поэтому при выполнении шага 2 приведенного алгоритма необходимо сначала подгрузить объекты листовой вершины V в оперативную память, произвести проверку, а после проверки — освободить занятую память. В листинге данные шаги были опущены с целью упрощения изложения.

На третьем шаге алгоритма происходит продвижение по дереву вниз. Этот шаг выполняется только для внутренних вершин дерева (для листовых вершин проверка выполняется на втором шаге и выполнение процедуры на нем прекращается).

Каждая внутренняя вершина дерева V делит пространство поиска на две части в зависимости от номера оси измерения (параметр $V.i$) и координаты по этому измерению (параметр $V.k$). Именно от этих величин зависит, какое из поддеревьев необходимо проверять дальше. Если i -й ключ объекта O больше значения $V.k$, то необходимо перейти в правое поддерево, иначе – в левое. Опять же с целью наглядности в листинге 2.16 была опущена проверка и загрузка внешних блоков. При переходе в одно из поддеревьев, анализируя флаг типа вершины, нужно проверить, не выгружено ли соответствующее поддерево во внешнюю память. Если

такая ситуация произошла, то перед переходом в соответствующее поддереву необходимо временно его подгрузить в оперативную память (и после всех проверок память освободить).

Алгоритм добавления нового объекта в LSD-дерево

Все объекты *LSD*-дерева хранятся в листовых вершинах. Поэтому добавление объекта происходит в листовую вершину. После такого добавления может произойти переполнение листа. В этом случае необходимо расщепить его и модернизировать структуру дерева, чтобы восстановить основные свойства, описанные ранее. Пример реализации процедуры вставки объекта показан в листинге 2.17.

Листинг 2.17

```
//=====
// Вставка нового объекта в дерево
// Параметры:
//   О - объект, который необходимо вставить в дерево
//=====
ВСТАВКА(О)
[1] // Проверка дерева на существование
    V = корень дерева
    Если V = NULL, то
        V = пустая листовая вершина
        Добавить О в V
        Корень дерева = V
        Завершить процедуру вставки
[2] // Поиск листа для вставки
    Если V – внутренняя вершина дерева, то
        i = V.i
        k = V.k
        Если  $K_i(O) \geq k$ , то
            V = Right(V)
        Иначе
            V = Left(V)
        Перейти к шагу 2
[3] // Вставка объекта в листовую вершину
    Добавить объект О в V
    Если количество объектов в V  $\leq N$ , то
        Завершить процедуру вставки
[4] // Расщепление переполненного листового узла
    Q = РАСЩЕПИТЬ_ЛИСТ(V)
    Vp = Parent(V)
    Заменить ссылку Vp->V на Vp->Q
    Удалить старую вершину V
[5] // Модернизация дерева
    Если Vp – вершина во внутреннем индексе, то
        m = Количество вершин во внутреннем индексе
        Если m  $\leq M$ , то
```

```

        Завершить процедуру вставки
    Иначе
        ВЫГРУЗИТЬ_ЧАСТЬ_ДЕРЕВА()
    Иначе
        В = внешний блок, в котором находится Vp
        Добавить Q в В
        h = количество вершин в блоке В
        Если h <= N, то
            Завершить процедуру вставки
        Иначе
            Q = РАСЩЕПИТЬ_ВНЕШНИЙ_БЛОК(В)
            Если Q ≠ NULL, то
                Vp = Parent(Q)
            Перейти к шагу 5
Конец ВСТАВКА

```

Алгоритм вставки объекта в *LSD*-дерево начинается с проверки существования дерева (шаг 1). Для этого корневая вершина сравнивается с *NULL*. Если дерево еще не существует, то необходимо создать пустую листовую вершину и поместить в нее добавляемый объект. Эта вершина и будет новым корнем дерева.

Если дерево является не пустым, то необходимо определить тот лист, в который должен разместиться вставляемый объект. Из свойств дерева можно сделать вывод, что при непустом дереве, в нем обязательно будет листовая вершина, соответствующая абсолютно любой части пространства (*LSD*-дерево, так же как и адаптивное *K-D-B*-дерево, покрывает все отведенное пространство). Процедура поиска листовой вершины полностью идентична той, что использовалась в листинге 2.16.

После того, как нужная вершина найдена, вставляемый объект добавляется в нее (шаг 3). Если при этом не нарушилось ограничение на объем листового листа (число в нем меньше или равно *N*), то процедуру вставки можно завершить.

Если же произошло переполнение листа дерева, то необходимо расщепить его на две части (шаг 4). Для этого вызывается процедура *РАСЩЕПИТЬ_ЛИСТ*, которой в качестве параметра передается переполненный узел. Действие этой процедуры чрезвычайно просто: она разбивает узел вдоль некоторой оси пространства на две части и создает еще одну внутреннюю вершину дерева, отвечающую за данное разбиение. Рассмотрим этот процесс подробнее на примере рис. 2.10.

На рисунке представлен случай, когда один из листовых узлов *V* оказался переполненным (в него поместили пятый объект, а ограничение *N* для данного дерева равно 4). Особенностью *LSD*-дерева является то, что процедура разбиения листовой вершины не нуждается в какой-либо информации о структуре дерева целиком. Она не затрагивает другие вершины и может опираться только на критерии сбалансированности и

оптимальности деления пространства (конкретные алгоритмы деления листовой вершины будут рассмотрены далее).



Рис. 2.10. Расщепление листовой вершины LSD-дерева

Допустим, процедура деления разбила вершину V перпендикулярно оси Ox на две равные части (как показано на рис. 2.10). В этом случае создается некоторая внутренняя вершина Q , в которой записывается номер оси разбиения и координата. Потомками этой вершины становятся два только что созданных листа. Именно эту вершину Q процедура расщепления возвращает в качестве результата своей работы. Остается просто включить ее в дерево (она замещает старую ссылку на вершину V).

Однако включение вершины Q в дерево также может вызвать некоторые проблемы. Рассмотрим все возможные случаи.

- Родительская вершина V_p находится в оперативной памяти (внутренний индекс), и число узлов во внутреннем индексе меньше M . При этом происходит простое добавление вершины Q во внутренний индекс (замена ссылки $V_p \rightarrow V$ на $V_p \rightarrow Q$).

- Родительская вершина V_p находится в оперативной памяти (внутренний индекс), но число узлов во внутреннем индексе уже равно максимально допустимому значению M . При этом после добавления вершины Q в дерево произойдет переполнение внутреннего индекса. Поэтому необходимо после вставки вызвать процедуру ВЫГРУЗИТЬ_ЧАСТЬ_ДЕРЕВА, которая определит некоторое поддерево, создаст из него внешний блок и выгрузит его во внешнюю память.

- Родительская вершина V_p находится во внешнем блоке и число узлов в нем меньше N . При этом происходит простое добавление вершины Q во внешний блок и замена ссылки $V_p \rightarrow V$ на $V_p \rightarrow Q$.

- Родительская вершина V_p находится во внешнем блоке, но число узлов в нем равно N . При этом добавление вершины Q во внешний блок также вызовет его переполнение и однозначно появится необходимость выполнения очередной процедуры РАСЩЕПИТЬ_ВНЕШНИЙ_БЛОК,

которая приведет всю структуру дерева в полное соответствие со свойствами *LSD*-дерева. Данная процедура делает из одного внешнего блока два блока меньшего размера и вернет новую вершину Q , которая является вершиной деления этих двух блоков (вершиной, для которой новые блоки будут дочерними). Чтобы свойства дерева не нарушились, необходимо разместить и эту вершину в дереве по аналогичному принципу (т. е. необходимо вернуться к началу шага 5 данной процедуры). Такой подход позволяет распространить расщепление внешних блоков к корню, если это является необходимым.

Однако иногда процедура *РАСЩЕПИТЬ_ВНЕШНИЙ_БЛОК* может восстановить свойства *LSD*-дерева без расщепления внешнего блока. При этом она вернет не указатель на вершину деления Q , а некоторую предопределенную константу *NULL*. Такой флаг будет означать, что все свойства дерева восстановлены, возвращаться и повторять шаг 5 данного алгоритма не нужно.

Алгоритм расщепления листовой вершины *LSD*-дерева

Рассмотрим более подробно алгоритм деления переполненной вершины (процедуру *РАСЩЕПИТЬ_ЛИСТ*). Как было отмечено, разработчики *LSD*-дерева не стали накладывать абсолютно никаких ограничений на эту процедуру. Деление листовой вершины может происходить вдоль любой из осей пространства по любой координате, в зависимости от того, какие принципы заложены в данную реализацию. Чередование осей измерения (как в случае оригинального *K-D*-дерева) здесь жестко не регламентируется.

Среди всего многообразия алгоритмов можно выделить несколько разных подходов к процедуре деления в зависимости от того, что именно положено во главу данного процесса.

1. *Стратегия, ориентированная на данные.* В процедурах деления данного типа выбор оси и координаты разбиения происходит исключительно на основе анализа находящихся в листовой вершине объектов. Примером воплощения такой стратегии может служить следующий алгоритм:

- в качестве оси разбиения выбираем ту ось пространства, вдоль которой разброс объектов является самым большим (например, выбрать ту ось пространства, по которой разность максимальной и минимальной координаты для некоторых объектов будет максимальна);
- в качестве координаты разбиения выбирается такое значение по выбранной оси, которое разделит все объекты листовой вершины приблизительно на две равные группы.

2. *Стратегия, зависящая от некоторых нюансов конкретной реализации.* В этом случае выбор параметров разбиения зависит от конкретных требований разработчика и может основываться на некоторых априорных знаниях о распределении объектов. Примерами

таких стратегий можно считать следующие:

- жесткое чередование осей разбиения (если предыдущая вершина производит деление пространства по оси Ox , то следующая должна делить по оси Oy); этот принцип похож на правила деления, заложенные в оригинальных $K-D$ -деревьях;
- приоритет некоторых осей по сравнению с другими (если известно, что запросы по некоторым ключам будут использоваться в два раза чаще других, то разбиение по соответствующим направлениям так же можно сделать в два раза чаще);
- ориентирование на геометрический размер листовых вершин (в некоторых приложениях может понадобиться разбиение листовых вершин по геометрическим признакам, таким как геометрический центр вершины или приближение получившихся вершин к форме квадрата).

Для примера рассмотрим листинг процедуры деления вершины по принципу наибольшей сбалансированности, т.е. деление вдоль некоторой оси таким образом, чтобы образовавшиеся вершины содержали приблизительно равное число вершин. Данная процедура представлена в листинге 2.18.

Листинг 2.18

```
//=====
// Расщепление листа на две части
// Параметры:
//   V – переполненная листовая вершина
//=====
РАСЩЕПИТЬ_ЛИСТ(V)
[1] // Вычисление минимальной и максимальной координат
    // по каждой из осей пространства
    Max = массив из k элементов, заполненный  $-\infty$ 
    Min = массив из k элементов, заполненный  $+\infty$ 
    Для всех объектов O вершины V выполнить
        Для каждого измерения i выполнить
            Если  $K_i(O) > \text{Max}[i]$ , то
                 $\text{Max}[i] = K_i(O)$ 
            Если  $K_i(O) < \text{Min}[i]$ , то
                 $\text{Min}[i] = K_i(O)$ 
[2] // Выбор оси с максимальным разбросом
    I = 0
    Для каждого измерения i выполнить
        Если  $(\text{Max}[I] - \text{Min}[I]) < (\text{Max}[i] - \text{Min}[i])$ , то
            I = i
[3] // Выбор координаты разбиения
    Arr = массив из (N+1) элементов
    Для каждого объекта  $O_j$  вершины V выполнить
         $\text{Arr}[j] = K_I(O_j)$ 
    Сортировать массив Arr
```


$$Key = \frac{Arr \left[\frac{N}{2} \right] + Arr \left[\frac{N}{2} + 1 \right]}{2}$$

[4] // Разбить вершину V на две части

V_1 = новая пустая листовая вершина

V_r = новая пустая листовая вершина

Для всех объектов O вершины V выполнить

Если $K_I(O) \geq Key$, то

Поместить O в V_r

Иначе

Поместить O в V_1

[5] // Создать родительскую вершину для новых листов

Q = новая внутренняя вершина

Ось разбиения для Q = I

Координата разбиения для Q = Key

Left(Q) = V_1

Right(Q) = V_r

[6] // Вернуть результат

Вернуть вершину Q

Конец РАСЩЕПИТЬ_ЛИСТ

В приведенном листинге выбрана стратегия деления листовых вершин, ориентированная на данные. На первом шаге для каждого измерения рассчитываются максимальные и минимальные координаты, а на втором — выбирается измерение I , для которого разность максимальной и минимальной координат максимальна. Это позволяет выбрать измерение с максимальной абсолютной разницей в координатах. Однако иногда стоит учитывать не абсолютную, а относительную разницу по измерениям или использовать некоторые другие функции распределения, основанные на плотности точек по осям пространства.

На третьем шаге выбирается координата разбиения. Для этого создается и сортируется массив координат по выбранному измерению. В качестве координаты разбиения выбирается среднее арифметическое двух точек, попавших в середину массива после сортировки. Это позволяет в большинстве случаев разбить листовую вершину на две части приблизительно равные по числу элементов.

На четвертом шаге алгоритма объекты старой вершины V распределяются по двум новым вершинам V_1 и V_r в соответствии с выбранными параметрами. После этого разбиение можно считать завершенным. Однако для полного завершения процедуры необходимо создать некоторую родительскую вершину для двух только что образованных листовых вершин. Это и происходит на пятом шаге приведенного алгоритма.

Алгоритм расщепления внешнего блока *LSD*-дерева

Переполняться в *LSD*-дереве могут не только листовые узлы, но и внешние блоки. Внешним блоком в *LSD*-дереве является объединение некоторых внутренних узлов (некоторого поддерева) в один общий блок для выгрузки во внешнюю память. При выполнении операций вставки и расщеплении листовых узлов могут появляться вершины, которые также будут включаться в тот или иной внешний блок (см. алгоритм листинга 2.18). Такое положение дел рано или поздно неизбежно приведет к переполнению этих блоков. В качестве критериев переполнения внешних блоков можно выбрать либо высоту поддерева, хранящегося в данном блоке, либо число узлов, размещенных в нем.

При переполнении внешнего блока он расщепляется на две части и образуются два новых внешних блока. При этом очень важным условием является сохранение свойств сбалансированности дерева – расстояние от корня до любой листовой вершины, измеренное в количестве внешних блоков, не может отличаться более чем на единицу. Чтобы данное условие не нарушалось, расщепление внешнего блока надо производить как выделение двух поддеревьев корневой вершины данного внешнего блока. Рассмотрим этот алгоритм на примере. Для этого предположим, что после некоторой операции у нас имеется внешний блок, показанный на рис. 2.11.

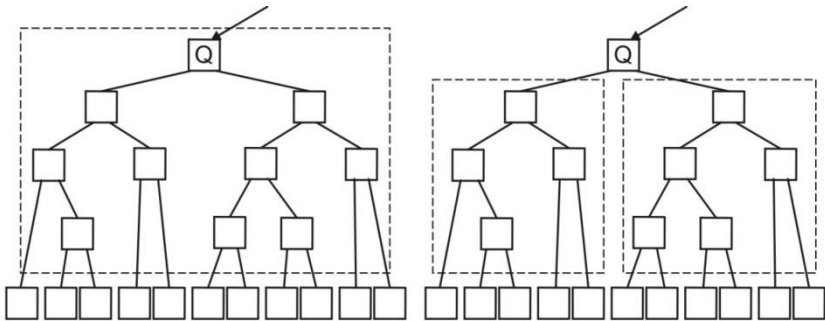


Рис. 2.11. Расщепление внешнего блока *LSD*-дерева

В этом блоке находится десять вершин. Если в качестве объема внешнего блока принято ограничение $H = 9$, то данный блок переполнен и его необходимо разделить на две части.

Алгоритм разбиения заключается в следующем шаге. Берем корневую вершину поддерева данного внешнего блока (в данном случае это вершина *Q*) и делаем ее вершиной деления. Правое поддерево этой вершины мы помещаем в один новый внешний блок, а левое – в другой. Таким образом, мы получаем два новых внешних блока, которые уже

удовлетворяют условию максимального объема (число вершин в каждом из них меньше N).

Однако остается открытым вопрос, куда должна перейти сама вершина Q . Если ее разместить в один из блоков, то это приведет к нарушению условия балансировки дерева (в этом случае один из внешних блоков станет как бы дочерним для другого, т. е. для доступа к некоторым узлам дерева придется проходить оба внешних блока, что увеличивает высоту дерева, измеренную во внешних блоках). Чтобы условие балансировки не нарушилось, необходимо вершину Q вынести из обоих внешних блоков и поместить во внутренний индекс или предыдущий внешний блок. Именно такая операция производится в листинге 2.18 на шаге 5.

У данного алгоритма есть один серьезный недостаток. Если поддерево, находящееся во внешнем блоке, является несбалансированным (т. е. у вершины Q один из потомков по количеству вершин гораздо больше другого), то алгоритм расщепления сделает неравномерное разбиение. Один из новых внешних блоков будет почти пустым, в то время как другой внешний блок будет иметь почти критическую величину по объему.

Развитием этой идеи является случай, когда один из потомков вершины Q является ссылкой на листовой узел (вырожденный случай). При этом один из внешних блоков должен получиться абсолютно пустым. Чтобы не создавать пустых блоков, можно немного модифицировать алгоритм расщепления, предложенный разработчиками. Для этого при вырожденном поддереве расщепление внешнего блока не производится. Процедура просто удаляет вершину Q , а в ее предке V_p ссылку меняет на то поддерево, которое является не пустым. После данной процедуры число вершин и глубина поддерева во внешнем блоке уменьшится на 1, что восстановит свойства *LSD*-дерева. Однако объекты из листовой вершины, которая ранее была дочерней вершиной для Q , были также удалены из дерева (в дереве нет ссылки на этот лист). Поэтому необходимо выполнить повторную вставку этих объектов в дерево. Не трудно показать, что такая модификация в большинстве случаев не только даст возможность избежать расщепления внешнего блока, но и повысит общую сбалансированность структуры.

Описанный алгоритм расщепления внешнего блока представлен в виде псевдокода в листинге 2.19.

Листинг 2.19

```
//=====
// Расщепление внешнего блока
// Параметры:
// В – переполненный внешний блок
//=====
```

РАСЩЕПИТЬ_ВНЕШНИЙ_БЛОК (В)

- [1] // Определение вершины Q и его предка
 Q = корневая вершина блока В
 Vp = Parent(Q)
 - [2] // Проверка на вырожденность левого потомка
 Если Left(Q) - листовая вершина, то
 $Q' = \text{Right}(Q)$
 Заменить ссылку $Vp \rightarrow Q$ на $Vp \rightarrow Q'$
 Для всех объектов O вершины Left(Q) выполнить
 ВСТАВКА (O)
 Удалить Q
 Удалить Left(Q)
 Выйти из процедуры и вернуть NULL
 - [3] // Проверка на вырожденность правого потомка
 Если Right(Q) - листовая вершина, то
 $Q' = \text{Left}(Q)$
 Заменить ссылку $Vp \rightarrow Q$ на $Vp \rightarrow Q'$
 Для всех объектов O вершины Right(Q) выполнить
 ВСТАВКА (O)
 Удалить Q
 Удалить Right(Q)
 Выйти из процедуры и вернуть NULL
 - [4] // Оба потомка не вырождены - разбить блок
 B' = новый внешний блок
 Для всех узлов V поддерева Left(Q) выполнить
 Исключить V из блока В
 Поместить V в блок B'
 Исключить вершину Q из блока В
 Выйти из процедуры и вернуть Q
- Конец РАСЩЕПИТЬ_ВНЕШНИЙ_БЛОК

Алгоритм выгрузки некоторого поддерева во внешний блок

Этот алгоритм вызывается после переполнения внутреннего индекса дерева. При вставке очередного элемента и расщепления листовых блоков может произойти ситуация, при которой во внутренний индекс добавится еще одна вершина. Если при этом число вершин в индексе окажется больше некоторого порогового значения M (объема внутреннего индекса), то появится необходимость часть вершин объединить в один блок и выгрузить его во внешнюю память. Однако, чтобы не нарушить свойств *LSD*-дерева, объединять вершины нужно не произвольно, а согласно принадлежности их определенному поддереву, т. е. при переполнении происходит выгрузка определенного поддерева.

В процедуре определения поддерева для выгрузки необходимо также учитывать тот факт, что после выполнения данной операции не должна нарушаться сбалансированность дерева по внешним блокам и новый блок не должен быть переполненным. Поэтому выбранное поддерево должно удовлетворять следующим условиям:

– любой путь от корня до листовых узлов, проходящий через выбранное поддерево, должен быть минимальным в смысле числа внешних блоков;

– число узлов в выбранном поддереве не должно превышать H .

На рис. 2.12 показана процедура выгрузки поддерева во внешний блок. После некоторой операции число узлов во внутреннем индексе стало равно 6. Если для данного дерева ограничение $M = 5$, то появляется необходимость в создании нового внешнего блока и выгрузки части внутреннего индекса в него. В качестве такого поддерева была выбрана часть индекса, начинающаяся с вершины V_s .

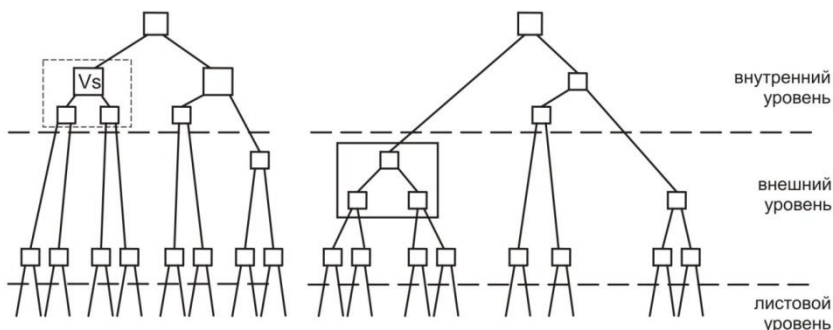


Рис. 2.12. Выгрузка части индекса во внешний блок

На практике может существовать несколько кандидатов для выгрузки. При этом лучше выбирать то поддерево, которое содержит число узлов, близкое к H (при условии, что H намного меньше M). Это позволит в дальнейшем больший промежуток времени обходиться без выгрузки во внешние блоки вершин внутреннего индекса при вставках элементов.

Существуют и более эффективные стратегии выбора поддерева для выгрузки, позволяющие построить оптимальное дерево. Однако они сложнее в реализации и требуют больших вычислительных ресурсов.

Рассмотрим один из возможных алгоритмов выгрузки поддерева во внешнюю память (листинг 2.20). В этой процедуре предполагается, что с каждой внутренней вершиной дерева дополнительно связаны следующие величины:

- $V.B_{min}$ – минимальная высота данного поддерева V , вычисленная во внешних блоках;
- $V.B_{max}$ – максимальная высота данного поддерева V , вычисленная во внешних блоках ($V.B_{max} - V.B_{min}$ всегда равно 0 или 1 для любого LSD -дерева);
- $V.S$ – число внутренних узлов в поддереве данной вершины V .

Перечисленные параметры могут постоянно храниться во всех вершинах внутреннего индекса или временно вычисляться перед процедурой выгрузки. Однако при постоянном хранении необходимо модернизировать процедуру вставки таким образом, чтобы при всех изменениях внутреннего индекса данные параметры пересчитывались для соответствующих вершин (при изменениях листовых блоков или внешнего индекса их пересчитывать не нужно, так как такие изменения не влияют на указанные величины).

Листинг 2.20

```
//=====
// Выгрузка поддерева во внешний блок
//=====
ВЫГРУЗИТЬ_ЧАСТЬ_ДЕРЕВА()
[1] // Инициализация
    V = корневая вершина дерева
[2] // Проверка на возможность выгрузки
    Если (V.Bmin = V.Bmax) И (V.S < H), то
        В = новый пустой внешний блок
        Поместить все вершины поддерева V в блок В
        Выйти из процедуры выгрузки
[3] // Перейти ниже по дереву
    Если (Left(V).Bmin < Right(V).Bmin) ИЛИ
        ((Left(V).Bmin = Right(V).Bmin) И
         Left(V).S > Right(V).S), то
        Если Left(V) – вершина внутреннего индекса, то
            V = Left(V)
            Перейти к шагу 2
        Иначе
            В = новый пустой внешний блок
            Добавить в блоке В ссылку на Left(V)
            Left(V) = ссылка на блок В
            Увеличить V.Bmin и V.Bmax от V до корня
            Перейти к шагу 1
        Иначе
            Если Right(V) – вершина внутреннего индекса, то
                V = Right(V)
                Перейти к шагу 2
            Иначе
                В = новый пустой внешний блок
                Добавить в блоке В ссылку на Right(V)
                Right(V) = ссылка на блок В
                Увеличить V.Bmin и V.Bmax от V до корня
                Перейти к шагу 1
    Конец ВЫГРУЗИТЬ_ЧАСТЬ_ДЕРЕВА
```

На первом шаге алгоритма происходит инициализация переменных. Поиск поддерева для выгрузки начинается с корня дерева, поэтому

изначально в переменную V заносится именно корневая вершина. Далее эта переменная будет содержать нового претендента для выгрузки, пока не будет найдена такая вершина, которая удовлетворит всем необходимым свойствам.

На шаге 2 данного алгоритма происходит проверка текущей вершины на возможность ее выгрузки. Вершина V считается хорошим претендентом, если в поддереве внутреннего индекса, начинающегося с V , содержится не более H вершин ($V.S < H$), и в то же время минимальная и максимальная высота всех возможных путей в дереве, проходящих через V , совпадает ($V.B_{min} = V.B_{max}$). Второе условие весьма важно, так как при его несоблюдении может нарушиться балансировка дерева по внешним блокам.

Если текущая вершина отвечает двум указанным условиям, то можно создавать еще один внешний блок и выгружать в него все поддерево V . Если же хотя бы одно из условий не выполняется, то необходимо перейти по дереву вниз, чтобы найти другого претендента для выгрузки.

Каждая вершина LSD -дерева содержит двух потомков. При переходе от вершины V к дочерней вершине выбирается тот ее потомок, который имеет меньшую высоту во внешних блоках (у которого величина $V.B_{min}$ меньше). При равенстве величин $V.B_{min}$ у обоих потомков следует выбирать того, у которого больше внутренних узлов (у которого величина $V.S$ больше). Данная проверка и переход происходят на шаге 3. После перехода к новому претенденту выполняется возврат к шагу 2 и повторяется циклическая проверка новой вершины V на возможность выгрузки.

Однако при перемещении вниз по дереву может возникнуть ситуация, при которой будет пройден весь внутренний индекс LSD -дерева, а претендента, удовлетворяющего всем условиям, найдено не будет. Для обработки этой ситуации создается новый пустой внешний блок дерева, который просто увеличивает высоту данной ветки, рассчитанную во внешних блоках. Данный этап также необходим, так как без него может нарушиться балансировка дерева.

Для демонстрации работы всех описанных процедур рассмотрим пример, представленный на рис. 2.13.

На рисунке изображено LSD -дерево со следующими ограничениями: внутренний индекс не должен содержать более 2-х вершин ($M = 2$), и внешний блок также не должен содержать более 2-х вершин ($H = 2$). Такие маленькие ограничения выбраны в учебных целях, чтобы нагляднее показать работу всех процедур деления. На практике выбор малых M и H приведет к падению производительности структуры.

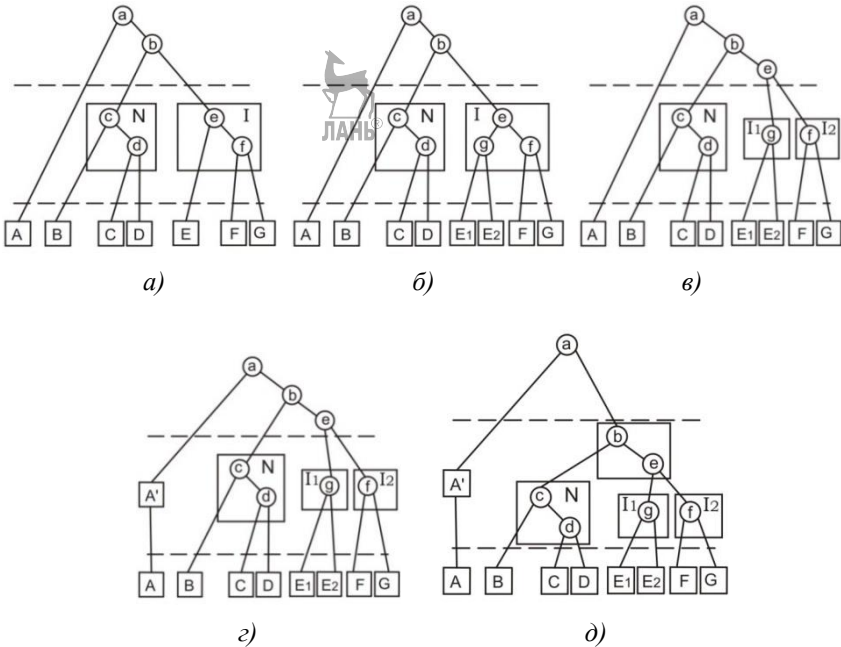


Рис. 2.13. Расщепление внешнего блока LSD-дерева: а – исходное дерево; б – деление листа E; в – деление внешнего блока I; г – создание пустого блока A'; д – создание внешнего блока J

Допустим, произошла очередная вставка объекта в дерево, показанная на рис. 2.13(а). При этом вставленный объект разместился в листовой вершине E и это привело к ее переполнению.

Чтобы восстановить нормальное состояние, вершина E разбивается на две новые вершины E1 и E2. Параметры разбиения заносятся в новую внутреннюю вершину g, которая становится дочерней для вершины e и размещается в том же самом внешнем блоке I (рис. 2.13(б)).

Однако при таком изменении произошло переполнение внешнего блока I. Поэтому будет вызвана процедура РАСЩЕПИТЬ_ВНЕШНИЙ_БЛОК (I), которая разобьет блок I на два – I1 и I2. При этом, согласно алгоритму листинга 2.19, внутренняя вершина g перейдет в блок I1, вершина f – в блок I2, а вершина e – во внутренний индекс (рис. 2.13(в)).

Перемещение вершины e во внутренний индекс приведет к его переполнению. Поэтому сразу за таким перемещением будет вызвана процедура ВЫГРУЗИТЬ_ЧАСТЬ_ДЕРЕВА (), которая должна создать новый внешний блок и выгрузить в него часть дерева (см. листинг 2.20). Однако в самом начале работы процедура столкнется с одной проблемой.

У вершины a левый потомок имеет высоту, измеренную во внешних блоках, равную 1, а правый – 2. Следовательно, согласно алгоритму, чтобы не нарушить балансировку дерева, необходимо добавить внешний блок для левого потомка вершины a . Такой блок добавляется (блок A'), однако в дереве нет ни одной вершины, которую можно было бы в него переместить (рис. 2.13(г)).

Создание блока A' не привело к восстановлению всех свойств дерева. Внутренний индекс по-прежнему содержит 3 вершины и нуждается в выгрузке. Поэтому работа процедуры ВЫГРУЗИТЬ_ЧАСТЬ_ДЕРЕВА продолжается. На этот раз вершина a имеет равную высоту во внешних блоках как для левого, так и для правого потомков. Однако выгрузить все поддерево A во внешний блок нельзя, т. к. в этом случае внешний блок будет содержать 3 вершины, что нарушит другое свойство дерева. Поэтому создается внешний блок для вершины b и в него выгружаются вершины b и c (рис. 2.13(д)). Теперь все свойства дерева восстановлены и процедуру вставки и расщепления можно завершить.

Алгоритм удаления объекта LSD-дерева

Процедура удаления по принципу действия полностью противоположна процедуре вставки. Сначала определяется листовой узел, в котором находится удаляемый объект, и из него этот объект исключается. После этого, если листовой узел оказался пустым, он также удаляется, как бы объединяясь со своим соседом (см. листинг 2.21).

Возможен и более сложный, но при этом и более эффективный алгоритм. Листовой узел удаляется не только тогда, когда он оказался пустым, но и если его заполненность оказалась меньше некоторого предельного порога, который также может указываться в программе. При этом объекты, которые находились в удаляемом листе, заново вставляются в дерево с помощью обычной процедуры вставки.

Листинг 2.21

```
//=====
// Удаление объекта LSD-дерева
// Параметры:
//   O – удаляемый объект
//=====
УДАЛЕНИЕ(O)
[1] // Проверка дерева на существование
    V = корень дерева
    Если V = NULL, то
        Завершить процедуру, объекта в дереве нет
[2] // Поиск листа, в котором находится O
    Если V – внутренняя вершина дерева, то
        i = V.i
```

```

k = V.k
Если  $K_i(O) \geq k$ , то
    V = Right(V)
Иначе
    V = Left(V)
Перейти к шагу 2
[3] // Удаление объекта из листовой вершины
    Удалить объект O из V
    Если количество объектов в V > 0, то
        Завершить процедуру удаления
[4] // Если пустой лист является корнем
    Если V – корень дерева, то
        Удалить V
        Корень дерева = NULL
        Выйти из процедуры удаления
[5] // Определение предка и соседнего узла
    Vp = Parent(V)
    Если V = Left(Vp), то
        Vs = Right(Vp)
    Иначе
        Vs = Left(Vp)
[6] // Если родитель пустого листа – корень
    Если Vp – корень дерева, то
        Удалить Vp
        Удалить V
        Корень дерева = Vs
        Выйти из процедуры удаления
[7] // Если Vp не корень, то заменить его на Vs
    Vpp = Parent(Vp)
    Удалить Vp
    Удалить V
    Заменить ссылку Vpp->Vp на Vpp->Vs
[8] // Распространение реорганизации вверх по дереву
    Если Vp находился во внутреннем индексе, то
        Выйти из процедуры удаления
    Иначе
        B – внешний блок вершины Vp
        Если B – не пустой блок, то
            Выйти из процедуры удаления
        Иначе
            СЛИТЬ_ВНЕШНИЙ_БЛОК(B)
Конец УДАЛЕНИЕ

```

Процедура удаления, как и другие процедуры, начинается с проверки существования дерева (шаг 1) и поиска листового узла, в котором находится удаляемый объект O (шаг 2). После определения нужного листа V объект O из него исключается и уничтожается. Однако данная операция может привести к необходимости перестроить

некоторую часть дерева. Рассмотрим все возможные варианты последствий удаления объекта из листовой вершины.

1. Листовая вершина V осталась не пустой после удаления из нее объекта O (шаг алгоритма 3). При этом действие процедуры наиболее простое. Необходимо завершить процедуру удаления без реорганизаций (если, конечно, не предусмотрен алгоритм объединения листовых вершин при их слабой заполненности).

2. После удаления образовался пустой узел, причем этот узел является корнем дерева (шаг алгоритма 4). Данная ситуация означает, что дерево стало пустым и необходимо просто присвоить корню специальное значение *NULL*.

3. Родитель V_r данного пустого листа V является корнем (шаг алгоритма 6). При этом корневая вершина V_r и пустой лист V удаляются, а вместо корня помещается соседняя вершина V_s .

4. Родитель V_r данного пустого листа V не является корнем дерева (шаг алгоритма 7). При этом V_r и V также удаляются, и в родителе вершины V_r указатель меняют так, чтобы он стал указывать не на V_r , а на V_s . Однако данный вариант самый сложный. После удаления вершины V_r из дерева может появиться необходимость дальнейшей его модернизации (шаг алгоритма 8).

Рассмотрим возможные варианты и реакции на них.

- Вершина V_r была во внутреннем индексе. При этом реорганизация не нужна, можно завершить процедуру удаления.
- Вершина V_r находилась во внешнем блоке B , но этот блок остался не пустым. При этом реорганизация также не нужна.
- Вершина V_r находилась во внешнем блоке B , который после удаления V_r стал пустым. При этом нужно попытаться слить внешний блок B с соседним для него блоком (если такой существует). Данная операция полностью противоположна операции расщепления, поэтому она не приведена здесь.

Как видно, операция удаления объектов из *LSD*-дерева повторяет аналогичные шаги операции вставки, только вместо процедур деления и добавления в ней используются процедуры удаления и слияния.

2.2.4. Quad-дерево

Quad-дерево (Quadtree) является еще одной структурой для доступа к точечным пространственным данным. Как и *K-D*-дерево, данная структура предназначена для индексирования и поиска данных в оперативной памяти. В оригинальных алгоритмах, предложенных Д. Л. Бентли и Р. А. Финкелем в 1974 году [21], не было предусмотрено никаких особенностей, позволяющих использовать ее для внешней памяти. Однако *Quad*-дерево нашло очень много применений, в том числе для использования в компьютерной графике. В настоящее время

разработано большое множество различных вариантов *Quad*-дерева, позволяющих применять данную структуру не только для хранения точечных объектов, но и пространственных областей, графических данных с разной точностью детализации, сложных форм и изменяющихся во времени объектов. Однако в данной главе будет рассмотрен базовый вариант данной структуры – точечное *Quad*-дерево.

Структура точечного *Quad*-дерева

Базовая идея, заложенная в точечное *Quad*-дерево, очень похожа на оригинальное *K-D*-дерево. В каждой вершине дерева находится точечный объект, который своими координатами делит пространство на части. Однако, в отличие от *K-D*-дерева, деление происходит не на две части некоторой гиперплоскостью, а на 2^k частей, где k – число измерений (число координат в точке). Первоначально *Quad*-деревья разрабатывались для двумерного пространства, хотя распространить эти идеи на три и больше измерений не составляет труда. Однако стоит помнить, что данная структура становится малоэффективной в пространствах большой размерности.

В данной главе рассматривается двумерное точечное *Quad*-дерево. В такой структуре в каждой вершине объект разбивает пространства на четыре части. Соответственно, в вершине должны находиться сам объект и четыре указателя на потомков. В каждом из потомков будут храниться те поддеревья (точки), которые попали в соответствующие части пространства. Пример *Quad*-дерева показан на рис. 2.14.

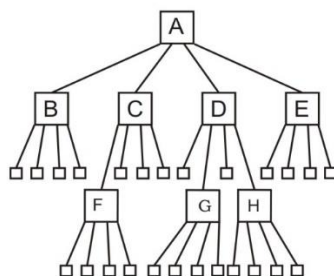
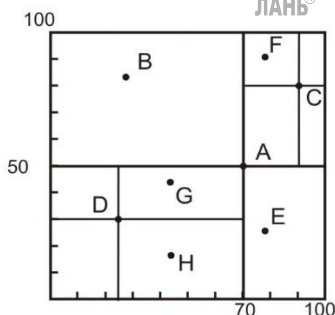


Рис. 2.14. Пример *Quad*-дерева

В корне дерева находится объект *A* с ключами 70 и 50. Он делит пространство на четыре части:

- северо-западное – все объекты, у которых первый ключ меньше 70, а второй – больше 50 (в нашем случае это объект *B*);
- северо-восточное – все объекты, у которых первый ключ больше 70, а второй – больше 50 (в нашем случае это объекты *C* и *F*);

- юго-восточное – все объекты, у которых первый ключ больше 70, а второй – меньше 50 (в данной части находится только объект *E*);
- юго-западное – все объекты, у которых первый ключ меньше 70 и второй меньше 50 (объекты *D, G, H*).

Объекты, попавшие в определенную часть пространства, полученного после деления корневой вершиной, сохраняются в соответствующем поддереве. Таким образом, получается как бы решетка, которая делит пространство в каждой вершине на четыре части.

Если в соответствующую часть пространства попал всего один объект, то он заносится в дочерний узел (в приведенном примере это северо-западное и юго-западное направления). В противном случае деление происходит рекурсивно с одной из вершин данного подпространства.

Алгоритм поиска объекта в Quad-дереве

Процедура поиска по *Quad*-дереву полностью определяется его структурой. Проходя от корня к листьям, в каждой вершине сравнивается объект, находящийся в ней, с искомым. Если ключи этих объектов совпадают, то поиск заканчивается успехом, иначе – происходит перемещение в соответствующий квадрант дерева. Если после такого перемещения процедура оказывается в пустой вершине, то это означает, что искомого объекта в дереве нет. Пример процедуры поиска на точное совпадение при наличии двух ключей в объекте показан в листинге 2.22.

Листинг 2.22

```
//=====
// Поиск объекта О в дереве по полному
// совпадению
// Параметры:
//   О = (K0, K1) – объект поиска
//=====
ПОИСК_ОБЪЕКТА(О)
[1] // Начальная инициализация
    V = корень дерева
[2] // Проверка на существование текущей вершины
    Если V = NULL, то
        Вернуть NULL
[3] // Проверка на совпадение с текущей вершиной
    Если (K0(V) = K0(О)) И (K1(V) = K1(О)), то
        Вернуть V
[4] // Продвижение по дереву
    Если K0(О) ≥ K0(V), то
        Если K1(О) ≥ K1(V), то
            V = Child_NE(V)
        Иначе
```

```

    V = Child_SE(V)
Иначе
    Если  $K_1(O) \geq K_1(V)$ , то
        V = Child_NW(V)
    Иначе
        V = Child_SW(V)
    Перейти к шагу 2
Конец ПОИСК_ОБЪЕКТА

```

В процедуре поиска объекта для обращения к дочерним узлам были использованы следующие обозначения:

Child_NE(V) – северо-восточный потомок вершины *V*;

Child_SE(V) – юго-восточный потомок вершины *V*;

Child_NW(V) – северо-западный потомок вершины *V*;

Child_SW(V) – юго-западный потомок вершины *V*.

Нетрудно заметить, что процедура поиска по точному совпадению очень сильно похожа на аналогичную процедуру в *K-D*-дереве. Это произошло из-за того, что структуры данных деревьев очень похожи и имеют схожие принципы построения. Процедуры диапазонного поиска или поиска ближайшего соседа также очень похожи на соответствующие процедуры, приведенные ранее для *K-D*-дерева, поэтому здесь они дублироваться не будут.

Алгоритм добавления нового объекта в Quad-дерево

Добавление объекта *O* с ключами K_1 и K_2 в *Quad*-дерево является очень простой операцией. Она очень похожа на добавление записи в обычное *K-D*-дерево. Дерево начинает строиться с добавления корня. Запись, которая добавляется в дерево первой, занимает место корня и делит все пространство на четыре квадранта в зависимости от своих координат.

При вставке следующих записей выполняется процедура, аналогичная поиску записи. Если в дереве обнаруживается вершина с такими же ключами, то процесс завершается, так как считается, что данный объект уже был добавлен в дерево ранее. Если же во время поиска был достигнут пустой лист, то новый объект вставляется на его место. Псевдокод процедуры вставки показан в листинге 2.22.

Листинг 2.22

```

//=====
// Вставка нового объекта в дерево
// Параметры:
//   O – запись, которую необходимо вставить в дерево
//=====
ВСТАВКА(O)
    [1] // Подготовка переменных
        V = Корень дерева

```

```

V0 = новая вершина с объектом O
Child_NE(V0) = NULL
Child_NW(V0) = NULL
Child_SE(V0) = NULL
Child_SW(V0) = NULL
[2] // Проверка существования дерева
    Если V = NULL, то
        Корень дерева = V0
        Выход из процедуры, вершина добавлена
[3] // Проверка на совпадение вершин
    Если (K0(V) = K0(O)) И (K1(V) = K1(O)), то
        Выход из процедуры, вершина уже есть в дереве
[4] // Поиск пустого листа
    Если K0(O) ≥ K0(V), то
        Если K1(O) ≥ K1(V), то
            Если Child_NE(V) = NULL, то
                Child_NE(V) = V0
                Выйти из процедуры
            Иначе
                V = Child_NE(V)
        Иначе
            Если Child_SE(V) = NULL, то
                Child_SE(V) = V0
                Выйти из процедуры
            Иначе
                V = Child_SE(V)
        Иначе
            Если K1(O) ≥ K1(V), то
                Если Child_NW(V) = NULL, то
                    Child_NW(V) = V0
                    Выйти из процедуры
                Иначе
                    V = Child_NW(V)
            Иначе
                Если Child_SW(V) = NULL, то
                    Child_SW(V) = V0
                    Выйти из процедуры
                Иначе
                    V = Child_SW(V)
    Перейти к шагу 3
Конец ВСТАВКА

```

Как видно, данная процедура строит несбалансированное дерево. Возможен случай, когда построенное дерево превратится в список, и поиск по нему будет не быстрее простого последовательного перебора. Однако Р. А. Финкель и Д. Л. Бентли в своей работе показали, что для случайного дерева, состоящего из N элементов, скорость поиска будет величиной, близкой к $O(\log_4 N)$. При этом сложность построения данного дерева приблизительно равна $(N * \log_4 N)$ [35].

Позднее Л. Деврой нашел ожидаемую высоту случайного дерева. Для точечного *Quad*-дерева она оказалась равна $(2c/d \cdot \ln N)$, где c – некоторая константа, равная 4.31107, а d – число измерений (ключей). Из этой формулы можно сделать ряд выводов, характеризующих как положительно, так и отрицательно *Quad*-дерево. Так, в формуле очень хорошо видно, что с ростом числа измерений высота дерева падает всего лишь линейно, в то время как накладные расходы на память растут по степенной функции. Именно поэтому *Quad*-дерево показывает небольшую производительность в пространствах большой размерности и практически никогда для таких размерностей не применяется.

Если большая часть объектов, которые будут находиться в дереве, известна заранее, то можно попробовать создать более сбалансированную структуру с лучшими показателями, чем случайное *Quad*-дерево. Для этого был предложен ряд подходов, отличающихся различными предобработками последовательности объектов и выбором очередного объекта для вставки в дерево.

Самый первый вариант такого алгоритма был предложен самими же создателями структуры. Полученную структуру они назвали *оптимальное точечное Quad-дерево*. Это дерево имело одно важное свойство – в каждом узле дерева не было ни одного поддеревя, которое вмещало бы в себя более половины всех дочерних узлов. Таким образом, максимальная высота такого дерева не может превосходить $\log_2 N$, т. е. оно не хуже обычного бинарного *K-D*-дерева.

Суть предложенных изменений заключалась в следующем. В каждом квадранте происходила сортировка всех объектов лексикографически (сначала по одному ключу, затем по другому), и в качестве точки-претендента на корневую вершину данного квадранта выбирался средний объект отсортированной последовательности. После этого все объекты распределялись по соответствующим потомкам данной вершины и процедура повторялась рекурсивно для каждого из поддеревьев. При таком построении гарантировано, что в самом худшем случае дерево будет иметь логарифмическую высоту по основанию 2, однако это не гарантирует получение абсолютно сбалансированного дерева (логарифмическая высота по основанию 4).

Одним из основных минусов оптимизированного точечного дерева является статичность. Дерево получается приемлемой высоты, но при динамических изменениях (процедурах вставки и удаления) высота дерева может меняться произвольно. Позднее были предложены алгоритмы, позволяющие добиться таких же показателей сбалансированности и при динамической вставке объектов.

Алгоритм удаления вершины Quad-дерева

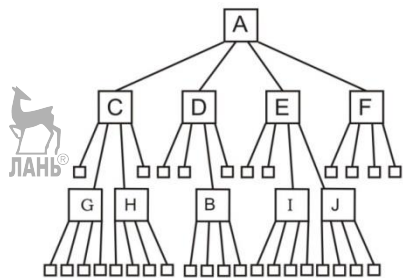
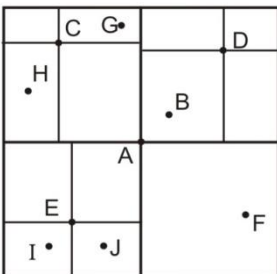
Удаление объектов *Quad*-дерева является не самой простой операцией. Так как удаляемый объект может находиться не только в

листовой вершине, откуда его легко можно удалить, но и в любой другой внутренней вершине дерева, то его удаление может потребовать перестроения того поддерева, которое начинается с данного объекта. При этом удаление корня дерева может затронуть все дерево целиком, в зависимости от применяемого алгоритма.

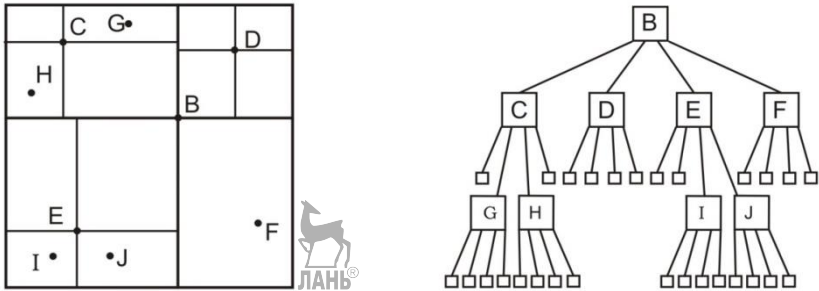
Существует несколько алгоритмов удаления вершин *Quad*-дерева. Простейший из них был предложен создателями структуры [21]. Он заключается в удалении найденной вершины и повторной вставке всех его потомков. Такой алгоритм может быть очень эффективным, если удаляются листовые вершины или узлы, близкие к ним. Однако чем ближе удаляемая вершина находится к корню, тем большую часть дерева придется перестроить. А при удалении корневой вершины этот алгоритм потребует перестроить все дерево целиком, что является весьма трудоемкой операцией.

Идеальным вариантом была бы замена некоторой удаляемой вершины A , имеющей координаты (K_{A0}, K_{A1}) , другой вершиной B с координатами (K_{B0}, K_{B1}) , такой, чтобы между линиями $x = K_{A0}$ и линией $x = K_{B0}$, так же как и между линиями $y = K_{A1}$ и $y = K_{B1}$, не было бы ни одного объекта (рис. 2.15(а)). В этом случае вершина A удаляется, а на ее место ставится вершина B (рис. 2.15(б)). Такая операция требует минимум действий и является наименее трудоемкой.

К сожалению, вершина, которой можно было бы заменить удаляемый узел, не всегда может быть найдена. Так, на рис. 2.15(б) при удалении вершины B нельзя найти такую вершину, которой можно было бы заменить B с сохранением остальной структуры дерева. Поэтому в общем случае нельзя обойтись без удаления и повторной вставки части вершин дерева. Однако желательно выбрать такую вершину-претендент, которая позволила бы минимизировать подобные операции.



а)



б)


Рис. 2.15. Удаление вершины A и замена ее вершиной B: а – структура до удаления вершины; б – структура после удаления вершины

Рассмотрим один из подобных алгоритмов. Удаление любой вершины можно рассматривать как удаление корня некоторого поддерева. Причем нетрудно показать, что процесс удаления не затронет остальную часть дерева, кроме той, которая начинается с данной вершины.

Поэтому в листинге 2.24 предполагается, что происходит удаление корневой вершины некоторого поддерева V , переданной в процедуру в качестве параметра.

Листинг 2.24

```
//=====
// Удаление объекта из Quad-дерева
// Параметры:
//   V – вершина, которую необходимо удалить
//=====
УДАЛЕНИЕ (V)
[1] // Инициализация переменных
    V1 = Child_NW(V)
    V2 = Child_NE(V)
    V3 = Child_SW(V)
    V4 = Child_SE(V)
    VNEW = NULL
[2] // Удалить вершину V, если она лист дерева
    Если V1 = NULL И V2 = NULL И V3 = NULL И
        V4 = NULL, то
        Заменить ссылку Parent(V) -> V на NULL
        Удалить V
        Завершить работу процедуры
[3] // Поиск четырех претендентов для замены V
    [3.1] Если (V1 ≠ NULL) И (Child_SE(V1) ≠ NULL), то
```



```

V1 = Child_SE(V1)
Перейти к шагу 3.1
[3.2] Если (V2 ≠ NULL) И (Child_SW(V2) ≠ NULL), то
      V2 = Child_SW(V2)
      Перейти к шагу 3.2
[3.3] Если (V3 ≠ NULL) И (Child_NE(V3) ≠ NULL), то
      V3 = Child_NE(V3)
      Перейти к шагу 3.3
[3.4] Если (V4 ≠ NULL) И (Child_NW(V4) ≠ NULL), то
      V4 = Child_NW(V4)
      Перейти к шагу 3.4
[4] // Выбор лучшего претендента
[4.1] Если Crit1(V, V1, V3, V2), то
      VNEW = V1
      Если Crit1(V, V4, V2, V3) И
        Crit2(V, V4, VNEW), то
        VNEW = V4
      Если VNEW ≠ NULL, то
        Перейти к шагу 5
[4.2] Если Crit1(V, V2, V4, V1), то
      VNEW = V2
      Если Crit1(V, V3, V1, V4) И
        Crit2(V, V3, VNEW), то
        VNEW = V3
      Если VNEW ≠ NULL, то
        Перейти к шагу 5
[4.3] Если Crit2(V, V1, V2), то
      VNEW = V1
      Иначе
        VNEW = V2
      Если Crit2(V, V3, VNEW), то
        VNEW = V3
      Если Crit2(V, V4, VNEW), то
        VNEW = V4
[5] // Замена вершины V на VNEW и повторная вставка
    // некоторых вершин
    K0min = min { K0(V), K0(VNEW) }
    K0max = max { K0(V), K0(VNEW) }
    K1min = min { K1(V), K1(VNEW) }
    K1max = max { K1(V), K1(VNEW) }
    K0(V) = K0(VNEW)
    K1(V) = K1(VNEW)
    ПРОВЕРКА_ДЕРЕВА(Child_NW(V), K0min, K0max, K1min, K1max)
    ПРОВЕРКА_ДЕРЕВА(Child_NE(V), K0min, K0max, K1min, K1max)
    ПРОВЕРКА_ДЕРЕВА(Child_SW(V), K0min, K0max, K1min, K1max)
    ПРОВЕРКА_ДЕРЕВА(Child_SE(V), K0min, K0max, K1min, K1max)

```

Конец УДАЛЕНИЕ

```

//=====
// Повторная вставка тех вершин дерева, которые попали в
// диапазон между  $K_0^{\min}$  и  $K_0^{\max}$  или  $K_1^{\min}$  и  $K_1^{\max}$ 
// Параметры:
//   V – текущая просматриваемая вершина
//    $K_0^{\min}$ ,  $K_0^{\max}$  – диапазон по первому ключу
//    $K_1^{\min}$ ,  $K_1^{\max}$  – диапазон по второму ключу
//=====
ПРОВЕРКА_ДЕРЕВА(V,  $K_0^{\min}$ ,  $K_0^{\max}$ ,  $K_1^{\min}$ ,  $K_1^{\max}$ )
[1] // Проверка самой вершины
    Если ( $K_0(V) \geq K_0^{\min}$  И  $K_0(V) \leq K_0^{\max}$ ) ИЛИ
        ( $K_1(V) \geq K_1^{\min}$  И  $K_1(V) \leq K_1^{\max}$ ), то
        Для всех вершин V' поддерева V
            Удалить V' из дерева
        ВСТАВКА(V')
    Выйти из процедуры
[2] // Проверка всех поддеревьев данной вершины
    Если ( $K_0(V) \geq K_0^{\min}$ ) ИЛИ ( $K_1(V) \leq K_1^{\max}$ ), то
        ПРОВЕРКА_ДЕРЕВА(Child_NW(V),  $K_0^{\min}$ ,  $K_0^{\max}$ ,  $K_1^{\min}$ ,  $K_1^{\max}$ )
    Если ( $K_0(V) \leq K_0^{\max}$ ) ИЛИ ( $K_1(V) \leq K_1^{\max}$ ), то
        ПРОВЕРКА_ДЕРЕВА(Child_NE(V),  $K_0^{\min}$ ,  $K_0^{\max}$ ,  $K_1^{\min}$ ,  $K_1^{\max}$ )
    Если ( $K_0(V) \geq K_0^{\min}$ ) ИЛИ ( $K_1(V) \geq K_1^{\min}$ ), то
        ПРОВЕРКА_ДЕРЕВА(Child_SW(V),  $K_0^{\min}$ ,  $K_0^{\max}$ ,  $K_1^{\min}$ ,  $K_1^{\max}$ )
    Если ( $K_0(V) \leq K_0^{\max}$ ) ИЛИ ( $K_1(V) \geq K_1^{\min}$ ), то
        ПРОВЕРКА_ДЕРЕВА(Child_SE(V),  $K_0^{\min}$ ,  $K_0^{\max}$ ,  $K_1^{\min}$ ,  $K_1^{\max}$ )
Конец ПРОВЕРКА_ДЕРЕВА

```

Для начала алгоритм проверяет, не является ли удаляемый узел листовой вершиной. Если все его потомки (V_1 , V_2 , V_3 и V_4) – пустые вершины, то узел V можно удалить из дерева и завершить процедуру. Дальнейшие преобразования в этом случае не нужны (шаг алгоритма 2).

Если же имеем дело с внутренней вершиной дерева, то необходимо не просто удалить ее, а подобрать для нее замену, т. е. такую вершину из данного поддерева, при замене на которую необходимо будет повторно вставить небольшое число объектов. Для выполнения данной операции производится поиск четырех претендентов на подобную замену (шаг 3). Для этого в каждом из дочерних узлов вершины V выбирается такая вершина V_i , которая находится в ближайшем квадранте. Так, в правом верхнем квадранте (шаг 3.2) происходит циклический переход в нем в юго-западном направлении до тех пор, пока не будет найдена конечная вершина (вершина, у которой нет потомков в этом направлении). На рис. 2.16 показан такой поиск для некоторого дерева. Сначала происходит переход к дочернему узлу V_2 . Так как этот узел не является конечным, поиск продолжается и переходит к дочернему узлу юго-западного направления – вершине V_2'' . Однако и данная вершина не является конечной, поэтому происходит еще один переход – к вершине

V_2''' . В результате всех операций, в качестве претендента на замену возвращается вершина V_2''' .

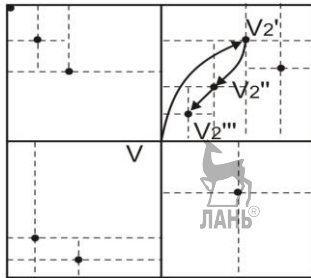


Рис. 2.16. Выбор претендента на замену в северо-восточном квадранте

Подобный алгоритм применяется для всех четырех квадрантов вершины V . В результате после выполнения шага алгоритма 3 получаем четыре претендента на замену данной вершины.

На четвертом шаге алгоритма выбирается из четырех претендентов наилучший. Для этого вводятся два критерия. Согласно первому критерию, некоторая вершина V_i является хорошим претендентом в том случае, если она находится по обоим своим ключам ближе к удаляемой вершине V , чем ее соседи. Так, на рис. 2.17(а) показан случай, в котором вершина V_2 является единственной, прошедшей по первому критерию. Вершину V_2 в этом случае можно считать наилучшим претендентом на замену удаляемой вершины V и перейти к следующему шагу алгоритма.

В листинге 2.24 первый критерий проверяется некоторой конструкцией $Crit1(V, V_b, V', V'')$. В данной записи первым параметром выступает удаляемая вершина, вторым – проверяемый претендент, третьим – сосед по горизонтальной оси, четвертым – сосед по вертикальной оси. Данная конструкция является истинной тогда и только тогда, когда истинным является следующее условие:

$$(V_i \neq NULL) \wedge ((V' = NULL) \vee (|K_0(V_i) - K_0(V)| < |K_0(V') - K_0(V)|)) \wedge ((V'' = NULL) \vee (|K_1(V_i) - K_1(V)| < |K_1(V'') - K_1(V)|)).$$

Однако не всегда первый критерий может однозначно определить лучшего кандидата. Например, на рис. 2.17(б) две вершины удовлетворяют этому критерию – вершина V_1 и V_4 , а на рис. 2.17(в) ни одну вершину нельзя назвать лучшей.

Если по первому критерию не удалось определить лучшего претендента (таких претендентов оказалось два или данному критерию не удовлетворяет ни одна вершина), используют второй критерий. На рис. 2.17(б) по второму критерию выбирают лучшего претендента из вершин V_1 и V_4 , а на рис. 2.17(в) – из всех четырех претендентов.

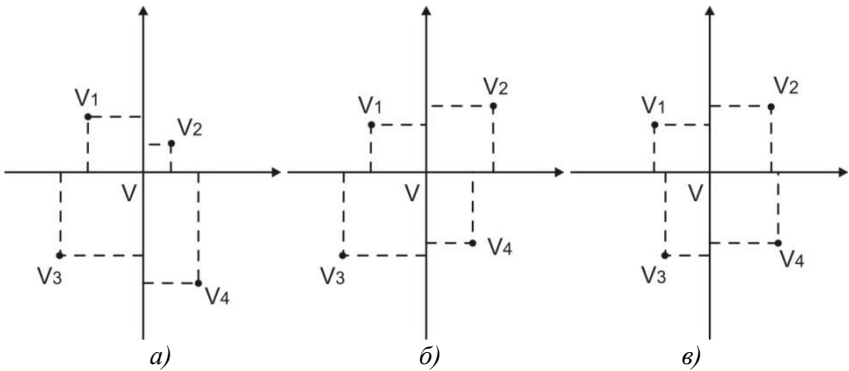


Рис. 2.17. Критерий 1: а – лучшая вершина V_2 ; б – лучшие вершины V_1 и V_4 ; в – лучших вершин нет

В качестве второго критерия предлагается использовать манхэттенское расстояние, т. е. из двух претендентов выбирают того, у которого сумма разностей координат с удаляемой вершиной является минимальной. В листинге 2.24 второй критерий проверялся с помощью предиката $Crit2(V, V', V'')$. Этот предикат должен быть истинным тогда и только тогда, когда вершина V' превосходит вершину V'' по второму критерию. Само же условие можно записать в следующей форме:

$$(V' \neq NULL) \wedge ((V'' = NULL) \vee (|K_0(V') - K_0(V)| + |K_1(V') - K_1(V)|) < (|K_0(V'') - K_0(V)| + |K_1(V'') - K_1(V)|)).$$

Данный алгоритм не является идеальным с точки зрения выбора претендента на замену удаляемой вершины. В ряде случаев он будет выбирать не самую лучшую вершину (однако почти всегда достаточно хорошую). Поиск наилучшего претендента может потребовать гораздо большего числа операций и быть более трудоемким, чем простая повторная вставка всех вершин поддерева. Данный же алгоритм является компромиссом между качеством выбора вершины претендента и скоростью работы процедуры в целом.

После выбора вершины V_i , которой можно заменить удаляемую вершину V , происходит данная замена и повторная вставка тех объектов дерева, которые попали в область сдвига. Эта операция выполняется на шаге 5 с помощью процедуры ПРОВЕРКА_ДЕРЕВА.

Область сдвига – это тот участок пространства, который находится по каждой из осей между старым и новым положением вершины V . На рис. 2.18 показана область сдвига с помощью штриховки. Все попавшие в нее вершины должны быть удалены и вставлены в дерево заново, так как после сдвига текущей вершины они стали занимать в дереве неправильные позиции.

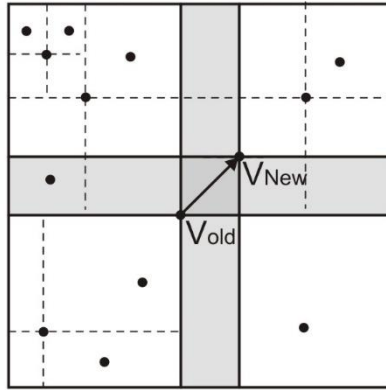


Рис. 2.18. Область сдвига

Если в область сдвига попадает не листовая вершина, то необходимо произвести удаление и повторную вставку не только этой вершины, но и всех ее дочерних вершин.

Следует также отметить, что при использовании процедуры, представленной в листинге 2.24, можно специально не удалять старую вершину V_i и повторно вставлять все ее дочерние объекты, так как такое удаление и повторная вставка произойдут автоматически в процедуре ПРОВЕРКА_ДЕРЕВА.

Разновидности Quad-деревьев

С момента появления Quad-дерева было опубликовано множество его вариантов, отличающихся как принципами хранения объектов и алгоритмами их обработки, так и сферой применения. Появились модернизации алгоритма, предназначенные для компьютерной графики, ГИС- и CAD-систем, многоключевого доступа в СУБД и других областей. Рассмотрим лишь некоторые из них, имеющие хоть какое-то отношение к точечной обработке.

Quad-деревья областей (Region Quad-tree). Эта структура была предложена Г. Самитом в 1984 году. Данный вид деревьев, строго говоря, не является точечным. С помощью него сохраняется информация о некотором геометрическом объекте сложной структуры с заранее заданной точностью. Однако на основе этого дерева был разработан ряд точечных модификаций, описанных далее.

В качестве примера рассмотрим некоторый сложный объект, показанный на рис. 2.19(а). В практических задачах это может быть территория страны на карте мира, информация о протяженном объекте ГИС- или CAD/CAM-системы.

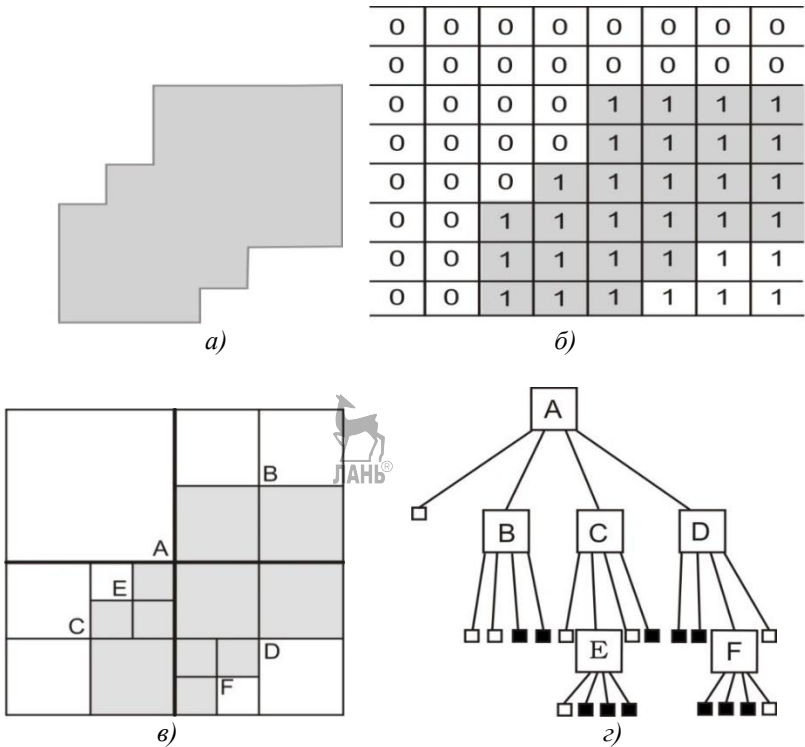


Рис. 2.19. Quad-дерево областей: а – исходный объект; б – матрица объекта; в – структура дерева; г – дерево областей

Для того чтобы данный объект представить в виде дерева, его необходимо каким-либо образом трансформировать. Для этого он заключается в описывающий его прямоугольник минимального размера. После этого прямоугольник разбивается регулярной решеткой (решеткой с одинаковым размером ячеек) до тех пор, пока не будет достигнута необходимая точность представления объекта. Точность представления контуров объекта как раз и зависит от размеров ячейки решетки.

Для представления объекта решетка заполняется специальным образом. Те ячейки решетки, которые содержат данный объект, помечаются 1, а которые не содержат – 0 (рис. 2.19(б)). В результате получаем матрицу исходного объекта. По этой матрице и строится Quad-дерево областей. Для этого все пространство ограничивающего прямоугольника сопоставляют с корневой вершиной дерева. Затем его разбивают на четыре равные части. Эти части соответствуют потомкам корневой вершины Quad-дерева областей.

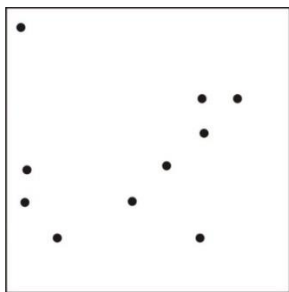
Для каждого потомка и соответствующего ему квадранта можно выделить три возможные ситуации.

1. В соответствующий квадрант не попала ни одна часть объекта (в нем нет ячеек решетки, помеченных 1). При этом потомок помечается пустым, и дальнейшие операции над ним не проводят. В рассматриваемом примере это, например, северо-западный потомок корневой вершины.

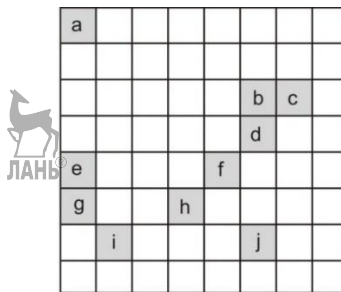
2. В квадранте содержатся только части исходного объекта (в нем нет ячеек решетки, помеченных 0). При этом подобная вершина помечается специальным флагом принадлежности объекту и дальнейшие операции над этим квадрантом также не проводятся. В примере на рис. 2.19 подобными вершинами, например, являются оба южных потомка вершины *B*.

3. В квадранте присутствуют как пустые ячейки, так и ячейки, принадлежащие объекту (в нем есть ячейки, помеченные как 1, так и 0). При этом необходимо произвести рекурсивное разбиение квадранта на четыре части заново и процедуру проверки повторить для каждого из полученных потомков.

MX Quad-дерево (MX Quad-tree). Первой попыткой адаптировать алгоритмы *Quad*-дерева областей для точечных данных явилось *MX Quad*-дерево. Оно применяется для дискретных данных. Если ключи некоторого распределения имеют дискретный характер (изменяются с некоторым шагом), то можно разбить пространство решеткой с данным шагом и ячейку, соответствующую индексируемым объектам, пометить 1, а все остальные – 0. При этом мы получаем решетку, аналогичную той, что представлена на рис. 2.19(б). Дальнейшее представление пространства в виде *Quad*-дерева ничем не отличается от описанного ранее для *Quad*-дерева областей. Пример *MX Quad*-дерева показан на рис. 2.20.



а)



б)

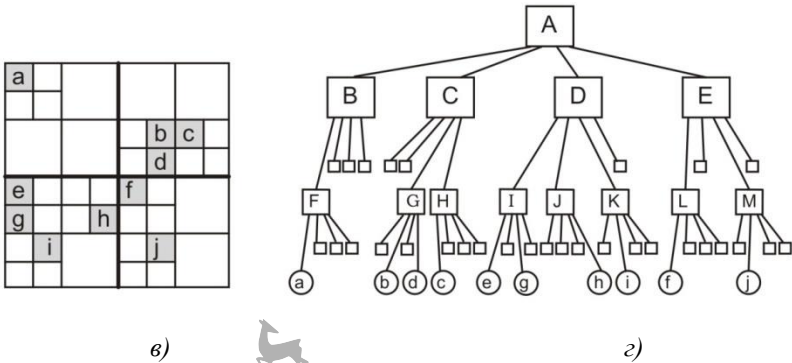


Рис. 2.20. MX Quad-дерево: а – размещение точечных объектов; б – матрица объектов; в – структура дерева; з – дерево областей

Одним из недостатков данной структуры является тот факт, что данная структура может применяться только для дискретных данных. Причем если дискретный интервал изменения ключей объекта очень маленький, то глубина дерева будет очень большой, даже при малом числе объектов в нем.

Строго говоря, глубина дерева зависит не от количества объектов в нем, а от интервала дискретизации, что естественно является недостатком.

PR Quad-дерево (PR Quad-tree). PR Quad-дерево является еще одним преобразованием алгоритмов Quad-дерева областей для точечных данных. Однако в отличие от MX Quad-деревьев в нем нет ограничения на дискретность данных и зависимости глубины дерева от интервала дискретизации.

Основная идея данного дерева также заключается в рекурсивном делении пространства на квадранты равного размера в каждом внутреннем узле. Однако, в отличие от MX Quad-дерева, рекурсивное деление прекращается не по достижению предела, равного интервалу дискретизации, а в том случае, если в соответствующем квадранте находится не более одного объекта (рис. 2.21).

В PR Quad-дереве все данные хранятся в листовых узлах, а внутренние узлы содержат только координаты деления и указатели на потомков. При реализации координаты можно опускать, так как точка деления пространства находится точно по середине квадранта и ее всегда можно вычислить.

Описанные структуры имеют как положительные, так и отрицательные стороны по сравнению с обычным точечным Quad-деревом. Одной из сильных сторон данных структур является упрощенная процедура удаления объекта дерева.

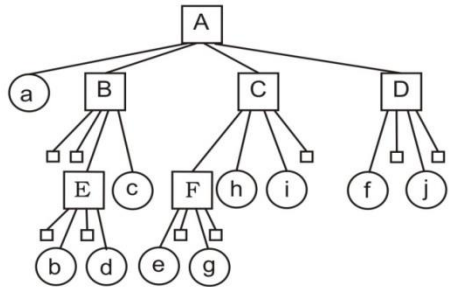
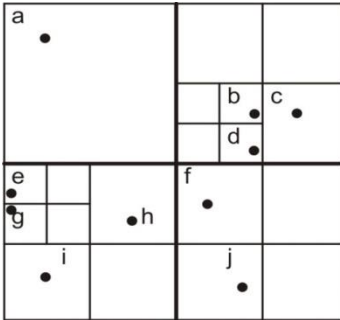


Рис. 2.21. PR Quad-дерево

Так как все объекты хранятся только в листовых узлах дерева, то ситуации, при которой после удаления объекта необходимо перестроить большую часть дерева, в нем не возникает. Объект просто удаляется из листа, и при необходимости удаляются пустые родительские вершины. Таким образом, процедура удаления упрощается до минимально возможной.

Однако у всех перечисленных в данном разделе структур есть один недостаток. При неравномерном распределении объектов в пространстве могут появляться ветви дерева очень большой длины, хотя общее число объектов в дереве будет небольшим. В *MX Quad*-дереве такие длинные ветви появляются в любом случае, а в *PR Quad*-дереве – только для близкорасположенных объектов.

Псевдо Quad-дерево (pseudo Quad-tree). В данном варианте дерева нет жесткого регулярного деления пространства решеткой. Этот вариант больше похож на оригинальное точечное *Quad*-дерево, чем на дерево областей. Как и в оригинальном варианте, в каждой внутренней вершине такого дерева находится точка, которая делит пространство на четыре квадранта. Однако в отличие от обычного дерева, точки внутренних вершин не соответствуют реальным объектам, а выбираются произвольно в соответствии с некоторым алгоритмом. Это позволяет избавиться от ряда недостатков, присущих как оригинальной структуре, так и различным модификациям *Quad*-деревя областей. Пример псевдо *Quad*-деревя приведен на рис. 2.22.

В отличие от оригинального точечного *Quad*-деревя, в рассматриваемом варианте объекты не разбросаны по всей структуре. Это позволяет упростить процедуру удаления объектов, как и в *PR Quad*-дереве. Однако, в отличие от *PR Quad*-деревя, в данной структуре нет требования к равенству размеров квадрантов любой внутренней вершины дерева. Это дает преимущества для неравномерных

распределений. При неудачном близком распределении объектов в пространстве в дереве не появляются очень длинные ветви.

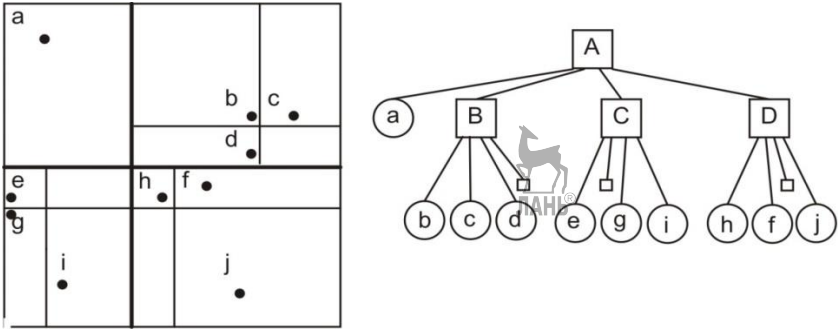


Рис. 2.22. Псевдо Quad-дерево

Данный вариант Quad-дерева был всесторонне исследован его изобретателями. В результате были найдены интересные свойства. Одно из них касается глубины дерева.

При любом распределении объектов в пространстве для псевдо Quad-дерева можно найти такую точку, которая разобьет это пространство на 2^d частей так, чтобы в каждой части было не более $(N / (d + 1))$ объектов, где N – общее число объектов в пространстве, d – число измерений (ключей). Отсюда можно найти максимальную глубину такого дерева. Она составит $(\log_{d+1} N)$. А так как максимальная глубина дерева является показателем сложности процедуры поиска по дереву, то псевдо Quad-дерево можно считать деревом со сложностью поиска, равным $O(\log_{d+1} N)$.

В случае двумерного пространства поиска, приведенные размышления означают, что псевдо Quad-дерево позволяет разбивать пространство так, что в каждом потомке любой вершины дерева находится не более трети всех объектов, а скорость поиска по такому дереву равна $O(\log_3 N)$. Ранее было показано, что скорость поиска в оригинальном точечном Quad-дереве равна $O(\log_2 N)$.

2.3. Многомерное хеширование

В восьмидесятых годах прошлого столетия различными исследователями было предложено адаптировать одномерное хеширование к многомерному случаю (Д. Нивергельт, Г. Хинтерберг, К. Севчик) [68]. Разработанный ими метод получил название *файлы-решетки*. Идея была настолько оригинальной, что ее подхватили

другие исследователи, и вскоре появилось большое количество различных вариантов реализации. В настоящее время разработано более двух десятков алгоритмов хранения многомерных данных на основе принципов хеширования информации.

Целью всех этих методов является разработка способов размещения информации о многомерных объектах в файле подобно тому, как они хранятся в самом этом пространстве. Для вычисления положения записей в файле, точно так же, как и для одномерного хеширования, применяется некоторая функция. Это позволяет искать записи с меньшим числом обращений к жесткому диску (у большинства схем данный показатель находится в диапазоне от 1 до 2).

Далее в этой главе будут описаны некоторые из алгоритмов многомерного хеширования, применяемых для точечных данных (*файлы-решетки, EXCELL, PLOP, MOLHPE*).

Все разнообразие методов многомерного хеширования можно разделить на ряд групп в соответствии с некоторыми принципами и критериями. Чаще всего во главу такого деления помещают критерий наличия дополнительного индекса, называемого каталогом. Согласно этому показателю, среди структур многомерного хеширования можно выделить те, которые для своей работы требуют построения некоторой дополнительной структуры, которая в дальнейшем используется в функции хеширования, и те, которые такого индекса не требуют.

Хеш-функция алгоритмов с каталогом представляет собой комбинацию математических расчетов и некоторой служебной информации, хранящейся в индексе (это могут быть шкалы деления пространства на ячейки или массив соответствия частей пространства некоторым блокам внешней памяти). Сам по себе индекс может достигать огромных размеров, что приводит к необходимости размещать его также во внешней памяти. Это накладывает определенный отпечаток на процедуру поиска. При определении местоположения объекта по его ключам сначала появляется необходимость обращения к соответствующей части каталога, а затем уже – непосредственно к внешнему блоку. В результате получается два обращения к внешней памяти для процедур поиска, что все равно является гораздо более быстрой процедурой по сравнению с иерархическими методами доступа. К данному классу алгоритмов относятся файл-решетка, *EXCELL, PLOP*.

Вторая группа алгоритмов позволяет обойтись хеш-функцией без каталога. При этом на вход некоторой математической функции подаются все ключи объекта поиска, и эта функция с помощью специальных математических методов и обработок вычисляет блок внешней памяти, в котором должен находиться данный объект. В результате получается, что в идеальном случае запись будет найдена за одно обращение к внешней памяти. К этому классу относятся многомерное линейное хеширование и *MOLHPE*.

Однако все методы хеширования без каталога подвержены одному существенному недостатку – любая хеш-функция рано или поздно приведет к переполнению внешнего блока, т. е. к коллизии. В методах с каталогом коллизии разрешаются с помощью видоизменения каталога (а следовательно, и самой хеш-функции) таким образом, чтобы в результате получить большее число внешних блоков, но без переполнения. При хешировании без каталога подобный подход не всегда возможен. Поэтому в подобных алгоритмах чаще всего переполненные блоки, соответствующие какому-то одному значению хеш-функции, разбиваются и объединяются в цепочку (метод цепочек). В общем случае это приводит к необходимости просмотра более одного внешнего блока при поиске объектов, хотя современные методы стараются избежать появления длинных цепочек. Среднее число доступов для методов без каталога может варьироваться в разных пределах, но для большинства алгоритмов оно находится в пределах 2.

2.3.1. Файл-решетка

Файл-решетка (Grid File) впервые был описан Д. Нивергельтом, Г. Хинтербергом и К. Севчиком в 1981 году [68]. Это был первый метод многоключевого доступа, основанный на принципах хеширования данных. В дальнейшем появилось множество различных вариантов файлов-решеток, улучшающих те или иные показатели и характеристики. О некоторых из таких улучшений будет упомянуто в дальнейшем.

Существует несколько структур, принципы которых опираются на решеточное деление пространства. Самыми популярными из них являются файлы-решетки и *EXCELL*. Во многом эти структуры похожи между собой. Главное их свойство заключается в том, что все пространство поиска разбивается решеткой на некоторые части (ячейки). Объекты, попавшие в одну и ту же ячейку, сохраняются в определенном блоке внешней памяти. При поиске с помощью решетки происходит определение блока, в котором может находиться поисковый объект, и данный блок загружается в память. После этого все объекты блока проверяются на равенство заданному объекту. При этом и файл-решетка, и *EXCELL* допускают размещение нескольких ячеек решетки в одном внешнем блоке, но запрещают нескольким блокам принадлежать одной и той же ячейке решетки. Это связано с политикой переполнения блоков при коллизиях и методах решения данной проблемы.

Несмотря на близость файлов-решеток и *EXCELL*, у данных методов есть и отличия. *EXCELL* допускает только регулярное деление пространства, что приводит к некоторым упрощениям процедур поиска (особенно поиск по области). Но он влечет за собой быстрый рост каталога. В то же время файл-решетка не накладывает никаких ограничений на размер ячеек и регулярность деления, что позволяет данной структуре быть более эффективной, особенно при неравномерном

распределении данных. Отсутствие жестких правил формирования решетки приводит к необходимости сохранять еще и дополнительную информацию, называемую массивами линейного разбиения.

Структура файла-решетки

При разработке файла-решетки был поставлен ряд задач и требований к разрабатываемой структуре. Самыми важными, по мнению разработчиков, должны были быть следующие требования.

Принцип двух дисковых запросов. Запрос на точное совпадение по всем ключам должен вернуть запись или флаг ее отсутствия в индексе за два обращения к внешней памяти: первое обращение – к нужной части каталога, второе – к нужному блоку внешней памяти.

Эффективные диапазонные запросы. Структура данных должна по возможности сохранять порядок, определенный для каждого ключевого диапазона, т. е. желательно, чтобы записи, имеющие близкие значения в диапазоне какого-либо ключа, располагались в близких участках памяти на физическом носителе.

В процессе проектирования структуры обе эти задачи были выполнены. В результате получилась структура данных многоключевого доступа, которая эффективно выполняет большинство типов запросов поиска. Единственным недостатком такой структуры оказалось использование большого количества памяти для внутреннего каталога.

Для пояснения общих принципов файлов-решеток будем рассматривать двухмерный ее вариант. Распространение идей на пространстве больших размерностей не составляет особых трудностей, однако, как показала практика, при большом числе измерений (большем 10) размер внутреннего индекса становится неоправданно большим. Поэтому оригинальный вариант файлов-решеток используется только для пространств малых размерностей.

Введем ряд понятий и определений. Пусть у нас дано пространство объектов с двумя ключами. Обозначим область ключей как оси Ox и Oy . Для получения решетки на каждую ось накладывается интервал

$$X = [x_0, x_1, \dots, x_n],$$

$$Y = [y_0, y_1, \dots, y_m].$$

В результате получаем решеточное деление пространства, показанное на рис. 2.23(а). Суть разбиения пространства решеткой заключается в получении такого распределения, при котором в каждой ячейке решетки будет не более N объектов. При этом число N является параметром структуры и выбирается таким образом, чтобы N объектов помещались в одном блоке внешней памяти (например, сегменте жесткого диска). Обычно N находится в диапазоне от 10 до 1000, в зависимости от размера внешнего блока и конечного объекта.

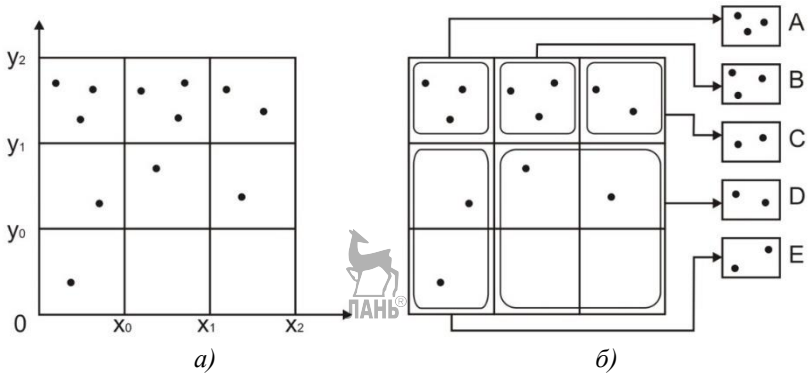


Рис. 2.23. Пример деления пространства решеткой с $N = 3$:
 а – решеточное деление; б – сопоставление с внешними блоками

Случаи, когда N мало (равно 1 или 2), для данной структуры плохо подходят, так как при этом есть вероятность нежелательных случайных эффектов, таких как, например, «парадокс дней рождения» (высокая возможность переполнения участка памяти при появлении всего двух записей). Однако примеры в данной главе для наглядности будут приводиться для малых N .

После получения решетки с каждой ячейкой сопоставляется один внешний блок, в котором и сохраняется вся информация об объектах данной ячейки, т. е. задача файлов-решеток заключается в установлении соответствия между ячейками пространства и внешними блоками, в которых и будут физически размещены объекты. Данная задача решается с помощью специального k -мерного массива (в нашем случае – двухмерного), который называется каталогом. Данный массив должен быть динамическим, иначе в результате проектирования получится полустатическая структура с жестко заданными пределами расширения.

Принцип двух дисковых запросов подразумевает, что все записи из одной ячейки решетки должны быть расположены в одном внешнем блоке памяти (иначе для поиска может понадобиться более двух обращений к внешней памяти). Однако очень неэффективно было бы обратное требование. Если каждому внешнему блоку соответствовала бы единственная ячейка решетки, то использование памяти было бы низким. Так, на рис. 2.23 четыре блока, расположенные в правом нижнем углу, являются редко заполненными или даже пустыми. Если для каждого из них выделять отдельные внешние блоки, то использование памяти в структуре будет практически нулевым. Гораздо более эффективно, с точки зрения памяти, для всех этих ячеек выделить один внешний блок и поместить все записи в него. Таким образом, получается объединение ячеек решетки в один внешний блок. Совокупность всех ячеек решетки,

сопоставленных с одним участком памяти (или, другими словами, пространство, занимаемое этими ячейками), называется *объединенной ячейкой*. Форма этих объединений влияет на скорость как минимум двух операций:

- диапазонный запрос;
- обновление после изменений в разделении решетки.

Учитывая акцент на эффективность обработки диапазонных запросов, а также решение основать систему на решеточном разделении пространства записей, у разработчиков не осталось других вариантов, кроме как ввести ограничение на представление объединенных ячеек в форме параллелепипеда или другого K -мерного прямоугольника. Назовем такое сопоставление ячеек решетки с участками памяти *блочным*.

На рис. 2.23 проиллюстрировано типичное блочное сопоставление. Каждый блок решетки указывает на свой участок памяти. Несколько ячеек решетки могут объединяться в прямоугольный блок и использовать один участок до тех пор, пока объединение этих ячеек формирует прямоугольную область в пространстве записей и общее число объектов в нем не превышает N .

Из всего изложенного следует, что самым главным в файле-решетке является каталог. Именно он позволяет разбивать пространство на ячейки, сопоставлять ячейки с внешними блоками, объединять ячейки друг с другом. По сути, каталог файла-решетки является хеш-функцией, потому что на его основе вычисляется блок внешней памяти, в котором находится тот или иной объект. Поэтому рассмотрим структуру каталога подробнее.

Каталог файла-решетки для K -мерного пространства состоит из следующих двух частей.

- K -одномерных массивов, называемых *линейными шкалами по осям*. Задача этих массивов заключается в хранении координат плоскостей, которые разбивают пространство решеткой. В описанном примере для двухмерного случая (рис. 2.23) шкалы представляют собой два массива $[x_0, x_1, x_2]$ и $[y_0, y_1, y_2]$.
- K -мерный массив сопоставления ячеек решетки и блоков внешней памяти, называемый *массивом решетки*. Для примера, представленного на рис. 2.23, данный массив выглядит следующим образом:

$$\begin{bmatrix} A & B & C \\ E & D & D \\ E & D & D \end{bmatrix}$$

В большинстве случаев линейные шкалы по осям являются небольшими массивами и хранятся в оперативной памяти, в то время как

массив решетки может достигать огромных размеров, поэтому его целесообразно размещать во внешней памяти.

Описанная структура не всегда может давать высокие показатели по эффективности. Чтобы структура обладала практической ценностью, алгоритмы, лежащие в ее основе, должны соответствовать следующим требованиям:

- выполнение принципа двух дисковых обращений для запросов по точному совпадению;
- эффективная обработка диапазонных запросов в больших линейно упорядоченных диапазонах;
- использование всего двух внешних блоков при расщеплении и слиянии ячеек решетки;
- поддержание корректной нижней границы заполнения системы на уровне средней заполненности участка данных.

Первые три пункта зависят от выбранных алгоритмов и будут описаны ниже, последний пункт об использовании памяти подтверждается многочисленными экспериментами, в том числе проведенными и самими разработчиками структуры [68].

Алгоритм поиска объекта

Файл-решетка применяется для индексирования многоключевых данных небольшой размерности. Рассмотрим два наиболее часто встречающихся запроса в приложениях, которые используют подобные данные – запрос по точному совпадению и запрос по диапазону ключей.

Запрос по точному совпадению. Это простейший вид поискового запроса. Его суть состоит в проверке, присутствует ли в индексе запись с определенным набором значений всех ключей. Для понимания задачи рассмотрим данный процесс на примере.

Пусть в нашем распоряжении имеется база данных подержанных автомобилей определенной марки. Наиболее часто необходимо обращаться к базе с поиском автомобилей по двум ключам – год выпуска и стоимость. Именно по этим ключам и была проиндексирована база с помощью алгоритмов файлов-решеток (рис. 2.24).

В результате индексирования данных, представленных на рисунке, был получен каталог файла-решетки со следующими элементами.

Линейные шкалы по осям:

$$X = [2000, 2005],$$

$$Y = [250, 500].$$

Массив решетки:



$$Grid = \begin{bmatrix} A & B & C \\ D & E & C \\ F & F & G \end{bmatrix}.$$

При составлении линейных шкал предполагалось отсутствие пределов по ключам (цена может быть сколь угодно большой и максимальный год выпуска может со временем меняться). В этом случае последние элементы по шкалам должны быть равны ∞ . Если решетка распространяется до бесконечности (как в рассматриваемом примере) и последние элементы шкал равны ∞ , то эти элементы можно опускать. Однако алгоритм поиска по шкале в этом случае нужно модифицировать, чтобы он учитывал этот факт.

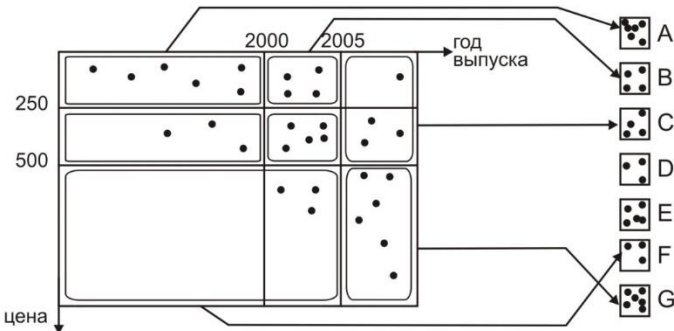


Рис. 2.24. Индекс файла-решетки для базы поддержанных автомобилей

Рассмотрим выполнение запроса на поиск объекта в этой базе, который был выпущен в 2009 году и стоит сейчас 400 тысяч руб. Первым шагом такого поиска является определение положения соответствующего объекта в массиве решетки. Для этого просматриваются линейные шкалы по осям для определения позиции объекта. По шкале X первый ключ объекта (год выпуска) находится после второго элемента, поэтому индекс по этой оси равен $i = 2$. По шкале Y (стоимость автомобиля) ключ располагается между первым и вторым элементом, поэтому индекс равен $j = 1$. В общем случае определение индекса по шкале происходит по следующим правилам:

- если ключ объекта меньше первого элемента шкалы, то индекс равен 0;
- если ключ объекта больше последнего элемента шкалы, то индекс равен n , где n – число элементов в данной шкале;
- если ключ объекта находится между элементами с позициями i_1 и i_2 , то индекс равен позиции i_2 .

Так как линейные шкалы по осям обычно располагаются в оперативной памяти, то данная операция не требует обращения к внешней памяти и выполняется достаточно быстро. Учитывая тот факт, что линейные шкалы являются отсортированными массивами значений, для ускорения поиска по ним можно использовать вместо простого перебора процедуру бинарного поиска.

После получения индексов по шкалам происходит определение номера внешнего блока, в котором может находиться объект поиска. Для этого нужно просто получить значение ячейки массива $Grid[j][i]$. Массив решетки $Grid$ может быть достаточно большим, поэтому чаще всего его размещают во внешней памяти. Поэтому для определения номера внешнего блока необходимо сделать одно дополнительное обращение к внешней памяти. Если массив $Grid$ не может быть целиком загружен в оперативную память (он занимает объем больше размера одного внешнего блока), то необходимо просто загрузить ту его часть, в которой находится элемент $Grid[j][i]$. Это сделать достаточно просто, придерживаясь определенных правил при выгрузке массива во внешнюю память.

В нашем примере $Grid[1][2] = C$. Это означает, что объект с ключами $K_0 = 2009$, $K_1 = 400$ может находиться только в блоке внешней памяти с номером C . На заключительном шаге поиска остается только загрузить этот блок и проверить все объекты в нем на совпадение с ключами поиска. Это будет второе обращение к внешней памяти во время процедуры поиска.

Таким образом, правило двух дисковых запросов полностью выполняется. Для поиска объекта по точному совпадению ключей необходимо сделать ровно два обращения во внешнюю память: первое – к каталогу для определения номера внешнего блока, второе – непосредственно к внешнему блоку.

Описанная процедура поиска объекта по точному совпадению ключей представлена в листинге 2.25.

Листинг 2.25

```
//=====
// Поиск объекта O по полному совпадению
// Параметры:
//   O = (K0, K1) - объект поиска
//=====
ПОИСК_ОБЪЕКТА(O)
[1] // Поиск в линейной шкале X
    n = количество элементов в шкале X
    i = 0
    Пока i < n выполнять
        Если K0(O) < X[i], то
            Прервать цикл
```

```

    i++
[2] // Поиск в линейной шкале Y
    n = количество элементов в шкале Y
    j = 0
    Пока j < n выполнять
        Если  $K_1(O) < Y[j]$ , то
            Прервать цикл
        j++
[3] // Поиск в массиве решетки
    N = Grid[j][i]
[4] // Поиск объекта во внешнем блоке N
    В = N-й внешний блок
    Загрузить В в оперативную память
    Для всех объектов  $O'$  из блока В проверить
        Если  $K_0(O) = K_0(O')$  И  $K_1(O) = K_1(O')$ , то
            Завершить процедуру и вернуть ИСТИНУ
[5] // Поиск неудачный
    Вернуть ЛОЖЬ
Конец ПОИСК_ОБЪЕКТА

```

Запрос по области (диапазонный запрос). Более общий тип запроса, поиск в ограниченной области, предполагает поиск объектов, попавших в некоторую область. Файл-решетка проектировался таким образом, чтобы эффективно выполнять также и данный тип запросов.

Рассмотрим пример диапазонного запроса. Для этого вернемся к базе автомобилей, показанной на рис. 2.24. Допустим, необходимо найти все автомобили, выпущенные в период между 2003 и 2009 годами и стоимостью до 500 тысяч рублей. Если формализовать данный запрос, то он будет выглядеть следующим образом:

$$\forall O (2003 \leq K_0(O) \leq 2009 \wedge 0 \leq K_1(O) \leq 500).$$

Для выполнения этого запроса необходимо найти все внешние блоки, в которых могут содержаться подобные записи. Для этого по каждой из осей пространства определяются минимальный и максимальный индексы решетки, содержащие подобные записи. Правила определения индексов совпадают с теми, которые использовались в запросе по точному совпадению.

Для оси O_x ключ должен находиться в пределах от 2003 до 2009. Линейная шкала по этой оси равна $X = [2000, 2005]$. Первый предел (число 2003) находится между первым и вторым числом в шкале. Поэтому начальный индекс примет значение $i_{start} = 1$. Второй предел (число 2009) больше последнего элемента в шкале, поэтому конечный индекс равен $i_{end} = 2$. Аналогично определяются начальный и конечный индексы по второй оси. В нашем примере они примут значения $j_{start} = 0$ и $j_{end} = 1$.

После определения индексов по всем осям пространства выбираются номера внешних блоков, находящихся в массиве решетки

Grid в заданных диапазонах, т. е. для рассматриваемого примера будут выбраны номера внешних блоков *B*, *C* и *E*.

На последнем шаге подгружаются внешние блоки с соответствующими номерами из внешней памяти и все их записи проверяются на соответствие запросу поиска.

Пример процедуры поиска показан в листинге 2.26. В качестве параметров процедура принимает область поиска $Reg = [K_0^{\min}, K_0^{\max}, K_1^{\min}, K_1^{\max}]$ и пустое множество $Res = \{\}$, в которое она должна поместить все найденные объекты, удовлетворяющие условию.

Листинг 2.26

```
//=====
// Поиск объектов по области
// Параметры:
// Reg = [ $K_0^{\min}$ ,  $K_0^{\max}$ ,  $K_1^{\min}$ ,  $K_1^{\max}$ ] - область поиска
// Res = {} - множество-результат (в него помещаются
//           объекты, удовлетворяющие запросу)
//=====
ПОИСК_В_ОБЛАСТИ(Reg, Res)
[1] // Поиск в линейной шкале X
    n = количество элементов в шкале X
    i_start = 0
    i_end = n
    Для всех i от 0 до n выполнять
        Если  $Reg.K_0^{\min} > X[i]$ , то
            i_start = i + 1
        Если  $Reg.K_0^{\max} < X[i]$ , то
            i_end = i
    Прервать цикл
[2] // Поиск в линейной шкале Y
    n = количество элементов в шкале Y
    j_start = 0
    j_end = n
    Для всех j от 0 до n выполнять
        Если  $Reg.K_1^{\min} > Y[j]$ , то
            j_start = j + 1
        Если  $Reg.K_1^{\max} < Y[j]$ , то
            j_end = j
    Прервать цикл
[3] // Поиск в массиве решетки
    Blocks = {} - пустое множество для номеров блоков
    Для всех i от i_start до i_end выполнять
        Для всех j от j_start до j_end выполнять
            Если Grid[j][i] не присутствует в Blocks
                Добавить Grid[j][i] в множество Blocks
[4] // Поиск объектов во внешних блоках
    Для всех блоков B из множества Blocks выполнить
        Загрузить B в оперативную память
```

Для всех объектов O' из блока B проверить
 Если $(K_0(O') \geq \text{Reg}.K_0^{\min})$ И
 $(K_0(O') \leq \text{Reg}.K_0^{\max})$ И
 $(K_1(O') \geq \text{Reg}.K_1^{\min})$ И
 $(K_1(O') \leq \text{Reg}.K_1^{\max})$, то
 Добавить O' в множество-результат Res
 Конец ПОИСК_В_ОБЛАСТИ



Процедура работает по тому алгоритму, который был описан выше. На первых двух шагах выполняется поиск начальных и конечных индексов по линейным шкалам, а на третьем – выбор номеров внешних блоков, в которых могут находиться объекты поиска. Однако стоит еще раз обратить внимание на ряд особенностей.

На третьем шаге при формировании списка внешних блоков-кандидатов (список *Blocks*) была добавлена проверка, не находится ли данный блок уже в списке. Без такой проверки некоторые блоки могут попасть в список несколько раз. Так, в описанном примере, если убрать данную проверку, то блок *C* дважды попадет в список. Это приведет к тому, что в дальнейшем этот блок дважды будет загружен в память и его объекты дважды будут проверяться. Все элементы этого блока, прошедшие проверку, также дважды попадут в множество результатов *Res*, что является нежелательным.

На четвертом шаге алгоритма выбранные внешние блоки загружаются в оперативную память и проверяются все их объекты на соответствие условию. Может показаться, что проверка на принадлежность объектов выбранных внешних блоков области поиска является лишней. Однако ее нельзя исключить по следующим двум причинам:

- ячейка решетки, из-за которой данный блок был выбран, может не целиком входить в область поиска (например, в приведенном примере блоки *C* и *E* принадлежат области поиска лишь частично);
- даже если некоторая ячейка массива решетки целиком принадлежит области поиска, внешний блок этой ячейки может выходить за пределы области (данная ситуация может возникать из-за возможности объединения ячеек в один блок).

Алгоритм добавления нового объекта

Динамичность любой структуры, прежде всего, зависит от поддержки этой структурой произвольной последовательности вставок и удалений объектов. Файл-решетка изначально разрабатывался как динамическая структура. Поэтому эти операции были предусмотрены сразу же в базовом варианте.

Первоначальное построение решетки происходит с помощью процедуры вставки. Объекты вставляются один за другим до тех пор,

пока не происходит переполнение какой-либо ячейки. После этого решетка модифицируется с помощью расщепления и вставки объектов можно продолжать далее.

Процедуру добавления и модификации решетки лучше всего рассмотреть на примере. Для иллюстрации всех действий выберем двухмерный файл-решетку с небольшой вместимостью одного внешнего (например, $N = 3$).

С самого начала имеется пустой файл-решетка, не содержащий ни одной записи. Обе линейные шкалы каталога этого файла являются пустыми массивами. Это означает, что пространство не пересекается ни одной прямой, и оно состоит всего из одной большой ячейки, соответствующей всему пространству поиска. Назовем эту ячейку A . Начальное состояние каталога файла-решетки будет следующим:

$$X = [], Y = [], \\ Grid = [A].$$

Если начинать вставлять объекты в такой файл, они будут размещаться в блоке A . Такой процесс будет продолжаться до тех пор, пока блок A будет содержать менее трех объектов. Добавление четвертого объекта вызовет переполнение блока A и единственной ячейки решетки. Переполнение любой ячейки приводит к процедуре деления. Эта процедура добавляет в решетку прямую, разбивающую переполненную ячейку на две части. Допустим, в качестве первой прямой разбиения была выбрана прямая, перпендикулярная оси X и проходящая через точку с координатой x_1 . В этом случае ячейка, связанная с блоком A , разбивается на две другие – A и B . Часть объектов остается в блоке A , другая часть – перемещается в новый блок B . В результате всех этих действий получится каталог, показанный на рис. 2.25(а). В линейной шкале X появился один элемент, соответствующий прямой $x = x_1$, а массив решетки стал состоять из двух ячеек – одна из них связана с блоком A , другая – с блоком B .

Продолжим добавлять объекты в получившийся файл. Допустим, было добавлено еще два объекта в решетку и оба эти объекта разместились в ячейку, связанную с блоком A . Это снова приведет к переполнению блока A и первой ячейки решетки. Поэтому опять будет запущена процедура деления, которая должна разбить переполненную ячейку на две части. На этот раз в качестве прямой разбиения выбирается прямая, перпендикулярная оси Y и проходящая через точку y_1 (рис. 2.25(б)).

Однако прямая $y = y_1$ разбивает на две части не только первую ячейку решетки, но и вторую. Объекты первой переполненной ячейки перераспределяются между старым блоком A и вновь созданным блоком C . Но во второй ячейке было мало объектов, и разбивать блок B на два новых смысла нет. Поэтому обе ячейки решетки, полученные в результате разбиения второй ячейки, объединяются в одну группу и

продолжают ссылаться на внешний блок B (рис. 2.25(б)). Такое поведение позволяет в процедуре деления ячеек не затрагивать те внешние блоки, которые в данный момент являются непереполненными, что удовлетворяет одному из принципов структуры, описанному ранее.

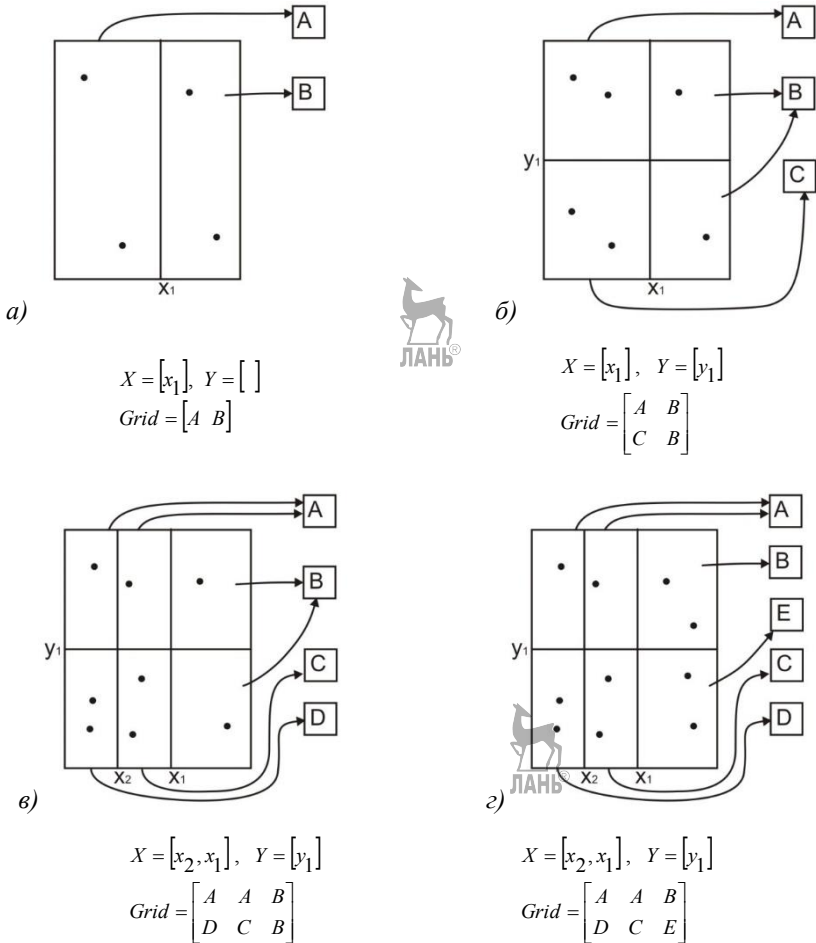


Рис. 2.25. Построение файла-решетки: а – деление решетки по оси X ; б – деление решетки по оси Y ; в – деление решетки по оси X ; г – расщепление внешних блоков

Аналогичная процедура будет проделана и при добавлении еще двух объектов в блок C (ячейка решетки с индексом $[1;0]$). В этом случае будет добавлена еще одна прямая, перпендикулярная оси X и проходящая через точку с координатой x_2 . Эта прямая разобьет не только ячейку $[1;0]$,

но и ячейку $[0;0]$ (рис. 2.25(в)). Однако, как и в прошлый раз, две верхние ячейки будут объединены в одну общую группу и продолжат ссылаться на один и тот же внешний блок A .

Выше были рассмотрены случаи, при которых переполнение происходило одновременно и во внешнем блоке, и в ячейке решетки. Однако возможен случай переполнения только внешнего блока. Например, очередное добавление двух объектов в ячейки $[0;2]$ и $[1;2]$ решетки, показанной на рис. 2.25(в). Такая ситуация приведет к переполнению внешнего блока B (он станет содержать четыре объекта, вместо трех разрешенных), однако ни одна из ячеек решетки переполнена не будет.

Подобная ситуация обрабатывается без расщепления решетки. Создается еще один внешний блок и часть ячеек из объединенной группы переносятся в новый внешний блок (рис. 2.25(г)). Такое перемещение сопряжено с одной сложностью. Важно разбить ячейки группы на две новые группы таким образом, чтобы обе группы образовывали области прямоугольной формы. В случае с двумя ячейками группы это получается естественным образом. Однако если изначально в группе было более трех ячеек, возможен вариант, при котором дальнейшая работа с каталогом решетки будет затруднена. Пример неправильного деления группы показан на рис. 2.26.

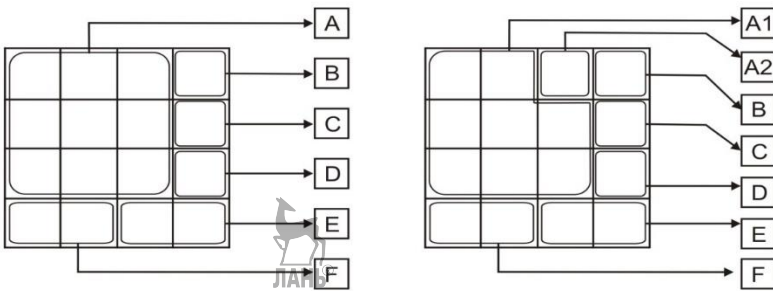


Рис. 2.26. Пример неправильного деления группы

На данном рисунке показан случай, когда имеется большая группа ячеек, соотношенная с одним внешним блоком A . При переполнении этого блока появляется необходимость разбить внешний блок A на блоки $A1$ и $A2$. При этом ячейки группы $[0;0] - [2;2]$ нужно перераспределить между этими блоками так, чтобы выполнились два условия:

- блоки $A1$ и $A2$ должны быть непереполненными (в идеале половина записей блока A должна перейти в $A1$, а вторая половина – в блок $A2$);
- группы ячеек, соотношенные с обоими блоками, образовывали области прямоугольной формы.

На примерах были рассмотрены основные принципы добавления объектов в файл-решетку. Формализуем теперь эти алгоритмы в виде процедур на условном языке программирования. Процедура вставки объекта в двухмерную решетку показана в листинге 2.27.

Листинг 2.27

```
//=====
// Вставка нового объекта
// Параметры:
//   О - объект, который необходимо добавить в решетку
//=====
ВСТАВКА(О)
[1] // Поиск нужной ячейки решетки (индексов i и j)
    n = количество элементов в шкале X
    i = 0
    Пока i < n выполнять
        Если  $K_0(O) < X[i]$ , то
            Прервать цикл
        i++
    n = количество элементов в шкале Y
    j = 0
    Пока j < n выполнять
        Если  $K_1(O) < Y[j]$ , то
            Прервать цикл
        j++
[2] // Проверка на присутствие объекта в решетке
    В = внешний блок с номером Grid[j][i]
    Загрузить В в оперативную память
    Для всех объектов  $O'$  из блока В проверить
        Если  $K_0(O) = K_0(O')$  И  $K_1(O) = K_1(O')$ , то
            Завершить процедуру
[3] // Добавление объекта
    Добавить О в блок В
    Если блок В переполнен, то
        РАСЩЕПЛЕНИЕ_БЛОКА(i, j)
Конец ВСТАВКА
```

Действия процедуры вставки объекта просты и интуитивно понятны. На первом шаге происходит определение ячейки решетки, в которой должен находиться вставляемый объект. Правила определения индексов полностью идентичны тем, которые использовались в процедуре поиска объекта.

На втором шаге процедуры необходимо проверить, не находится ли данный объект уже в решетке. Повторная вставка одного и того же объекта может привести к нежелательным последствиям. Если объекта O в решетке нет, то на заключительном третьем шаге алгоритма происходит вставка его во внешний блок, связанный с найденной

ячейкой решетки. Однако необходимо проверить число объектов в этом внешнем блоке. Если произошло переполнение блока, то необходимо произвести деление блока или решетки. Для этого вызывается процедура РАСЩЕПЛЕНИЕ_БЛОКА, описанная далее.

Расщепление внешних блоков и ячеек решетки

Вставка объектов в решетку рано или поздно приведет к переполнению ее элементов и необходимости процедуры расщепления. Как было показано ранее, существует два типа расщеплений. Первый из них заключается в расщеплении только внешних блоков. Он возможен, когда несколько ячеек объединены в одну группу и ссылаются на один и тот же внешний блок. При этом ячейки этой группы перераспределяются между двумя группами и внешний блок расщепляется на две части. Данный вид расщепления наиболее часто выполняется в реальных системах. В большинстве случаев процесс реакции на переполнение заканчивается именно им. В рассмотренном ранее примере данная ситуация показана в переходе от рис. 2.25(в) к рис. 2.25(г). Данное решение не приводит к значительным перестроениям решетки и обрабатывается относительно просто.

Второй вариант расщепления – расщепление самой решетки. Он выполняется, когда с переполненным блоком связана всего одна ячейка (первый вариант расщепления при этом невозможен). Появляется необходимость разбить определенный столбец или строку массива решетки на две части. Выбираются некоторое измерение, по которому будет произведено разбиение, и координата на этом измерении. В массиве решетки такое разбиение приводит к делению всех ячеек по этому измерению, однако внешние блоки делятся только для переполненной ячейки. Процедура деления показана в листинге 2.28.

Листинг 2.28

```
//=====
// Расщепление переполненного блока
// Параметры:
// (iB, jB) – индекс ячейки решетки, связанной
// с переполненным блоком
//=====
РАСЩЕПЛЕНИЕ_БЛОКА(iB, jB)
[1] // Поиск группы ячеек
    В = внешний блок с номером Grid[jB][iB]
    Если В не переполнен, то
        Завершить процедуру
    G = {istart, iend, jstart, jend} – группа ячеек,
        связанных с внешним блоком В
[2] // Расщепление
    Если группа G содержит более одной ячейки, то
        РАСЩЕПЛЕНИЕ_ГРУППЫ(G)
```

```

        Иначе
            РАСЩЕПЛЕНИЕ_РЕШЕТКИ(iВ, jВ)
        Перейти к шагу 1
    Конец РАСЩЕПЛЕНИЕ_БЛОКА

//=====
// Расщепление группы ячеек
// Параметры:
//   G = {istart, iend, jstart, jend} - группа ячеек, связанных
//   с переполненным блоком
//=====
РАСЩЕПЛЕНИЕ_ГРУППЫ(G)
    [1] // Расщепление блока
        В = внешний блок с номером Grid[jstart][istart]
        В' = новый внешний блок
    [2] // Выбор границы расщепления
        Если istart ≠ iend, то
            isplit = istart + (iend - istart) / 2
            jsplit = jstart
        Иначе
            isplit = istart
            jsplit = jstart + (jend - jstart) / 2
    [3] // Расщепление группы
        Для всех i от isplit до iend выполнить
            Для всех j от jsplit до jend выполнить
                Grid[i][j] = В'
    [4] // Перенос части объектов из В в В'
        Если isplit = 0, то
            K0min = 0
        Иначе
            K0min = X[isplit - 1]
        Если jsplit = 0, то
            K1min = 0
        Иначе
            K1min = Y[jsplit - 1]
        Для всех объектов О из блока В выполнить
            Если (K0(О) ≥ K0min) и (K1(О) ≥ K1min), то
                Перенести О из блока В в блок В'
    Конец РАСЩЕПЛЕНИЕ_ГРУППЫ

//=====
// Расщепление решетки
// Параметры:
//   iВ, jВ - индекс ячейки решетки, связанной с
//   переполненным блоком
//=====
РАСЩЕПЛЕНИЕ_РЕШЕТКИ(iВ, jВ)
    [1] // Выбор параметров расщепления
        n - номер оси расщепления
        Ksplit - ключ расщепления

```

```

[2] // Перераспределение объектов
В = внешний блок с номером Grid[jB][iB]
В' = новый внешний блок
Для всех объектов О из блока В выполнить
    Если  $K_n(O) \geq K_{split}$ , то
        Перенести О из блока В в блок В'
[3] // Расщепление решетки
Если n = 0, то
    Добавить  $K_{split}$  в массив X между элементами iB и iB+1
    Добавить столбец в массив Grid между iB и iB+1
    Для всех j от 0 до максимума выполнить
        Grid[j][iB+1] = Grid[j][iB]
        Grid[jB][iB+1] = В'
Иначе
    Добавить  $K_{split}$  в массив Y между элементами jB и jB+1
    Добавить строку в массив Grid между jB и jB+1
    Для всех i от 0 до максимума выполнить
        Grid[jB+1][i] = Grid[jB][i]
        Grid[jB+1][iB] = В'
Конец РАСЩЕПЛЕНИЕ_РЕШЕТКИ

```

Алгоритм расщепления состоит из трех процедур. Первая из них, процедура РАСЩЕПЛЕНИЕ_БЛОКА, определяет вид расщепления, который можно провести. На первом шаге приводится поиск группы ячеек, связанных с переполненным внешним блоком. Такой поиск достаточно просто реализуется. Для этого необходимо проверить все соседние ячейки на соответствие тому же самому блоку. В результате получаем группу, характеризующуюся диапазоном индексов по всем осям пространства:

$$G = \{i_{start}, i_{end}, j_{start}, j_{end}\}.$$

Если данная группа состоит всего из одной ячейки (т.е. $i_{start} = i_{end}$ и $j_{start} = j_{end}$), то в данном случае можно провести только расщепление решетки (вызвать процедуру РАСЩЕПЛЕНИЕ_РЕШЕТКИ). В противном случае – можно провести расщепление внешнего блока без расщепления самой решетки. Для этого вызывается процедура РАСЩЕПЛЕНИЕ_ГРУППЫ. Как было показано ранее, в большинстве случаев расщепление внешних блоков без деления решетки приводит к необходимому результату.

После процедур деления необходимо снова вернуться к проверке на переполненность текущего блока. В ряде случаев процедура расщепления может не привести к нужному результату и деление придется повторить.

Процедура РАСЩЕПЛЕНИЕ_ГРУППЫ вызывается, если с данным блоком связано более одной ячейки решетки. Действие процедуры заключается в распределении ячеек группы на две части так, чтобы каждая из частей образовывала область прямоугольной формы. В листинге 2.28 приведен простейший алгоритм этой операции. Для

этого в качестве оси деления выбирается линия, которая разобьет группу на две равные части по одной из осей (сначала такое разбиение пытаются провести по оси Ox , затем – по Oy). В результате образуется две приблизительно равные по числу группы ячеек. Одна группа остается связанной со старым блоком, вторая – переносится во вновь созданный пустой блок.

Такой подход является самым простым с точки зрения реализации. Однако он не всегда является эффективным. Самое хорошее распределение можно получить, перебрав все возможные варианты деления и выбрав тот, который распределит объекты по блокам как можно более равномерно. Ориентация на равномерное распределение объектов в блоках, а не ячеек в группах, является самой эффективной с точки зрения дальнейшей работы с файлом-решеткой.

Третья процедура листинга 2.28 – процедура деления решетки РАСЩЕПЛЕНИЕ_РЕШЕТКИ. Это самая трудоемкая процедура с точки зрения используемых ресурсов. Она выполняет модификацию всей решетки (и массива шкал, и массива самой решетки *Grid*).

На первом шаге этой процедуры выбирают ось деления решетки и координата деления. Данная операция жестко не регламентируется, выбор зависит от конкретной задачи и ее нюансов. При равнозначности ключей оси расщепления рекомендуется чередовать. Это позволяет сделать решетку более правильной формы, а поиск – более эффективным. Подобная тактика, например, использовалась в рассмотренном ранее примере (рис. 2.25).

Для чередования осей можно завести глобальную переменную, в которой будет находиться номер оси, по которой происходило последнее расщепление. При входе в процедуру РАСЩЕПЛЕНИЕ_РЕШЕТКИ на основе значения данной переменной будет выделено следующее измерение из списка возможных измерений.

Иногда известны измерения, по которым распределение объектов наиболее сильно меняется, или измерения, запросы по которым будут выполняться чаще. При этом можно реализовать более интеллектуальный порядок следования осей расщепления по сравнению с простым чередованием.

После выбора оси расщепления необходимо выбрать координату на этой оси, по которой пройдет новая плоскость решетки. В примере (рис. 2.25) в качестве такой координаты выбиралась середина переполненной ячейки. Это может оказаться очень полезным при геометрическом поиске или если заранее известно, что объекты в пространстве поиска распределены равномерно. В противном случае лучшей тактикой является выбор такой координаты, которая позволит распределить объекты переполненной ячейки приблизительно на две равные группы.

После выбора оси и координаты разбиения происходит распределение объектов переполненного блока в соответствии с заданными параметрами (РАСЩЕПЛЕНИЕ РЕШЕТКИ – шаг 2 процедуры) и модификацией самой решетки (шаг 3). Во время модификации решетки появляется еще одна плоскость деления (новый элемент в линейной шкале по осям) и, соответственно, новый столбец или строка в массиве решетки *Grid* (соответствующий данному делению). Так как большинство ячеек из старого столбца или строки были непереполненными и не нуждались в делении, происходит их объединение с новыми соседями. Исключение составляет единственная ячейка, которая и вызвала процедуру расщепления.

Процедура расщепления является наиболее трудоемкой среди всех алгоритмов обработки файлов-решеток. Поэтому во многих последующих модификациях были предприняты попытки откладывать данную процедуру как можно дольше. Некоторые из таких модификаций будут описаны далее.

Алгоритм удаления объекта

Удаление объекта из файла-решетки противоположно по своим действиям вставке объекта. Сначала определяется внешний блок, в котором находится удаляемый объект, и этот объект из него исключается. Если при этом число записей в блоке больше некоторого порогового значения, то процедуру удаления можно завершить. Если записей осталось мало, то данный блок объединяется с каким-либо из своих соседей. После объединения блоков проверяют возможность удаления строки или столбца решетки. Код процедуры показан в листинге 2.29.

Листинг 2.29

```
//=====
// Удаление объекта
// Параметры:
//   0 - удаляемый объект
//=====
УДАЛЕНИЕ(0)
[1] // Поиск ячейки решетки (определение i и j)
    n = количество элементов в шкале X
    i = 0
    Пока i < n выполнять
        Если  $K_0(0) < X[i]$ , то
            Прервать цикл
        i++
    n = количество элементов в шкале Y
    j = 0
    Пока j < n выполнять
        Если  $K_1(0) < Y[j]$ , то
            Прервать цикл
```



```

j++
[2] // Удаление объекта
    В = внешний блок с номером Grid[j][i]
    Загрузить В в оперативную память
    Для всех объектов О' из блока В проверить
        Если О = О', то
            Исключить О' из блока В
            Удалить О'
        Прервать цикл
    Выгрузить В во внешнюю память
[3] // Проверка на возможность слияния блоков
    СЛИЯНИЕ_БЛОКА(i, j)
Конец УДАЛЕНИЕ

//=====
// Процедура слияния внешних блоков
// Параметры:
//   i, j - индекс ячейки, из которой удалили объект
//=====
СЛИЯНИЕ_БЛОКА(i, j)
[1] // Проверка возможности слияния
    В = внешний блок с номером Grid[j][i]
    В' = блок-кандидат на слияние с В
    Если В' не найден, то
        Выйти из процедуры
    n = количество объектов в блоке В
    n' = количество объектов в блоке В'
    Если (n+n')/N > Порог, то
        Выйти из процедуры
[2] // Слияние блоков
    Для всех объектов О из блока В' выполнить
        Перенести О из блока В' в блок В
    Удалить блок В'
    Заменить все ссылки В' в массиве Grid на В
[3] // Проверка на возможность слияния решетки
    СОКРАЩЕНИЕ_РЕШЕТКИ(i, j)
Конец СЛИЯНИЕ_БЛОКА

//=====
// Процедура удаления строк и столбцов
// Параметры:
//   iCur - номер столбца
//   jCur - номер строки
//=====
СОКРАЩЕНИЕ_РЕШЕТКИ(iCur, jCur)
[1] // Проверка возможности удаления столбца
    MergeLeft = MergeRight = ИСТИНА
    n = количество элементов в шкале Y
    Для всех j от 0 до n проверить
        Если iCur = 0 ИЛИ Grid[j][iCur-1] ≠ Grid[j][iCur], то

```

```

MergeLeft = ЛОЖЬ
Если  $i_{Cur} = n-1$  ИЛИ  $Grid[j][i_{Cur}+1] \neq Grid[j][i_{Cur}]$ , то
    MergeRight = ЛОЖЬ
[2] // Удаление столбца, если это возможно
Если MergeLeft = ИСТИНА, то
    Удалить из шкалы X элемент с индексом ( $i_{Cur}-1$ )
    Удалить из массива Grid столбец с индексом  $i_{Cur}$ 
Если MergeRight = ИСТИНА, то
    Удалить из шкалы X элемент с индексом  $i_{Cur}$ 
    Удалить из массива Grid столбец с индексом  $i_{Cur}$ 
[3] // Проверка возможности удаления строки
MergeUp = MergeDown = ИСТИНА
n = количество элементов в шкале X
Для всех i от 0 до n проверить
    Если  $j_{Cur} = 0$  ИЛИ  $Grid[j_{Cur}-1][i] \neq Grid[j_{Cur}][i]$ , то
        MergeUp = ЛОЖЬ
    Если  $j = n-1$  ИЛИ  $Grid[j_{Cur}+1][i] \neq Grid[j_{Cur}][i]$ , то
        MergeDown = ЛОЖЬ
[4] // Удаление строки, если это возможно
Если MergeUp = ИСТИНА, то
    Удалить из шкалы Y элемент с индексом ( $j_{Cur}-1$ )
    Удалить из массива Grid строку с индексом  $j_{Cur}$ 
Если MergeDown = ИСТИНА, то
    Удалить из шкалы Y элемент с индексом  $j_{Cur}$ 
    Удалить из массива Grid строку с индексом  $j_{Cur}$ 
Конец СОКРАЩЕНИЕ_РЕШЕТКИ

```

По аналогии с процедурой вставки на первом шаге алгоритма определяют индексы ячейки решетки i и j , в которой должен находиться удаляемый объект O . После определения координат ячейки из решетки извлекается номер внешнего блока, этот блок загружается в оперативную память, и объект O удаляется из него (второй шаг алгоритма).

Первые два шага просты и интуитивно понятны. Самым сложным является обработка ситуации, при которой часть блоков становятся пустыми или редко заполненными. Как и при вставке объектов, здесь возможны два случая:

- объединение двух внешних блоков в один (при этом ячейки решетки, которые ссылались на эти блоки, объединяются в одну общую группу);
- удаление строки или столбца решетки.

В реальной ситуации чаще всего используется первый вариант, так как при удалении строк и столбцов необходимо выполнение множества условий. И только если файл-решетка в некоторой системе является сокращающейся (постоянно уменьшается количество хранимых записей), то возможен случай объединения строк и столбцов самой решетки. Рассмотрим эти два вида объединения более подробно.

Объединение внешних блоков. Процесс проверки на возможность слияния внешних блоков запускается при вызове процедуры `СЛИЯНИЕ_БЛОКА` из листинга 2.29. Первое, что необходимо выполнить в процедуре – выбрать некоторый соседний блок для B' , с которым можно было бы провести слияние (шаг процедуры 1). Если такого блока найти не удалось, то слияние в данном случае невозможно. Также слияние невозможно, если в результате объединения двух блоков получится переполненный блок. В практических реализациях вводят некоторую пороговую величину слияния *Porog*, и процедуру объединения выполняют только в том случае, если наполнение новой ячейки не превысит этот порог. Значение переменной *Porog* рекомендуется выбирать в районе 70%. Если значение будет близко к 100%, то велика вероятность того, что после процедуры слияния при следующей вставке придется данный блок снова разбивать на две части. В результате получаем лишние операции слияния/деления.

Если все условия для слияния двух внешних блоков выполнены, то происходит перенос объектов из одного блока в другой и удаление опустевшего блока (шаг алгоритма 2). При этом необходимо поменять ссылки в массиве *Grid* так, чтобы они указывали на один объединенный блок. Важно заметить, что такую процедуру вовсе не обязательно проводить с просмотром всего массива *Grid* (а размеры массива решетки могут быть очень большими). Достаточно проверить только близкие группы для ячейки (i, j) .

На последнем шаге алгоритма происходит вызов процедуры `СЛИЯНИЕ_РЕШЕТКИ`, которая должна проверить возможность удаления строки или столбца из решетки после очередного слияния ячеек. Такую проверку можно опустить, если известно, что файл будет расширяющимся или постоянным (число вставок и удалений будет приблизительно одинаковым), так как в этом случае вероятность появления ситуации, при которой будет возможно удалить целую строку или столбец, близка к нулю.

В процедуре объединения внешних блоков самым сложным и ответственным является алгоритм выбора блока-кандидата для слияния с данным. Несмотря на кажущуюся простоту, данный шаг содержит в себе ряд сложностей. Главным условием для выбора кандидата является то, что в результате объединения двух блоков должна получиться группа ячеек прямоугольной формы. В реальных условиях не все окружающие блоки могут подойти по этому критерию. Поэтому при выборе кандидата на слияние необходимо исключать из рассмотрения те варианты, которые приведут к группе ячеек сложной, не прямоугольной формы.

В листинге 2.29 этот шаг подробно не раскрыт, так как он во многом зависит от конкретных параметров и методов реализации. Однако рассмотрим два наиболее общих подхода – слияние соседних ячеек и слияние ячеек-близнецов.

Слияние ячеек-близнецов. Две ячейки называются близнецами в данном контексте, если они образовались из одной общей ячейки в процессе деления решетки. Так, для решетки, показанной на рис. 2.25(г), ячейки, связанные с блоками C и D , являются ячейками-близнецами, так как они образовались из одной ячейки при делении линией $x = x_2$, в то же время ячейки C и E близнецами не являются. Поэтому при удалении объектов из блока C он может быть слит только с блоком D .

Нетрудно заметить, что, придерживаясь данного правила слияния, все время будут получаться группы ячеек прямоугольной формы. Также данный подход приводит к тому, что в сокращающихся файлах условие удаления строк или столбцов выполняется чаще, чем при использовании каких-либо других методик. Однако у данного подхода есть два существенных недостатка.

1. Сложность определения, какие из ячеек решетки являются близнецами. Данный процесс детерминирован, если у нас сохранено дерево деления решетки или его можно восстановить, основываясь на каких-либо фактах (например, оно легко восстанавливается при регулярном делении пространства решеткой). Однако в сложных случаях это не всегда можно сделать.

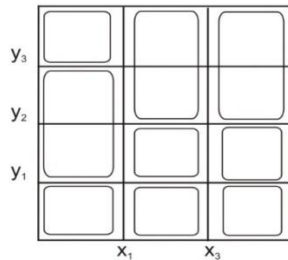
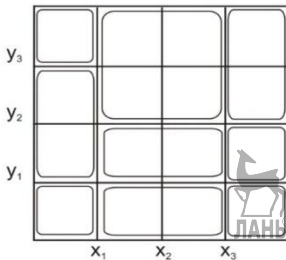
2. Возможен случай, когда в решетке рядом будут находиться две почти пустые ячейки, однако они не будут слиты, так как не будут являться близнецами. Это может повлиять на процент использования памяти самой структурой.

Слияние соседних ячеек. В данном способе кандидатом на слияние является абсолютно любой блок, который находится по соседству с заданным и который при слиянии образует группу ячеек прямоугольной формы. Возвращаясь к рис. 2.25(г), можно заметить, что, придерживаясь данной методики, блок C может быть слит как с блоком D , так и с блоком E . Однако он не может быть слит с блоком A , так как при этом образуется группа ячеек непрямоугольной формы.

Удаление строк и столбцов из решетки. Постоянное удаление объектов из файла может привести к тому, что целая строка или столбец решетки станет лишним. В этом случае его можно удалить из решетки без особого ущерба для нее. Такой процесс называется *сокращением решетки*. На рис. 2.27 показано такое сокращение. В результате последовательности некоторых объединений, все ячейки второго столбца оказались объединены с соседними справа ячейками третьего столбца. При этом деление пространства линией $x = x_2$ потеряло свою необходимость. Для сокращения удаляется элемент x_2 из линейной шкалы X и второй столбец из массива решетки *Grid*. Такое преобразование позволяет не только уменьшить накладные расходы на использование памяти, но и увеличить скорость запросов поиска.

В листинге 2.29 показан один из возможных вариантов реализации подобной процедуры. В процедуру передается номер столбца и строки,

которые нужно проверить на возможность удаления. Сначала проверяется столбец. Для этого блок каждой ячейки данного столбца сравнивается с соседними блоками левого и правого столбцов соответственно (шаг алгоритма 1). Это делается для того, чтобы определить факт возможности удаления данного столбца и слияния его либо с левым, либо с правым. Если находится хотя бы одно отличие по номерам блоков, то слияние с соответствующим столбцом становится невозможным, что и помечается специальным флагом. Если в результате полного просмотра один из флагов оказался истинным, столбец удаляется (шаг 2).



$$X = [x_1, x_2, x_3], \quad Y = [x_1, x_2, x_3]$$

$$Grid = \begin{bmatrix} A & B & B & C \\ D & B & B & C \\ D & E & E & F \\ G & H & H & I \end{bmatrix}$$

$$X = [x_1, x_3], \quad Y = [x_1, x_2, x_3]$$

$$Grid = \begin{bmatrix} A & B & C \\ D & B & C \\ D & E & F \\ G & H & I \end{bmatrix}$$

Рис. 2.27. Сокращение решетки

На третьем и четвертом шаге алгоритма проводится аналогичная проверка с той лишь разницей, что проверяется возможность удаления не столбца, а строки. Соответственно при удалении данная строка будет объединяться либо с той, что находится над ней, либо с той, что находится под ней.

В целом, как уже описывалось ранее, ситуации, приводящие к удалению целого столбца или строки решетки, в реальных приложениях происходят очень редко. Поэтому, если известно, что файл будет расширяющимся или стабильным по размеру, данную процедуру можно опускать.

Параметры файлов-решеток

При практической реализации файлов-решеток может возникнуть ряд вопросов, не рассмотренных ранее. Так, могут появиться вопросы, каким образом реализовать массив решетки, чтобы затем легко и

эффективно производились изменения, можно ли организовывать параллельный доступ к файлу из разных потоков, не появится ли при этом нежелательных последствий. Не менее важными могут оказаться вопросы по выбору размера внешнего блока или порога слияния. Эти и некоторые другие аспекты будут рассмотрены далее.

Выбор структуры хранения массива решетки. Логически массив решетки представляет собой n -мерный массив, элементами которого являются целые числа (номера внешних блоков). Однако физически такую структуру можно реализовать по-разному.

Первый способ – физически реализовать каталог точно так же, как она выглядит логически – в виде n -мерного массива целых чисел. В данном случае доступ к элементам будет происходить естественным способом с помощью обычных операторов матричного обращения. Это самый простой способ и в то же время самый эффективный с точки зрения скорости прямого обращения к элементам. Однако расширение решетки (вставку дополнительных строк и столбцов в середину массива) реализовать в этом случае будет не совсем просто. Для подобных операций возникнет необходимость не только выделения дополнительной памяти под новую строку/столбец, но и перенос части массива от места вставки и до конца массива. Аналогично при удалении строк и столбцов также появляется необходимость в переносе части данных в обратном направлении. Это связано с тем, что массивы не поддерживают простой способ удаления и вставки данных в середину массива.

Второй способ – реализовать каталог в виде одномерного массива целых чисел. При этом обращение к некоторому элементу решетки будет выполняться с помощью формулы расчета адреса. Так, в двумерном случае для определения индекса можно воспользоваться следующей формулой:

$$Pos = j \cdot NumCol + i,$$

где Pos – номер элемента в одномерном массиве;

i, j – номер столбца и строки соответственно в массиве решетки;

$NumCol$ – число столбцов в массиве решетки.

Данный способ по своим характеристикам практически не отличается от предыдущего. Для него характерны те же самые достоинства и недостатки.

Может возникнуть идея реализовать массив решетки в виде некоторой списковой структуры. Списки поддерживают простой механизм вставки и удаления элементов в середину списка без перемещения данных в памяти. Однако списки являются структурами последовательного доступа, что неприемлемо для реализации каталога файлов-решеток. К тому же списки для своей организации требуют дополнительной памяти (это также становится очень важным, если учесть, что размер каталога может достигать огромных размеров).

Разработчики файлов-решеток проанализировали все известные методы реализации массивов в памяти и пришли к выводу, что реализация каталога в виде массива будет самой эффективной, несмотря на возможные сложности операции редактирования решетки. В большинстве приложений операции разделения и слияния в каталоге происходят гораздо реже, чем операции прямого доступа и перехода. Таким образом, достаточно эффективно работает обычный массив. Операции деления и слияния могут приводить к перезаписи всего каталога, а это занимает больше времени для массива, чем для списков. Однако при использовании массивов старый каталог может быть использован для доступа к данным, пока новый каталог создается. Кроме того, операция прямого доступа осуществляется быстрее в массивах, и память используется более эффективно, поэтому большую часть каталога можно хранить в оперативной памяти. Получается, что преимущества обычного размещения массива перевешивают недостатки времени обработки.

Если планируется использовать систему в высокодинамичных приложениях, в которых деление решетки предполагается очень частым, стоит воспользоваться некоторыми модификациями, при которых применяется либо система без каталога (PLOP-хеширование), либо регулярная решетка с предварительным делением. Во втором случае каталог представляет собой n -мерный массив, размеры ячеек которого определены самым коротким интервалом в каждой линейной шкале по каждому измерению (рис. 2.28). Подобная технология применяется в расширяемом хешировании. Изменение решетки приводит к изменению в структуре каталога только в том случае, если происходит деление самого маленького интервала (при этом размер каталога удваивается). Во всех остальных случаях деление проводится без перемещения данных по файлу. В данной модификации происходит замена последовательности структурных изменений одним большим предварительным делением. На рис. 2.28 показано изменение решетки при переполнении и делении второго столбца.

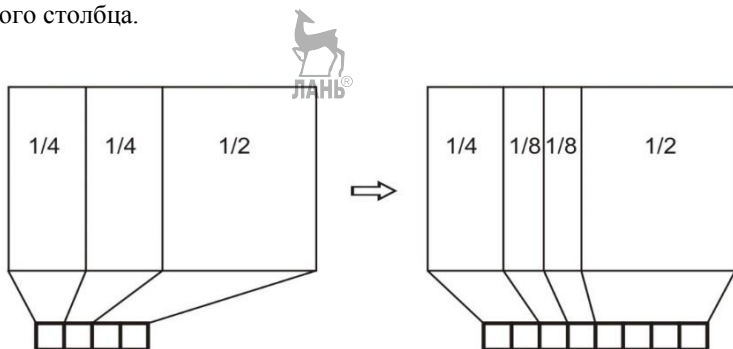


Рис. 2.28. Представление решетки массивом с удвоением размера

Однако подобный способ лучше использовать только при однородном распределении данных. В противном случае постоянное удвоение решетки может привести к чрезмерно большому размеру каталога. В расширяющемся хешировании однородность обеспечивается рандомизированной хеш-функцией, даже если данные распределены в пространстве записей неоднородно. Исходя из того, что файл решетки был спроектирован для эффективных ответов на диапазонные запросы, а функции-рандомизаторы неприменимы к таким запросам, такой подход не может быть использован для создания однородно распределенных данных.

Следовательно, любые неоднородности в данных приводят к большим размерам каталога, и такой подход не может быть применен.

Приоритет ключей. Ранее говорилось, что чередование осей измерения при расщеплении пространства может быть организовано в зависимости от приоритета ключей. Так, если известно заранее, что первый ключ в запросах поиска будет использоваться чаще, то при построении решетки можно производить по этому измерению делений в два раза больше.

Однако при практическом использовании зачастую сложно сказать, какое измерение является более приоритетным, а какое – менее. Особенно сложно бывает вычислить относительное превосходство, т. е. во сколько раз больше нужно проводить деление по тому или иному ключу. Для решения подобной задачи можно использовать динамические приоритеты.

При использовании файла-решетки можно собирать информацию о числе запросов к системе и применении в этих запросах тех или иных ключей. На основе данной статистики в дальнейшем можно вычислить приоритет ключей, который будет использоваться для адаптации параметров деления и слияния. Идею можно развить вплоть до того, что неактивные ключи могут быть установлены в положение «только для слияния», после чего они будут постепенно изменяться. Когда их деление будет уменьшено до единственного интервала, соответствующее измерение в каталоге решетки может быть вообще удалено или назначено другому атрибуту.

Параллельный доступ. Большинство систем хранения и обработки информации можно значительно ускорить, используя в них параллельный доступ к данным. Если структура является статической (не поддерживает операции вставки/удаления/изменения), то параллельный доступ к данным является операцией тривиальной, ведь операции поиска не должны блокировать друг друга. Блокировки могут возникать только в динамических структурах, поддерживающих изменение хранящихся данных.

Контроль за параллельностью в древовидных структурах осложнен наличием корня, который используется всеми операциями доступа. Если

один процесс выполняет операцию изменения вблизи корня (например, вставка и удаление в сбалансированных деревьях), другие процессы могут быть замедлены в связи с соблюдением протоколов блокировки, даже если доступ производится для несвязанных между собой данных.

В то же время для файлов-решеток (а также и других структур, основанных на вычислении адреса) пути доступа к отдельным участкам памяти не связаны, что позволяет упростить протоколы параллельного доступа. Единственная операция, которая занимает исключительно важный ресурс, – это модификация решетки. Однако эту процедуру можно проводить в теновом режиме. При этом построение новой версии происходит не с помощью замены старой, а независимо от нее. Пока новая решетка не будет построена, операции поиска могут использовать старый вариант. После внесения всех изменений решетки просто меняются местами, и операция модификации заканчивается.

Размер внешнего блока решетки. Еще одним вопросом, который возникает при реализации структуры, является вопрос о числе записей в одном внешнем блоке. С одной стороны, чем больше записей разместить в одном внешнем блоке, тем реже будет появляться необходимость в делении решетки. Но, с другой стороны, все операции поиска заканчиваются загрузкой нужного внешнего блока в память и проверкой всех его объектов на соответствие условиям поиска. Поэтому большое число записей негативно может сказаться на скорости поиска.

Обычно число записей внешнего блока выбирают таким, чтобы размер этого блока равнялся размеру единицы обращения к устройству хранения (например, кластеру или сектору жесткого диска). Однако выбор очень маленького числа записей в блоке (менее 10) может привести к частым операциям деления, что в реальных приложениях сильно замедляет время работы системы.

Может возникнуть вопрос, а как влияет число записей на другие показатели файлов-решеток, например на среднюю заполненность файла. Опыты, проведенные разными исследователями, показали, что средняя заполненность памяти в оригинальных файлах-решетках не зависит от размера внешнего блока и числа записей в нем и приблизительно равна 70%.

Разновидности файлов-решеток

Файлы-решетки были одними из первых структур, использовавших идеи хеширования в многомерном пространстве. Впоследствии появилось достаточно много разновидностей файлов-решеток, отличающихся числом решеток и алгоритмами деления/слияния. Авторы практически всех разновидностей пытались либо увеличить скорость доступа, изменив каким-либо образом каталог решетки, либо хоть частично избавиться от сложностей при делении решетки. Рассмотрим наиболее популярные варианты. Некоторые методы хеширования, также

похожие на файлы решетки (*EXCELL*, *PLOP*), будут рассмотрены в следующих разделах.

Двухуровневый файл-решетка. Данный вариант был разработан К. Гинричем в 1985 году [42]. В нем основное внимание уделено замедлению скорости поиска вследствие большого объема каталога.

Основная идея двухуровневого файла-решетки заключается в построении еще одной решетки, которая используется для управления основным каталогом. Таким образом, получаем каталог, состоящий из двух решеток. Первая из них, называемая *корневой решеткой*, содержит в своих ячейках ссылки на страницы внешней памяти, содержащие части основной решетки. Каждая часть основной решетки имеет такую же структуру и назначение, как и в оригинальных файлах-решетках. Пример двухуровневого файла-решетки показан на рис. 2.29.

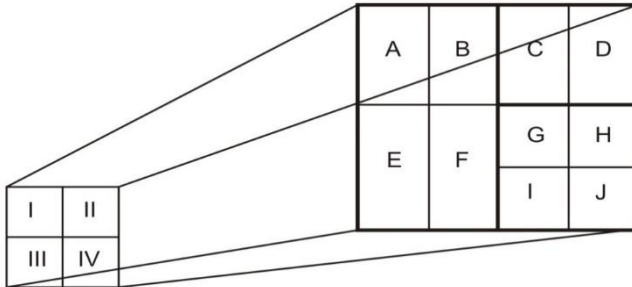


Рис. 2.29. Двухуровневый файл-решетка

В двухуровневых файлах поиск начинается с корневой решетки. В ней определяется блок внешней памяти, в котором хранится нужная часть основной решетки, из которой в свою очередь извлекается информация о внешнем блоке с записями файла. Такой процесс добавляет дополнительные расходы на память и усложняет процедуры модификации решетки. Однако благодаря второму уровню решетки ускоряется процесс поиска по решетке. Также из-за наличия второго уровня решетки деление какой-либо строки или столбца может затрагивать не весь каталог, а только его небольшую часть. Это позволяет уменьшить скорость роста каталога, однако полностью проблему не решает.

Двухуровневый файл-решетка не нарушает принципа двух дисковых запросов для процедуры поиска по точному совпадению. Дело в том, что корневая решетка обычно занимает мало места и размещается в оперативной памяти.

Двойной файл-решетка. В 1988 году А. Хитфлез, Г. Сикс и П. Видмар предложили еще один вариант улучшения свойств файлов-решеток [45]. Их изменения позволили повысить процент использования

памяти благодаря созданию еще одной самостоятельной решетки. Вторая решетка действительно являлась самостоятельной и никак не зависела от первой. Она так же, как и первая была предназначена для хранения некоторых объектов, т. е. в отличие от двухуровневого варианта, описанного выше, в ней не было иерархического построения каталога.

Исследователи в своих работах показали, что процент использования памяти при таком распределении повысился до 90%, что несравнимо больше по сравнению с оригинальным вариантом (в оригинальной схеме процент использования равняется 70%).

Все объекты в двойном файле распределяются между двумя решетками. Часть объектов находится в первой из них, другая часть – во второй. Конечно, при этом усложняется процедура поиска. Для выполнения запроса необходимо выполнить поиск как в одной решетке, так и в другой (т. е., по сути, выполнить два поиска в разных решетках). Однако улучшение распределения, качества решетки и повышение процента использования памяти приводят к значительным преимуществам, которые в ряде случаев превосходят этот минус. На рис. 2.30 показан пример такого файла.

Для того чтобы преимущества использования двух каталогов были существенными, необходимо изменить процедуру вставки. При добавлении объектов в решетку сначала предпринимается попытка вставки его в первую решетку *Grid1*. Если подобная операция должна привести к расщеплению решетки, то вставку отменяют и пытаются добавить объект во вторую решетку *Grid2*. Это позволяет избежать лишних делений решетки и повысить как процент использования памяти, так и скорость работы подобной операции.

Однако рано или поздно возникнет ситуация, когда вставка некоторого объекта должна привести к модификации как *Grid1*, так и *Grid2*. При этом для вставки выбирают ту решетку, деление которой приведет к меньшему увеличению общего числа блоков. После проведения процедуры деления решетки возможно выполнение перемещения части записей между решетками так, чтобы это улучшило общие показатели каталога.

Существует несколько стратегий использования двух файлов. В некоторых из них предполагается ориентация на большую наполненность первого файла (называемого основным) и использование второй решетки просто как источника резервного хранения при модификациях. Такой способ позволяет в большинстве случаев поиска по точному совпадению ограничиться просмотром только первой решетки (если запись действительно окажется в ней). После деления решетки стараются подобрать такой алгоритм, который позволит большую часть объектов перенести в первую решетку, а ячейки второй объединить или даже удалить.

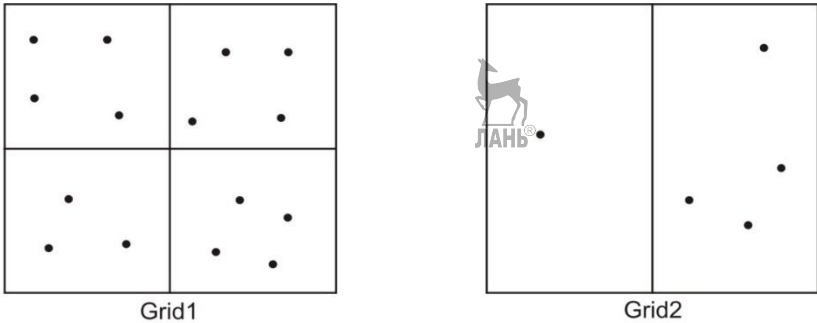


Рис. 2.30. Двойной файл-решетка

В других вариантах поддерживают приблизительно равный процент записей в обеих решетках и основной упор делают на поиск таких алгоритмов, которые позволят распределять объекты так, чтобы необходимость модификации решетки (слияние или деление) возникала как можно реже.

В данной части были рассмотрены лишь две базовые модификации файлов-решеток. Однако существует большое число вариантов, улучшающих те или иные их показатели. В частности, есть варианты, развивающие идеи иерархического каталога (многоуровневые файлы-решетки, *Buddy Tree*), или идеи объединения нескольких самостоятельных решеток (многослойные файлы-решетки). Рассмотрение этих структур выходит за рамки данного издания.

2.3.2. Хеширование EXCELL

Структура *EXCELL* (*EX*extendible *CELL*) была предложена М. Таминеном в 1984 году [86]. Его подход очень похож на обычные файлы-решетки. Предложенная структура, как и файлы-решетки, является методом хеширования с каталогом и обеспечивает доступ к записям за два обращения к внешней памяти.

Ключевым отличием файлов-решеток от *EXCELL* является то, что при переполнении некоторой ячейки в первых происходит деление только одного интервала на две части (вставка одного значения в линейную шкалу и добавление одного столбца или строки в массив решетки), в то время как в *EXCELL* происходит деление всех интервалов по соответствующему направлению и удвоение каталога. Данное изменение позволяет ускорить некоторые обработки и поиски. Однако удвоение каталога при переполнении имеет и отрицательные последствия (особенно для неравномерного распределения данных).

Большинство понятий, принципов и алгоритмов, описанных в разделе, посвященном файлам-решеткам, подходят и для *EXCELL*.

В данном параграфе будут описаны только самые важные отличия двух структур и показано, как эти отличия можно реализовать.

Общие принципы построения EXCELL

Самыми важными моментами, позволяющими говорить о *EXCELL* как о самостоятельной структуре данных, а не некотором подвиде файлов-решеток, являются следующие два отличия:

- регулярное деление пространства;
- удвоение каталога при расщеплении решетки.

Регулярное деление пространства. Под регулярным делением пространства решеткой понимается тот факт, что все интервалы линейных шкал по некоторому измерению являются одинаковыми (но они не обязательно должны совпадать по разным измерениям). Таким образом, можно утверждать, что в *EXCELL* пространство в каждом измерении делится на некоторое число равных по размеру частей.

Данное свойство позволяет сделать ряд упрощений и модификаций, положительно сказывающихся на производительности структуры в целом. Главное из них – это отсутствие линейных шкал. Действительно, если все интервалы по некоторому измерению равны, то отпадает необходимость хранить точки, по которым произведено деление решеткой. Поэтому каталог *EXCELL* упрощается по сравнению с файлами-решетками. Теперь она состоит не из двух частей (линейные шкалы в оперативной памяти и решетка во внешней памяти), а из одной – только массив решетки. Из данного факта можно сделать несколько выводов.

1. Поиск по точному совпадению ускоряется. В файлах-решетках первым шагом такого поиска было определение индекса в массиве-решетке по каждому измерению. Для этого необходимо провести поиск в массиве-шкале, который в лучшем случае является логарифмическим. В *EXCELL* линейные шкалы отсутствуют, а индекс по измерению определяется по следующей формуле:

$$i = \lfloor K_i / N_i \rfloor,$$

где i – индекс в массиве решетки по i -му измерению;

K_i – i -й ключ объекта;

N_i – число ячеек в массиве решетки по i -му измерению.

Вычисление по этой формуле гораздо быстрее любого поиска по массиву, поэтому общая скорость процедуры поиска объекта по точному совпадению возрастает.

2. Отказ от линейных шкал приводит к уменьшению затрат оперативной памяти. Для файлов-решеток линейные шкалы хранятся в оперативной памяти, так как размещение их во внешней памяти привело бы к необходимости третьего обращения к внешней памяти в процедурах поиска. Сами по себе линейные шкалы являются не очень большими по сравнению с общим объемом хранимых данных. Однако размер

оперативной памяти всегда ограничен, и наличие лишних субданных в ней нежелательно. Отказ от линейных шкал позволяет освободить практически всю оперативную память под нужды приложения и перенести весь индекс во внешнюю память. Если же требуется увеличить скорость работы, то можно освободившуюся оперативную память использовать для кэширования некоторой наиболее часто используемой части каталога.

3. Равный размер ячеек решетки позволяет облегчить диапазонный поиск. В файлах-решетках использовались ячейки разных размеров. При этом линейные размеры в большинстве реализаций зависят от распределения данных. Поэтому для определения внешних блоков, попавших в заданный диапазон поиска, необходимо выполнять ряд манипуляций со шкалами и массивом. В *EXCELL* все ячейки имеют равный размер и не зависят от распределения данных в пространстве (это одновременно является как положительной, так и отрицательной стороной). Поэтому геометрические запросы (ориентированные на области) в нем выполняются гораздо быстрее.

Удвоение каталога при расщеплении решетки. Равенство интервалов разбиения приводит к необходимости изменения обработки ситуации с переполнением ячеек. Если предел по числу объектов в некоторой ячейке решетки превышает, нельзя просто разбить один столбец или строку массива решетки на две части. Такое разбиение приведет к нарушению равенства интервалов разбиения. Поэтому появляется необходимость при переполнении разбивать все ячейки решетки вдоль выбранного разбиения. Этот факт позволяет сразу же заметить ряд сложностей и неэффективных ситуаций в работе *EXCELL*.

1. При переполнении всего одной ячейки размер решетки увеличивается в два раза. При этом происходит разбиение всех ячеек решетки и производительность дальнейшего выполнения процедур поиска несколько снижается.

2. Процедура деления затрагивает не часть массива каталога (как это было в файлах-решетках), а весь массив целиком. Появляется необходимость перестроения абсолютно всего каталога, что может занимать много времени.

3. Если распределение данных в пространстве является неравномерным, то подобное деление приведет к появлению большого числа пустых и слабо заполненных ячеек, которые необходимо объединять в группы. Само по себе такое объединение не сильно влияет на производительность процесса поиска, но при этом происходит неоправданное разрастание каталога (массива решетки). Поэтому для неравномерных распределений данных *EXCELL* практически не подходит.

Однако, несмотря на все описанные недостатки, есть у процесса удвоения каталога и положительные стороны. Если данные распределены

равномерно в пространстве поиска, то за переполнением и делением некоторой ячейки решетки скорее всего произойдут переполнения и деления остальных ячеек такого же размера. Такой вывод можно сделать согласно теории вероятности и закону больших чисел. Наполнение всех ячеек при равномерном распределении будет приблизительно одинаковым. Поэтому, если некоторая ячейка переполнена и нуждается в делении, остальные ячейки того же размера также близки к порогу деления. В файле-решетке переполнение каждой ячейки обрабатывается самостоятельно в момент, когда это переполнение происходит. Таким образом, появляется период времени, в который растущий файл-решетка часто делится и его каталог модифицируется. Как было отмечено ранее, процесс деления обладает большой трудоемкостью и требует больших вычислительных и временных затрат.

EXCELL при подобных обстоятельствах как бы заменяет последовательность частичных делений решетки одним большим делением. При этом ее размер увеличивается в два раза. Однако нетрудно показать, что такое деление эффективно только при равномерном распределении данных в пространстве поиска.

Алгоритм добавления нового объекта

Рассмотрим процесс построения решетки на примере, который уже использовался для файлов-решеток. Используя те же самые данные в качестве входных параметров, можно показать основные нюансы и отличительные черты *EXCELL* от файлов-решеток. В качестве параметров решетки выберем сходные с теми, которые использовались в прошлом разделе: пространство является двумерным, ключи – равнозначны, при делении происходит чередование осей, вместимость блока решетки равна трем ($N = 3$).

Построение начинается с пустого файла. При этом все пространство поиска находится в одной ячейке и связано с одним внешним блоком (который в начальный момент также является пустым). На этом этапе все очень похоже на файл-решетку.

Начнем добавлять объекты в файл. Первые три объекта будут размещены в единственном на данный момент внешнем блоке (назовем его блок *A*). Четвертый объект вызовет переполнение ячейки и появится необходимость деления. Как и в файлах-решетках, выделяют два типа деления – деление внешних блоков и деление массива решетки. В нашем случае решетка состоит всего из одной ячейки, которая и переполнилась. Поэтому единственно возможным является деление массива решетки. Для этого пространство разбивается по оси *Ox* на две равные по размеру части (рис. 2.31(a)) и решетка перестраивается. Число ячеек в ней увеличивается в два раза.

На представленном рисунке данный процесс позволил избавиться от переполнения ячейки решетки. Появилось два внешних блока (*A* и *B*), и

все объекты распределились между ними. Однако, в отличие от файлов-решеток, для *EXCELL* это не всегда справедливо. Так, если бы все четыре объекта находились в правой части пространства, то первая ячейка решетки осталась бы переполненной даже после деления. Поэтому, в отличие от файлов-решеток, после окончания процедуры деления необходимо проверить, не осталось ли переполненных ячеек. Если такие ячейки имеются, то для них необходимо снова повторить процедуру деления.

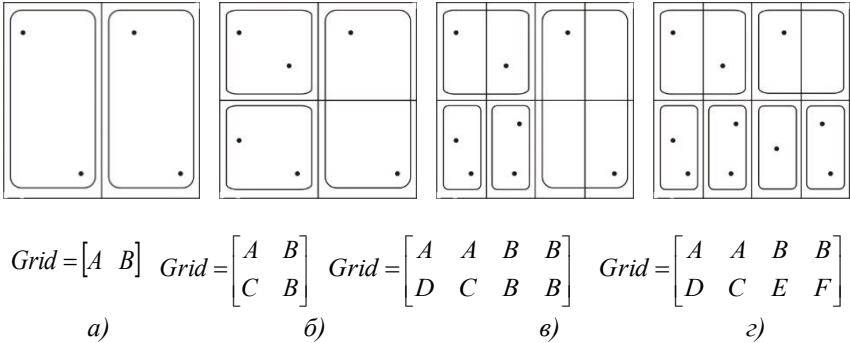


Рис. 2.31. Построение решетки *EXCELL*: а – деление решетки по оси Ox ; б – деление решетки по оси Oy ; в – деление решетки по оси Ox ; г – расщепление внешних блоков

Добавим еще пару объектов, которые разместятся в ячейке решетки, связанной с внешним блоком A . При этом снова происходит переполнение ячейки и опять появляется необходимость деления решетки. Так как все измерения в приведенном примере являются равнозначными, то необходимо разбивать решетку по оси Oy на две равные части. В результате получится распределение, показанное на рис. 2.31(б). Первая переполненная ячейка (связанная с блоком A) будет разбита на две самостоятельные ячейки, связанные с блоками A и C , а вторая ячейка – на две сгруппированные ячейки, связанные с блоком B .

Продолжим добавлять объекты. Разместим еще два из них в ячейку $[1;0]$, связанную с блоком C . Такая ситуация снова приведет к необходимости деления решетки. В файлах-решетках деление происходило только первого столбика массива решетки. Однако в *EXCELL* есть жесткое ограничение, связанное с регулярностью решетки. Поэтому делению подвергаются все ячейки по оси Ox на две равные части. Те из них, которые не переполнены в данный момент, объединяются в группы (по тем же принципам, что и объединение в файлах-решетках). В результате получаем массив, представленный на рис. 2.31(в).

На данном шаге наглядно представлено основное отличие – при переполнении массива решетки в *EXCELL* происходит расщепление всех ячеек массива и удвоение размера каталога. Сначала может показаться такое поведение затратным. Однако удвоение решетки не привело к удвоению числа внешних блоков. Непереполненные ячейки были объединены в группы. Число блоков увеличилось незначительно.

Также стоит заметить, что удвоение размера решетки позволяет избавиться от некоторых модификаций в будущем. Так, при вставке нескольких объектов во внешний блок *B* произойдет его переполнение. Однако данная ситуация ограничится только делением внешних блоков без модификации решетки в целом.

Обработка переполнения блока *B* заслуживает особого внимания. Сначала группа ячеек разбивается на две части по оси *Oy*. Две верхние ячейки остаются в блоке *B*, две нижние – переносятся в блок *E*. Разбиение группы по оси *Ox* было бы более сбалансированным, однако в *EXCELL* при делении группы ячеек нужно следовать правилу: разбиение группы ячеек должно проводиться в той же последовательности, что и деление массива решетки. Именно поэтому деление блока *B* происходит по оси *Oy*. При следующем переполнении блок *E* будет снова разбит на две части – блоки *E* и *F*. Однако из-за того, что размер решетки был уже удвоен, это не приведет к модификации массива. Данная операция ограничится простым делением внешнего блока. В результате получаем распределение, показанное на рис. 2.31(г).

Алгоритм удаления объекта

Рассмотрим еще одну операцию – удаление объектов из решетки. Как и другие операции, удаление объектов имеет много общего с аналогичной процедурой для файлов-решеток. Сама процедура удаления в целом практически ничем не отличается. Основные нюансы заключены в объединении внешних блоков и ячеек решетки. Дополнительные свойства *EXCELL* накладывают ряд новых ограничений, которые и будут рассмотрены в данном параграфе.

Как и в случае с файлами-решетками, структура поддерживает два вида объединения. Первый из них, выполняющийся наиболее часто, – это объединение внешних блоков. В процессе этой операции два внешних блока, являющиеся пустыми или почти пустыми, объединяются в один общий блок. При этом ячейки решетки, ссылающиеся на эти блоки, также объединяются в одну группу. Однако, в отличие от файлов-решеток, *EXCELL* позволяет объединять только так называемые ячейки-близнецы (те ячейки решетки, которые были получены в результате деления одной и той же ячейки). Так, если удалить объект из ячейки [1;2] (см. рис. 2.31(г)), то данная ячейка и связанный с ней блок *E* станут пустыми. Однако объединить блоки *C* и *E* нельзя, так как они получены в результате деления разных ячеек решетки. Объединение возможно

только блоков E и F . В дальнейшем, если удаление объектов продолжится, получившийся блок можно будет объединить с блоком B , что позволит сгруппировать четыре ячейки в одну общую группу.

Нетрудно заметить, что объединение в *EXCELL* полностью симметрично делению. Объединяются только те блоки, которые в свое время были получены с помощью деления соответствующих групп ячеек. Поэтому, в отличие от файлов-решеток, чередование объединений и делений блоков не приведет к ухудшению структуры решетки.

Второй вид объединений – это слияние решеток. При объединении ячеек в группы может возникнуть ситуация, при которой некоторое деление станет неактуальным. Данное явление на практике случается редко и возможно только в постоянно сокращающемся файле. Необходимым и достаточным условием слияния решетки является следующий факт: в решетке не должно быть ни одной самостоятельной ячейки, все ячейки должны быть объединены в группы. При выполнении этого условия можно провести слияние решетки. В процессе слияния интервал ячейки вдоль последней оси деления увеличивается в два раза и происходит двукратное уменьшение решетки.

Рассмотрим пример возможного слияния (рис. 2.31(г)) 4-х самостоятельных ячеек решетки. Слияние возможно только в том случае, если все ячейки будут объединены в группы. Данное явление может произойти, если в результате удаления объектов из блока F произойдет его объединение с блоком E , а также, в результате удаления объектов из C , он объединится с D . В этом случае все ячейки решетки будут входить в какие-либо группы. Последнее расщепление решетки происходило вдоль оси Ox , значит, и слияние так же должно происходить по этой оси. В процессе слияния интервал по оси Ox увеличивается в два раза и, соответственно, число ячеек уменьшается в два раза. При этом из массива решетки каждый второй столбец удаляется. В результате получаем структуру, показанную на рис. 2.31(б).

2.3.3. Многомерное линейное хеширование с частичным расширением (MOLHPE)

В предыдущих подпараграфах были представлены алгоритмы, использующие в функции хеширования некоторые дополнительные данные, называемые каталогом. В этой главе описывается другой способ хеширования, основанный на функции без каталога.

Схема *MOLHPE* была предложена в 1986 году Г. Крейгиллом и Б. Сигиром [51]. В ней авторы развили идеи линейного хеширования с частичными расширениями (схема Ларсона) на многомерный случай. Название алгоритма *MOLHPE* расшифровывается как многомерное линейное хеширование с частичным расширением (*Multidimensional Order-preserving Linear Hashing with Partial Expansions*).

Схемы хеширования с каталогом, рассмотренные ранее, обладают рядом недостатков. В предыдущих параграфах указывалось, что размер каталога растет сверхлинейно по отношению к росту хранимых данных даже при равномерном распределении данных. Если же данные распределяются неравномерно, то во многих вариантах файлов-решеток рост размера каталога может стать экспоненциальным. Этот факт ограничивает применение данных структур в ряде приложений. В *MOLHPE* от дополнительных данных (каталога) отказываются полностью, и это позволяет искоренить данный недостаток.

Чтобы избавиться от использования каталога, необходимо ввести такую функцию хеширования, которая позволяла бы вычислять адрес размещения объекта во внешней блоке, основываясь только на ключах этого объекта (не используя дополнительные данные, такие как массив решетки, которые нужно будет сохранять в каталоге). Такую функцию легко найти для статичных данных. При этом можно подобрать некоторую математическую формулу, которая позволит разбивать данные на группы практически равного размера. Однако в большинстве приложений данные имеют динамический характер (т. е. постоянно добавляются и удаляются объекты, изменяются существующие записи). Поэтому необходимо использовать динамическую функцию, адаптирующуюся к числу хранимых данных. Для одномерного случая такие функции были впервые предложены В. Литвиным. Однако эти схемы требовали скачкообразного увеличения пространства, используемого для хранения данных. Это плохо сказывалось на таком показателе, как процент использования памяти структурой (отношение общего количества памяти, занимаемого индексом, к объему хранимых в нем данных). Позднее П. А. Ларсон предложил схему с линейным ростом индекса [59]. Именно эта схема и легла в основу многомерного хеширования *MOLHPE*.

Помимо проблемы роста каталога, файлы-решетки обладают еще одним существенным недостатком – процедура вставки рано или поздно приводит к необходимости деления решетки. Деление ячеек решетки практически во всех вариантах происходит с частичной или полной модификацией массива решетки, которая занимает много времени и требует значительных затрат ресурсов системы. В различных модификациях файлов-решеток исследователи пытались уменьшить вероятность наступления такого события или заменить череду небольших делений одним большим. Однако полностью убрать этот недостаток невозможно, так как он непосредственно вытекает из факта использования каталога в структуре.

В *MOLHPE* не используется каталог, и поэтому расширение решетки происходит только путем модификации функции хеширования. Это не требует никаких затрат ресурсов системы и происходит практически мгновенно.

Однако не все идеально в схеме *MOLHPE*, есть и ряд недостатков. В частности, отказ от каталога и использование универсальной функции хеширования приводит к появлению необходимости использования совершенно других механизмов обработки коллизий. В частности, при наступлении факта переполнения внешнего блока в файлах-решетках происходит расщепление решетки, и обработка коллизии на этом заканчивается. Аналогичные действия не подходят для хеширования без каталога, так как они будут вызывать появление большого числа пустых блоков. Поэтому в *MOLHPE* и ряде других схем без каталога для обработки коллизий используется метод цепочек. При переполнении внешнего блока данные добавляются в дополнительные блоки, которые объединяются в цепочку блоков. В то же время расщепление решетки происходит только при выполнении некоторого заранее заданного условия (например при достижении порога по проценту использования памяти).

Общие принципы хеширования *MOLHPE*

В общих чертах хеширование *MOLHPE* похоже на файлы-решетки. Все пространство поиска разбивается некоторой решеткой на ячейки. С каждой ячейкой связывается некоторый внешний блок, в котором сохраняются все объекты, попавшие в эту ячейку. Однако алгоритмы *MOLHPE*, выполняющие данные принципы, кардинально отличаются от всех рассмотренных ранее.

Первое отличие заключается в структуре функции хеширования. Для схем с каталогом характерно наличие линейных шкал (явных или неявных), по которым определяется индекс элемента некоторого массива решетки. В массиве решетки находится номер внешнего блока, который и возвращается как значение функции хеширования.

Хеширование *MOLHPE* вычисляет адрес внешнего блока иначе. На основе некоторых математических функций, используемых в одномерном хешировании, определяется индекс ячейки решетки, в которой должен размещаться данный объект. Для этого к каждому ключу объекта применяют некоторое математическое преобразование H_i (функция линейного хеширования для i -го измерения). Затем полученные индексы подставляют в некоторую функцию заполнения пространства G , которая и возвращает номер внешнего блока, содержащего данный объект. Таким образом получается, что функция хеширования в *MOLHPE* представляет собой комбинацию некоторых математических вычислений и абсолютно не использует никаких дополнительных данных в виде каталога. Более подробное описание математической реализации функции хеширования будет представлено далее. Наглядное представление поиска номера внешнего блока для двумерного случая показано на рис. 2.32.

Объект поиска:

$$O = (K_0, K_1)$$

Определение индекса в решетке:

$$l_0 = H_0(K_0)$$

$$l_1 = H_1(K_1)$$

Определение номера внешнего блока:

$$n = G(l_0, l_1)$$

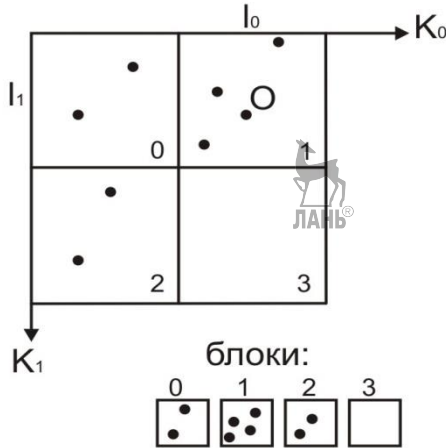


Рис. 2.32. Функция хеширования

Еще одним отличием *MOLHPE* от методов с каталогом является способ обработки коллизий (переполнений). Коллизия – это ситуация, при которой ряд записей попадает в одну и ту же ячейку решетки, что приводит к переполнению внешнего блока, связанного с этой ячейкой.

Допустим, имеется схема хеширования, в которой во внешнем блоке может располагаться не более трех объектов. Индексирование 7 записей показано на рис. 2.33(а). При этом блок с номером 3 является пустым (однако даже пустой блок должен присутствовать в системе), а блок с номером 1 содержит три записи. Добавление еще одной записи в блок 1 приведет к коллизии и его переполнению. В случае алгоритмов хеширования с каталогом произошло бы расщепление решетки, в результате которого появился бы еще один столбец (или строка) в массиве решетки. При этом появляется много дополнительных ячеек, которые являются пустыми или почти пустыми. Для файлов-решеток такие ячейки объединяются в группы и ссылаются на один и тот же внешний блок. Информация о таких объединениях находится в каталоге. Таким образом, использование внешней памяти (если не считать затрат на увеличение каталога) не увеличивается.

MOLHPE не использует каталог для хранения соответствия ячеек решетки и внешних блоков. Вместо этого используется прямое сопоставление – каждой ячейке решетки соответствует свой внешний блок. Даже если в ячейке нет ни одной записи, для нее все равно выделяется пустой внешний блок (так сделано, например, для ячейки 3, показанной на рис. 2.33(а)). Поэтому подобный алгоритм обработки коллизий приведет к ухудшению процента использования памяти структурой и уменьшению скорости доступа к записям.

Разработчики *MOLHPE* пошли по другому пути. В случае переполнения происходит расщепление ячейки решетки только в том случае, если общий процент использования памяти индексом превысил некоторое пороговое значение. Алгоритм расширения выполняется по той же самой схеме, которую использовал П. А. Ларсон в линейном хешировании [59]. Подробнее данный алгоритм будет рассмотрен далее.

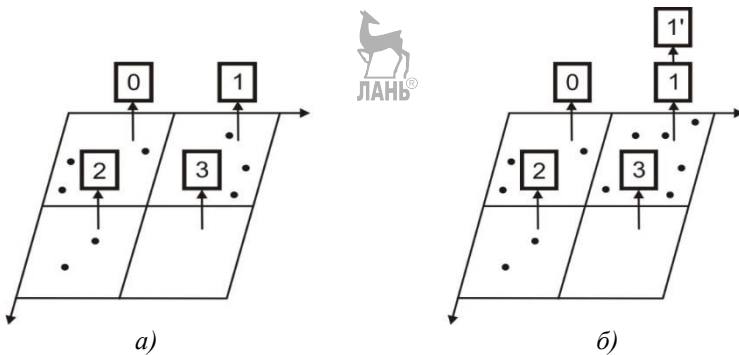


Рис. 2.33. Обработка переполнения: а – распределение записей по внешним блокам; б – объединение блоков в список

Если же число записей в индексе меньше порогового значения, то происходит выделение дополнительного внешнего блока под те объекты, которые не поместились в переполненный блок, и оба эти блока объединяются в линейный список. Результат данной операции представлен на рис. 2.33(б).

Объединение внешних блоков в список при неравномерном распределении может привести к появлению цепочек, состоящих более чем из двух элементов. Для поиска некоторых объектов необходимо будет сделать больше двух обращений к внешней памяти. Однако в среднем все же число обращений к внешней памяти для поиска объектов по точному совпадению в *MOLHPE* будет меньше двух.

Частичные расширения в хешировании MOLHPE

Рассмотрим процесс расщепления ячеек решетки *MOLHPE*, называемый *расширением решетки*. Данный алгоритм является многомерной модификацией одномерного линейного хеширования, которое предложил В. Литвин и в дальнейшем развил П. А. Ларсон [59]. Поэтому рассмотрение схемы расширений начнем с одномерного случая.

Одномерное линейное хеширование. Оригинальная схема линейного хеширования была предложена В. Литвиным в 1980 году [58]. В ней рассматривается файл как набор N блоков с адресами $0, 1, \dots, (N-1)$ от начала. Каждый блок может содержать M записей, попавших в соответствующую ячейку. При переполнении некоторого блока проверяется достижение порога наполнения. Если порог достигнут не был, то происходит размещение записей, вызвавших переполнение, в новом блоке и связывание этого блока с первоначальным в список (метод цепочек).

Если же произошло достижение порога наполнения файла, то выполняется расщепление блока P . При этом файл расширяется на одну страницу с адресом $(N+P)$ и функция хеширования меняется таким образом, чтобы часть записей блока P перешла в блок $(N+P)$. Пример расширения показан на рис. 2.34.

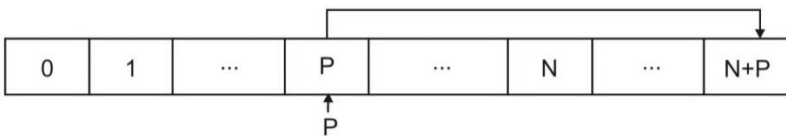


Рис. 2.34. Расширение в линейном хешировании

Последовательность расщепления задана заранее как $0, 1, \dots, (N-1)$, т. е. сначала расщепляется страница с номером 0, затем – с номером 1 и т. д. Переменная P , по сути, содержит адрес страницы, которая должна будет расщеплена следующей. Если все N страниц расщеплены и размер файла удвоился, то переменная P сбрасывается в 0, и процесс расщепления начинается заново.

Линейное хеширование с частичным расширением (LHPE). У предыдущей схемы есть один существенный недостаток. В результате расширений получается неравномерное распределение записей в файле. Те ячейки, которые были расщеплены недавно, заполнены приблизительно в два раза меньше тех, расщепление которых произошло давно. Это связано с тем, что при расщеплении ячейки P приблизительно половина ее записей перешла в ячейку $(N+P)$. Чтобы достичь более равномерного распределения, П. А. Ларсон предложил разделить блоки на группы, равные по числу элементов [59]. При расщеплении блока P ,

расщепляется каждый P -й блок всех групп. Если в файле было выделено k групп, то из каждой ячейки будет перенесено приблизительно M/k объектов. Это позволяет достичь более равномерного распределения объектов по файлу. Однако большое число групп может привести к замедлению работы хеширования. Во многих случаях достаточно число групп выбирать равное двум. Рассмотрим процесс расщепления для двух групп более подробно.

Первоначально файл содержит $2N$ блоков, где N – число полных расщеплений (расщепление называется полным, если в результате его выполнения произошло удвоение размера файла). При этом указатель на страницу, которая должна быть расщеплена следующей, установлен $P = 0$. Как и при линейном хешировании, указатель при каждом расщеплении увеличивается на 1, пока не достигнет значения N . После этого он сбрасывается снова в 0. Однако так как изначально записей в файле было $2N$, а не N , то при расщеплении N первых страниц объем файла увеличится всего в 1,5 раза. Поэтому данное расширение считается частичным, а не полным.

Пусть текущая расщепляемая страница равна P . При выполнении первого частичного расширения происходит выделение нового блока ($2N+P$) и перенос в него приблизительно $1/3$ записей из блока P и $1/3$ записей из блока $(N+P)$. Таким образом, получаем более равномерное распределение, чем при обычном расширении (рис. 2.35(а)).

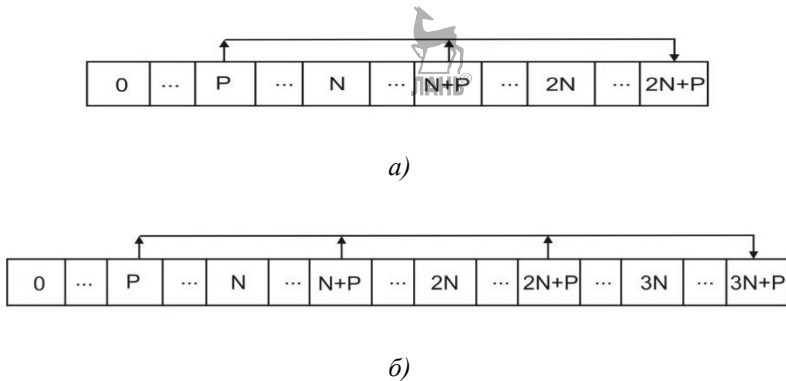


Рис. 2.35. Расширение в LHPE: а – первое частичное расширение; б – второе частичное расширение

Расщепление в процессе второго частичного расширения затрагивает уже не две, а три ячейки. При этом расщепление группы P вызывает перераспределение объектов ячеек P , $(N+P)$ и $(2N+P)$, как показано на рис. 2.35 (б).

Предположим, что в нашей схеме используется деление с двумя частичными расширениями. Обобщение алгоритма для произвольного числа частичных расширений не составит труда, хотя два частичных расширения являются наиболее простой и в то же время эффективной схемой.

a)

6)

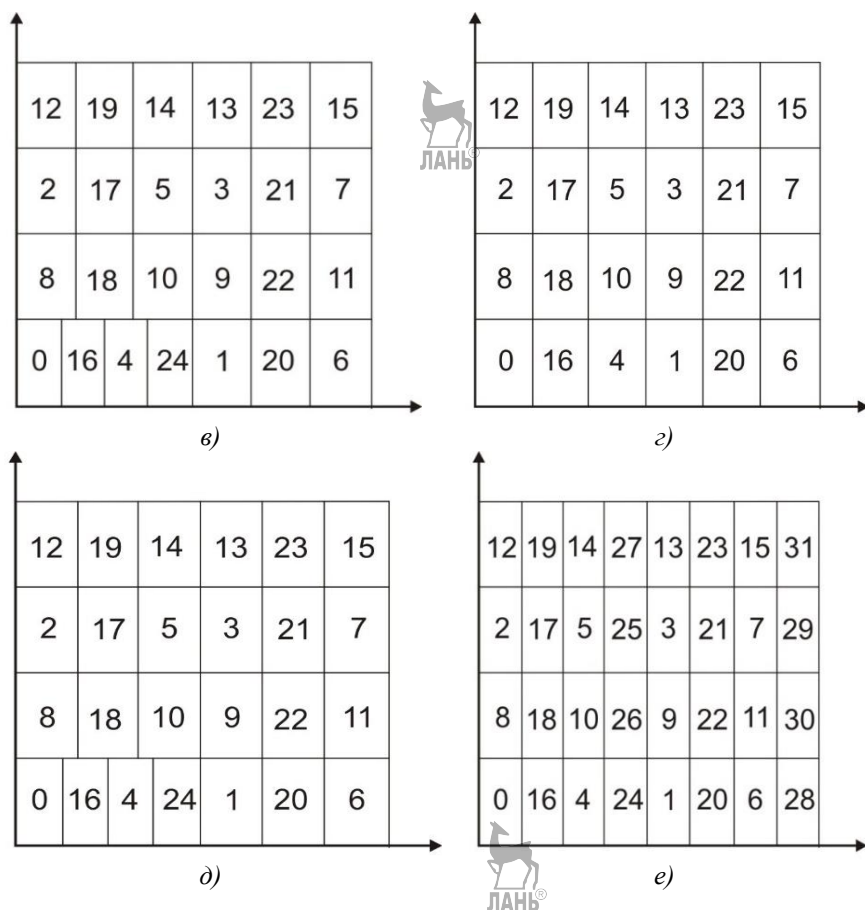


Рис. 2.36. Расширение решетки MOLHPE: а – начальное состояние решетки; б – первое расширение в первой фазе $P = [0, 0]$; в – второе расширение в первой фазе $P = [0, 1]$; г – состояние решетки после окончания первого частичного расширения; д – первое расширение во второй фазе $P = [0, 0]$; е – состояние решетки после окончания полного расширения

Для определения измерения, вдоль которого производится расширение в данный момент, используется следующая формула:

$$s = L \bmod d + 1,$$

где s – номер измерения для расширения;

L – число полных расширений;

d – число измерений пространства.

Для рассматриваемого примера на данном шаге необходимо использовать в качестве текущего измерения ось Ox (первое измерение).

Как и в *LHPE*, первое частичное расширение затрагивает всего две ячейки. Для определения номера этих ячеек используются следующие формулы:

$$G(l_1, l_2, \dots, l_a, \dots, l_d),$$

$$G(l_1, l_2, \dots, l_a + 2^{l_a - 1}, \dots, l_d),$$

где l_i – номер колонки или столбца по i -му измерению, в которой находится расширяемая ячейка;

d – число измерений пространства;

l_a – активное измерение (по которому расширяется решетка);

L_a – число полных расширений активного измерения;

G – адресная функция, рассмотренная в следующем разделе.

Указатель на ячейку, которая должна быть расширена следующей, хранится в переменной P . Однако если при одномерном хешировании этот указатель являлся номером блока, то в многомерном варианте он представляет собой массив из d чисел. Каждое число является индексом строки или столбца в решетке, которые идентифицируют расширяемую ячейку. В нашем примере изначально $P = [0; 0]$. При этом расширение должно затронуть ячейку $G(0, 0)$ и $G(0 + 2^{L_0 - 1}, 0)$. Согласно рис. 2.37(б), на котором представлена схема соответствия функции G , это ячейки с номерами 0 и 4. Данные ячейки расширяются еще одной ячейкой с номером $G(0 + 2^{L_0}, 0)$, и записи трех ячеек будут перераспределены между собой. Результат такого деления показан на рис. 2.36(б).

Сразу же после расширения указатель P должен быть перемещен к следующей ячейке. Однако сделать это не так просто, как в одномерном случае. Для перемещения указателя используется алгоритм, представленный в листинге 2.30.

Листинг 2.30

```
//=====
// Перемещение указателя на текущую расширяемую
// ячейку
// Параметры:
//   P – текущее значение указателя
//=====
СЛЕДУЮЩИЙ_УКАЗАТЕЛЬ(P)
[1] // Выбор измерения
    Если s ≠ 1, то
        i = 1
    Иначе
        i = 2
[2] // Переход к следующей ячейке по выбранному
    // измерению
    P.li = P.li + 1
    Если P.li = 2Li, то
```

```

P.li = 0
Если i = d, то
    i = s
Иначе Если i = s - 1, то
    i = s + 1
Иначе
    i = i + 1
Перейти к шагу 2
Конец СЛЕДУЮЩИЙ_УКАЗАТЕЛЬ

```

Суть данной процедуры заключается в обходе всех ячеек решетки. Расширение происходит последовательно, в порядке следования измерений (однако активное измерение s пропускается и меняется в самую последнюю очередь). Это позволяет поочередно пройти и расширить все ячейки решетки. К концу частичного расширения данная процедура снова приведет указатель P к начальному виду (т. е. для двумерного случая в конце частичного расширения указатель примет вид $P = [0, 0]$).

Согласно процедуре СЛЕДУЮЩИЙ_УКАЗАТЕЛЬ, после расширения ячейки, показанной на рис. 2.35(б), указатель P примет вид $P = [0, 1]$. Это означает, что при следующем расширении будет выполнено расщепление ячеек $G(0, 1)$ и $G(0+2^{L_0-1}, 1)$, т. е. блоков с номерами 2 и 5. По аналогии с прошлым расширением будет создана новая ячейка $G(0+2^{L_0}, 1)$, и записи этих ячеек будут перераспределены по трем блокам (рис. 2.35(в)).

Расширения ячеек будут продолжаться до тех пор, пока в решетке не останется ни одной нерасщепленной ячейки. При этом указатель P снова вернется к первоначальному состоянию $[0, 0]$. Данный факт будет означать, что первое частичное расширение закончено. Результирующая решетка после окончания первого частичного расширения показана на рис. 2.35(г). При этом общий объем индекса увеличился в 1,5 раза.

При одномерном линейном хешировании с частичными расширениями второе частичное расширение затрагивало уже не две, а три ячейки. При этом только приблизительно 1/3 записей из этих ячеек переходила во вновь созданный блок. Точно так же и в многомерном случае во втором расширении участвуют следующие три ячейки:

$$\begin{aligned}
 &G(l_1, l_2, \dots, l_a, \dots, l_d), \\
 &G(l_1, l_2, \dots, l_a + 2^{L_a-1}, \dots, l_d), \\
 &G(l_1, l_2, \dots, l_a + 2^{L_a}, \dots, l_d).
 \end{aligned}$$

Приблизительно по 1/3 записей из каждой из них переходит в новую ячейку, связанную с блоком:

$$G(l_1, l_2, \dots, l_a + 2^{L_a-1} + 2^{L_a}, \dots, l_d).$$

Если снова вернуться к рассматриваемому примеру, второе частичное расширение начнется с расщепления блоков:

$$G(0, 0) = 0,$$

$$\begin{aligned} G(0 + 2^1, 0) &= 4, \\ G(0 + 2^2, 0) &= 16. \end{aligned}$$

Часть записей каждого из этих блоков будет перенесена в блок с номером $G(0 + 2^1 + 2^2, 0) = 24$. Результат подобного расширения представлен на рис. 2.36(д).

Расщепление всех ячеек в процессе второго частичного расширения приведет к полному расширению. Размер индекса при этом увеличится в два раза по сравнению с первоначальным значением. Решетка после завершения полного расширения показана на рис. 2.36(е).

После завершения полного расширения поменяется активная ось пространства, и следующие расширения уже будут проводиться по второму ключу.

Адресная функция MOLHPE

Основным элементом любой схемы хеширования является адресная функция. Именно она вычисляет номер внешнего блока по ключам объекта. При рассмотрении хеширования *MOLHPE* будем предполагать, что значения всех ключевых полей объекта являются числовыми величинами в диапазоне от 0 до 1. Данный факт упростит понимание схемы в целом. Если это не так, то перевод к такому формату в большинстве случаев не составляет особого труда (с помощью деления всех ключей на максимально возможное значение по данной оси).

Адресная функция *MOLHPE* является сложной комбинацией более простых функций хеширования. В качестве составных частей она использует адресную функцию одномерного линейного хеширования с частичными расширениями, предложенную П. А. Ларсоном. Поэтому рассмотрение функций хеширования в данном разделе начнем с одномерного случая.

Адресная функция одномерного линейного хеширования.

Разработано большое число функций, но не все из них можно применять в структурах данных. Очень важным для индексирования является выбор такой функции, которая обеспечивает эффективное выполнение запросов поиска. Для этого желательно, чтобы функция хеширования сохраняла порядок следования записей во внешних блоках. Это означает, что все записи, расположенные в блоке n , должны иметь ключевое поле меньше, чем записи, расположенные в блоке $(n+1)$.

Чаще всего используют различные модификации следующей функции хеширования:

$$H(K) = \begin{cases} \sum_{j=0}^L b_j 2^j, & \text{если } \sum_{j=0}^L b_j 2^j < 2^L + P \\ \sum_{j=0}^{L-1} b_j 2^j, & \text{иначе} \end{cases},$$

где K – ключевое поле объекта в диапазоне от 0 до 1, в двоичном диапазоне имеющее вид $\sum b_j 2^{-(j+1)}$;

b_j – j -й бит ключа K ;

L – число расширений с момента создания файла (уровень файла, который показывает, сколько раз файл удваивался с момента создания);

P – номер блока, который должен быть расширен следующим.

Данная функция используется в одномерном линейном хешировании. При хешировании с частичными расширениями необходимо в формулу ввести зависимость от номера фазы (номера частичного расширения). Для схемы хеширования с двумя частичными расширениями адресная функция представлена в листинге 2.31.

Листинг 2.31

```
//=====
// Адресная функция хеширования LHPE для двух частичных
// расширений
// Параметры:
//   K – ключевое поле объекта ( $K = \sum b_j 2^{-(j+1)}$ )
//=====
ЛИНЕЙНЫЙ_АДРЕС (K)
[1] // Положение группы
    pos =  $\sum_{j=0}^{L-2} b_j 2^j$ 
[2] // Если положение записи до первого частичного
    // расширения
    Если (pos  $\geq$  P) И (EXT = 1), то
        Вернуть (pos + 2L - 1 * bL - 1)
[3] // Если положение записи после второго расширения
    Если (pos < P) И (EXT = 2)
        Вернуть (pos + 2L - 1 * bL - 1 + 2L * bL)
[4] // Если положение записи между первым и вторым
    // расширением
    remainder =  $K - \sum_{j=0}^{L-2} b_j 2^{-(j+1)}$ 
    q =  $\left\lfloor \frac{2 * remainder}{3} \right\rfloor$ 
    Вернуть (pos + 2L - 1 * q)
Конец ЛИНЕЙНЫЙ_АДРЕС
```

В процедуре вычисления адреса используется ряд предопределенных переменных. Эти переменные могут быть объявлены как глобальные для всей структуры или могут находиться в определенной ячейке памяти и передаваться как параметры в функцию:

L – число полных расширений индекса. Каждое полное расширение состоит из двух частичных расширений. В результате полного расширения размер файла удваивается в два раза. Также можно вывести

формулу зависимости размера файла от параметра L . В момент завершения полного расширения файл содержит ровно 2^L записей;

P – указатель на блок индекса, который должен быть расширен следующим. В адресной функции этот параметр используется для того, чтобы определить, затронуло ли первое или второе частичное расширение блок с нужным ключом. Это необходимо делать, потому что вычисление адреса будет отличаться для разных фаз расширения;

EXT – номер частичного расширения, в котором находится сейчас индекс. В нашем варианте с двумя частичными расширениями эта переменная может принимать значение 1 или 2.

Процедуру листинга 2.31 можно условно разделить на три части. Первая часть соответствует случаю, когда запись расположена в ячейке, которую не коснулось еще первое частичное расширение. Это актуально для первого частичного расширения ($EXT = 1$) и тех записей, которые находятся в блоках с номерами большими или равными P . Номер блока в этом случае зависит от первых L бит ключевого поля. Формула вычисления имеет следующий вид

$$pos = \sum_{j=0}^{L-1} b_j 2^j.$$

Вторая часть процедуры соответствует ситуации, при которой для заданного блока уже было проведено второе частичное расширение (номер блока меньше P и в данный момент индекс находится во втором частичном расширении). При этом определения позиции необходимо использовать на один бит больше из ключевого поля. Формула примет вид

$$pos = \sum_{j=0}^L b_j 2^j.$$

Самая последняя ветвь алгоритма соответствует случаю, при котором блок уже был расширен в процессе первого частичного расширения, но второе частичное расширение до него еще не дошло. В этом случае остаток ключевого поля (часть, которая начинается с $(L-1)$ бита) анализируется на попадание в одну из трех групп, и по его значению определяется позиция блока с записью.

Адресная функция MOLHPE. В MOLHPE используется функция хеширования G , которая зависит от рассмотренной ранее функции одномерного линейного хеширования с частичными расширениями. В данной схеме предполагается, что все ключевые поля имеют одинаковый приоритет и расширяются поочередно. Это является существенным ограничением. Поэтому на практике для ситуаций с разным приоритетом ключей необходимо использовать другие схемы хеширования.

Как и для одномерного хеширования, в многомерном случае есть понятие уровня файла L . Уровень файла – число полных расширений по всем измерениям. Каждое полное расширение по любому измерению увеличивает значение L на 1, а размер файла – в два раза. Нетрудно заметить, что если расширения по осям чередуются, то значение L всегда больше числа расширений по конкретным осям пространства. Для вычисления числа полных расширений по некоторому j -му измерению можно воспользоваться следующей формулой:

$$L_j = \begin{cases} \left\lfloor \frac{L}{d} + 1 \right\rfloor, & \text{если } j \in \{0, 1, \dots, s-1\} \\ \left\lfloor \frac{L}{d} \right\rfloor, & \text{если } j \in \{s, \dots, d-1\} \end{cases}.$$

В формуле значение s – текущая ось пространства, вдоль которой происходит расширение в настоящий момент времени. Как было показано ранее, ее можно определить с помощью формулы:

$$s = L \bmod d + 1.$$

В хешировании *MOHPE* используется адресная функция G , впервые примененная Е. Д. Отто в 1984 году для хеширования *МЕН* [69]. Данная функция вычисляет номер внешнего блока, основываясь только на ключах объекта. Результатом работы функции является число от 0 до $(2^L - 1)$. Однако в *МЕН* было организовано хеширование с каталогом, а в данном контексте эта функция используется для адресации блоков в схеме без каталога.

Функция G в качестве своих параметров принимает коэффициенты, полученные в результате одномерного хеширования отдельных ключей записи, т. е. сначала все ключи объекта хешируются с помощью процедуры, показанной в листинге 2.32.

$$l_j = \text{ЛИНЕЙНЫЙ_АДРЕС}(K_j), \quad j = 0, 1, \dots, d-1.$$

В результате получаем некоторый набор чисел l_j , который и подставляется в саму функцию G :

$$G(l_0, l_1, \dots, l_{d-1}) = \begin{cases} l_z \prod_{j \in M} J_j + \sum_{j \in M} c_j l_j, & \text{если } \max(l_0, l_1, \dots, l_{d-1}) \neq 0 \\ 0, & \text{иначе} \end{cases},$$

где z – максимальный уровень по ключам записи:

$$z = \max\{j \in \{0, 1, \dots, d-1\} \mid \lfloor \log_2 l_j \rfloor = \max_{0 \leq k \leq d-1} \{\lfloor \log_2 l_k \rfloor\}\};$$

M – весь набор измерений, кроме z :

$$M = \{0, 1, \dots, d-1\} \setminus \{z\};$$

t – уровень измерения z для хешируемой записи:

$$t = \lfloor \log_2 l_z \rfloor;$$

J_j – число блоков до искомого в j -м измерении:

$$J_j = \begin{cases} 2^{t+1}, & \text{если } j < z; \\ 2^t, & \text{иначе} \end{cases};$$

c_j – число дополнительных блоков в j -м измерении:

$$c_j = \prod_{\substack{r=j+1, \\ r \neq z}}^{d-1} J_r.$$

Пример расчета функции G для решетки пространства размером 4×4 показан на рис. 2.37.

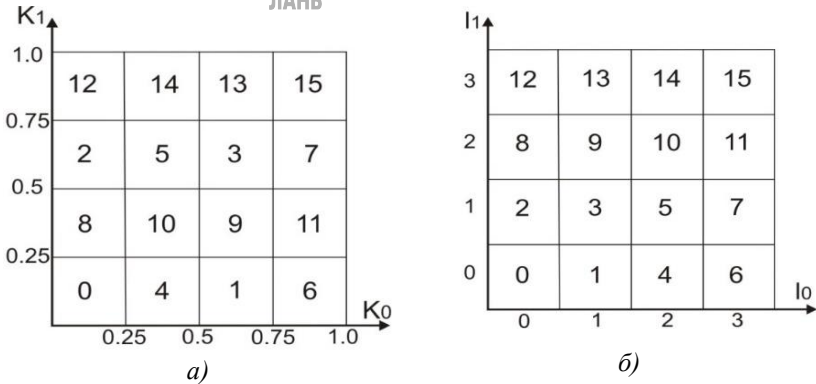


Рис. 2.37. Значения функции G : а – зависимость функции G от ключевых полей; б – зависимость функции G от индексов по осям

Слева (рис. 2.37(а)) показана зависимость вычисленных адресов функцией G от ключевых полей объекта, а справа (рис. 2.37(б)) – зависимость функции от индексов, вычисленных с помощью процедуры ЛИНЕЙНЫЙ_АДРЕС.

Алгоритм поиска объекта

Для пояснения принципов поиска записей в индексе рассмотрим один из самых простых типов поисков – поиск записи по точному совпадению всех ключей. Данный вид поиска хорошо иллюстрирует основные принципы работы схемы, при этом он избавлен от сложностей реализации, способных затруднить общее понимание метода.

Пусть дан некоторый точечный объект P . Необходимо ответить на вопрос, присутствует ли данный объект в индексе. Алгоритм, решающий поставленную задачу для двумерного случая, показан в листинге 2.32.

Листинг 2.32

```
//=====
// Поиск точечного объекта Р в индексе
// Параметры:
//   Р – объект поиска
//=====
ПОИСК (Р)
```

```

[1] // Вычисление адреса блока
    10 = ЛИНЕЙНЫЙ_АДРЕС (K0(P))
    11 = ЛИНЕЙНЫЙ_АДРЕС (K1(P))
    h = G(10, 11)
[2] // Проверка блока с номером h
    В = внешний блок с номером h
    Для всех объектов P' из блока В проверить
        Если P' = P, то
            Выйти из процедуры и вернуть P'
[3] // Проверка следующего блока в цепочке
    Если В не последний блок в цепочке блоков, то
        h = номер следующего блока в цепочке после В
        Перейти к шагу 2
    Иначе
        Выйти из процедуры и вернуть NULL
Конец ПОИСК

```

Данная процедура возвращает найденный объект, если он присутствует в индексе. Если объект найден не был, то процедура возвращает *NULL*, что является флагом неудачного поиска.

Поясним работу процедуры на примере. Пусть у нас дано некоторое распределение объектов, показанное на рис. 2.38. При этом задано ограничение на максимальное число объектов в одном внешнем блоке, равное 3. При таких параметрах все точечные объекты, находящиеся в одной ячейке с объектом *P*, не удастся разместить в одном внешнем блоке и в индексе появится цепочка блоков для данной ячейки.

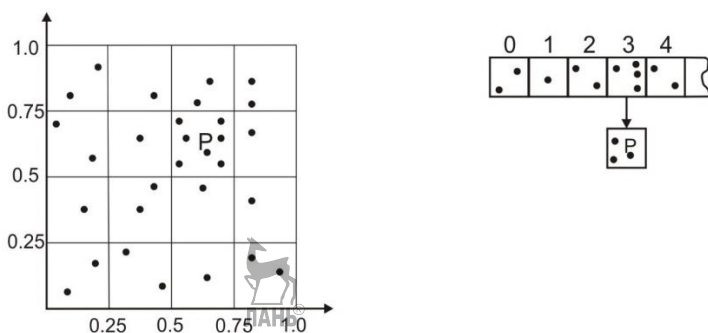


Рис. 2.38. Пример поиска объектов в индексе

Рассмотрим процедуру поиска объекта *P* с ключами (0,6; 0,7). На первом шаге происходит вычисление внешнего блока, в котором может находиться данный объект. Для этого каждый ключ объекта подставляется в адресную функцию:

$$I_0 = \text{ЛИНЕЙНЫЙ_АДРЕС}(0,6) = 1;$$

$$I_1 = \text{ЛИНЕЙНЫЙ_АДРЕС}(0,7) = 1.$$

Получив линейный адрес каждого ключа в отдельности, все коэффициенты подставляются в общую функцию хеширования для вычисления номера внешнего блока. Согласно данным зависимости номера блока от линейных адресов по шкалам, представленным на рис. 2.37(б), получаем:

$$h = G(1, 1) = 3.$$

На втором шаге алгоритма происходит загрузка внешнего блока с номером 3 в оперативную память и проверка всех его объектов на соответствие поисковой записи. В рассматриваемом нами случае в блоке 3 нет записи с заданными ключами, поэтому данный перебор не даст положительных результатов.

На третьем шаге алгоритма происходит переход к следующему блоку цепочки. Если в цепочке непроверенных блоков больше нет, то делается вывод об отсутствии поисковой записи в индексе. Цепочка 3 содержит еще один внешний блок. Именно он принимается за текущий проверяемый блок, и происходит переход ко второму шагу алгоритма. При этом снова происходит загрузка этого блока в оперативную память и проверка всех его объектов. На этот раз такая проверка даст положительный ответ, и процедура вернет найденный объект в качестве результата своей деятельности.

Алгоритм добавления нового объекта

Частично алгоритм добавления объекта в индекс уже был описан в предыдущих параграфах. Этот процесс полностью базируется на принципах разрешения коллизий с помощью построения цепочек блоков и линейным увеличением решетки с частичными расширениями при достижении некоторого порогового значения по проценту заполнения. Данный алгоритм представлен в листинге 2.33.

Листинг 2.33

```
//=====
// Добавление нового объекта в индекс
// Параметры:
//   Р - добавляемый объект
//=====
ДОБАВИТЬ (Р)
[1] // Вычисление адреса блока, который должен
    // содержать Р
    10 = ЛИНЕЙНЫЙ_АДРЕС (К0 (Р))
    11 = ЛИНЕЙНЫЙ_АДРЕС (К1 (Р))
    h = G(10, 11)
[2] // Проверка блока с номером h
    В = внешний блок с номером h
    Для всех объектов Р' из блока В проверить
        Если Р' = Р, то
            Выйти из процедуры, Р уже в индексе
```

```

[3] // Проверка следующего блока в цепочке
    Если В не последний блок в цепочке блоков, то
        h = номер следующего блока в цепочке
        Перейти к шагу 2
[4] // Вставка объекта Р
    n = количество объектов в блоке В
    Если n < максимально возможного количества, то
        Добавить Р в блок В
    Иначе
        В' = новый пустой внешний блок
        Добавить В' в цепочку после блока В
        Добавить Р в блок В'
[5] // Проверка на необходимость расширения
    К = коэффициент заполнения
    Если  $K \geq K_{\text{порог}}$ , то
        РАСШИРИТЬ_ИНДЕКС()
Конец ДОБАВИТЬ

```

Представленная процедура первыми тремя шагами очень похожа на алгоритм из листинга 2.32. Точно так же, как и в предыдущем примере, происходит определение номера внешнего блока, который должен содержать объект P (шаг 1), и поиск этого объекта в цепочке блоков (шаги 2 – 3). Если объект P будет найден в цепочке блоков, то вставку производить не нужно, так как P уже был проиндексирован ранее. Если же соответствие найдено не будет, то происходит включение P в индекс. Для этого на четвертом шаге проверяется возможность включения P в последний блок цепочки. Если блок не переполнен, то в него вставляют данный объект, иначе – добавляют еще один пустой блок в цепочку, который будет местом хранения P .

Добавление нового объекта в индекс может изменить процент заполнения структуры так, что придется выполнять некоторое частичное расширение. Для этого на пятом шаге процедуры ведется проверка коэффициента заполнения и сравнение его с пороговым значением. Коэффициент заполнения легко рассчитывается по формуле

$$K = \frac{[\text{Количество объектов в индексе}][\text{Размер объекта}]}{[\text{Общий размер индекса}]}$$

Если порог был превышен, то индекс расширяется на одну новую ячейку. Процесс расширения был подробно описан ранее.

Алгоритм удаления объекта

Удаление объектов из индекса является операцией, полностью противоположной добавлению. На первых шагах процедуры удаления происходит точно такой же поиск, как и в листинге 2.32. Если объект не был найден, то процедура завершается, так как удаляемый объект

отсутствует в индексе. Если же объект удастся обнаружить, то он просто исключается из внешнего блока B и удаляется из памяти.

После удаления объекта из внешнего блока необходимо переформатировать всю цепочку, в которой был найден объект. Если в начале цепочки внешних блоков есть не до конца заполненные элементы, то в них переносятся объекты из последнего блока цепочки. Это позволяет увеличить скорость поиска объектов в дальнейшем.

Если после переформатирования цепочки блоков некоторый блок окажется пустым, то он удаляется из индекса. Однако это не касается самого первого блока цепочки, на который ссылается индекс. Цепочка должна состоять как минимум из одного внешнего блока, даже если в этом блоке нет ни одного объекта.

На последнем шаге удаления необходимо проверить коэффициент заполнения индекса и сравнить его с нижним порогом слияния. Если коэффициент заполнения окажется меньше порога, то необходимо проделать операцию, обратную расширению ячеек – слияние. В процессе этой операции индекс уменьшается на одну ячейку и происходит объединение двух цепочек внешних блоков в одну.

Развитие идеи MOLHPE для неравномерного распределения

Многочисленные эксперименты показали, что хеширование MOLHPE при неравномерном распределении данных в пространстве сильно уменьшает свою производительность. Так, число доступов к диску для операций поиска может значительно превосходить аналогичный показатель для файлов-решеток. Это ограничивает область применения данной структуры только приложениями с равномерным распределением данных.

Однако существует ряд модификаций, развивающих идеи MOLHPE и позволяющих индексировать неравномерные данные почти с такой же эффективностью, как и равномерные. Одной из таких модификаций является применение техники квантилей.

MOLHPE разбивает пространство поиска решеткой, используя равноудаленные точки по каждой оси. Таким образом, схема является очень эффективной при равномерном распределении, так как в этом случае вероятность значительного переполнения какой-либо одной ячейки очень мала. Однако чем дальше распределение от равномерного, тем хуже будет производительность структуры. Вполне очевидным фактом является наблюдение, что преодолеть этот недостаток можно с помощью выбора точек деления, зависящих от конкретного распределения объектов.

Рассмотрим практический пример. Допустим, имеется такое распределение объектов, при котором большая их часть расположена в левом нижнем углу. Понятно, что при равноудаленных точках деления пространства ячейки, расположенные в правой и верхней частях, будут

практически пустыми, в то время как ячейки левой нижней части будут содержать длинные цепочки внешних блоков.

Если использовать для данной ситуации метод квантилей, то пространство будет делиться несколько иначе. Допустим, нам известны функции распределения объектов по осям (f_0 для нулевого ключа и f_1 – для первого). Теперь предположим, что заполнение индекса начинается с пустого файла, и наступил момент, когда необходимо расширить единственную ячейку. В качестве деления выбирается измерение, соответствующее ключу K_0 . Если известна функция распределения f_0 , то делить пространство по середине является малоэффективным. Лучше выбрать такой вариант, при котором приблизительно половина записей останется в одной ячейке, а вторая половина – перейдет во вновь созданную. Допустим, такое распределение соответствует 1/2 квантиля (рис. 2.39(а)).

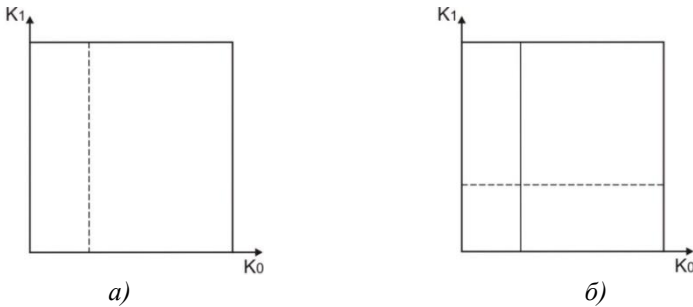


Рис. 2.39. Применение метода квантилей: а – первое деление по оси K_0 ; б – второе деление по оси K_1

При втором расщеплении уже используется другое измерение пространства, однако принцип остается прежним.

В результате использования метода квантилей после полного расширения будет создано четыре ячейки пространства, разных по размеру, но с приблизительно равным числом содержащихся в них объектов (рис. 2.39(б)).

На рис. 2.40 показан файл уровня $L = 4$ при завершении полного расширения, где каждая ось была разделена в квантилях 1/4, 1/2 и 3/4.

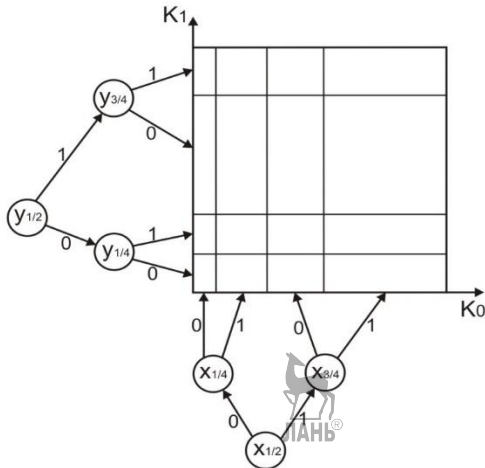


Рис. 2.40. Пример файла уровня $L = 24$

Как изображено на рисунке, точки раздела хранятся в бинарных деревьях, которые имеют небольшой объем и могут быть сохранены в оперативной памяти.

Данный метод является очень эффективным средством индексирования неравномерных распределений. В дальнейшем он был развит для динамического формирования точек деления, в котором распределение данных по осям заранее неизвестно.

2.3.4. Многомерное линейное хеширование с сохранением порядка - PLOP

В предыдущих подпараграфах были описаны два подхода к реализации многомерного хеширования – с применением дополнительной структуры (файлы-решетки и *EXCELL* [68, 42, 45, 86]) и без нее (*MOLHPE* [51]). У каждой из них есть свои достоинства и недостатки. К сожалению, не существует схемы с идеальными показателями для абсолютно всех случаев. Поэтому работа над созданием новых структур и алгоритмов продолжается постоянно.

В 1988 году была предложена новая схема хеширования, в которой Г. Кригел и Б. Сигер попытались устранить недостатки предыдущих методов. Новый алгоритм получил название PLOP хеширование (*Piecewise Liner Order Preserving hashing*) [53]. Основными принципами, которыми руководствовались разработчики, было устранение следующих недостатков предыдущих методов.

- Размер каталога в файле-решетке составляет $O(n^{1+(d-1)/d*b})$ при равномерном распределении записей в пространстве (d – размерность пространства, b – размер блока, n – число записей в индексе). Если же

распределение записей является неравномерным и имеет ярко выраженные области концентрации, то размер может быть еще больше. В этом случае методы хеширования без каталогов имеют явное преимущество.

- Расширение каталога при переполнении в файлах-решетках является очень дорогостоящей операцией. Оно может потребовать $O(N^{1-1/d})$ операций доступа к диску (где N – число блоков в решетке). Поэтому разработчики нового метода предпочли метод линейного расширения, использующийся в *MOLHPE*.

- Функция хеширования *MOLHPE* и принцип расширения приводят к равномерному расширению решетки. Однако при наличии ярких областей сосредоточения объектов приходится расширять полупустые или даже абсолютно пустые блоки, в то время как переполненные участки выстраиваются в цепочки большой длины. Ситуацию немного улучшает применение квантилей [52], однако полностью устранить данный недостаток в *MOLHPE* оказалось невозможно. Файлы-решетки оказываются в более выгодном положении, так как в них при переполнении точка нового расщепления продиктована наличием переполненных областей, а не некоторым заранее заданным порядком, зависимым от предопределенной функции.

Разработчики попытались взять самое лучшее от двух методов и построить схему хеширования, свободную от перечисленных недостатков. Разработанный ими метод является хешированием без каталога, поэтому в нем нет дорогостоящих операций деления пространства и лишних затрат памяти на хранение массива решетки. Однако в нем впервые был применен алгоритм произвольного выбора деления группы переполненных блоков, независимый ни от каких предопределенных функций.

Общие принципы хеширования *PLOP*

В первом приближении хеширование *PLOP* похоже на хеширование *MOLHPE*. Все пространство поиска разбивается на ячейки некоторой решеткой, которая в процессе добавления и изменения индексированных данных увеличивается линейным способом. Причем для увеличения коэффициента заполнения пространства используется принцип линейного расщепления с частичными расширениями, предложенный П. А. Ларсон для одномерного случая [59]. С каждой ячейкой решетки связана цепочка внешних блоков, содержащих попавшие в ячейку объекты. Адресная функция хеширования в *PLOP* также была позаимствована из *MOLHPE*.

Однако *MOLHPE* имеет ряд ограничений и недостатков, особенно в неравномерных распределениях. Некоторые из них не характерны файлам-решеткам – индексным методам со шкалами и каталогом. Поэтому разработчики, взяв за основу *MOLHPE*, ввели в него ряд

элементов схем с каталогами, что позволило повысить эффективность схемы на разных распределениях данных.

Первое изменение коснулось линейных шкал по осям пространства. В оригинальном описании алгоритма *MOLHPE* шкала является линейной. Деления по осям рассчитываются по некоторой формуле и не зависят от реального распределения данных в пространстве. Этот метод хорошо подходит для равномерного распределения данных в пространстве, но он абсолютно неприменим во всех остальных случаях. Поэтому в *PLOP* были предложены произвольные деления. Это накладывает ряд ограничений и заставляет выделять память для хранения шкал. Однако произвольные деления по шкалам имеют несравненный плюс – появляется возможность строить шкалы по осям пространства, зависящие от данных, а не от некоторой математической формулы. Именно так ведут себя файлы-решетки.

Еще одно изменение в хешировании *PLOP* – это порядок расщеплений. Во всех схемах хеширования без каталога, существовавших до этого метода, применялся жесткий порядок расщеплений. Это означает, что последовательность блоков, в которой они должны расщепляться, была жестко задана. Это неудобство вытекало из адресной функции хеширования. Дело в том, что, применяя некоторую математическую формулу непосредственно над данными, мы получаем индекс блока внешней памяти, в котором должны размещаться те или иные данные. Поэтому и расщепление блоков нужно проводить так, чтобы в индексе существовали все те блоки, которые может выдать адресная функция непосредственно над данными.

В хешировании *PLOP* адресная функция обрабатывает не данные, а индексы шкал по осям. Поэтому появляется возможность произвольного расщепления блоков, зависящее от переполненных ячеек, а не от предопределенной последовательности.

Адресная функция

Для начала рассмотрим адресную функцию хеширования *PLOP*. Так как в этой схеме применяется линейное расщепление блоков с частичным расширением, то функция хеширования будет зависеть от L – уровня файла, который показывает, сколько раз файл был увеличен в 2 раза. Ось s , относительно которой производится следующее расширение, меняется циклически. Ее всегда можно рассчитать, используя формулу:

$$s = L \bmod d + 1.$$

Как уже было отмечено, адресная функция хеширования *PLOP* позаимствована из хеширования *MOLHPE*. Она точно так же является сложной составной величиной. Однако в ее использовании скрыто одно из основных отличий. В хешировании *MOLHPE* в адресную функцию G подставлялись коэффициенты, полученные после применения функций одномерного линейного хеширования, предложенных В. Литвиным

и П. А. Ларсоном. В методе PLOP хеширования одномерные схемы не используются. Индексы для функции G получаются непосредственно из шкал по осям пространства. Это позволяет строить произвольные расщепления, зависящие только от тех индексов, которые получены из шкал.

При этом сама адресная функция хеширования не претерпела каких-либо серьезных изменений. Изменился только первый шаг – получение коэффициентов для функции G . Как и в других схемах, помимо понятия уровня файла (L – количество полных расширений по всем направлениям), можно выделить и понятие уровня по определенной оси:

$$L_j = \begin{cases} \left\lfloor \frac{L}{d} + 1 \right\rfloor, & \text{если } j \in \{0, 1, \dots, s-1\} \\ \left\lfloor \frac{L}{d} \right\rfloor, & \text{если } j \in \{s, \dots, d-1\} \end{cases}.$$

Эта величина показывает, сколько раз происходило полное расширение по каждой из осей пространства. Нетрудно заметить, что если расширения по осям чередуются, то значение L всегда больше числа расширений по конкретным осям пространства.

Сам индекс, который соответствует номеру внешнего блока с объектом поиска, можно рассчитать по следующим формулам:

$$G(i_0, i_1, \dots, i_{d-1}) = \begin{cases} i_z \prod_{j \in M} J_j + \sum_{j \in M} c_j i_j, & \text{если } \max(i_0, i_1, \dots, i_{d-1}) \neq 0 \\ 0, & \text{иначе} \end{cases},$$

где z – максимальный уровень по ключам записи:

$$z = \max\{j \in \{0, 1, \dots, d-1\} \mid \lfloor \log_2 i_j \rfloor = \max_{0 \leq k \leq d-1} \{\lfloor \log_2 i_k \rfloor\}\};$$

M – весь набор измерений кроме z :

$$M = \{0, 1, \dots, d-1\} \setminus \{z\};$$

t – уровень измерения z для хешируемой записи

$$t = \lfloor \log_2 i_z \rfloor;$$

J_j – число блоков до искомого в j -м измерении

$$J_j = \begin{cases} 2^{t+1} & \text{если } j < z; \\ 2^t & \text{иначе} \end{cases};$$

c_j – число дополнительных блоков в j -м измерении

$$c_j = \prod_{\substack{r=j+1 \\ r \neq z}}^{d-1} J_r.$$

Как видно, данный вариант адресной функции практически ничем не отличается от той, которая использовалась для хеширования *MOLHPE* ранее.

Главная особенность функции G заключается в том, что она позволяет разделить некоторую область пространства данных на две части. Это в точности тот самый процесс, который происходит в файле-решетке при расширении каталога. Однако в отличие от

файлов-решеток процесс деления происходит не за одну итерацию, а с помощью процедуры линейного расширения, что позволяет сделать операцию расщепления простой и эффективной в вычислительном плане.

Шкалы по осям пространства



Как было описано, все многомерное пространство данных покрыто ортогональной сеткой. Чтобы можно было построить такую сетку, необходимо выбрать принцип формирования шкал по каждой оси. В хешировании *PLOP* разделяющие точки определяются бинарными деревьями. Для каждой оси строится свое дерево, поэтому в итоге получаем d бинарных деревьев (где d – размерность пространства).

Каждый внутренний узел дерева-шкалы содержит в себе точку на оси, представляющую собой $(d-1)$ -мерную гиперплоскость, которая разделяет пространство на две прямоугольные области. Каждый лист ассоциируется с d -мерной областью пространства $S(i,j)$, окруженной двумя соседними разделяющими гиперплоскостями (i – номер гиперплоскости от 0 до m (где m – число областей на данной оси), j – измерение пространства). Таким образом, получаем, что каждая область $S(i,j)$ адресуется некоторым целочисленным индексом i , хранящимся в соответствующем листе, $0 \leq i < m$, $0 \leq j < d$. Пример деления пространства показан на рис. 2.41.

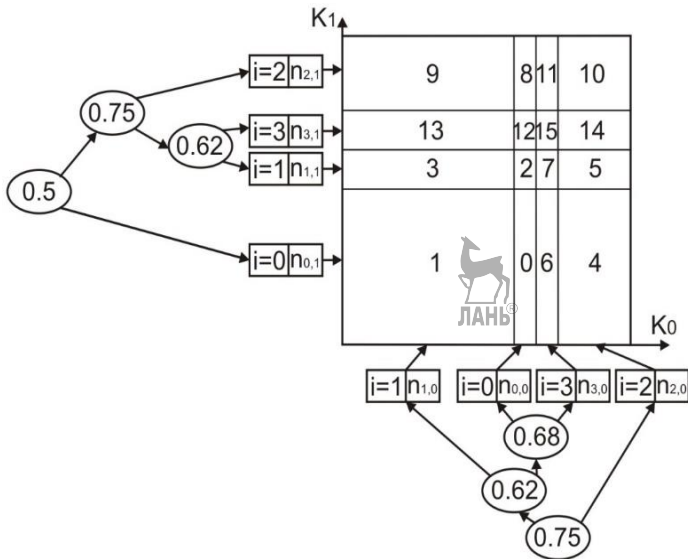


Рис. 2.41. Пространство данных, образованное решеткой *PLOP*

Все пространство данных представляет собой набор d -мерных прямоугольников, которые получились в результате деления гиперплоскостями шкал. Как и в других схемах, все d -мерные точки, находящиеся в одной ячейке, хранятся в одной странице. Адрес этой страницы вычисляется с использованием описанной ранее адресной функции G , в которую как раз и подставляются номера i всех областей $S(i,j)$, чье пересечение образует соответствующую ячейку. В этом и заключается отличие адресной функции $PLOP$ от $MOLHPE$ (в $MOLHPE$ в адресную функцию подставляются результаты линейного хеширования отдельных ключей с помощью формулы Ларсона).

Кроме индекса i каждый лист содержит n_{ij} – количество точек в области $S(i,j)$. Эта информация используется для расширения файла при переполнениях.

Для наглядности на рис. 2.42(а) показаны адреса страниц пространства данных в зависимости от индексов i_j , а на рис. 2.42(б) – адреса внешних блоков, полученные для распределения рис. 2.42.

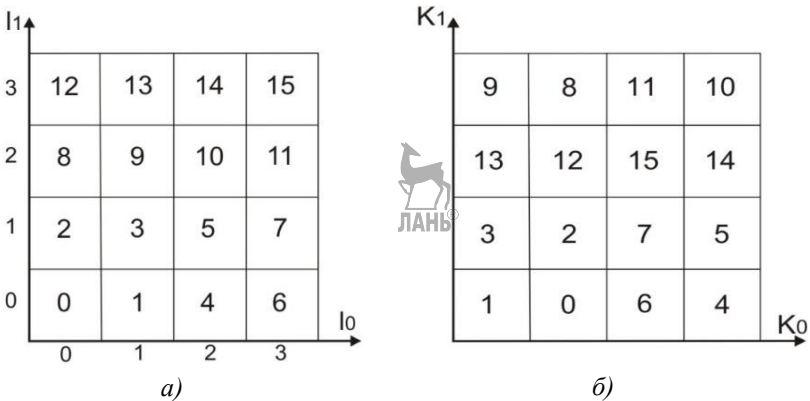


Рис. 2.42. Пример распределения номеров блоков: а – зависимость от индексов; б – зависимость реального распределения

Расширение решетки при переполнении

Динамическое поведение $PLOP$ хеширования при переполнениях и расщеплениях решетки лучше объяснить на примере. Пусть дан файл с уровнем $L = 2$, $s = 1$, хранящий двухмерные записи. Так как $L = 2$, файл состоит из 4 страниц (рис. 2.43).

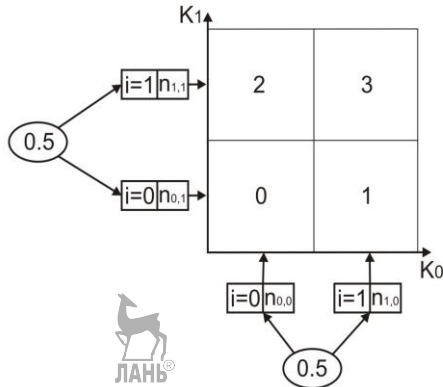


Рис. 2.43. Файл уровня 2

Пусть в некоторый момент произошло переполнение блоков и появилась необходимость расщепления решетки. Контрольная функция расщепления может быть разработана в соответствии с конкретной ситуацией, например, можно следить за показателем использования памяти структурой или максимальной длиной цепочек блоков.

Когда появляется необходимость расширить решетку, функция расщепления выбирает на текущей оси ту область $S(i, j)$, которая содержит максимальное число элементов. Эта операция является тривиальной, так как в листовых узлах бинарного дерева каждой шкалы, помимо номера i , хранится число записей в данной области.

Если уровень файла $L = 2$ (рис. 2.43), то текущей осью расширения является нулевое измерение, т. е. ось Ox . Допустим, максимально число записей оказалось в первой области. Тогда будет расширена $S(1, 0)$. В этом случае добавится новая разделяющая точка в бинарное дерево оси Ox (рис. 2.44(a)). Пусть это будет середина отрезка ($x = 0,75$). Так как данная схема расширяет область линейно, страница за страницей, время вставки ограничивается временем, которое нужно для расширения файла на одну страницу. Однако при следующем переполнении расширения будут происходить по этой же точке до тех пор, пока все ячейки области $S(1, 0)$ не будут расщеплены (рис. 2.44(a)).

После полного расщепления области $S(1, 0)$ расщепление продолжится по оси Ox . Это обусловлено тем, что расщепление данной области не привело к удвоению количества областей по данной оси. Поэтому полное расширение по оси нельзя считать завершенным.

При следующем переполнении появится необходимость снова выбрать область для расщепления. И опять же ее можно будет выбрать в зависимости от конкретной ситуации в файле, а не согласно жесткому предопределенному порядку.

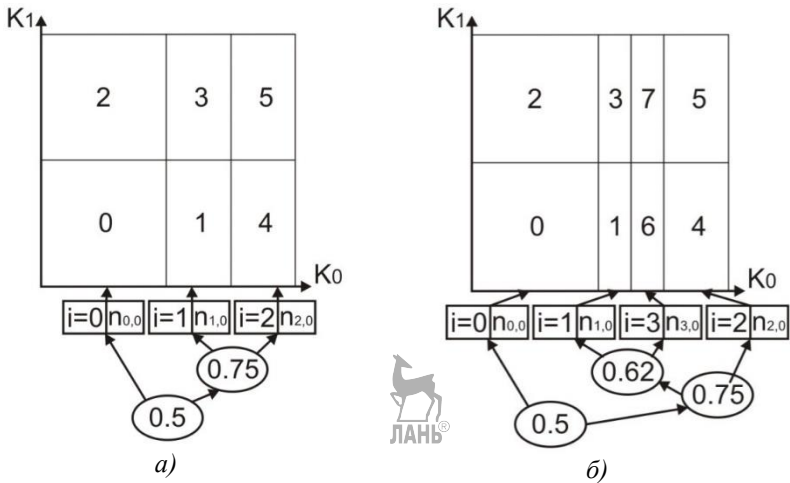


Рис. 2.44. Расширение файла: а – первое расширение;
б – второе расширение

Допустим, первая область снова оказалась самой переполненной и для расщепления снова будет выбрана $S(1,0)$. И пусть снова в качестве точки деления будет выбрана середина отрезка. Тогда в дереве шкалы оси Ox появится новая вершина 0,625 (рис. 2.44(б)).

После завершения деления области $S(1,0)$ количество областей на оси Ox удвоится. Поэтому следующие расширения будут проводиться вдоль оси Oy .

Сжатие решетки при удалении объектов

Если записи в файл не добавляются, а удаляются, то наступит момент, когда появится необходимость слияния некоторых блоков. Если слияние не производить, то процент использования памяти структурой может понизиться, и это негативно скажется на производительности алгоритмов.

Для сжатия используют ту же самую ось, что и для расширения. Формула вычисления номера измерения была приведена в данном разделе ранее. На выбранной оси определяется пара соседних областей с наименьшим числом объектов. Если показатель заполнения обеих областей меньше некоторого порогового значения, то необходимо производить сжатие.

Для примера в качестве исходного возьмем файл, показанный на рис. 2.44(б). Допустим, самыми редко заполненными оказались области с индексами 0 и 1. При этом сначала модифицируется бинарное дерево,

т. е. объединяются два листа с индексами 0 и 1, а соответствующая разделяющая точка 0.5 удаляется (рис. 2.45(а)).

Затем производится первый шаг слияния: страницы 2 и 3 на рис. 2.44(б) объединяются. При этом записи страницы 2 переносятся в страницу 3. На этом же шаге происходит перемещение записей последней страницы (страница 7) в освободившуюся вторую страницу. Таким образом, можно гарантировать, что в процесс слияния будет вовлечено максимум три страницы и при этом размер индекса уменьшится на одну страницу (рис. 2.45(а)).

Второй шаг завершает слияние. Объединяются страницы 0 и 1, а также записи из последней 6-й ячейки переносятся в освобожденную нулевую. В результате получим решетку, показанную на рис. 2.45(б).

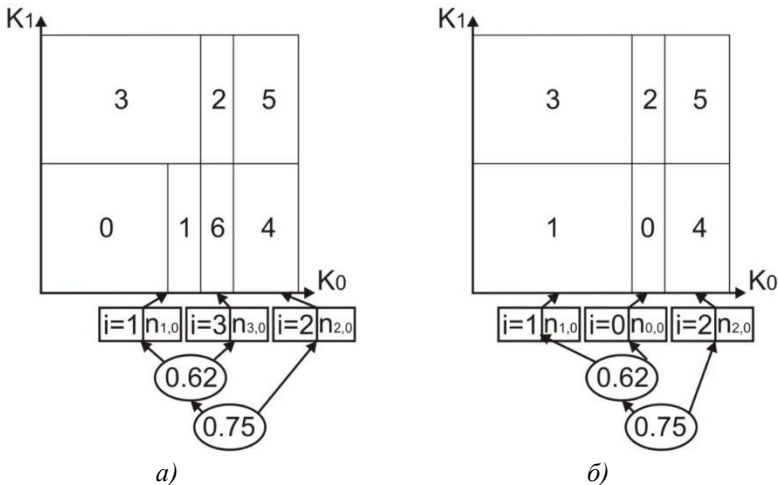


Рис. 2.45. Слияние областей файла: а – первый шаг слияния; б – окончательные модификации

Для операций расширения и объединения существует одно важное правило: нельзя выполнять обе эти операции в одно время. Необходимо завершить первую операцию перед началом второй. Это ограничение вытекает из того факта, что области расщепления и слияния выбираются произвольно и в практических ситуациях могут не совпадать.

Основные характеристики

Для определения области применения данной структуры рассмотрим ее основные характеристики. Это позволит более точно определить положительные и отрицательные стороны тех нововведений, которые были внесены в *PLOP* хеширование.

Процент использования памяти. Основная парадигма схем хеширования заключается в том, что наибольшая производительность достигается при максимально возможном равномерном распределении записей в файле. Неравномерное распределение записей влияет на производительность операции доступа при линейном расширении файла.

PLOP хеширование предполагает увеличение файла линейным способом с частичными расширениями. При этом страницы расположены в k группах, $k < m$, где k находится в диапазоне числа записей в файле. Группа страниц, которая в данный момент расширяется линейно, адресуется указателем gp (как и в случае хеширования *MOLHPE*). За один шаг разделяется надвое одна из страниц группы gp . После удвоения числа страниц в выбранной группе указатель gp переадресовывается на новую группу страниц, которая будет расширяться также линейно. Для достижения хорошей производительности выбирается группа с наибольшим количеством записей. Это позволяет повысить процент использования памяти по сравнению с другими структурами многомерного хеширования без каталога. Следует подчеркнуть, что *PLOP* – это первая *MDH* схема без каталога, в которой можно свободно выбирать группу страниц, которая будет расширяться после текущей.

Размер каталога. Размер каталога файлов-решеток достигает величины $O(n^d)$. Это достаточно большая величина. Поэтому каталог в файлах-решетках хранится на жестком диске. Это усложняет процедуры модификации, так как появляется необходимость значительного изменения страниц жесткого диска.

В хешировании *PLOP* каталог как таковой отсутствует. Однако, в отличие от хеширования *MOLHPE*, в нем появляются шкалы по осям в виде бинарных деревьев. Это вносит некоторые коррективы в алгоритмы. Однако размер шкал в практических применениях настолько мал, что они без особых проблем помещаются в оперативной памяти. Поэтому никакие модификации не требуют значительных затрат.

Запрос на точное совпадение. Одним из самых частых запросов является запрос на точное совпадение. При этом файл-решетка справляется с такой задачей ровно за два обращения к жесткому диску. Данный показатель не зависит ни от размера файла, ни от наличия или отсутствия записи в индексе. Первое обращение тратится на поиск нужного участка каталога, второе – на непосредственную загрузку нужного блока.

Хеширование *PLOP* не содержит каталога, поэтому в нем нет первого обращения. Оно автоматически рассчитывает номер внешнего блока и загружает его.

Однако в хешировании *PLOP* иначе обрабатывается ситуация переполнения. Записи, попавшие в один блок и не поместившиеся в нем, объединяются в цепочку блоков. Таким образом, появляются ситуации, в

которых для поиска может понадобиться два и больше обращений к жесткому диску, что может быть хуже, чем в файлах-решетках.

Однако многочисленные исследования показали, что при практическом применении хеширования *PLOP* число длинных цепочек оказывается невелико. В то же время имеется достаточно большая вероятность нахождения нужной записи с первого обращения к внешней памяти. Поэтому средний показатель числа обращений к внешней памяти для запросов поиска по точному совпадению незначительно больше 1. Однако при самом неблагоприятном стечении обстоятельств этот показатель может достигать $O(n^{1-1/d})$.

Операция вставки. Еще одной операцией, по которой можно сравнивать структуры хеширования, является операция вставки объектов. Эта процедура является узким местом для многих схем хеширования. Если объект разместился в некотором блоке, который заполнен не максимально, то особых трудностей данная операция не предоставляет ни в одном методе хеширования. Однако если появляется переполнение и необходимость расщепления блоков, то некоторые схемы требуют серьезного перестроения всего индекса.

Так, файлы-решетки при переполнении могут потребовать перестроения всего каталога, что приведет к значительным трудозатратам. Сложность такой операции может достигать $O(n^{d-1})$.

В то же время хеширование *PLOP* в большинстве случаев потребует всего нескольких обращений. Однако если расщепление затрагивает самую длинную цепочку при самом неудачном построении индекса, то может потребоваться $O(n^{1-1/d})$, хотя даже эта величина намного меньше трудозатрат файлов-решеток. Поэтому, с точки зрения динамических структур с очень частым изменением объектов, хеширование *PLOP* является более предпочтительным.

2.4. Кривые, заполняющие пространство

Еще одной группой алгоритмов, предназначенных для точечных данных, являются так называемые «кривые, заполняющие пространство» (*space filling curves*). Исторически именно эта группа алгоритмов была первой попыткой ускорить операции обработки многомерных данных. Однако назвать эти алгоритмы многомерными в полной мере нельзя. Основная суть всех методов, рассматриваемых в этом разделе, сводится к преобразованию многомерной задачи в одномерную и применению обычных одномерных структур, таких как бинарные или *B*-деревья, одномерное хеширование.

Для одноключевых данных существует большой набор различных алгоритмов и структур, достаточно хорошо решающих задачи

индексирования и поиска. Большинство из них базируется на том факте, что в одномерном случае можно построить соотношение порядка, т. е. для каждой пары записей можно однозначно сказать, какая из них больше или меньше в зависимости от ключевого атрибута. Однако в многоключевом или многомерном случае такого порядка не существует. Поэтому все одномерные схемы не подходят для него.

Чтобы появилась возможность применения одномерных схем индексирования, необходимо ко всем ключевым полям многомерной записи применить некоторую функцию, которая преобразует их в одно единственное общее цифровое поле. Таким образом, можно избавиться от многомерности и использовать хорошо исследованные одномерные структуры.

Другими словами, необходимо использовать функцию, которая позволит выстроить все многоключевые записи в некотором линейном порядке. Функции, выполняющие подобное преобразование, были названы *кривыми, заполняющими пространство*. Такое название они получили из-за своего графического представления. Если попытаться изобразить кривые, полученные из данных функций, они будут пронизывать все пространство, все его части, как бы заполняя его собой полностью.

Совершенно очевидно, что на практике создать кривую для непрерывного пространства практически нереально. Однако в компьютерной обработке данных практически всегда используются дискретные величины с определенной точностью. Это связано с тем, что ключевые поля объектов хранятся в памяти ЭВМ в некотором двоичном виде с заранее определенной разрядной сеткой. Поэтому необходимости в создании универсальной функции заполнения непрерывного пространства нет, достаточно использовать конечные функции, работающие с битовым представлением ключей.

Иногда в литературе можно встретить название данных методов как пиктографическое индексирование. Данный термин появился из-за некоторой схожести общих принципов представления многокритериального пространства и изображений в компьютере. При индексировании все n -мерное пространство условно разбивается на маленькие ячейки, каждая из которых имеет размер, равный точности рассматриваемой задачи. Точно так же изображение при оцифровке разбивается на маленькие точки, называемые пикселями. Из-за этой схожести иногда ячейки пространства также называют пикселями. Кривая, заполняющая пространство, при этом как бы покрывает все пиксели, проходя через них. Распрямляя ее в одномерную линию, получаем простое линейное упорядочивание всех элементов многомерного пространства, которое уже можно использовать в обычных одномерных схемах.

В настоящее время существует достаточно большое число функций, выполняющих подобное действие. Самые популярные из них будут рассмотрены далее в этом разделе. Некоторые другие схемы можно посмотреть в статье Г. Сегана, опубликованной в 1994 году [79]. Описание пространственных запросов и их модификаций для кривых, заполняющих пространство, хорошо представлено в работе Г. Самета [76].

В рассматриваемых примерах данной главы будет предполагаться, что после преобразования в одномерное пространство для индексирования используется обычное бинарное дерево, хотя на практике можно использовать любую другую структуру (2-3-дерево, B-дерево или даже одномерное хеширование).

Процессы добавления и удаления объектов рассматриваться в этой главе не будут, так как они сводятся к аналогичным операциям в бинарном дереве. Так, для вставки объекта в индекс просто рассчитывается суперключ, и уже он вставляется в бинарное дерево по обычному алгоритму вставки.

Точно так же не будет рассматриваться процедура поиска объекта по точному совпадению. В нем все начинается с формирования суперключа и поиска вершины в дереве по нему. Если суперключ в дереве был найден, то соответствующая ему вершина и есть та самая, которая удовлетворяет запросу поиска, иначе запрошенного объекта в индексе нет.

Пространственные запросы, такие как поиск по области или поиск ближайшего соседа, имеют свои особенности в реализации. Это связано с тем, что в этих запросах на первое место выходит пространственное размещение и пространственная составляющая ключей объектов. Их нельзя свести к обычным одномерным процедурам. Поэтому эти процедуры будут рассмотрены более подробно в следующих параграфах.

2.4.1. Упорядочивание по ключам

Самым простым вариантом функции заполнения пространства является упорядочивание по ключам [76]. В этом случае для получения единственного общего ключа многомерного объекта используется простая конкатенация (слияние) всех его ключевых полей. В общем случае данная функция будет иметь вид

$$K = K_1 K_2 \dots K_n,$$

где K – единый суперключ, используемый в схемах одномерного индексирования;

K_i – i -й ключ многомерного объекта;

n – размерность пространства.

Графическое представление данной функции для двумерного случая показано на рис. 2.46. Для большей наглядности на рис. 2.46(а)

представлен вариант индексирования всего четырех ячеек пространства, а на рис. 2.46(б) – индексирование в общем случае.

Нетрудно заметить, что данная функция практически равноценна сортировке записей с лексикографическим упорядочиванием ключей. Действительно, при размещении двумерных объектов в индексе большую роль будет играть ключ K_1 , а ключ K_2 разделяет объекты только при равенстве первых ключей. Данный подход иногда бывает полезным. Особенно тогда, когда ключи, действительно, имеют разный уровень важности и их можно выстроить в порядке убывания приоритета. При этом поиск объектов по точному совпадению ключей будет выполняться достаточно быстро и эффективно.

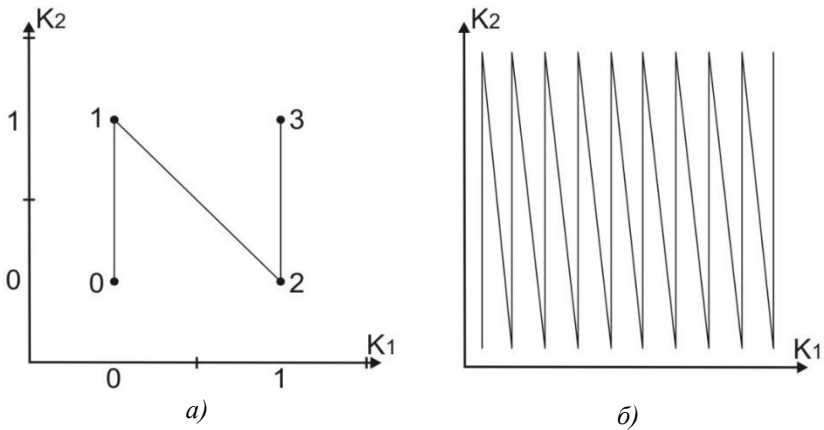


Рис. 2.46. Графическое представление функции упорядочивания ключей:
а – индексирование 4-х ячеек; б – общий случай

Еще одним несомненным плюсом данной схемы является простота метода. Реализация побитовой конкатенации нескольких ключей является тривиальной операцией и легко реализуется на компьютере.

Однако, несмотря на перечисленные положительные стороны, данный метод на практике применяется редко. Дело в том, что многомерные запросы, такие как диапазонный запрос или поиск ближайшего соседа, используют пространственные соотношения ключей. Данные запросы очень хорошо выполняются в том случае, если объекты, которые находятся рядом в пространстве, будут находиться рядом и в индексной структуре. Однако для данной схемы это не всегда так.

Для примера рассмотрим поиск объектов по области. Процедура, выполняющая данное действие, представлена с помощью псевдокода в листинге 2.34.

```

//=====
// Поиск объектов в области
// Параметры:
//   R1,R2 - область поиска, заданная двумя точками
//   V - текущая вершина поиска (первоначально - корень)
//   {Res} - множество-результат (в него помещаются
//           объекты, удовлетворяющие запросу)
//=====
ПОИСК_В_ОБЛАСТИ(V, R1, R2, {Res})
    [1] // Проверка текущей вершины
        Если  $(K1(R1) \leq K1(V) \leq (K1(R2))$  и
            $(K2(R1) \leq K2(V) \leq (K2(R2)))$ , то
            Добавить V в множество результата {Res}
    [2] // Проверка поддеревьев данной вершины
        Если SuperKey(V) > SuperKey(R1), то
            ПОИСК_В_ОБЛАСТИ(Left(V), R1, R2, {Res})
        Если SuperKey(V) < SuperKey(R2), то
            ПОИСК_В_ОБЛАСТИ(Right(V), R1, R2,
                               {Res})
Конец ПОИСК_В_ОБЛАСТИ

//=====
// Функция заполнения пространства
// Параметры:
//   V - вершина, для ключей которой нужно вычислить один
//       общий одномерный ключ
//=====
SuperKey(V)
    K = K1(V) . K2(V)
    Вернуть в качестве результата K
Конец SuperKey

```

Разберем работу этой процедуры и ее слабые стороны на конкретном примере. Пусть у нас дано двухмерное пространство, изображенное на рис. 2.47. При этом значение каждого из ключей может меняться от 0 до 9 и принимать только целые числа. Это значит, что для индексирования объектов данного пространства достаточно его разбить всего на 100 ячеек (с шагом дискретности ключей).

Функция заполнения пространства в нашем примере для наглядности будет работать не с битовым представлением ключей, а с их десятичным видом. При этом она выполняет конкатенацию второго ключа к первому. В результате применения этой функции получаются числа от 0 до 99. Так, если некоторый объект *B* на рис. 2.47 имеет координаты (2;8), то после применения функции он станет обладать суперключом 28.

Рассмотрим процесс выполнения запроса по области, который должен вернуть все объекты с ключом K_1 в диапазоне от 1 до 5, и K_2 – от 1 до 3. Эта область показана на рис. 2.47 пунктиром и ограничивается двумя точками R_1 и R_2 , которые передаются в качестве параметров в процедуру поиска.

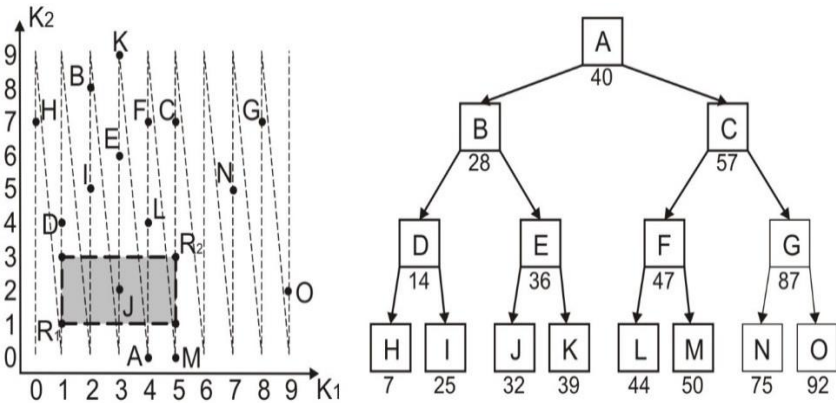


Рис. 2.47. Пример индексирования двумерных данных

Процесс начинается с корня (с вершины A). Данная вершина не принадлежит указанному диапазону, поэтому, согласно условию шага 1, она не будет добавлена в множество результатов Res . Однако ее суперключ равен 40, и он больше суперключа R_1 (11) и меньше суперключа R_2 (53). Поэтому, согласно условию шага 2, необходимо проверить оба поддерева вершины A , т. е. вызывать для них рекурсивно эту же процедуру.

Таким образом, получаем, что при выполнении диапазонного запроса, решение о том, принадлежит ли точка указанной области, принимается на основе обычных ключей. Однако перемещения по дереву и проверка его поддеревьев основываются на суперключях вершин. Причем в дереве необходимо обойти те вершины, которые принадлежат диапазону, образованному суперключами границ области.

Обход дерева приведет к тому, что в множество результатов Res будет добавлена вершина J , которая действительно принадлежит указанному диапазону. Однако, чтобы ответить на этот запрос, процедуре понадобится обойти большую часть дерева (на рис. 2.47 та часть дерева, которую обойдет процедура, выделена жирным). Это происходит из-за главного недостатка данной функции заполнения – многомерные объекты, которые находятся близко в пространстве, будут значительно разнесены при индексировании в дереве. Другие функции заполнения

пространства, рассмотренные далее, пытаются уменьшить негативный эффект данного свойства вследствие более гибкой группировки объектов.

2.4.2. Кривая z -порядка

Еще одним видом кривой является кривая z -порядка. Данный способ является не самым эффективным, однако он намного превосходит описанный вариант и при этом достаточно просто реализуется программно.

В литературе можно встретить несколько названий данной кривой и структур, основанных на ней. Помимо традиционного названия кривой z -порядка (z -ordering [72]), также используются позиционные коды (*locational codes* [17]), N -дерево (N -tree [89]), *quad*-коды (*quad codes* [35]) и т. д. Это связано с большой популярностью данного метода.

Метод получения суперключа в кривой z -порядка сводится к побитовому перемешиванию всех ключевых полей. Если рассматривается n -мерное пространство с n ключами, каждый из которых состоит из m бит, то формирование общего ключа будет иметь следующий вид:

$$K = K_{1,m-1}, \dots, K_{n,m-1}, K_{1,m-2}, \dots, K_{n,m-2}, \dots, K_{1,0}, \dots, K_{n,0},$$

где K – единый суперключ, используемый в схемах одномерного индексирования;

K_{ij} – j -й бит i -го ключа многомерного объекта.

Как можно заметить из формулы, происходит формирование ключа с помощью все того же принципа слияния. Однако в отличие от предыдущей схемы сливаются ключевые поля не полностью, а побитно. Сначала выбираются старшие биты ключей и сливаются вместе, потом – следующие и т. д. (нумерация бит в формуле предполагается справа налево). В результате получается число, которое в n раз больше размера каждого из ключей. Так, если рассматривается двумерное пространство, а размер ключей равен 32 бита, то результирующий ключ будет иметь размер 64 бита.

Графическое представление данной функции для двумерного случая показано на рис. 2.48. Для большей наглядности на рис. 2.48(а) представлен вариант индексирования всего 16 ячеек пространства, а на рис. 2.48(б) – индексирование в общем случае.

Данная функция не может считаться идеальной, однако, как можно видеть из рис. 2.48, она приводит к более адекватной группировке объектов. Те точки, которые находятся близко в пространстве, с большой долей вероятности будут близко находиться и на итоговой кривой, а следовательно, будут находиться в одной ветке бинарного дерева.

Внесенные изменения в функцию заполнения пространства не изменили алгоритмы работы с ней. Так, процедура поиска объектов по области, представленная в листинге 2.34, будет точно так же работать и

для этой функции. В данном же разделе рассмотрим функцию поиска ближайшего соседа (листинг 2.35).

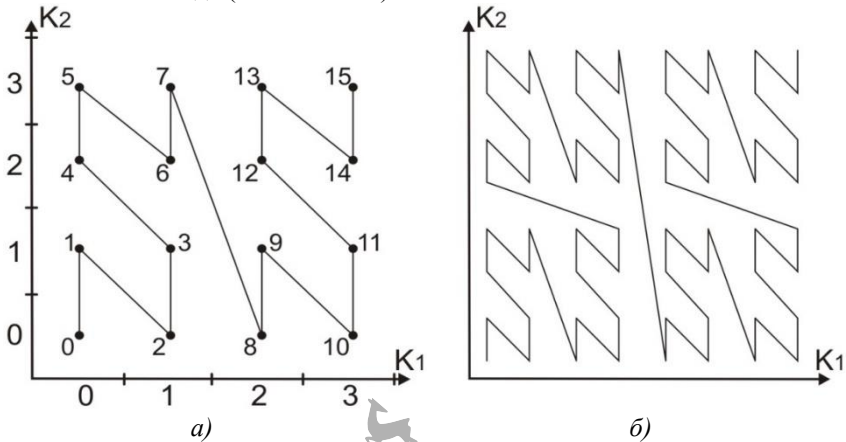


Рис. 2.48. Графическое представление функции z-порядка:
а – индексирование 16 ячеек; б – общий случай

Листинг 2.35

```
//=====
// Поиск ближайшего соседа
// Параметры:
//   V – текущая проверяемая вершина
//   P – вершина, для которой ищется ближайший сосед
//   P' – текущее приближение (первоначально – корень)
//=====
ПОИСК СОСЕДА(V, P, P')
[1] // Проверка текущей вершины
    Если Dist(V,P) < Dist(P,P'), то
        P' = V
[2] // Проверка левого поддерева
    R1 – левая граница области поиска
    K1(R1) = K1(P) - Dist(P,P')
    K2(R1) = K2(P) - Dist(P,P')
    Если SuperKey(V) > SuperKey(R1), то
        ПОИСК СОСЕДА(Left(V), P, P')
[3] // Проверка правого поддерева
    R2 – правая граница области поиска
    K1(R2) = K1(P) + Dist(P,P')
    K2(R2) = K2(P) + Dist(P,P')
    Если SuperKey(V) < SuperKey(R2), то
        ПОИСК СОСЕДА(Right(V), P, P')
Конец ПОИСК СОСЕДА
```


Разберем работу этой процедуры на конкретном примере. Пусть у нас дано двумерное пространство, изображенное на рис. 2.49. При этом значение каждого из ключей может меняться от 0 до 7 и принимать только целые числа. Это значит, что каждый ключ может быть представлен 3-битным числом.

Поиск начинается с корня дерева. Процедуре поиска передается в качестве параметров текущая проверяемая вершина (корень дерева, в рассматриваемом примере это узел A), объект, для которого ищется ближайший сосед (пусть это будет объект P с координатами $(2;2)$) и текущее лучшее приближение (первоначально лучшим приближением к результату поиска является корневая вершина A , так как она единственная затронута на этом шаге в поиске). Причем третий параметр должен передаваться в процедуру по ссылке, так как в процессе своей работы он будет меняться, уточняться и в него будет возвращен результат работы.

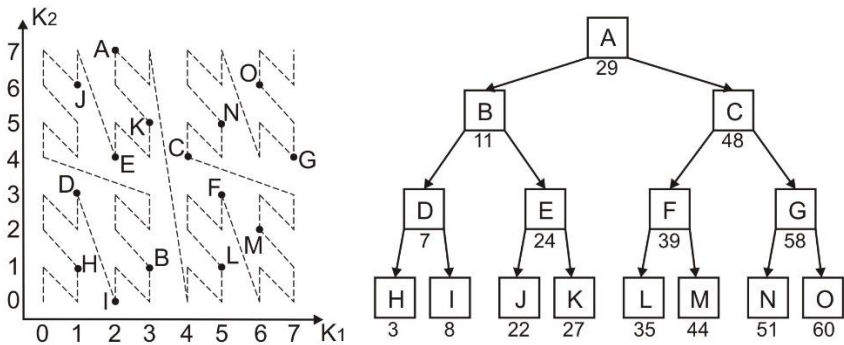


Рис. 2.49. Пример индексирования двумерных данных

Алгоритм работы процедуры поиска ближайшего соседа похож на поиск по области, рассмотренный в прошлом разделе. На каждом шаге работы процедуры вычисляется область, которая может вмещать в себя результат. Область представляет собой квадрат с центром в точке P и радиусом, равным расстоянию от этой точки до лучшего на данный момент приближения (сторона квадрата, таким образом, равна двум подобным расстояниям).

В процессе поиска и перехода по дереву будет выбираться все лучшее и лучшее приближение к P и уменьшаться область поиска. Таким образом, в большинстве случаев произойдет перебор небольшой части дерева. Так, в рассматриваемом примере необходимо просмотреть всего лишь небольшую ветку левого поддерева.

2.4.3. Кривая Гильберта

Одной из самых эффективных в настоящее время считается кривая Гильберта. Данная кривая для индексирования пространственных данных применяется сравнительно недавно [23, 47], хотя сам метод возник еще в XIX веке.

В 1890 году Джузеппе Пеано (Giuseppe Peano) открыл плоскую кривую с удивительным свойством «заполнения пространства». Такая кривая заполняла единичный квадрат и проходила через каждую его точку по меньшей мере один раз. А в 1891 году Давид Гильберт [46] открыл вариант кривой Пеано, основанной на делении каждой стороны единичного квадрата на две равные части, что делит квадрат на четыре меньшие части. Затем каждый из четырех получившихся квадратов, в свою очередь, аналогично делится на четыре меньших квадрата и т. д. На каждой стадии такого деления Гильберт построил кривую, которая обходила все имеющиеся квадраты [15].

Графическое представление данной функции для двумерного случая показано на рис. 2.50. Слева представлен случай для четырех ячеек. Иногда данную кривую называют кривой первого порядка.

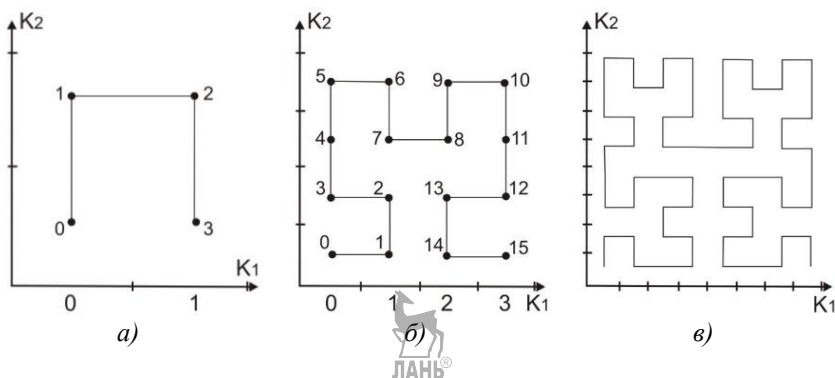


Рис. 2.50. Графическое представление кривой Гильберта: а – 4 ячейки; б – 16 ячеек; в – общий случай

Для получения кривой второго порядка (покрывающей 16 ячеек) необходимо на каждой линии кривой первого порядка построить углубление, имеющее такую же форму, как и оригинальная кривая. Получившаяся кривая показана на рис. 2.50(б). Аналогично можно получить кривую любого порядка, которая покроеет пространство нужного размера. На рис. 2.50(в) показана кривая третьего порядка. На практике чаще всего применяют кривую 32-го порядка (по числу бит в ключах многомерного объекта). Как можно видеть из представленных

рисунков, кривая Гильберта лучше всего группирует объекты. Однако существует большое количество проблем при реализации этого метода.

Первой сложностью является генерация кода произвольного порядка. Выше было дано определение кода через его графическое отображение. Именно так он был изображен в оригинальном документе. К рисункам прилагалась формула рекурсивного определения кривой n -го порядка через кривую $(n-1)$ -го порядка. Долгое время это был единственный алгоритм получения кода произвольного порядка. Однако этот алгоритм плохо подходит для индексирования многомерных данных. Во-первых, рекурсивный алгоритм формирует кривую заполнения квадрата единичного размера. Поэтому для представления реальных многомерных объектов необходимо перед формированием кода производить перевод ключевых полей к единичному отрезку. Во-вторых, рекурсивный алгоритм имеет достаточно большую вычислительную сложность.

В последнее время появилось много вариантов формирования кодов Гильберта из битового представления ключей, приспособленных для программного выполнения. Одним из них является древовидное представление кода [62]. На рис. 2.51 представлено дерево для формирования кода Гильберта третьего порядка. Использование этого дерева рассмотрим на примере. Допустим, необходимо получить код Гильберта для точки P с координатами $(110;100)$. Формирование кода начинается с корня. Из обоих ключей точки P берутся по одному биту, и в корневой вершине дерева выбирается соответствующий им потомок. Таким потомком оказалась ветвь дерева с номером '10'. Этот номер становится началом кода, а для формирования его дальнейшей части происходит переход по дереву.

Далее выбираются вторые биты ключей – $(1;0)$. Для них выбирается потомок, которым оказывается ветвь с номером '11'. Эти два бита добавляются к уже имеющимся, и происходит дальнейший переход по дереву. Процесс заканчивается в листовой вершине. Для рассматриваемой вершины P получившийся код будет иметь вид '101110'.

Для кривых большого порядка дерево может иметь огромный размер. Однако хранить его целиком в памяти необязательно. Можно формировать его ветви динамически. Такое получение кода было модифицировано в схему состояний [60], которую можно реализовать программно без использования рекурсии.

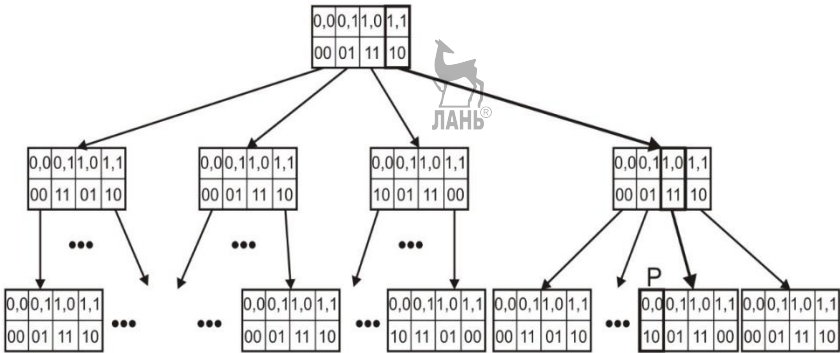


Рис. 2.51. Древоидное представление кода Гильберта

На рис. 2.52 показан пример сформированного бинарного дерева для двумерного случая и трехбитных ключей. Построение дерева и его использования в одномерных запросах происходит точно так же, как и для других кривых заполнения пространства.

Кривая Гильберта формирует более качественные коды, которые лучше учитывают пространственное положение объектов. Однако, в отличие от предыдущих схем, пространственные запросы для этого метода имеют более сложное исполнение. Так, запрос по области уже нельзя проводить, используя верхнюю и нижнюю границы. Кривая Гильберта может многократно входить в область поиска и выходить из нее. При этом рассчитать нижнюю и верхнюю границу, которая использовалась в листинге 2.33, очень сложно, а ее использование является неэффективным.

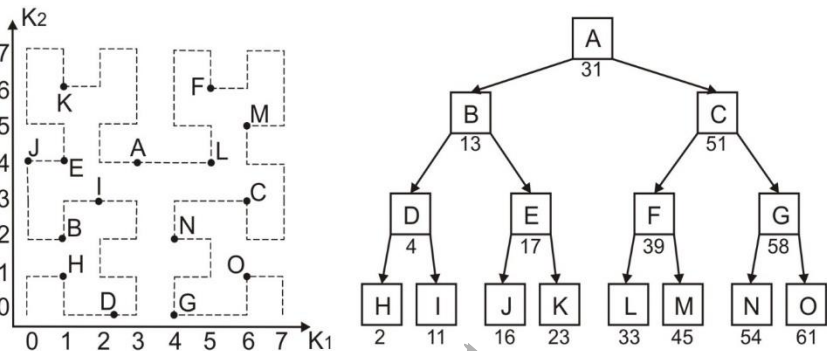


Рис. 2.52. Пример индексирования двумерных данных

В настоящее время разработано несколько алгоритмов поиска по области. Одни из них основываются на принципе рекурсивного деления

пространства при формировании кода Гильберта [23]. В других происходит поиск сегментов кривой, которые потом ищутся в бинарном дереве [62].

2.4.4. Кривая, основанная на кодах Грея

В заключение кратко рассмотрим еще одну кривую, основанную на кодах Грея [28, 29]. Первоначально коды Грея были разработаны для использования в электронных системах. Их отличительной особенностью является то, что соседние числа отличаются друг от друга ровно одним битом. Это позволяло избавиться от ложных срабатываний при переключениях схемы.

В настоящее время коды Грея используются во многих отраслях, а не только в электронике (выявление ошибок в системах связи, генетические алгоритмы, индексирование данных и т. д.). На рис. 2.53 представлено графическое представление кривой, полученной на основе кода Грея. Слева изображен сегмент для индексирования 16 ячеек, а справа – общий случай.

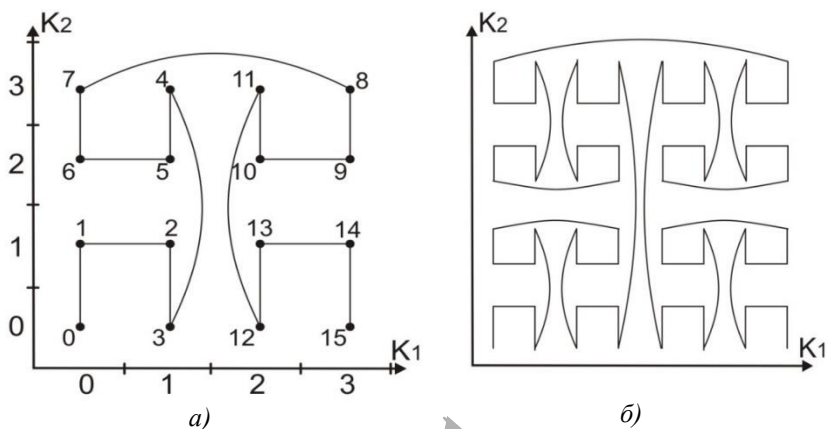


Рис. 2.53. Графическое представление кривой на основе кода Грея:
а – индексирование 16 ячеек; б – общий случай

Математически получение кода Грея по всем ключам объекта можно выразить следующей формулой:

$$K_z = \text{CodeZ}(K_1, K_2, \dots, K_n),$$

$$K = \text{GrayCode}(K_z),$$

где K – единый суперключ, полученный с помощью кода Грея;

K_z – ключ, полученный по алгоритму перемешивания бит;

GrayCode – функция получения кода Грея из двоичного числа.

Первым шагом получения кода является вычисление кривой z -порядка. Для этого перемешиваются биты всех ключей, начиная со старшего. Подробно этот алгоритм был рассмотрен в предыдущих главах.

После получения единого двоичного представления к нему применяется кодирование Грея. Примеры кодов Грея для двухбитных ключей, сгенерированные по рассмотренному алгоритму, показаны в табл. 2.2.

Таблица 2.2

Ключи	Код Грея	Ключи	Код Грея
(00, 00)	0000	(10, 00)	1100
(00, 01)	0001	(10, 01)	1101
(00, 10)	0110	(10, 10)	1010
(00, 11)	0111	(10, 11)	1011
(01, 00)	0011	(11, 00)	1111
(01, 01)	0010	(11, 01)	1110
(01, 10)	0101	(11, 10)	1001
(01, 11)	0100	(11, 11)	1000

Код Грея программно легко вычисляется из двоичного представления числа. Для этого используется побитовая операция «исключающее ИЛИ» числа с самим собой, но сдвинутым вправо на один бит.

По своим свойствам код Грея немного уступает кривой Гильберта. Однако он проще в генерации и имеет более высокую производительность, чем кривые z -порядка.

Резюме

Материалы второй главы позволяют системно оценить многомерные структуры, работающие с точечными объектами. Выделено две группы структур, ориентированных на иерархический доступ и доступ с помощью методов хеширования. Они достаточно полно покрывают все знания в этой области. Третья группа методов, реализующая точечный доступ с помощью кривых, заполняющих пространство, на данном этапе представляет больше теоретический интерес. Но вариант эффективного применения их не исключен.

ГЛАВА 3. ПРОСТРАНСТВЕННЫЕ МЕТОДЫ ДОСТУПА

Все многомерные методы доступа, описанные в предыдущей главе, разработаны для наборов точечных данных и поддерживают запросы поиска только для них. Ни один из этих методов без модификации не сможет работать с многомерными объектами, имеющими определенный размер. Однако очень много приложений, в которых необходимо размещать информацию об объектах, геометрическими размерами которых пренебрегать нельзя. К таким приложениям, например, можно отнести геоинформационные системы (ГИС), в которых объекты в простейшем случае представляют двухмерные фигуры, или CAD-системы с трехмерными деталями.

При переходе от точечных объектов к пространственным появляется ряд проблем, решение которых является не всегда тривиальным. Например проблема неправильных геометрических форм объекта (о ней уже было рассказано в главе 1) или проблема перекрытия минимальных описывающих прямоугольников (*MBR*) разных объектов. В частности, перекрытие *MBR* объектов не позволяет однозначно разделять объекты на два непересекающихся подмножества, что является очень важным свойством практически всех алгоритмов, описанных ранее. Причем пересечение *MBR* может возникнуть даже в том случае, если сами объекты и не имеют общих точек. Пример такого пересечения показан на рис. 3.1. На этом рисунке ни один объект не имеет общих точек, однако *MBR*₂, *MBR*₆ и *MBR*₇ пересекаются между собой.

Для адаптации точечных методов и модификации их для пространственных объектов применяется одна из следующих технологий:

- 1) преобразование в точечные объекты (*object mapping*);
- 2) перекрытие областей (*overlapping region*);
- 3) усечение объектов (*clipping*);
- 4) многослойность (*multiple layer*).

Все методы, рассматривающие объект как некоторую многомерную структуру, имеющую определенные геометрические размеры, можно объединить в один класс алгоритмов, получивший название пространственных методов доступа (*Spatial Access Methods – SAM*).

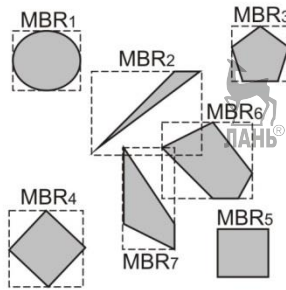


Рис. 3.1. Пример MBR

На данный момент алгоритмов данной категории разработано гораздо больше, чем даже для точечных данных. Однако, как и в случае с *PAM*, единой их классификации не существует. Первую простейшую версию классификации предложили Б. Сигер и Г. Кригел в 1988 году [80]. Позднее (1991 г.) Г. Кригел ее немного модифицировал и добавил новые возможности и параметры. В табл. 3.1 представлен окончательный вид данной классификации.

Таблица 3.1

Подход	Базовый тип MBR			
	решетка	интервал (прямоуголь- ник)	сфера	многоуголь- ник
Преобразо- вание в точечные объекты	<i>Z-K-D-B+</i> - дерево, <i>BANG</i> - файл, <i>hB</i> -дерево	Любой метод класса <i>PAM</i> , за исключением <i>BANG</i> -файлов и <i>hB</i> -деревьев		<i>P</i> -дерево (Jagadish)
Перекрытие областей		<i>R</i> -дерево, <i>R*</i> -дерево, <i>SKD</i> -дерево, <i>GBD</i> -дерево, <i>R</i> -дерево Хильберта	Сфери- ческое дерево	<i>P</i> -дерево (Schiewietz), <i>KD2B</i> -дерево
Усечение объектов		<i>EXCELL</i> , Расширенное <i>K-D</i> -дерево, <i>R+</i> -дерево		Дерево- решетка
Многослой- ность		Многоуровне- вый файл- решетка, <i>R</i> -файл		

В данной главе будет представлено описание основных структур для пространственных объектов, выделены их свойства, преимущества и недостатки по сравнению с другими алгоритмами.

3.1. Методы преобразования пространственных объектов

Индексирование объектов, имеющих геометрические размеры, является более сложной задачей. Однако существует несколько попыток привести сложные пространственные объекты к более простому виду, который можно использовать в рассмотренных ранее схемах, таких как точечные структуры данных или даже алгоритмы одномерного индексирования.

Чтобы появилась возможность использовать более простые схемы индексирования, необходимо преобразовать пространственные объекты к более простому виду. Такой подход получил название *метода трансформации*.

В настоящее время разработано несколько принципиально различающихся схем преобразований. Первым способом является попытка представить объект с некоторыми геометрическими размерами как точку, не имеющую геометрических размеров. При этом для индексирования появляется возможность использования методов и алгоритмов, рассмотренных в предыдущей главе. Такой метод получил название *преобразования в пространство большей размерности* [44, 80].

Второй подход к технике трансформации – это преобразование пространственных объектов не в один объект, а в целый набор более простых объектов. Такими объектами могут быть точки. В этом случае объект разбивается на множество точек конечного размера (размер зависит от точности метода), и эти точки уже индексируются как независимые объекты. Одна из подобных схем уже была рассмотрена нами в прошлой главе (*Quad-деревья областей* [79]).

Еще одним способом трансформации является преобразование объектов в отрезки линий заполнения пространства, которые проходят через этот объект. Все перечисленные варианты будут рассмотрены далее.

3.1.1. Преобразование в пространство большей размерности

Простейшим вариантом представить пространственный объект как точку является перевод его в пространство большей размерности. Так, пусть имеется прямоугольник в двухмерном пространстве. При некотором видоизменении его можно представить как точку, имеющую

четыре координаты, т. е. перейти к четырехмерному пространству. Для этого можно выбрать две точки этого прямоугольника, находящиеся на главной его диагонали, и записать их координаты через запятую. Таким образом, получается, что любой двухмерный прямоугольник представляется как точка с четырьмя координатами. Способ представления через координаты углов называется *метод конечной трансформации (endpoint transformation)*.

Существует и другой алгоритм перевода прямоугольника в точку. Для этого необходимо вычислить его геометрический центр (точка пересечения диагоналей) и приписать к нему длину и ширину, поделенные пополам (радиусы протяженности по осям). Таким образом, снова получается четыре координаты, т. е. прямоугольник преобразовался в четырехмерную точку. Этот способ получил название *метод серединной трансформации (midpoint transformation)*.

В общем случае для представления объекта в n -мерном пространстве его необходимо заключить в минимальный ограничивающий прямоугольник и его координаты перевести в точку в пространстве с удвоенным числом измерений (т. е. перейти к $(2n)$ -мерному пространству). После получения точечного представления для хранения и обработки можно использовать любые структуры, рассмотренные в предыдущей главе.

Рассмотрим пример описанного преобразования. Так как четырехмерное пространство нельзя наглядно представить на двумерном листе, рассмотрим преобразование протяженных объектов одномерного пространства (отрезков) в их точечное представление в двухмерном. Для преобразования будем использовать метод конечной трансформации. На рис. 3.2 представлен пример такого преобразования.

Рис. 3.2(а) содержит три отрезка (O_1 , O_3 , O_4) и одну точку (O_2). Это яркие представители пространственных объектов одномерного пространства. Рассмотрим, как эти объекты можно перевести в точки пространства большего измерения.

Объект O_1 начинается в координате $K = 1$ и заканчивается при $K = 3$. Приняв эти величины как значения ключей в двухмерном пространстве, получаем точку O_1 с координатами $(1; 3)$, изображенную на рис. 3.2(б). Аналогично можно получить отображение отрезков O_3 и O_4 .

Отдельно рассмотрим преобразование объекта O_2 . Он уже в одномерном пространстве имеет вид точки. Однако перенести его без изменений в двумерное пространство нельзя, иначе может нарушиться пространственное преобразование и запросы. Поэтому для перевода в двухмерную точку его представляют как отрезок нулевой длины. Этот отрезок начинается в точке $K = 4$ и заканчивается в ней же. После преобразования в двухмерное пространство данный объект станет точкой O_2 с координатами $(4; 4)$.

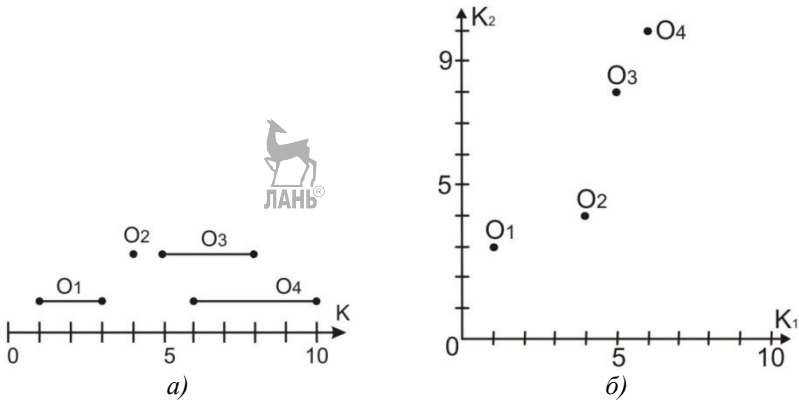


Рис. 3.2. Пример трансформации объектов в точки:
 а – пространственные объекты одномерного пространства;
 б – точечное представление в двухмерном пространстве

После преобразования объекта к точке проводится индексирование уже известными методами. При этом операции вставки и удаления полностью зависят от выбранного метода индексирования. Однако операции поиска, особенно те из них, которые основаны на пространственном положении первоначальных объектов, необходимо преобразовывать перед выполнением к определенному виду. Рассмотрим эти операции более подробно.

Поиск по точному совпадению. Данный вид поиска является самым простым в исполнении. Допустим, у нас имеется преобразование, показанное на рис. 3.2. Необходимо найти отрезок, начинающийся в точке $K = 6$ и заканчивающийся в точке $K = 10$.

Процесс поиска начинается с преобразования запроса поиска в пространство большей размерности по тем же самым алгоритмам, которые использовались при индексировании объектов. Таким образом, объект поиска будет преобразован в точку с координатами (6; 10). После получения точки выполняется многомерный запрос на точное совпадение, в процессе которого будет найден результат – объект O_4 (см. рис. 3.3).

Если в процессе поиска в многомерном пространстве не будет найдено соответствующих точек, это будет означать, что отрезков с подобными параметрами проиндексировано не было. Подобная ситуация возникнет, если произойдет поиск отрезка, начинающегося в координате $K = 5$ и заканчивающегося в координате $K = 7$ (рис. 3.3).

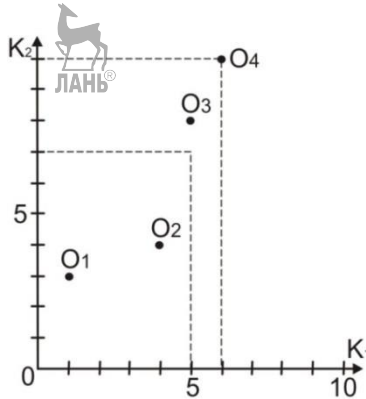


Рис. 3.3. Запрос на точное совпадение

Как можно заметить, запрос на точное совпадение в исходном пространстве преобразуется к точно такому же запросу на точное совпадение в пространстве большей размерности.

Поиск по области. Данный вид поиска состоит в нахождении всех объектов, имеющих хотя бы одну общую точку с заданной областью. Данный вид запроса уже невозможно выполнить без определенных модификаций. Рассмотрим этот запрос на том же самом примере.

Областью поиска в одномерном пространстве является отрезок. Для примера рассмотрим поиск всех объектов, которые имеют пересечение с отрезком от 4 до 7. Нижнюю границу обозначим $l = 4$, верхнюю – $u = 7$.

Графически процедура поиска представлена на рис. 3.4. На первой оси (K_1) откладывается область поиска, и она распространяется вверх до прямой $K_1 = K_2$. После пересечения с этой прямой левая граница области продлевается горизонтально влево, а правая – вертикально вверх. Полученная часть пространства (на рисунке она заштрихована) содержит те объекты, которые удовлетворяют параметрам поиска. В нашем случае это точка O_2 и отрезки O_3 и O_4 .

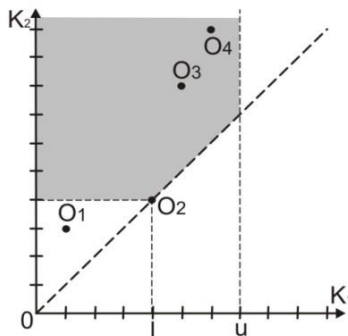


Рис. 3.4. Запрос по области

Нетрудно заметить, что данный вид поиска можно свести к поиску по области в многомерном пространстве. Для этого в качестве границы области по первому ключу K_1 выбирают диапазон $[0; u]$, а по второму ключу K_2 – $[l; \infty]$. На рис. 3.4 не вся указанная область заштрихована (нижняя часть под диагональю не входит в пространство результатов). Однако это не повлияет на результат, так как эта часть не может содержать точек при указанных правилах преобразования объектов.

Поиск содержащихся объектов. Этот запрос можно сформулировать двумя способами: «найти все объекты, которые полностью содержатся в указанной области» или «найти те объекты, которые полностью вмещают в себя указанную область». Рассмотрим эти два запроса более подробно.

Для примера возьмем область, нижняя граница которой равна $l = 3$, а верхняя – $u = 9$. Правила графического формирования области результата представлены на рис. 3.5. Границы области продлеваются вверх до диагонали, и после пересечения с ней меньшая граница направляется вертикально вверх, а большая – горизонтально влево.

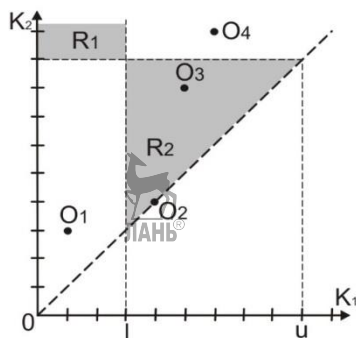


Рис. 3.5. Запрос содержащихся объектов

В результате таких манипуляций получаем две области R_1 и R_2 . Область R_1 содержит все те объекты, которые полностью вмещают в себя область поиска. В нашем примере подобных отрезков нет. Область R_2 содержит те объекты, которые полностью помещаются в указанную область и не выходят за ее границы. В рассматриваемом примере это точка O_2 и отрезок O_3 .

Данный вид запроса точно так же, как и предыдущий сводится к поиску точек по области в пространстве большей размерности.

Концепция преобразования пространственных объектов в точки большей размерности выглядит достаточно элегантно и просто. Однако у нее есть ряд ограничений и недостатков. Первый из них уже был рассмотрен выше – пространственные запросы к подобным объектам нельзя непосредственно выполнять в индексной структуре. Перед их

использованием необходимо трансформировать и изменять эти запросы к другим видам. Причем не для всех запросов найдено красивое и простое преобразование, как было показано выше.

Второй недостаток – изменение характера распределения объектов в пространстве. В прошлой главе было показано, что ряд схем индексирования точечных объектов достаточно хорошо справляется с равномерным распределением объектов. Однако они плохо подходят к неравномерным распределениям и распределениям с ярко выраженным центром. Однако процесс трансформации очень часто приводит именно к таким распределениям даже в том случае, если изначально объекты были равномерно распределены. Так, при использовании конечной трансформации получается распределение, при котором нет ни одного объекта в пространстве ниже главной диагонали [36].

Еще одним недостатком подобной схемы является тот факт, что объекты, которые находились рядом в оригинальном пространстве, могут оказаться далеко друг от друга в трансформированном. Для уменьшения этого недостатка были разработаны различные методы преобразований [35, 76]. Однако ни один из них не может полностью изменить ситуацию.

В заключение стоит отметить, что преобразование в точечные объекты происходит с двукратным увеличением числа измерений. Это также является недостатком указанных схем, так как в настоящее время разработано очень мало структур, способных индексировать данные в пространствах большой размерности.

3.1.2. Кривые, заполняющие пространство для объемных объектов

В последнем параграфе второй главы было рассмотрено несколько кривых, заполняющих пространство, которые преобразовывали многомерные точечные объекты в эквивалентные им одномерные с последующей индексацией простыми одномерными структурами. Этот же метод можно применить и для пространственных объектов. Для этого пространство разбивается решеткой на множество одинаковых ячеек. Каждая ячейка имеет размер, определяющий точность решения поставленной задачи. После этого через все пространство проводят некоторую кривую, заполняющую ее полностью, и получают линейное упорядочивание ячеек пространства. Остается только применить одномерную схему индексирования ячеек, чтобы получить готовый индекс.

Однако, в отличие от многомерных точек, преобразованные в кривую пространственные объекты переходят не в точки на этой кривой, а в целые отрезки (или связанные последовательности точек). Причем в общем случае каждому объекту будет соответствовать не один отрезок, а целый набор таких отрезков. Пример подобного преобразования с помощью кривой z -порядка показан на рис. 3.6.

$O_1: 3$
 $O_2: 48, 49, 50, 51; 54$

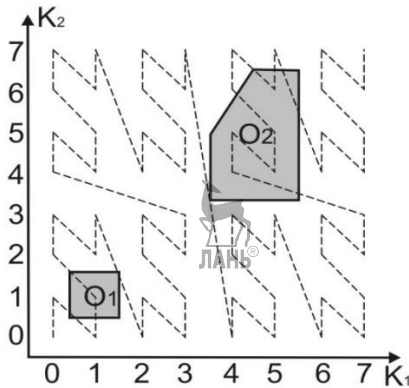


Рис. 3.6. Пример преобразования пространственного объекта с помощью кривой z -порядка

Кривая z -порядка, предложенная Оренштейном и Морретом в 1984 году [72], является самой популярной кривой для индексирования пространственных объектов. Простейший алгоритм подобного преобразования был показан выше. Однако этот алгоритм практически не используется. Если размер ячейки очень мал (в задаче используется большая точность), подобное разбиение приведет к генерации тысяч объектов на кривой, которые необходимо будет индексировать в одномерной структуре. Это приведет к значительному разрастанию индекса и плохой его эффективности.

Для того чтобы разработать более эффективный алгоритм, рассмотрим способ формирования кривой z -порядка и вычисление z -кодов способом деления пространства на квадраты. При этом для формирования двоичного представления z -кода любой ячейки пространства достаточно выполнить разбиение пространства до нужного предела подобно тому, как это происходит при формировании *Quad*-дерева (именно поэтому кривую z -порядка и ее коды иногда в литературе называют *quad*-кривой).

Рассмотрим подобный процесс на примере формирования z -кода для объекта O_1 , представленного на рис. 3.6. Первоначально пространство разбивается на четыре равные части с помощью центральной точки (рис. 3.7(а)). Каждому квадрату присваивается свой определенный код: квадрат в нижнем левом углу имеет код 00, в верхнем левом – 01, верхнем правом – 11, нижнем правом – 10.

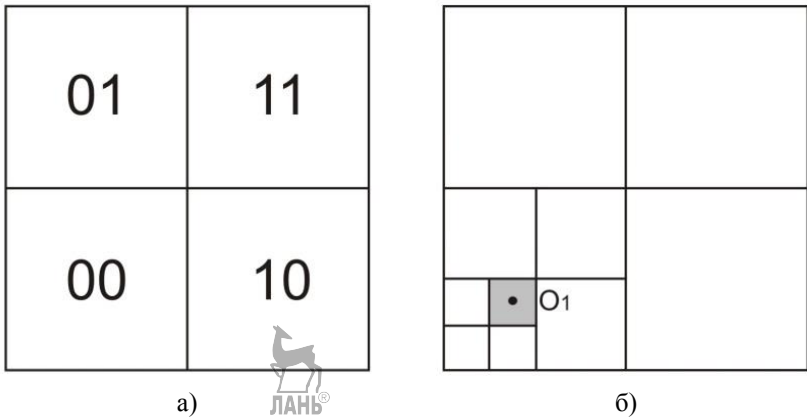


Рис. 3.7. Пример рекурсивного получения z -кода:

a – кодирование квадратов; b – рекурсивное деление пространства

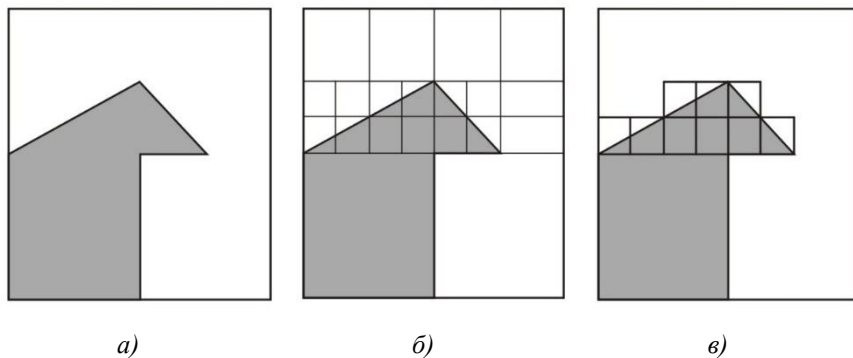
Далее определяется квадрат, в котором находится искомый объект, и он снова разбивается по той же самой схеме. Процесс разбиения продолжается до тех пор, пока не будет достигнут нужный предел (рис. 3.7(б)). Для получения z -кода объекта происходит переход по квадратам к объекту и конкатенация битовой строки. В результате для рассматриваемого примера мы получаем код 000011. В десятичном виде этому коду соответствует значение 3, которое было получено традиционным способом на рис. 3.6.

Теперь рассмотрим способ формирования одномерного представления пространственного объекта с помощью z -кодов. Для этого воспользуемся объектом, изображенным на рис. 3.8(а). Данный алгоритм основан на том же принципе, который только что использовался для получения z -кода точки. Пространство разбивается на четыре части, каждая из которых кодируется двумя битами. Каждая из получившихся частей точно так же рекурсивно разбивается, и этот процесс продолжается до тех пор, пока не выполнится одно из следующих условий:

- получившийся после очередного разбиения квадрат не содержит ни одного объекта (пустой квадрат);
- получившийся после разбиения квадрат полностью относится к некоторому объекту (квадрат объекта);
- достигнут предел точности, требуемый в данной задаче.

Пример подобного разбиения показан на рис. 3.8(б). Так, нижний правый квадрат не разбивался ни разу, так как он не содержит ни одного объекта. В то же время для индексирования верхней части объекта

потребовалось произвести разбиения до тех пор, пока не был достигнут порог точности.



*Рис. 3.8. Формирование z-кода для пространственного объекта:
 а – пространственный объект; б – рекурсивное деление пространства;
 в – кодирование объекта*

После получения набора квадратов, из которых состоит индексруемый объект (рис. 3.8(в)), происходит их кодирование, т. е. получение для каждой из них соответствующего z-кода и помещение этих кодов в одномерную структуру индексирования (например, бинарное дерево). Таким образом, происходит индексирование всех объектов пространства. Однако в отличие от способа, показанного на рис. 3.6, данный вариант является более экономичным и эффективным. Так, для кодирования левой нижней части объекта по предыдущей схеме понадобилось бы добавить 16 ячеек вместо одной.

Пространственные запросы к результирующему дереву практически ничем не отличаются от тех процедур, которые были рассмотрены для точечных объектов в предыдущей главе. Однако, в заключение необходимо отметить одно важное свойство полученных z-кодов. Часть из этих кодов имеет неполный формат (например, состоит не из 6 бит, а только из 2). При этом В. Гейд и В. Рикерт в своей работе показали, что если z-код некоторого объекта является префиксом z-кода второго объекта, то это означает, что второй объект целиком содержится в первом [40]. Используя это свойство z-кода, можно легко работать с неполными кодами, получающимися при индексировании пространственных объектов методом деления на квадраты.

3.2. Структуры с перекрытием областей

Главной идеей метода перекрывающихся областей является разрешение различным узлам дерева частично перекрывать друг друга, т. е. соответствовать одной и той же части пространства. После применения такого допущения появляется возможность без каких-либо проблем размещать в узлах структуры данных объекты, имеющие некоторые пространственные размеры. При этом нет необходимости разбивать объект на более мелкие части с целью уменьшения числа пересечений их *MBR*. Многомерный объект просто помещается в минимальный покрывающий прямоугольник и размещается в одном из узлов структуры целиком.

Применяя такой способ для размещения многомерных объектов, можно адаптировать некоторые точечные методы, описанные в предыдущей главе, для пространственных объектов. Однако для этого необходимо модифицировать и алгоритмы поиска. При этом следует учитывать тот факт, что области, соответствующие дочерним узлам некоторого узла V , могут пересекаться. Поэтому в процедуре поиска, достигнув этого узла, уже нельзя просто определить одного из потомков (поддерево) для дальнейшего поиска. Необходимо также проверить *MBR* других узлов на возможность нахождения в них искомого объекта. Таким образом, даже в процедуре поиска по точному совпадению объекта по всем ключам, возможно, придется проверить несколько поисковых путей от корня к листу, в отличие от точечного случая без перекрытия, при котором достаточно было всего лишь одного прохода. Это усложняет процедуру поиска, однако является необходимой платой за некоторые достоинства данного подхода.

Приняв идею перекрытия областей как догму, мы получаем практически готовые алгоритмы для многомерных баз данных. Однако при более тщательном рассмотрении можно заметить ряд серьезных проблем, которые могут свести к минимуму преимущества от использования подобных индексов в некоторых практических применениях.

Первой и самой важной проблемой является проблема процента перекрытия областей. Нетрудно заметить, что чем большие площади *MBR* пространственных объектов пересекаются, тем менее эффективным становится поиск. В худшем случае может оказаться, что пересечение областей привело к полному перебору всех элементов структуры при поиске, а значит, применение такого индекса несет только дополнительные расходы по памяти, не давая никакого выигрыша во времени поиска.

Типичным примером такого неэффективного индекса может служить база данных объектов большого размера, например, некоторая

картографическая система. Объектами такой системы могут выступать бассейны некоторых рек. Если такие объекты описать минимальными покрывающими прямоугольниками (*MBR*), то их площадь окажется значительной по отношению к общей площади рассматриваемого пространства. При этом перекрытие *MBR* рек и *MBR* озер окажется практически полным. При поиске некоторого объекта по его координатам в такой базе данных нельзя ожидать значительного выигрыша в скорости.

Для таких баз данных предпочтительно использовать либо другие способы, либо применять разбиение объекта на более мелкие составляющие, что даст меньший процент перекрытия описывающих областей.

Еще одной проблемой, связанной с данным методом, может служить случайная вставка многомерных объектов в уже сформированное дерево. Если алгоритмы не предусматривают процедуру уменьшения перекрытия узлов, то при вставке нового элемента в некоторый узел дерева может произойти значительное увеличение *MBR* этого узла. Любое увеличение *MBR* является нежелательным, так как потенциально является источником увеличения площади перекрытия разных областей. Очень хорошо данную проблему описали А. Гуттман и Д. Грeen в своих работах [36, 38]. Позднее было предложен ряд мер, уменьшающих перекрытие областей [38]. Главной их особенностью является добавление в процедуру вставки некоторой модификации, позволяющей анализировать ситуацию и выбирать для вставки тот элемент, при помещении объекта в который увеличение процента перекрытия будет незначительным.

Рано или поздно вставка новых элементов в дерево приводит к переполнению некоторого узла (число элементов узла становится максимальным, и добавление нового объекта в него становится невозможным). При этом проводится процедура расщепления переполненного узла на два новых. Процедура расщепления, в случае своей неэффективной работы, также может являться источником перекрывающихся областей.

Большинство алгоритмов данного подраздела по своей структуре очень похожи друг на друга и отличаются только алгоритмами вставки и удаления узлов. Практически все они ведут свое существование от метода, получившего название *R*-дерево [36].

Далее в подразделах данной главы будет описан оригинальный алгоритм *R*-дерева, а также ряд его модификаций, улучшающих те или иные его свойства.

3.2.1. *R*-дерево

R-дерево (*R-Tree*) – это индексная структура для доступа к пространственным данным, предложенная А. Гуттманом (Калифорнийский университет, Беркли) в 1984 году [36]. *R*-дерево допускает произвольное выполнение операций добавления, удаления и

поиска данных без периодической переиндексации. При этом дерево получается сбалансированным, что является одним из важных свойств любой иерархической структуры данных.

Далее в данном параграфе будет описана структура дерева и основные алгоритмы. Также будут даны практические рекомендации по выбору тех или иных параметров дерева.

Структура R-дерева

R-дерево – это сбалансированное по высоте дерево, сходное с B+-деревом, листовые узлы которого содержат ссылки на конечные объекты. Если индексная структура находится на жестком диске, то каждый узел соответствует дисковой странице. Структура разработана так, чтобы для пространственного поиска требовалось посещение как можно меньшего числа узлов. Индексная структура полностью динамическая – добавление и удаление может выполняться одновременно с поиском, и никакой периодической реорганизации структуры производить не нужно.

Для организации такой индексной структуры используют пространственную базу данных, состоящую из набора записей, каждой из которых соответствует некоторый уникальный идентификатор. Этот идентификатор используют как средство ссылки на запись из индекса. В качестве идентификатора может выступать некоторое уникальное число или номер записи в файле (второй вариант предпочтительнее, так как работает быстрее, однако для него присущи некоторые недостатки, связанные с удалением записей из файла).

Если принять описанные условия, то каждый листовой узел дерева будет состоять из элементов, имеющих вид:

[MBR, идентификатор_записи],

где *идентификатор_записи* ссылается на запись в БД, а *MBR* – это *n*-мерный прямоугольник, который является минимальным охватывающим прямоугольником для пространственного объекта, со сторонами параллельными осям координат. Обычно *MBR* задают в виде:

$$MBR = \{I_1, I_2, \dots, I_n\},$$

где *n* – это число размерностей, а *I_i* – это интервал с закрытыми концами *[a, b]*, характеризующий размер объекта по соответствующей оси координат *i*. В принципе, *I_i* может иметь в качестве конечной точки $\pm\infty$. При этом предполагается, что объект по такому измерению распространяется бесконечно.

Внутренние узлы дерева содержат элементы, имеющие похожую структуру:

[MBR, ссылка_на_потомка],

где *ссылка_на_потомка* – это адрес узла низшего уровня в R-дереве (дочернего по отношению к данному), все записи внутри которого покрываются прямоугольником *MBR*.

Нетрудно заметить, что и листовые узлы, и внутренние представляют собой набор из элементов описанной структуры. Причем даже в простейшей реализации таких элементов должно быть больше одного. Обозначим за M – максимальное число элементов в любом узле, а m – минимальное. Причем для реализации основных алгоритмов необходимо выполнение условия $m \leq M/2$. Практические рекомендации по выбору конкретных значений M и m будут приведены ниже.

R -дерево, описанное выше, должно удовлетворять следующим требованиям.

1. Каждый узел дерева содержит не меньше m и не больше M записей. Исключение может составлять только корень.
2. Корень, если он не является листом, содержит как минимум двух потомков. Максимальное количество элементов в корне также ограничивается значением M .
3. Для каждой индексной записи листового узла MBR является минимальным прямоугольником, который полностью вмещает в себя пространственный объект, на который ссылается запись.
4. Для каждой индексной записи внутреннего узла дерева MBR является минимальным прямоугольником, охватывающим все MBR дочерних узлов.
5. Все листовые узлы дерева расположены на одном уровне (дерево является сбалансированным).
6. Каждый объект упоминается в дереве ровно один раз.

На рис. 3.9 показан пример структуры R -дерева и проиллюстрированы отношения ограничения и перекрытия, которые могут существовать между его прямоугольниками.

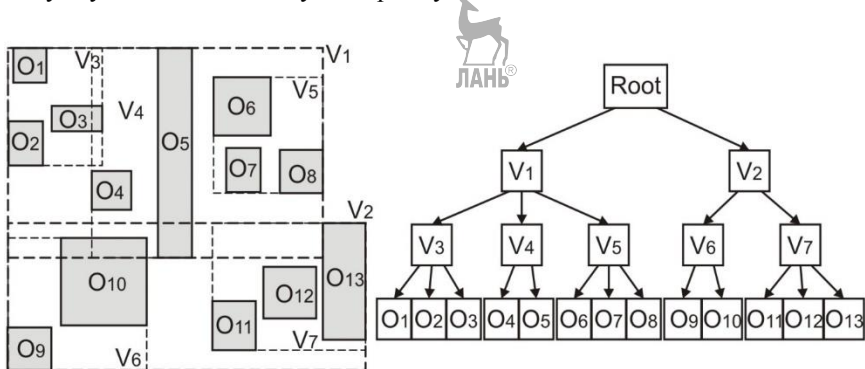


Рис. 3.9. Пример R -дерева

Имея представление о свойствах древовидной структуры, можно оценить его высоту при числе элементов N . Из свойств, описанных выше, следует, что каждый узел дерева содержит как минимум m потомков.

Поэтому наибольшая высота R -дерева, содержащего N индексных записей, будет не больше $\lceil \log_m N \rceil - 1$. При этом максимальное число узлов в таком дереве будет $\lceil N/m \rceil + \lceil N/m^2 \rceil + \dots + 1$.

В наихудшем случае использование пространства памяти, в которой хранится индексная структура, будет m/M . Однако алгоритмы построения дерева разработаны таким образом, что структура будет стремиться содержать более m записей в узле. Это уменьшает высоту дерева и увеличивает полезное использование памяти.

Практические расчеты показывают, что если m больше 4, то дерево получается широким и почти все пространство используется для листьев, содержащих индексные записи.

Алгоритм поиска объекта в R-дереве

Алгоритм поиска в R -дереве очень похож на алгоритм поиска по B -дереву. Он также начинается в корне и опускается по нему к листовому узлу, выбирая в зависимости от заданных параметров поиска то или иное поддереву. Главное же отличие состоит в том, что возможен вариант, при котором более одного поддереву текущего узла участвует в поиске. Такая ситуация связана с применением метода размещения многомерных объектов, разрешающего пересекаться ограничивающим областям разных элементов. Данный факт может привести к многократному уменьшению скорости поиска, однако алгоритмы построения и изменения дерева стараются поддерживать дерево в наиболее оптимальном виде.

В листинге 3.1 представлен один из возможных вариантов рекурсивной процедуры поиска объектов, имеющих хотя бы одну общую точку с областью поиска S .

Листинг 3.1

```
//=====
// Процедура поиска в R-дереве по области S
// Параметры:
//   V - текущая вершина (первоначально это корень)
//   S - область поиска
//   Res - множество результатов поиска
//=====
ПОИСК(V, S, Res)
[1] Если V не является листом, то
    Проверить все записи V', находящиеся в узле V
    Если MBR(V') пересекается с S, то
        Вызвать ПОИСК(V', S, Res)
[2] Если V является листом, то
    Проверить все записи O, находящиеся в узле V
    Если MBR(O) пересекается с S, то
        Добавить запись O в множество Res
Конец ПОИСК
```



Изначально, при вызове процедуры, ей передаются в качестве параметров корень дерева (V), область поиска (S) и пустое множество (Res), в которое будут помещаться найденные объекты.

Процедура поиска состоит из двух частей. Первая часть выполняется только для внутренних узлов дерева. Процедура перебирает все дочерние узлы данного узла V и для каждого из них проверяет, пересекается ли его описывающий прямоугольник с областью поиска. Если такое пересечение наблюдается, то процедура вызывается рекурсивно для этого узла. В противном же случае, если между областью поиска и MBR потомка нет общих точек, такой потомок просто пропускается.

Вторая часть процедуры поиска вступает в действие при достижении листового уровня. Как и в предыдущем случае, происходит перебор всех элементов узла с целью проверки на пересечение их MBR с областью поиска. При нахождении подобных элементов они добавляются в множество результатов Res .

Рассмотрим описанный алгоритм на примере, показанном на рис. 3.10. Область поиска соответствует заданному прямоугольнику $ABCD$.

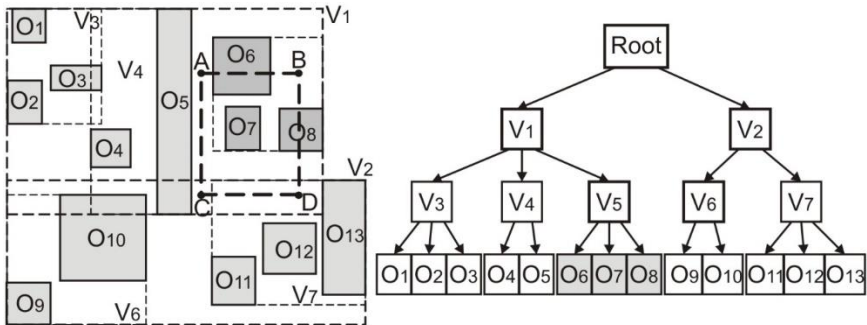


Рис. 3.10. Пример поиска в R-дереве

Первоначально процедура поиска вызывается для корня. Так как корень является внутренней вершиной, то для него выполняется первая ветка алгоритма поиска. Она проверяет узлы V_1 и V_2 на пересечение с заданной областью. Как нетрудно заметить, оба этих узла имеют общие точки с областью поиска, и поэтому для обоих из этих узлов рекурсивно вызывается процедура ПОИСК.

ПОИСК для вершины V_1 перебирает элементы V_3 , V_4 , V_5 , причем только V_5 имеет пересечение с прямоугольником $ABCD$. Поэтому вершины V_3 и V_4 пропускаются и далее не рассматриваются. Дальнейший вызов процедуры для вершины V_5 выдаст в качестве результата три

элемента – O_6 , O_7 , O_8 , которые и будут добавлены в множество результата Res .

Аналогичным образом будет просмотрена ветка V_2 . Из ее потомков только V_7 имеет общие точки с $ABCD$. Однако не один из элементов V_7 не пересекается с областью поиска. Данная ветка поиска оказалась ложной.

В результате поиска мы получаем список элементов, удовлетворяющих заданному запросу:

$$Res = \{O_6, O_7, O_8\}.$$

Другие виды поиска выполняются аналогично описанному. Для примера приведем листинг еще одной процедуры, выполняющей поиск объекта по точному совпадению с образцом. Процедура возвращает лист, в котором находится запрошенный объект, или $NULL$, если такого объекта в дереве нет.

Листинг 3.2

```
//=====
// Поиск листа, в котором находится объект O
// Параметры:
//   V – вершина, начиная с которой производится поиск
//   O – объект, который нужно найти.
//=====
ПОИСК_ОБЪЕКТА(V, O)
[1] Если V не является листом, то
    Проверить все записи V', находящиеся в узле V
    Если MBR(V') полностью содержит MBR(O), то
        L = ПОИСК_ОБЪЕКТА(V', O)
        Если L ≠ NULL, то
            Вернуть L
[2] Если V является листом, то
    Проверить все записи O', находящиеся в узле V
    Если O'=O, то
        Вернуть V
[3] Вернуть NULL
Конец ПОИСК_ОБЪЕКТА
```

Процедура очень похожа на ту, что представлена в листинге 3.1. Как и в предыдущем случае, первый шаг проверяет, является ли рассматриваемая вершина V листовой. Если эта вершина находится на внутреннем уровне дерева, алгоритм просматривает все ее дочерние элементы. Однако, в отличие от предыдущего алгоритма, проверяется не просто пересечение MBR узла дерева с запрошенным узлом, а полное его включение. Это ужесточение условия необходимо для отбрасывания веток поиска, с которыми объект пересекается, но не входит в них.

Если некоторый узел может содержать поисковый объект, то процедура выполняется для него рекурсивно. При этом необходимо проанализировать, что вернул этот рекурсивный вызов. Если

возвращенное значение равно *NULL*, то алгоритм продолжается дальше. Иначе – необходимо завершить процедуру поиска, так как листовой узел, содержащий объект *O*, уже найден. Это является еще одним отличием от предыдущего алгоритма.

Второй шаг алгоритма предназначен для листовой вершины. Он перебирает все элементы, содержащиеся в данном узле дерева, и проверяет их на равенство поисковому объекту. При нахождении соответствия процедура возвращает текущий лист.

И, наконец, если не один из предыдущих поисков не дал результата, то на третьем шаге необходимо вернуть значение *NULL*, идентифицирующее тот факт, что поиск в данном элементе не привел к положительному результату.

Алгоритм добавления нового объекта в R-дерево

Добавление нового объекта в *R*-дерево похоже на процедуру вставки в *B+*-дерево. Новая индексная запись добавляется в листовой узел. Если узел переполняется, то происходит его деление, в результате которого у предка появляется еще один потомок. Если предок также оказывается переполненным, то и он делится и т. д. Таким образом, вставка одного объекта может повлиять на структуру дерева в целом.

Процедура вставки объекта представлена в листинге 3.3.

Листинг 3.3

```
//=====
// Процедура вставки элемента в R-дерево
// Параметры:
//   O – объект, который нужно вставить в дерево
//=====
ВСТАВКА (O)
[1] L = ВЫБОР_ЛИСТА (O)
[2] Если число элементов в L меньше M, то
    Добавить объект O в узел L
    L" = NULL
    Иначе
        L" = ДЕЛЕНИЕ_УЗЛА (L, O)
[3] КОРРЕКТИРОВКА_ДЕРЕВА (L, L")
Конец ВСТАВКА
```

Первое, что делает процедура вставки элемента *O* в *R*-дерево, это ищет листовой узел, в который необходимо поместить данный объект (шаг 1). Процедура поиска такого листа является очень важным шагом, так как неправильно выбранная позиция может привести к неэффективности структуры в целом. Один из возможных вариантов этой процедуры показан в листинге 3.4.

После того, как узел для вставки выбран, производится непосредственно само размещение объекта в нем (шаг 2). При этом, если в листовом узле L есть место для новой записи, объект O помещается в него и процедура заканчивает свою работу. В противном случае, если узел L уже содержит максимально возможное число записей, то происходит деление узла на два новых L и L'' , которые содержат старые записи узла L и добавляемый объект O . Существует несколько вариантов реализации процедуры деления узла. Каждый из них имеет свои сильные и слабые стороны. Все они будут описаны далее.

После вставки объекта в дерево и возможного расщепления узла необходимо корректировать дерево (шаг 3). Процедура корректировки включает расширение границ минимального описывающего прямоугольника (MBR) для текущего узла и всех его предков. Также эта процедура распространяет расщепления узлов вверх по дереву, если это необходимо. Пример такой процедуры показан в листинге 3.5.

Рассмотрим алгоритмы упомянутых процедур подробнее.

Листинг 3.4

```
//=====
// Процедура поиска листа для вставки O в R-дерево
// Параметры:
//   O – объект, который нужно вставить в дерево
//=====
ВЫБОР_ЛИСТА (O)
[1] V = корень R-дерева
[2] Если V является листом, то
    Завершить процедуру и вернуть V
[3] Для всех потомков V' вершины V выбрать
    V'' = потомок, для которого min(MBR(V', O) - MBR(V'))
[4] V = V''
    Перейти к шагу 2
Конец ВЫБОР_ЛИСТА
```

Процедура начинает свой поиск с корня дерева (шаг 1), занося его в некоторую переменную V , отвечающую за текущий элемент поиска. Затем производится проверка вершины V , является ли она листом. Если данная вершина размещена на листовом уровне, то процедура завершает свою работу, возвращая в качестве результата вершину V (шаг 2).

На третьем шаге процедура перебирает всех потомков вершины V . Цель данного шага заключается в том, чтобы выбрать того потомка, чей MBR увеличится наименьшим образом при помещении в него объекта O (в идеале – вообще не изменится). В спорных ситуациях, когда найдено более одного потомка с одинаковым увеличением MBR , выбирать необходимо тот, который имеет меньшую площадь.

После выбора дочернего узла его заносят в переменную V и процедуру поиска повторяют с шага 2.

Теперь представим листинг еще одной процедуры, которая уже упоминалась ранее – процедура корректировки дерева.

Листинг 3.5

```
//=====
// Процедура корректировки R-дерева после вставки объекта
// Параметры:
//   L – вершина, MBR которой необходимо корректировать
//   L" – вершина, образовавшаяся при делении L на две
//       части. Если деление не производилось, L" = NULL.
//=====
КОРРЕКТИРОВКА_ДЕРЕВА (L, L")
[1] V = L, V" = L"
[2] Если V является корнем,
    Если V" ≠ NULL, то
        Root = новый корневой узел
        Поместить в Root элементы V и V"
        Выйти из процедуры
[3] P = Parent(V)
    PV = запись в узле P о потомке V
    Скорректировать MBR(PV)
[4] Если V" ≠ NULL, то
    PV" = новая запись о узле V"
    Если число элементов в P меньше M, то
        Добавить объект PV" в узел P
    Иначе
        P" = ДЕЛЕНИЕ_УЗЛА(P, PV")
[6] V = P
    V" = P"
    Перейти к шагу 2
Конец КОРРЕКТИРОВКА_ДЕРЕВА
```

Как было отмечено, процедура корректировки изменяет *MBR* всех вершин дерева, которые расположены выше листа с вставленным объектом. Второй и не менее важной функцией процедуры корректировки является распространение деления вершин вверх по дереву в случае, если будет происходить переполнение на внутренних узлах дерева.

В качестве параметров в процедуру передаются два новых узла, которые получились при вставке объекта в дерево. Если разбиение не произошло, то первым параметром передается старый узел, а второй параметр приравнивается в *NULL*.

На первом шаге процедура заносит переданные параметры в переменные V и V'' . Эти переменные будут отвечать за текущие вершины в дереве, которые необходимо исправить.

После этого происходит сравнение вершины V корня дерева. Если данная вершина является корнем, то это означает, что изменения уже распространились до верха дерева и необходимо просто завершить процедуру корректировки. Однако стоит учитывать один момент: если после предыдущих манипуляций произошло расщепление корня на два узла (переменная $V'' \neq \text{NULL}$), то необходимо создать новый корень дерева, узлами-потомками которого будут V и V'' .

Если предыдущий пункт не выполнен, то происходит корректировка. Для этого определяется предок узла V , а также запись в нем об этом узле. После этого MBR найденной записи изменяется таким образом, чтобы включать в себя все MBR дочерних элементов узла V , но при этом не содержать лишних областей.

Четвертый шаг алгоритма выполняется только в том случае, если предыдущие действия вызвали деление узла. В этом случае у нас в переменной V'' будет находиться вершина с элементами, которые пока еще не помещены в дерево. Для этой вершины необходимо создать запись $P_{V''}$, которая будет содержать минимальный описывающий прямоугольник для данной вершины и ссылку на саму вершину. Эту запись и нужно разместить в предке узла V .

Однако при помещении в узел P записи $P_{V''}$ необходимо помнить, что данная операция может привести к переполнению и тогда придется разбивать узел P на два новых.

После всех описанных операций в переменные V и V'' заносятся новые значения P и P'' соответственно, и алгоритм повторяется заново с шага 2.

Алгоритм удаления объекта из R-дерева

Для того, чтобы структуру можно было считать полностью динамической, необходима поддержка двух операций: добавление новых элементов и удаление уже существующих. Добавление элементов было рассмотрено выше. В листинге 3.6 представлена процедура удаления. Кроме удаления объекта из дерева, она должна корректировать дерево для сохранения его свойств.



Листинг 3.6

```
//=====
// Процедура удаления объекта из R-дерева.
// Параметры:
//   O - объект, который нужно удалить.
//=====
УДАЛЕНИЕ(O)
[1] V = корень дерева
    L = ПОИСК_ОБЪЕКТА(V,O)
    Если L = NULL, то
```

- Завершить процедуру удаления
- [2] Удалить объект O из L
 $V = L$
 $Q = \text{пустое множество}$
 - [3] Если узел V является корнем, то
 Перейти к шагу 7
 - [4] $P = \text{Parent}(V)$
 $PV = \text{запись в узле } P \text{ о потомке } V$
 - [5] Если число элементов в V меньше m , то
 Удалить PV из P
 Переместить все элементы из V в множество Q
 Удалить V
 Иначе
 Скорректировать $MBR(V)$
 - [6] $V = P$
 Перейти к шагу 3
 - [7] Если у корня всего один потомок, то
 Удалить корневой узел
 Сделать новым корнем этого потомка
 - [8] Вставить узлы из множества Q обратно в дерево
- Конец УДАЛЕНИЕ

Первое, что производит процедура удаления объекта O из R -дерева, это ищет листовой узел, в котором находится данный объект. Для этого используется процедура поиска ПОИСК_ОБЪЕКТА, описанная в листинге 3.2. В качестве параметров ей передается вершина, с которой нужно начать поиск (в нашем случае это корень) и объект поиска. Если объект не будет найден, то данная процедура вернет $NULL$. При этом необходимо завершить и процедуру удаления.

На втором шаге удаляется объект O из узла L и готовятся временные переменные для коррекции дерева. В переменную V (текущая вершина для коррекции) заносится листовой узел L , а в переменную Q – пустое множество (это множество вершин, которые необходимо потом вставить в дерево заново).

Далее необходимо проверить, является ли вершина V корнем. Если V – корневая вершина, то шаги 4–6 нужно пропустить и перейти сразу к седьмому пункту алгоритма. Иначе – находим предка для вершины V и в нем определяем запись, ссылающуюся на $V(P_V)$.

Если в рассматриваемом узле число записей меньше минимального (m), то необходимо удалить этот узел из дерева. При этом все элементы из V помещаются в множество Q (чтобы потом снова быть размещенными в дереве, но в других вершинах) и из вершины P удаляется элемент, ссылающийся на удаленную вершину (удаляется P_V).

Если же записей в вершине V больше, чем заданный параметр m , то удалять вершину не нужно. При этом необходимо просто скорректировать MBR узла таким образом, чтобы он охватывал все

прямоугольники дочерних узлов, но при этом не включал лишнего пространства (после удаления узлов вполне вероятно можно будет сузить *MBR*, который хранится в записи P_i).

После проделанных операций необходимо распространить сделанные изменения вверх по дереву (скорректировать *MBR* узла предка или, возможно, даже удалить его, если он оказался не заполненным до предела m). Для этого в переменную V заносится предок текущей вершины и повторяется алгоритм с шага 3.

После того, как все изменения дойдут до корня, алгоритм продолжится с шага 7. Исходя из свойств R -дерева, описанных в начале данного параграфа, корень должен иметь не меньше двух потомков. Поэтому необходимо просто проверить число дочерних узлов у корня и при нахождении там всего одного потомка сделать его новым корнем дерева.

Последнее, что необходимо выполнить в процедуре удаления, это вставить временно удаленные узлы из множества Q обратно в дерево. Данная процедура выполняется полностью аналогично описанной ранее процедуре ВСТАВКА за исключением: вершины из множества Q необходимо разместить на тех же уровнях, на которых они были до процедуры удаления. Этого требования необходимо придерживаться для того, чтобы не нарушить сбалансированность дерева (одно из свойств R -дерева заключается в том, что все листовые узлы находятся в нем на одном уровне).

Алгоритм разбиения узла

Изменение данных прикладной задачи требует частого изменения индексной структуры. Для добавления новой записи в уже заполненный узел R -дерева, содержащий M записей, необходимо распределить $M+1$ элемент между двумя узлами. Процедура разбиения узла может быть вызвана не только при добавлении новых элементов в индекс, но и при перестройке дерева, при удалении ненужной записи, при обновлении данных или даже при его корректировке.

Алгоритм, выполняющий деление узла, особенно важен, так как плохое разбиение может сильно затруднить операции поиска по дереву. Разбиение узла без учета критериев оптимальности построения дерева приводит к увеличению времени работы процедуры поиска конкретного объекта, а следовательно, к ухудшению работы индексной структуры в целом. При плохом разбиении узлы дерева разрастаются вдоль осей координат и захватывают много пространства, не содержащего ни одного объекта. Такой пример показан на рис. 3.11. С одной стороны вариант (а) обеспечивает нулевое перекрытие двух узлов дерева. Однако суммарная площадь этих узлов будет значительно больше самих узлов, что вызовет многократное ложное срабатывание процедуры поиска.

При большой площади пространства, соответствующей узлам дерева, запросу поиска на промежуточных стадиях работы может удовлетворять большое число записей (более одной), хотя, в конечном счете, на каждом уровне интересует только одна. Следовательно, алгоритм будет ветвиться и обходить дерево неоптимальным путем, включая обход ненужных узлов, что может сильно отразиться на скорости работы индексной структуры. Кроме того, обход ненужных узлов потребует дополнительного расхода оперативной памяти. В условиях большого числа запросов это обстоятельство также может стать критичным.

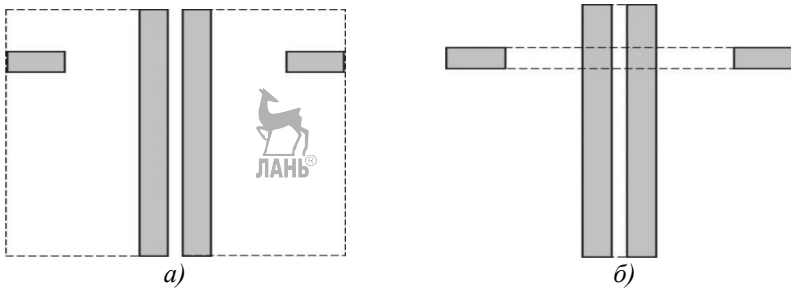


Рис. 3.11. Пример возможного разбиения узла на два новых: а – плохое разбиение; б – разбиение с пересечением

За 25 лет существования R -деревьев было предложено достаточно много различных модификаций алгоритма деления. Здесь будут приведены только некоторые из них.

Исчерпывающий перебор. Конечно, самый простой из алгоритмов, при котором получается максимально оптимальное дерево, это алгоритм полного перебора: для нахождения минимальной площади покрытия, а также всех остальных параметров оптимальности деления необходимо произвести все возможные разбиения $M+1$ записи на две группы и выбрать наилучшее.

Задача разбиения в общем случае является NP -полной. В настоящее время неизвестно полиномиального алгоритма выполнения такой операции. Этот вариант применим при M не более 5, но при $M = 50$, например, алгоритм работает очень долго. Поэтому данный подход применяется редко, а во всех вариантах построения R -деревьев используются эвристические подходы, дающие на реальных данных не всегда оптимальные разбиения (но практически всегда достаточно эффективные для решения конкретной задачи).

Квадратичный алгоритм. Данный алгоритм был предложен основателем R -деревьев А. Гуттманом. В нем осуществлена попытка

найти такое деление, при котором площадь охватывающих прямоугольников будет минимальна. Однако при этом не гарантируется, что это будет действительно наилучший вариант. Алгоритмическая сложность изменяется по квадратичному закону относительно M и по линейной – относительно числа измерений.

Процедура деления узла представлена в листинге 3.7.



Листинг 3.7

```
//=====
// Процедура деления узла
// Параметры:
//   О - объект, который не поместился в вершину L
//   L - вершина, которую необходимо разбить на две
//=====
ДЕЛЕНИЕ_УЗЛА (L, O)
[1] Q = множество всех элементов из L и элемент O
    Удалить все элементы из L
    Создать новую вершину L"
[2] O1, O2 = ВЫБОР_ПЕРВОЙ_ПАРЫ(Q)
    Добавить O1 в L, O2 в L"
[3] n = количество элементов в множестве Q
    n1 = количество элементов в вершине L
    n2 = количество элементов в вершине L"
    Если n = 0, то
        Завершить процедуру и вернуть L"
    Если m-n1 ≥ n, то
        Переместить все объекты из Q в вершину L
        Завершить процедуру и вернуть L"
    Если m-n2 ≥ n, то
        Переместить все объекты из Q в вершину L"
        Завершить процедуру и вернуть L"
[4] O' = ВЫБОР_СЛЕДУЮЩЕГО(L, L", Q)
    d1 = MBR(L, O') - MBR(L)
    d2 = MBR(L", O') - MBR(L")
    Если (d1 < d2) или (d1 = d2 и n1 < n2), то
        Добавить O' в L
    Иначе
        Добавить O' в L"
[5] Перейти к шагу 3
Конец ДЕЛЕНИЕ_УЗЛА
```



Алгоритм начинается с подготовительных операций (шаг 1). Все элементы, которые нужно будет распределить между двумя новыми вершинами, переносятся в множество Q . Создается еще одна пустая вершина L'' . Таким образом мы получаем две пустые вершины (L и L'') и множество элементов Q , которые необходимо распределить по этим вершинам.

Затем алгоритм выбирает пару элементов, которые покрывают наибольшую площадь (соответствующая процедура представлена в листинге 3.8). Для этого рассчитывается коэффициент

$$A = MBR(O_1, O_2) - MBR(O_1) - MBR(O_2),$$

где $MBR(O_1, O_2)$ – площадь прямоугольника, охватывающего обе записи, а $MBR(O_1)$ и $MBR(O_2)$ – площади прямоугольников соответствующих объектов. Этот коэффициент показывает неэффективность объединения двух данных объектов в одну группу. Элементы, на которых достигается максимум коэффициента A , становятся первыми элементами двух будущих групп.

Листинг 3.8

```
//=====
// Процедура выбора первой пары элементов для деления
// Параметры:
//   Q – список всех нераспределенных элементов
//   L, L'' – вершины, между которыми происходит
//           распределение
//=====
ВЫБОР_ПЕРВОЙ_ПАРЫ(L, L'', Q)
[1] Для каждой пары O1 и O2 из множества Q вычислить
     A = MBR(O1, O2) - MBR(O1) - MBR(O2)
[2] O1, O2 = та пара, для которой A максимально
     Исключить O1 и O2 из множества Q
     Вернуть O1 и O2
Конец ВЫБОР_ПЕРВОЙ_ПАРЫ
```

Оставшиеся записи распределяются в группы по одной (шаг 4 алгоритма ДЕЛЕНИЕ_УЗЛА). Для этого вызывается процедура ВЫБОР_СЛЕДУЮЩЕГО, на которую и возложена задача выбора элемента из множества Q , который будет распределен следующим. Примеры реализации данной процедуры показаны в листингах 3.9–3.11.

После выбора элемента для вставки он добавляется в вершину, ограничивающий прямоугольник которой потребует минимального увеличения площади (если d_1 и d_2 – увеличение площади ограничивающих прямоугольников при добавлении элемента O' в вершины L и L'' соответственно, то вставку нужно произвести в ту вершину, для которой d меньше). При равенстве увеличения площадей ($d_1 = d_2$), для вставки выбирается та вершина, в которой меньше число записей.

Описанное действие продолжается до тех пор, пока не будут выбраны все элементы из множества Q . Однако на каждом необходимо проверять выполнимость условия минимального наполнения узла (число элементов в любой вершине дерева, кроме корня, должно быть не меньше m). Если на каком-то шаге окажется, что для выполнения этого

условия необходимо все оставшиеся в Q элементы переместить в одну из вершин, то необходимо сделать это и завершить процедуру деления узла (шаг 3).

Далее приведем несколько стратегий выбора следующего элемента для распределения между вершинами, на который ссылается шаг 4. В листинге 3.9 приведен алгоритм, предложенный Гуттманом в оригинале статьи по R -деревьям.

Листинг 3.9

```
//=====
// Процедура выбора следующего элемента для
// распределения
// Параметры:
//   Q      - список всех нераспределенных элементов
//   L, L'' - вершины, между которыми происходит
//             распределение
//=====
ВЫБОР_СЛЕДУЮЩЕГО (L, LL, Q)
  [1] Для всех элементов O' из множества Q вычисляем
        d1 = MBR (L, O') - MBR (L)
        d2 = MBR (L'', O') - MBR (L'')
        a = |d1-d2|
  [2] O = запись, для которой значение A максимально
        Исключить O из множества Q
        Вернуть O
Конец ВЫБОР_СЛЕДУЮЩЕГО
```

Представленный в листинге алгоритм достаточно прост. Для каждого нераспределенного элемента высчитывается площадь охватывающего прямоугольника, который получится после присоединения этого элемента к каждой группе (значения d_1 и d_2). Элемент с наибольшей разницей площадей обеих групп выбирается как следующий элемент.

Существует несколько другая стратегия, предложенная позже (листинг 3.10): выбирается элемент, присоединение которого в какую-либо группу минимально увеличит площадь охватывающего прямоугольника группы.

Листинг 3.10

```
//=====
// Процедура выбора следующего элемента для распределения
// Параметры:
//   Q      - список всех нераспределенных элементов
//   L, L'' - вершины, между которыми происходит
//             распределение
//=====
ВЫБОР_СЛЕДУЮЩЕГО (L, LL, Q)
```

```

[1] Для всех элементов  $O'$  из множества  $Q$  вычисляем
       $d1 = MBR(L, O') - MBR(L)$ 
       $d2 = MBR(L'', O') - MBR(L'')$ 
       $d = \min\{d1, d2\}$ 
[2]  $O$  = запись, для которой значение  $d$  минимально
      Исключить  $O$  из множества  $Q$ 
      Вернуть  $O$ 
Конец ВЫБОР_СЛЕДУЮЩЕГО

```

Линейный алгоритм. Линейный алгоритм в классическом варианте отличается от квадратичного системой выбора двух первых элементов групп и выбором следующего элемента. Сама же процедура деления вершины из листинга 3.7 остается без каких-либо изменений. Главным преимуществом данного алгоритма является то, что он линеен и по числу элементов M , и по числу измерений.

Процедура выбора следующего элемента выполняет простой выбор одной из оставшихся записей без проверки каких-либо условий. Она представлена в листинге 3.11.



Листинг 3.11

```

//=====
// Процедура выбора следующего элемента для распределения
// Параметры:
//   Q      - список всех нераспределенных элементов
//   L, L'' - вершины, между которыми происходит
//             распределение
//=====
ВЫБОР_СЛЕДУЮЩЕГО (L, L'', Q)
  [1]  $O$  = первая запись из множества  $Q$ 
      Исключить  $O$  из множества  $Q$ 
      Вернуть  $O$ 
Конец ВЫБОР_СЛЕДУЮЩЕГО

```

Процедура выбора первой пары элементов также максимально упрощена, чтобы иметь линейные характеристики. По каждой из осей координат выбираются две записи со следующими характеристиками: первая – чей охватывающий прямоугольник имеет наименьшую из верхних границ, вторая – это чей охватывающий прямоугольник имеет наибольшую из нижних границ. Вычисляется и нормализуется разница между этими границами (нормализуется путем деления разницы на общий размер распределения по данному измерению). После проделанных операций в качестве начальной пары выбирается та, которая имеет наибольшую длину нормализованной разницы.

Листинг 3.12

```
//=====
// Процедура выбора первой пары элементов для деления
// Параметры:
//   Q      - список всех нераспределенных элементов
//   L, L"   - вершины, между которыми происходит
//             распределение
//=====
ВЫБОР_ПЕРВОЙ_ПАРЫ(L, L", Q)
  [1] Для каждого измерения  $d$  вычисляется
      O1 = прямоугольник с максимальной нижней границей
      O2 = прямоугольник с минимальной верхней границей
      
$$d = \frac{[\text{верхняя граница } O2] - [\text{нижняя граница } O1]}{[\text{размер по измерению}]}$$

  [2] O1, O2 = та пара, для которой  $d$  максимально
      Исключить O1 и O2 из множества Q
      Вернуть O1 и O2
Конец ВЫБОР_ПЕРВОЙ_ПАРЫ
```

Алгоритм начального построения дерева

Структура R -дерева является динамической, и для нее разработаны процедуры вставки и удаления. Более того, само дерево строится путем поочередного добавления элементов. Однако такое построение может привести к большим перекрытиям между узлами, а глобальная оптимизация дерева при добавлении объектов в алгоритмах не предусмотрена.

На практике взаимодействие с индексными структурами состоит из двух основных этапов: начального построения дерева и последующей оперативной работы. При этом во многих случаях во время второго этапа производится только поиск объектов без вставки новых и удаления старых.

В работе А. В. Скворцова предлагается учесть этот факт с помощью специально разработанных алгоритмов начального построения дерева, получивших название глобальных алгоритмов построения [12]. При этом алгоритмы работы с деревом остаются прежними. В частном случае, когда вначале ничего не строится, получается обычное R -дерево.

Алгоритм начального построения дерева, близкого к оптимальному, представлен в листинге 3.13.



Листинг 3.13

```
//=====
// Процедура начального построения дерева
// Параметры:
//   Q      - список всех элементов
```

```

//=====
ПОСТРОИТЬ_ДЕРЕВО(Q)
    [1] Root = корень дерева (пустая вершина)
        N = количество элементов в множестве Q
        Lev = количество уровней строящегося дерева
    [2] Вызвать алгоритм ПОСТРОИТЬ_УЗЕЛ(Root, Lev-1, Q)
Конеч ПОСТРОИТЬ_ДЕРЕВО

//=====
// Процедура построения узла по заданному
// набору вершин
// Параметры:
//   V - вершина, потомков которой нужно построить
//   Lev - количество уровней от вершины V до листьев
//   Q - список элементов, распределяемых по потомкам V
//=====
ПОСТРОИТЬ_УЗЕЛ(V, Lev, Q)
    [1] Если Lev = 0, то
        V является листовым узлом
        Поместить в V все объекты из множества Q
        Выйти из процедуры
    [2] N = количество элементов в множестве Q
         $K = \lceil \sqrt[L+1]{N} \rceil$  - количество потомков данного узла
    [3] {G} = РАЗДЕЛИТЬ_НА_ГРУППЫ(Q, K, Lev)
    [4] Для каждой группы Gi из множества групп {G}
        Создать пустую вершину Vi
        Сделать узел Vi потомком узла V
        Вызвать процедуру ПОСТРОИТЬ_УЗЕЛ(Vi, Lev-1, Gi)
Конеч ПОСТРОИТЬ_УЗЕЛ

```

Алгоритм глобального построения дерева интуитивно понятен. Для построения необходимо вызвать процедуру ПОСТРОИТЬ_ДЕРЕВО из листинга 3.13, передав ей в качестве параметров множество всех элементов. Эта процедура создаст корень дерева *Root* и рассчитает количество уровней строящегося дерева. Так как вершина *R*-дерева может содержать максимально *M* потомков, то глубина дерева находится по формуле $\log_M(N)$. Причем данное выражение необходимо округлить в большую сторону. Так, если мы строим дерево из 1000 вершин, а число потомков вершины выбрано $M = 5$, то глубина дерева получится равной 5.

После расчета глубины дерева производится вызов процедуры, которая должна распределить все элементы множества *Q* по потомкам переданного ей узла. Эта процедура также представлена в листинге 3.13.

Процедура ПОСТРОИТЬ_УЗЕЛ является рекурсивной. Она строит не только потомков корневого узла, но и всех остальных, вплоть до листовых вершин, при помощи вызова самой себя. Поэтому в качестве необходимых параметров она принимает не только текущий узел *V* и

список объектов Q , но еще и число уровней Lev , которое осталось до листьев дерева.

Самое первое, что необходимо сделать в процедуре построения узла, это проверить, не достиг ли алгоритм листового уровня (шаг 1). Если это произошло, то параметр Lev будет равен 0. При этом необходимо просто переместить все элементы из множества Q в вершину V и выйти из процедуры.

Если уровень листьев достигнут не был, то процедура рассчитывает число групп, на которое необходимо разбить множество элементов Q , чтобы можно было построить оптимальное дерево (шаг 2), и вызывает процедуру РАЗДЕЛИТЬ_НА_ГРУППЫ, которая и произведет такое деление (шаг 3). Процедура деления является центральной во всем алгоритме. От ее работы зависит, насколько оптимальным будет построенное дерево. А. В. Скворцов [12] предложил несколько вариантов этой процедуры, представленных в следующем подпункте.

Процедура РАЗДЕЛИТЬ_НА_ГРУППЫ возвращает набор групп $\{G\}$. На последнем шаге разбиения для каждой группы G_i создается вершина V_i , дочерняя по отношению к V , и для нее вызывается рекурсивно процедура ПОСТРОИТЬ_УЗЕЛ.

Базовый алгоритм деления на группы. Ранее был рассмотрен алгоритм деления множества объектов на две группы с близким к минимальному пересечению. Для этого по некоторому принципу выбиралось два базовых объекта из множества, и все остальные объекты распределялись последовательным присоединением к двум образовавшимся группам.

Глобальные алгоритмы деления на группы используют обобщение данного подхода. Из общего множества объектов выбирается K базовых объектов (по числу групп, которые необходимо создать). Оставшиеся объекты распределяются по группам, используя аналогичные описанным итерационные процедуры.

В листинге 3.14 представлен базовый вариант данного алгоритма для двумерного случая.

Листинг 3.14

```
//=====
// Процедура разбиения множества объектов Q на K групп
// Параметры:
//   Q - список всех нераспределенных элементов
//   K - количество групп, которое должно получиться в
//       результате разбиения
//   Lev - количество уровней от текущей вершины до
//         листьев
//=====
РАЗДЕЛИТЬ_НА_ГРУППЫ(Q, K, Lev)
```

[1] Для каждого измерения вычислить

Min_i = минимальная граница по i -му измерению
 Max_i = максимальная граница по i -му измерению
 N_i = количество групп, на которое нужно разбить i -е измерение

- [2] Разбить каждое измерение равностоящими линиями $\text{Min}_i + j * (\text{Min}_i + \text{Max}_i) / N_i$, где j меняется от 0 до N_i
 Создать ячейки, образованные решеткой
- [3] // Удаление пустых ячеек
 Для каждой ячейки Cell выполнить
 Если ячейка Cell пуста, то
 Удалить ячейку Cell
- [4] // Создание недостающих ячеек
 Пока количество ячеек $< K$ выполнять
 Cell = ячейка, содержащая более одного объекта
 Разбить ячейку Cell на две новые ячейки
- [5] // Построение групп
 Создать K групп $\{G\}$
 Для каждой группы G_i из множества $\{G\}$ выполнить
 O = любой объект из i -й ячейки пространства
 Поместить O в группу G_i
 Сделать размер группы G_i равным O
 Удалить O из множества Q
- [6] // Распределение оставшихся объектов по группам
 Для каждого объекта O из множества Q выполнить
 Поместить O в подходящую для него группу
 Удалить O из множества Q
- [7] Вернуть набор групп $\{G\}$

Конец РАЗДЕЛИТЬ_НА_ГРУППЫ

Представленный алгоритм в целом состоит из двух больших блоков. В первом блоке (шаги 1–5) производится разбиение пространства на ячейки с помощью равноотстоящих друг от друга линий (см. рис. 3.12). После этого в качестве базовых для каждой группы выбирается по одному произвольному объекту, попадающему в построенные ячейки. Если ячеек не хватает (число ячеек меньше количества групп, на которое нужно разбить множество объектов), то необходимо разделить любую из ячеек с несколькими объектами пополам и взять из получившихся частей по объекту.

После выполнения этих шагов получаем K групп, содержащих по одному базовому объекту. Далее необходимо все оставшиеся объекты (не базовые) поместить в образовавшиеся группы по принципу минимального увеличения их площадей (аналогично тому, как это было описано в процедуре деления узла в листинге 3.7). При этом добавление объекта в группу не должно привести к переполнению группы (число объектов должно быть меньше M^{Lev}), а также к ситуации, когда оставшихся объектов недостаточно для обеспечения минимально допустимого числа объектов в других группах (m^{Lev}).

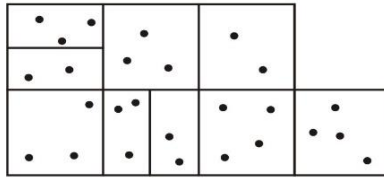


Рис. 3.12. Пример деления

Рассмотрим двумерный случай более подробно на примере распределения точек, представленных на рис. 3.12. При этом имеется всего два измерения – высота и ширина, а число групп разбиения примем равным девяти ($K = 9$).

Первым шагом алгоритма является вычисление минимального и максимального значения по каждому измерению. Пусть минимальным значением для ширины будет 0, а максимальным – W . Аналогично для высоты за минимальное значение примем 0, а за максимальное – H . Следующим этапом является выбор числа ячеек, на которое нужно разбить все пространство по высоте и ширине. Если в рассматриваемой задаче пространство представляет собой квадратную область ($W = H$), то число ячеек можно выбрать следующим образом:

$$N_{\text{длина}} = N_{\text{высота}} = \lfloor \sqrt{K} \rfloor.$$

Однако в реальных ситуациях область чаще всего представляет собой прямоугольник, у которого длина не равна высоте ($W \neq H$). При этом, чтобы пространство разбить линиями, равноотстоящими друг от друга, необходимо выбрать разное число ячеек по высоте и длине. Один из вариантов такого выбора представлен ниже.

$$N_{\text{ширина}} = \lfloor \sqrt{K * W/H} \rfloor$$

$$N_{\text{высота}} = \lfloor K / N_{\text{ширина}} \rfloor.$$

Для рассматриваемого примера (рис. 3.12) $N_{\text{ширина}} = 4$, а $N_{\text{высота}} = 2$ (так как на рисунке высота меньше ширины в два раза, т. е. $W/H = 2$).

Рассчитав параметры деления, производим само разбиение (шаг 2). Для этого проводим три вертикальные линии (через $W/4$) и одну горизонтальную (посередине). В результате получаем решетку из 8-ми ячеек.

Следующим шагом алгоритма является удаление пустых ячеек, т. е. ячеек, в которые не попало ни одного объекта. Такой ячейкой в нашем случае является верхняя левая. В результате остается всего 7 ячеек. Так как после деления необходимо получить 9 групп, то необходимо две

любые ячейки разбить пополам. В качестве критерия выбора кандидатов на разбиение можно принять во внимание тот факт, что предпочтительно разбивать ячейки, в которых попало максимальное число объектов.

На пятом шаге алгоритма создается 9 групп (по одной для каждой непустой ячейки), и в каждую из них помещается любой из объектов этой ячейки. После этого набор групп $\{G\}$ сформирован и остается только распределить по ним все остальные объекты (шаг 6) путем последовательного присоединения объектов к группам. Этот алгоритм подробно рассмотрен в предыдущем параграфе в процедуре ДЕЛЕНИЯ_УЗЛА. Единственное, что необходимо помнить при последовательном распределении объектов по группам – каждая группа должна содержать не более M^{lev} и не менее m^{lev} объектов. Для этого на каждом шаге распределения нужно проверять следующие условия:

- если группа, в которую необходимо добавить очередной объект, переполнена, то выбирается другая группа для добавления;
- если осталось нераспределенными ровно столько объектов, сколько необходимо для выполнения условия минимальности заполнения узлов, то оставшиеся объекты распределяются только по незаполненным группам.

Одним из недостатков работы такого алгоритма является недостаточно высокое качество разбиения на неравномерных распределениях, а также в случаях, когда число объектов незначительно превосходит K (например, при построении нижних уровней дерева). Это проявляется в ситуациях, когда на шестом шаге работы алгоритма некоторые группы уже заполнены, но при этом в соответствии с критерием выбора в них необходимо добавить очередной объект. Так как это сделать невозможно, приходится добавлять объекты в некоторые другие группы, что в конечном итоге приводит к сильному перекрытию образовавшихся групп.

В качестве одного из вариантов решения данной проблемы автором предлагается временно удалять объекты, которые нельзя поместить в требуемые группы, из списка участвующих в глобальном алгоритме. При этом после окончания его работы все эти объекты необходимо поместить в дерево, используя уже обычные динамические алгоритмы построения R -дерева.

Как показало проведенное А. В. Скворцовым экспериментальное исследование работы данного алгоритма, число объектов, которые нельзя разместить в требуемые группы, чаще всего незначительно, и они эффективно размещаются по окончании работы глобального алгоритма динамическим способом.

Клеточный алгоритм деления на группы. Как показало экспериментальное исследование, описанный выше алгоритм хорошо ведет себя на равномерных распределениях непересекающихся объектов. В других же случаях лучше использовать другой вариант алгоритма.

В нем на основе клеточного разбиения плоскости строятся группы объектов, и только затем анализируются их число и наполнение. При необходимости полученные группы делятся пополам или объединяются с другими. Данная процедура представлена в листинге 3.15.

Листинг 3.15

```
//=====
// Процедура разбиения множества объектов Q на K групп
// Параметры:
//   Q - список всех нераспределенных элементов
//   K - количество групп, которое должно получиться в
//       результате разбиения
//   Lev - количество уровней от вершины до листьев
//=====
РАЗДЕЛИТЬ_НА_ГРУППЫ(Q, K, Lev)
[1] Для каждого измерения вычислить
    Mini = минимальная граница по i-му измерению
    Maxi = максимальная граница по i-му измерению
    Ni = количество групп, на которое нужно
        разбить i-е измерение
[2] Разбить каждое измерение равностоящими линиями
    Mini + j * (Mini + Maxi) / Ni, где j меняется от 0 до Ni
    Создать ячейки, образованные решеткой
[3] // Уничтожение неполных ячеек
    Для каждой ячейки Celli выполнить
        Если ячейка Celli пустая, то
            Удалить ячейку Celli
        Если Celli содержит меньше mLev объектов, то
            Cellj = ячейка, у которой
                MBR(Cellj, Celli) минимальна для всех j
            Объединить Celli и Cellj
[4] // Уничтожение лишних ячеек
    Пока количество ячеек > K выполнять
        Celli = ячейка с минимальным количеством объектов
        Cellj = ячейка, у которой
            MBR(Cellj, Celli) минимальна для всех j
        Объединить Celli и Cellj
[5] // Создание недостающих ячеек
    Пока количество ячеек < K выполнять
        Cell = ячейка с максимальным количеством объектов
        ДЕЛЕНИЕ_ЯЧЕЙКИ(Cell)
[6] // Распределение переполненных ячеек
    Для каждой ячейки Celli выполнить
        Если Celli содержит больше MLev объектов, то
            Cellj = ближайшая к Celli ячейка, в
                которой число объектов меньше MLev
            O = объект из ячейки Celli, такой, что
                MBR(Cellj, O) минимально для всех вариантов
            Удалить O из Celli
```

```

        Поместить O в Cellj
[7] // Создание групп
    Для каждой ячейки Celli выполнить
        Создать группу Gi
        Поместить в группу Gi все объекты из Celli
[8] Вернуть набор групп {G}.
Конец РАЗДЕЛИТЬ_НА_ГРУППЫ

```

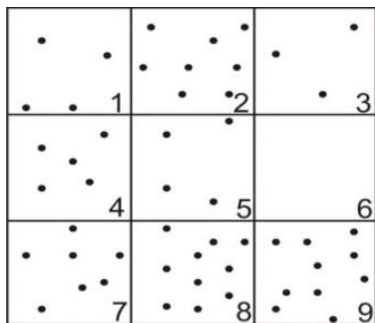
Для пояснения работы алгоритма снова рассмотрим двумерный случай. Распределение объектов в пространстве и шаги алгоритма представлены на рис. 3.13. При этом число групп разбиения равно восьми ($K = 8$), а максимальное и минимальное число объектов – 9 и 4 соответственно ($M^{Lev} = 9, m^{Lev} = 4$).

Первые два шага алгоритма очень похожи на предыдущую реализацию базового алгоритма. Они разбивают пространство равноотстоящими линиями на ячейки. Однако в них есть некоторое различие. Если при вычислении числа ячеек по высоте и длине в предыдущем алгоритме использовалось округление в меньшую сторону (поэтому в общем случае после такого разбиения получалось меньше ячеек, чем требуется), то в этой реализации алгоритма используется округление в большую сторону.

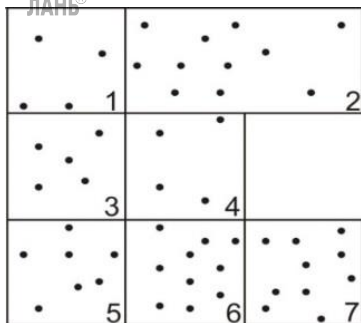
В нашем случае поле имеет квадратную форму, поэтому число ячеек по ширине и высоте можно вычислить по формуле

$$N_{\text{длина}} = N_{\text{высота}} = \lceil \sqrt{K} \rceil = 3.$$

Таким образом, все поле разбивается на три части по высоте и на три – по длине. Всего после такого разбиения получается 9 ячеек (рис. 3.13(a)).



a)



б)

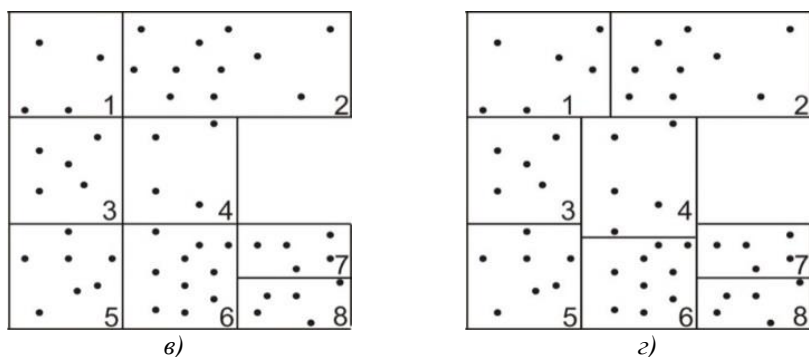


Рис. 3.13. Клеточное разбиение: а – начальное разбиение; б – разбиение после объединения; в – разбиение после деления; г – разбиение после перераспределения

На третьем шаге алгоритма уничтожаются неполные группы. Для этого просматриваются все образовавшиеся ячейки и проверяется число объектов, попавших в них. Если в какой-то из ячеек нет ни одного объекта, то такая группа удаляется (в нашем случае это ячейка с номером 6). Для ячеек с недостаточным числом объектов (ячейки, число объектов в которых меньше m^{Lev}) выполняется слияние с некоторой другой ячейкой. При этом для слияния необходимо выбирать такую ячейку, чтобы площадь минимального охватывающего прямоугольника получившейся группы была минимальной из всех возможных. В результате выполнения третьего шага алгоритма получится распределение, показанное на рис. 3.13(б).

Четвертый шаг алгоритма выполняет функцию удаления лишних ячеек. Так как при разбиении всего пространства на ячейки производилось округление в большую сторону, такая ситуация вполне могла возникнуть. Если число ячеек больше заданного (больше K), необходимо выбрать группу с минимальным числом объектов и объединить ее с другой, используя критерий, описанный выше.

Однако из-за объединений ячеек на третьем шаге может возникнуть и обратная ситуация – число ячеек меньше заданного K . В этом случае необходимо произвести противоположное действие – выбрать ячейки с максимальным числом объектов и разделить их на две части. Для этого вызывается процедура ДЕЛЕНИЕ_ЯЧЕЙКИ, абсолютно аналогичная той, что представлена в листинге 3.7. Деление пространства после выполнения пяти шагов алгоритма представлено на рис. 3.13(в).

После выполнения всех описанных шагов рассматриваемое пространство разбито ровно на K ячеек, причем выполняется условие минимального наполнения – в каждой ячейке не меньше m^{Lev} объектов.

Однако при этом может сложиться ситуация, при которой число объектов в ячейке может превысить верхний предел заполнения M^{Lev} . Так, в нашем примере (рис. 3.13(в)) таких ячеек две – ячейка с номером 2 (11 объектов вместо 9) и ячейка с номером 6 (10 объектов вместо 9). Для всех таких ячеек необходимо выполнить перераспределение части объектов между другими ячейками.

Для выполнения перераспределения выбирается ячейка, ближайшая к данной (например ближайшая в смысле расстояния между центрами) и имеющая меньше M^{Lev} объектов. После этого из переполненной ячейки выбирается объект, перемещение которого в выбранную ячейку вызовет минимальное увеличение ее площади, и производится его перемещение. Данный процесс повторяется до тех пор, пока будут существовать ячейки с числом объектов больше M^{Lev} . Распределение ячеек после выполнения этой операции представлено на рис. 3.13(г).

На последнем шаге алгоритма создаются группы для каждой ячейки и в них помещаются соответствующие объекты. После этого набор групп возвращается из процедуры деления.

Одним из недостатков работы такого алгоритма является увеличение времени работы на неравномерных распределениях, так как при этом часто происходит появление пустых и переполненных групп, а также приходится выполнять перемещение объектов по группам. В худшем случае алгоритм может иметь квадратичную сложность при многократном перемещении объектов по группам на последних шагах.

Алгоритм деления на группы «разделяй и властвуй». Рассмотрим еще один вариант алгоритма разбиения, основанный на стратегии «разделяй и властвуй», в которой происходит последовательное разбиение всего множества на две части до тех пор, пока не появится требуемое число частей.

Листинг 3.16

```
//=====
// Процедура разбиения множества объектов Q на K групп
// Параметры:
//   Q – список всех нераспределенных элементов
//   K – количество групп, которое должно получиться в
//       результате разбиения
//   Lev – количество уровней от вершины до листьев
//=====
РАЗДЕЛИТЬ_НА_ГРУППЫ(Q, K, Lev)
[1] // Деление на две группы
    K1 = ⌊ K/2 ⌋
    K2 = K - K1
    {G1, G2} = РАЗДЕЛИТЬ_НА_ДВЕ_ГРУППЫ(Q, K1, K2, Lev)
[2] // Рекурсивное деление первой полученной части
    Если K1 > 1, то
```



```

    {G'} = РАЗДЕЛИТЬ_НА_ГРУППЫ(G1, K1, Lev)
Иначе
    {G'} = G1
[3] // Рекурсивное деление второй полученной части
    Если K2 > 1, то
        {G''} = РАЗДЕЛИТЬ_НА_ГРУППЫ(G2, K2, Lev)
    Иначе
        {G''} = G2
[4] {G} = все элементы множества {G'} и {G''}
    Вернуть набор из множеств {G}
Конец РАЗДЕЛИТЬ_НА_ГРУППЫ

//=====
// Процедура разбиения множества объектов Q на две группы
// Параметры:
//   Q - список всех нераспределенных элементов
//   K1 : K2 - отношение деления объектов
//   Lev - количество уровней от вершины до листьев
//=====
РАЗДЕЛИТЬ_НА_ДВЕ_ГРУППЫ(Q, K1, K2, Lev)
[1] // Выбор оси координат для разбиения
    Для каждой оси вычисляем
        O1 = объект с максимальной нижней границей
        O2 = объект с минимальной нижней границей
        O3 = объект с максимальной верхней границей
        O4 = объект с минимальной верхней границей
        Max' = верхняя граница O1
        Min' = нижняя граница O2
        Max'' = нижняя граница O3
        Min'' = верхняя граница O4
    M = (Max' - Min') / (Max'' - Min'')
    I - номер оси, для которой M максимально
[2] // Разбиение на три группы
    N = количество объектов в множестве Q
    Сортировать объекты множества Q по оси I
    Разбить множество Q на три группы
        В первую группу поместить первые α*N объектов
        Во вторую - последние α*N объектов
        В третью - оставшиеся объекты
    G1 = меньшая по количеству объектов группа
    G2 = большая по количеству объектов группа
    G = оставшаяся группа
[3] // проверка завершения распределения
    N1 = количество элементов в множестве G1
    N2 = количество элементов в множестве G2
    N = количество элементов в множестве G
    Если N = 0, то
        Завершить процедуру и вернуть {G1, G2}
    Если N1 = K1*MLev, то
        Переместить все объекты из G в G2

```

```

    Завершить процедуру и вернуть  $\{G_1, G_2\}$ 
    Если  $N_2 = K_2 * M^{Lev}$ , то
        Переместить все объекты из  $G$  в  $G_1$ 
        Завершить процедуру и вернуть  $\{G_1, G_2\}$ 
    Если  $K_1 * m^{Lev} - N_1 \leq N$ , то
        Переместить все объекты из  $G$  в  $G_1$ 
        Завершить процедуру и вернуть  $\{G_1, G_2\}$ 
    Если  $K_2 * m^{Lev} - N_2 \leq N$ , то
        Переместить все объекты из  $G$  в  $G_2$ 
        Завершить процедуру и вернуть  $\{G_1, G_2\}$ 
[4] // Распределение объектов
     $O$  = любой объект из  $G$ 
    Если  $MBR(G_1, O) - MBR(G_1) \geq MBR(G_2, O) - MBR(G_2)$ , то
        Добавить  $O$  в  $G_2$ 
    Иначе
        Добавить  $O$  в  $G_1$ 
    Удалить  $O$  из  $G$ 
    Перейти к шагу 3
Конец РАЗДЕЛИТЬ_НА_ДВЕ_ГРУППЫ

```

Процедура деления множества объектов на группы первым своим шагом разбивает объекты на две группы в отношении $(\frac{K}{2}) / (K - |\frac{K}{2}|)$. Нетрудно заметить, что если число групп является четным, то алгоритму необходимо будет разбить множество объектов пополам, иначе – на две неравные части. Для разбиения используется процедура РАЗДЕЛИТЬ_НА_ДВЕ_ГРУППЫ, представленная в том же листинге и описанная ниже.

Пример разбиения представлен на рис. 3.14. После разбиения всего множества объектов на две группы G_1 и G_2 , каждая из этих групп разбивается рекурсивно дальше (если число групп в них не равно 1) с помощью вызова той же самой процедуры для каждого из этих множеств.

Основной частью приведенного алгоритма является функция РАЗДЕЛИТЬ_НА_ДВЕ_ГРУППЫ, делящая множество объектов на две группы в заданном соотношении деления $K_1 : K_2$, и с соблюдением ограничения на минимальное и максимальное число объектов в группах (от m^{K_1Lev} до M^{K_1Lev} для первой группы, и от m^{K_2Lev} до M^{K_2Lev} для второй).

Данный алгоритм похож на описанный ранее для процедуры ДЕЛЕНИЕ_УЗЛА.

На первом шаге алгоритма выбирается ось координат, по которой будут поделены объекты на группы. Для этого выбирается такая ось координат i , на которой достигается максимальное значение выражения

$$\max_{\text{оси } i} (\max_{\text{фигуры } j} R'_{j,i} - \min_{\text{фигуры } j} R''_{j,i}) / (\max_{\text{фигуры } j} R''_{j,i} - \min_{\text{фигуры } j} R'_{j,i}),$$

где $R'_{j,i}$, $R''_{j,i}$ – соответственно меньшее и большее значение i -й координаты минимального объемлющего j -ю фигуру прямоугольника.

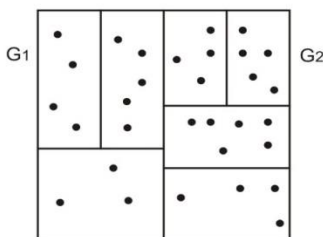


Рис. 3.14. Пример разбиения по алгоритму «разделяй и властвуй»

Выбрав таким образом ось координат, необходимо разделить по ней все объекты на три части по квантилям уровня α и $1-\alpha$ ($\alpha \in [0; 0,5]$ – параметр разделения по группам). Из наименьшей по значениям части образовать первую группу объектов, а из наибольшей – вторую. Оставшиеся объекты распределить по группам в шагах 3 и 4.

На втором шаге алгоритма присутствует некоторый эмпирический коэффициент α . Автором алгоритма, после серии экспериментального моделирования, было установлено, что уменьшение значения α приводит к некоторому улучшению структуры R -дерева на неравномерных распределениях и распределениях со значительным перекрытием объектов. С другой стороны, это приводит к снижению скорости работы алгоритма и уменьшению качества разбиения на точечных и регулярных наборах данных.

Суммируя достоинства и недостатки поведения алгоритма при разных значениях α на различных распределениях объектов, А. В. Скворцов выбрал эмпирическое значение параметра разделения для универсального применения, равное 0,45. При этом если на практике имеются некоторые сведения о реальном распределении объектов, то значение параметра α , вероятно, стоит оценить дополнительно в зависимости от требований конкретной задачи.

На третьем шаге алгоритма проверяется возможность завершения работы алгоритма. Если все объекты распределены, то необходимо закончить и вернуть образовавшиеся группы. Если одна из групп в результате дальнейших операций будет заполнена не полностью (число объектов в группе $N_1 < K_1 m^{Lev}$ или $N_2 < K_2 m^{Lev}$), то вставить в нее все оставшиеся объекты и закончить процедуру. Если одна из групп в результате дальнейших операций будет переполнена (количество объектов в группе $N_1 > K_1 m^{Lev}$ или $N_2 > K_2 m^{Lev}$), то вставить в другую группу все оставшиеся объекты и закончить.

Если же ни одно из этих условий не выполнено, то необходимо взять любой нераспределенный объект и поместить его в группу, размер которой увеличится в минимальной степени. После этого алгоритм заикликивается с помощью возврата к третьему шагу.

Алгоритм уточнения разбиения

Основной проблемой структуры R -дерева является множественное перекрытие узлов дерева. При большом проценте перекрытия структура может оказаться неэффективной в применении. Для улучшения этого показателя используется множество различных методов и алгоритмов. Одним из них является использование процедуры уточнения разбиения после расщепления узлов. Такой подход может использоваться как в обычных динамических алгоритмах манипуляции с R -деревьями (в алгоритме ДЕЛЕНИЕ_УЗЛА при вставке объектов в дерево), так и в алгоритмах разбиения, применяемых в глобальных алгоритмах. Один из вариантов алгоритма уточнения представлен в листинге 3.17.

Листинг 3.17

```
//=====
// Процедура уточнения разбиения
// Параметры:
//   {G} – набор групп, полученных после разбиения
//   m – минимальное количество объектов в группе
//   M – максимальное количество объектов в группе
//=====
УТОЧНИТЬ_РАЗБИЕНИЕ({G}, m, M)
[1] // Выбор группы с объектом для перемещения
    G' = группа максимального размера,
        количество объектов в которой больше m
[2] // Выбор объекта для перемещения
    O' = пустой объект
    Для всех объектов O из группы G' выполнить
        Если (O целиком входит в группу G) и (G' ≠ G) и
            (количество объектов в G < M) и
            (MBR(O) > MBR(O')), то
                O' = O
                G'' = G
[3] // Если объект не выбран, то подобрать другой
    Если O' = пустой объект, то
        k' = максимально возможное в системе число
        Для всех групп G (G' ≠ G и число объектов в G < M)
            Для всех объектов O из группы G'
                k = min(MBR(O, G) - MBR(G))
            Если k < k', то
                O' = O
                G'' = G
[4] // Проверка завершения
```

```

Если  $O'$  = пустой объект, то
    Завершить процедуру
Если  $O'$  перемещался ранее в этой процедуре, то
    Завершить процедуру
[5] // Перемещение объекта
    Переместить  $O'$  в  $G''$ 
[6] // Переход на следующую итерацию
    Если итераций было сделано меньше  $M$ , то
        Перейти на шаг 1
    Иначе
        Завершить процедуру
Конец УТОЧНИТЬ_РАЗБИЕНИЕ

```

Процедура уточнения должна вызываться сразу же после распределения объектов по группам. При этом в процедуре предпринимается попытка переместить часть объектов из одной группы в другую, улучшая при этом свойства разбиения.

На первом шаге алгоритма выбирается группа G' , имеющая самый большой размер и при этом содержащая больше минимально возможного числа объектов. Именно из этой группы необходимо попытаться переместить объекты. Для этого выбирается объект, целиком входящий в какую-либо другую группу (шаг 2). Если таких объектов нет (шаг 3), то необходимо выбрать такой объект, перемещение которого в некоторую другую группу вызовет минимальное увеличение площади этой группы. При выборе группы, в которую планируется перенести объект, необходимо учитывать правило максимального наполнения (число объектов в группе после переноса не должно превысить M).

Если на предыдущих шагах так и не было выбрано ни одного объекта для переноса или был выбран объект, уже переносившийся на предыдущей итерации алгоритма, то процедуру уточнения разбиения необходимо прервать, полагая, что разбиение прошло успешно.

Если же объект для перемещения был выбран, то необходимо выполнить такое перемещение и вернуться к первому шагу алгоритма (однако число итераций алгоритма не должно превысить M).

Данный алгоритм не делает разбиение оптимальным, однако вызов процедуры на реальных распределениях позволяет значительно улучшить структуру строящегося дерева.

Выбор параметров R -дерева

R -дерево как структура индексирования пространственных объектов стало почти стандартом для промышленных СУБД. Различные СУБД используют различные варианты R -деревьев в качестве индексных структур. Однако при конкретной реализации структуры в конечном приложении зачастую встает вопрос – какие параметры выбрать для

реализации, какой из алгоритмов лучше использовать? Далее в этом параграфе будут даны некоторые рекомендации на этот счет.

Максимальное число элементов в узле. При описании структуры R -дерева было отмечено, что одним из параметров дерева является минимально возможное (m) и максимально допустимое (M) количество элементов в узле. При выборе этих параметров необходимо руководствоваться следующими соображениями:

Чем больше значение M , тем сильнее будет ветвиться дерево, а следовательно, его глубина будет меньше. Если предположить, что индексная структура разрабатывается для внешней памяти, то уменьшение глубины дерева ведет к уменьшению обращений к диску (если учесть, что проверка узла дерева вызывает одно обращение к диску). Поэтому сильноветвящееся дерево (при большом M) будет более эффективным для внешней памяти.

С другой стороны, процедура поиска вынуждена просматривать абсолютно все элементы вершины. Поэтому при очень большом M индексная структура может вырождаться просто к последовательному поиску. К тому же на сравнение с элементами вершины расходуется процессорная мощность. Поэтому чем больше M , тем больше нагрузка на процессор в процедурах поиска.

Исходя из описанных фактов, можно сделать следующие выводы: если разрабатываемая индексная структура целиком размещается в оперативной памяти, то значение M стоит выбирать небольшим, порядка 4–10 элементов в вершине. Если же индексная структура хранится во внешней памяти, то значение M стоит вычислять по следующей формуле:

$$M = \lfloor Cluster / ElSize \rfloor,$$

где $Cluster$ – размер кластера жесткого диска (например, 512 или 1024);

$ElSize$ – размер одного элемента.

Так, если один элемент занимает 16 байт, то в качестве верхней границы стоит взять $M = 32$ элементов в вершине.

Минимальное число элементов в узле (m). Данный параметр зависит от M и, как было описано ранее, не может превышать $\lceil M/2 \rceil$. Минимальный же предел параметра m равен 2 (в узле не может быть меньше двух потомков, если это не корневой узел).

При выборе минимальной границы заполнения узла (m) стоит руководствоваться следующими соображениями.

Маленькое значение параметра m облегчает процедуру разделения узла, потому что исчезает необходимость повторной вставки элементов.

В то же время маленькое значение нижней границы может привести к неэффективному использованию памяти. По исследованию А. Гуттмана [36], наименее плотные индексы могут потреблять приблизительно на 50% больше места, чем самые плотные.

В практических применениях наиболее часто используемой операцией является процедура поиска элементов. Поэтому нижней границу заполнения узла стоит выбирать равной $\lfloor M/2 \rfloor$.

Выбор алгоритма деления узла. Центральным звеном при построении дерева является процедура разбиения узла пополам (ДЕЛЕНИЕ_УЗЛА). От эффективности этой процедуры зависит оптимальность построения дерева в целом. При неоптимальной структуре дерева появляется неоднозначность поиска элементов. Возможны ситуации, когда уже на уровнях, близких к корню R -дерева, охватывающие прямоугольники пересекаются не по пустому множеству данных, что значительно усложняет процедуру поиска. С проблемой качества изменения R -дерева можно бороться с помощью «исчерпывающего» алгоритма деления. Использование данного алгоритма для деления узла изменяет структуру R -дерева лучшим из возможных способов, что, конечно, отражается на дальнейшем поиске данных в лучшую сторону, но, в свою очередь, существенно замедляет работу индексной структуры. Применение данного алгоритма оправдано при малом числе записей в узле, а также в ситуациях, когда структура дерева редко меняется, т. е. при индексировании неподвижных (например, жилых домов, складов и т. д.) или слабоподвижных пространственных объектов (например, небесной карты звезд).

Для работы с большим количеством данных удобно пользоваться алгоритмом «линейной стоимости». Его применение оправдано в ситуациях, когда в узлах дерева находится достаточно много записей (больше 10) и число данных очень велико, так как он выполняется максимально быстро (например, индексирование машин, самолетов и т. п.). Но структура дерева, получаемая при использовании этого алгоритма, далека от оптимального варианта. Охватывающие прямоугольники часто получаются большими по площади, имеют пересечения друг с другом по непустому множеству записей, что приводит к значительному замедлению процедур поиска.

Выбор глобального алгоритма построения R -дерева. Из описанных глобальных алгоритмов базовый и клеточный наиболее хорошо работают на равномерных распределениях, когда объекты мало пересекаются между собой (например точечные объекты или объекты малого размера, равномерно распределенные по всему пространству). Но на неравномерных распределениях заметно ухудшение работы этих алгоритмов по сравнению с существующими вариантами R -деревьев. В то же время алгоритм «разделяй и властвуй» одинаково хорош как на равномерных, так и на неравномерных распределениях. Практически во всех тестах он обходит остальные алгоритмы по качеству получающегося построения. Только при построении R -дерева на наборе очень больших объектов этот алгоритм строит дерево, незначительно уступающее по проценту перекрытия классическому алгоритму построения R -дерева.

Данные результаты были получены на основе моделирования различных наборов данных, проведенного А. В. Скворцовым [12].

Алгоритм уточнения. Алгоритм уточнения разбиения может применяться как составная часть алгоритма деления узла при вставке объектов и при глобальном построении. При использовании данного подхода к обычному R -дереву достигается сокращение перекрытия потомков на 7–15%. Применение же уточнения в составе глобальных алгоритмов практически не дает никакого эффекта, за исключением уменьшения перекрытия потомков на регулярном наборе данных.

Различные варианты R -деревьев

В данном параграфе был описан первоначальный вариант R -дерева, опубликованный А. Гуттманом в 1984 году. Однако заложенные идеи оказались настолько удачными, что в настоящее время появилось множество вариантов и модификаций этого дерева.

Самой популярной модификацией R -дерева является R^* -дерево, предложенное в 1990 году [20]. В нем предлагается модифицировать процедуру вставки элементов, добавив в нее повторную вставку объектов в случае переполнения. Подробнее эта процедура описана в следующих параграфах.

Еще одним интересным вариантом являются упакованные R -деревья (*packed R -tree*), предложенные в 1985 году Н. Руссопулосом и Д. Лефкером [75]. Этот вариант разрабатывался для статических данных, которые известны заранее и в последующем не изменяются. В этом случае при построении дерева можно вычислить оптимальное разбиение пространства и построить R -дерево минимального размера.

Не менее интересным является предложение П. Остерома использовать вместо минимальных ограничивающих прямоугольников (MBR) минимальные ограничивающие сферы. Структура, представляющая собой иерархию n -мерных сфер, получила название *sphere tree* [70].

И. Камел и К. Фалоутсос в 1994 году предложили структуру, названную *Hilbert R -tree* [50]. Важнейшим свойством этой модификации является то, что внутренние узлы дерева содержат не только MBR своих потомков, но еще и значения кривой, заполняющей пространство (кривой Гильберта). Это позволяет улучшить процедуру вставки новых объектов в уже построенное дерево. Помимо изменения процедуры добавления объектов и расщепления узлов, И. Камел и К. Фалоутсос улучшили алгоритмы поиска. Данная модификация R -дерева хорошо подходит для точечных объектов или объектов, имеющих небольшие размеры. Однако при использовании объектов большого размера может наблюдаться даже некоторое ухудшение свойств по сравнению с оригинальным R -деревом.

Позднее В. Наг и Т. Камеда обсудили в своих работах возможность применения принципов, используемых в B -деревьях, к алгоритмам

R-деревьев [67]. А еще чуть позже М. Кронакер и Д. Банкс использовали похожий метод для внедрения идей *B-link*-деревьев в *R*-деревья [49].

3.2.2. *R*-дерево Грина

R-дерево является самой популярной структурой для работы с прямоугольными объектами. Однако в алгоритмах оригинального варианта *R*-дерева есть ряд недостатков. В частности, во всех алгоритмах динамического изменения дерева (добавление новой записи, удаление записи, разбиение группы объектов на два множества) в качестве главного критерия выбирается критерий уменьшения площади узлов дерева. Это очень важный показатель. Чем меньше площадь узлов, тем меньше ложных срабатываний при поиске объектов в дереве. Однако, как показано на рис. 3.15, алгоритмы, предложенные А. Гуттманом, не всегда справляются с поставленной задачей.

В 1989 году Д. Грин предложил некоторые усовершенствования в динамических алгоритмах *R*-дерева [38]. В частности, при разбиении объектов в качестве критерия выбирается не уменьшение площади дочерних объектов, а их геометрическое разделение. Такой подход во многих случаях дает лучшие результаты, чем оригинальные алгоритмы А. Гуттмана (особенно на неравномерных распределениях).

R-дерево Грина имеет ту же самую структуру, что и оригинальное *R*-дерево, описанное ранее. Единственное отличие заключается в алгоритме ДЕЛЕНИЕ_УЗЛА (листинг 3.7–3.12). Все остальные алгоритмы Д. Грином изменены не были.

Недостатки оригинального *R*-дерева

Рассмотрим одно из «неудачных» распределений объектов, при котором алгоритмы разбиения множества объектов на две группы, предложенные А. Гуттманом, выдают не самые лучшие результаты (случай с неравномерным распределением объектов в пространстве).

Рис. 3.15(а) показывает разбиения объектов на две группы с использованием квадратичного алгоритма (листинг 3.7), если в качестве m было выбрано 50% от M ($M = 6$, $m = 3$). В качестве первой пары, являющейся основой для будущих групп, будут выбраны объекты A и B . Затем последовательными присоединениями к объекту B будут включены ближайшие объекты C , D и E . Так как дальнейшее последовательное присоединение может привести к тому, что в первой группе окажется меньше m объектов, то оставшиеся объекты F и G добавляются в группу K объекта A . Как наглядно показано на рис. 3.15(а), получившееся разбиение оказалось не самым лучшим из всех возможных.

Если уменьшить ограничение на минимальное заполнение вершины до 30% от максимального ($M = 6$, $m = 2$), то результирующее разбиение получится с меньшим перекрытием (рис. 3.15(б)). Однако вообще избежать его не получится даже в этом случае.

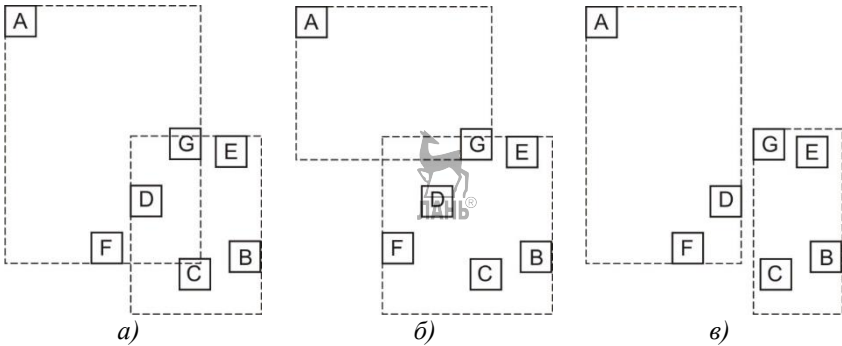


Рис. 3.15. Примеры разбиения объектов на группы: а – квадратичный алгоритм Гуттмана при $t = 50\%$ от M ; б – квадратичный алгоритм Гуттмана при $t = 30\%$ от M ; в – алгоритм Грина

Алгоритм Грина

Чтобы уменьшить перекрытие получающихся при разбиении групп, Д. Грин предложил свой вариант алгоритма разбиения объектов на группы, основанный на геометрическом разделении объектов вдоль одной из осей координат (листинг 2.18).

Листинг 3.18

```
//=====
// Процедура деления узла
// Параметры:
//   О – объект, который не поместился в вершину L
//   L – вершина, которую необходимо разбить на две
//=====
ДЕЛЕНИЕ_УЗЛА (L, O)
  [1] Q = множество всех элементов из L и элемент O
      Удалить все элементы из L
      Создать новую вершину L"
  [2] i = ВЫБОР_ОСИ(Q)
  [3] РАЗДЕЛИТЬ_НА_ГРУППЫ(Q, i, L, L")
Конец ДЕЛЕНИЕ_УЗЛА

//=====
// Выбор оси, по которой будет произведено деление
// Параметры:
```

```

// Q - множество всех объектов
//=====
ВЫБОР_ОСИ(Q)
[1] O1, O2 = ВЫБОР_ПЕРВОЙ_ПАРЫ(Q)
[2] Для каждого измерения вычисляется

$$d_i = \frac{[\text{верхняя граница } O_1] - [\text{нижняя граница } O_2]}{[\text{размер по измерению}]}$$

[3] Вернуть i, для которого величина di максимальна
Конец ВЫБОР_ОСИ

//=====
// Деление множества объектов на две группы
// Параметры:
// Q - первоначальное множество всех объектов
// i - измерение, вдоль которого необходимо произвести
// деление
// L - вершина для объектов первой группы
// L'' - вершина для объектов второй группы
//=====
РАЗДЕЛИТЬ_НА_ГРУППЫ(Q, i, L, L'')
[1] Сортировать объекты множества Q по i-й координате
[2] Поместить первые  $\lfloor (M+1)/2 \rfloor$  объектов в вершину L
Поместить последние  $\lfloor (M+1)/2 \rfloor$  объектов в L''
[3] Если  $(M+1) \% 2 \neq 0$ , то
O = оставшийся объект из множества Q
Если  $MBR(L, O) - MBR(L) \leq MBR(L'', O) - MBR(L'')$ , то
Добавить O в L
Иначе
Добавить O в L''
Конец РАЗДЕЛИТЬ_НА_ГРУППЫ

```

Основная суть алгоритма заключается в следующем: необходимо выбрать некоторую координатную ось и разбить все объекты на две группы с помощью гиперплоскости по этой оси (в двумерном случае – с помощью линии).

Для выбора оси разбиения используется процедура ВЫБОР_ОСИ. Сначала данная процедура выбирает два объекта, расположенных как можно дальше друг от друга (шаг 1). Для этого можно использовать процедуру ВЫБОР_ПЕРВОЙ_ПАРЫ, описанную ранее в листинге 3.18. Затем для каждого измерения вычисляется некоторый показатель, являющийся нормализованной величиной расположения выбранных объектов по данной оси. Он рассчитывается как результат деления расстояния между выбранными объектами на общую длину измерения для всех объектов множества (шаг 2). И, наконец, в качестве оси разбиения выбирается та ось, по которой рассчитанный показатель является максимальным (шаг 3).

Так для ситуации, изображенной на рис. 3.15, в качестве пары объектов будут выбраны объекты A и B , а в качестве оси разбиения – ось x .

После выбора оси разбиения выполняют деление объектов на две группы (процедура РАЗДЕЛИТЬ_НА_ГРУППЫ). Для этого сортируют все объекты по значению нижней границы описывающего их прямоугольника для выбранной оси (шаг 1). Первые $\lfloor (M+1)/2 \rfloor$ объекта из отсортированного списка добавляют в первую группу, а последние $\lfloor (M+1)/2 \rfloor$ объектов – во вторую (шаг 2). На рис. 3.15(в) после этого шага в первую группу попадут объекты A, D, F , а во вторую – B, C, E .

Нетрудно заметить, что если $(M + 1)$ является величиной нечетной, то останется еще один объект, который не попал ни в первую, ни во вторую группу. Это объект G . Его помещают в ту группу, чей ограничивающий прямоугольник MBR увеличится наименьшим образом при добавлении.

3.2.3. R^* -дерево

Ранее были описаны два варианта R -дерева (оригинальное и R -дерево Грина). Эти структуры очень похожи. Единственное различие в них – это процедура деления узла при переполнении. В оригинальном R -дереве используется критерий уменьшения площади получившихся после деления групп, а в R -дереве Грина – критерий геометрического разделения.

В 1990 г. Н. Бекман и ряд других ученых предложили еще один вариант, получивший название R^* -дерево [20]. Как и предыдущие варианты, R^* -дерево отличается от первоначального варианта только процедурой вставки новых элементов. Все алгоритмы поиска в дереве остаются прежними. Однако, учитывая результаты проведенных исследований, новая процедура вставки позволяет создавать структуру дерева более оптимальную, чем алгоритмы, описанные в предыдущих главах.

Недостатки предыдущих вариантов

В предыдущем параграфе были представлены случаи, при которых оригинальные алгоритмы построения R -дерева, описанные А. Гуттманом, создают дерево неоптимальной структуры. В частности, на рис. 3.15 показано дерево с большим перекрытием дочерних узлов, а любое перекрытие усложняет процедуру поиска, делая ее более трудоемкой.

R -дерево Грина в некоторых ситуациях строит более оптимальное дерево, как было показано ранее. Однако и этот алгоритм не всегда лучшим образом производит деление. Рассмотрим один из таких случаев. На рис. 3.16(а) представлен набор объектов, которые необходимо разбить

на две группы. Особенностью этого распределения является присутствие объектов большого и маленького размера.

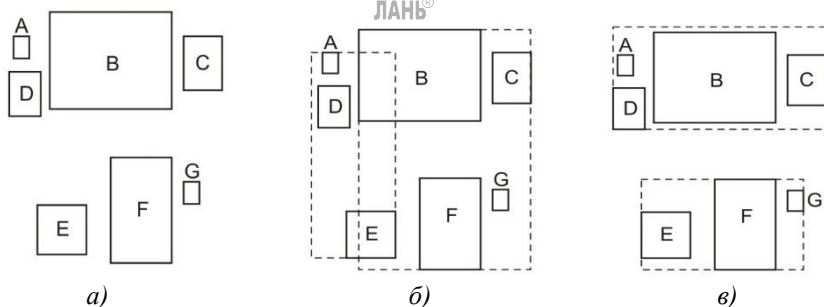


Рис. 3.16. Примеры разбиения объектов на группы: а – исходные объекты; б – разбиение Грина; в – корректное разбиение

Алгоритм Грина на первом шаге выбирает два объекта в качестве базовых. В представленном примере в качестве таковых будут выбраны объекты *A* и *G*. Это произойдет из-за того, что они имеют маленькую внутреннюю площадь, а ограничивающий прямоугольник, включающий оба эти объекта, очень большой. С другой стороны, объекты *B* и *F* имеют большую собственную площадь и поэтому, хотя их крайние точки и расположены дальше всех, на роль базовых объектов они не подойдут.

Следующим шагом алгоритма является выбор оси разбиения. Нормализованное отношение длины для оси *x* объектов *A* и *G* больше, чем для оси *y*, поэтому разбиение произойдет по горизонтальной оси. Результат такого разбиения представлен на рис. 3.16(б), хотя интуитивно понятно, что более корректное разбиение можно выполнить так, как показано на рис. 3.16(в).

Для решения подобных задач необходимо изменить ряд принципов и критериев в процедурах вставки объекта в дерево и деления узла, что и было сделано в *R**-дереве.

Алгоритм добавления нового объекта

Обе структуры — и оригинальное *R*-дерево, и *R**-дерево — являются недетерминированными по положению объектов в самом дереве. Данный факт вытекает из того, что положение объектов в дереве зависит от порядка, в котором эти объекты были добавлены в дерево. Из-за этого *R*-дерево не всегда может иметь структуру, эффективную для процедур поиска. Конечно, ранее была описана процедура оптимизации (перестроения дерева в наиболее оптимальном виде), которую можно использовать при вставке объектов (локальная оптимизация) или

вызывать во время простоя системы (глобальная оптимизация). Однако эта процедура является весьма трудоемкой и при частых изменениях в дереве может отнимать значительное время.

Однако в результате многочисленных экспериментов и исследований был замечен интересный факт. Неэффективная структура чаще всего получается, если неудачно выбираются самые первые элементы для вставки. При этом если в дальнейшем просто удалить некоторые существующие объекты в дереве и снова их вставить, то структура дерева может значительно улучшиться. Так, Н. Бекман провел простой опыт. Он создал дерево при помощи вставки 20 000 равномерно распределенных в пространстве объектов. Затем, измерив производительность этого дерева по наиболее популярным типам поисковых запросов, он удалил 10 000 объектов, которые вставлялись первыми, и снова вставил их в дерево. Результаты оказались поражающими. Такая нехитрая комбинация действий позволила увеличить производительность дерева от 20 до 50% в зависимости от типа поискового запроса.

Данный подход действительно дает хорошие результаты. Достаточно удалить приблизительно половину элементов в дереве и затем снова их вставить, чтобы структура дерева немного улучшилась. Однако этот метод можно применять только для статичных деревьев (которые редко меняются).

Для того чтобы применить динамическую реорганизацию дерева, было предложено в R^* -дереве принудительно повторно вставлять объекты в процедуру добавления объектов. Процедура вставки объекта в R^* -дерево представлена в листинге 3.19.

Листинг 3.19

```
//=====
// Процедура вставки объекта в  $R^*$ -дерево
// Параметры:
//   O - объект, который нужно вставить в дерево
//=====
ВСТАВКА(O)
    [1] Lev = номер листового уровня у данного дерева
    [2] ВСТАВКА_ВЕРШИНЫ(O, Lev, Истина)
Конец ВСТАВКА

//=====
// Процедура вставки элемента в  $R^*$ -дерево на уровне Lev
// Параметры:
//   Vadd - элемент, который нужно вставить в дерево
//   Lev - уровень, на котором должен
//         разместиться Vadd
//   IsFirst - флаг, первый ли раз вызывается процедура
//            в процессе вставки элемента
```

```
//=====
ВСТАВКА_ВЕРШИНЫ(Vadd, Lev, IsFirst)
[1] V = ВЫБОР_ВЕРШИНЫ(Vadd, Lev)
[2] Если число элементов в V меньше M, то
    Добавить элемент Vadd в узел V
    КОРРЕКТИРОВКА_ДЕРЕВА(V, NULL)
    Выйти из процедуры
[3] Q = множество всех элементов из V и элемент Vadd
    Если Lev = 0 или IsFirst = Ложь, тогда
        {V', V''} = ДЕЛЕНИЕ_УЗЛА(Q)
        КОРРЕКТИРОВКА_ДЕРЕВА(V', V'')
        Выйти из процедуры
[4] ПЕРЕВСТАВКА(Q, V)
Конец ВСТАВКА_ВЕРШИНЫ

//=====
// Процедура повторной вставки элементов
// вершины
// Параметры:
// Q - множество всех элементов вершины
// V - вершина, элементы которой перевставляются
//=====
ПЕРЕВСТАВКА(Q, V)
[1] Для каждого элемента из Q вычислить
    Di = расстояние от центра i-го элемента до V
[2] Сортировать Q по величине Di в порядке убывания
[3] Оставить в множестве Q только первые p элементов
    Удалить из V элементы, находящиеся в Q
    Скорректировать MBR вершины V
[4] Для каждого элемента V' из множества Q
    ВСТАВКА(V', Level(V), Ложь)
Конец ПЕРЕВСТАВКА
```

Описанная процедура ВСТАВКА не производит никаких действий. Она просто вызывает процедуру ВСТАВКА_ВЕРШИНЫ, которая предназначена как для вставки объектов в листовые узлы дерева, так и для вставки временно удаленных внутренних узлов дерева. Для вставки конечного объекта просто необходимо передать этот объект процедуре, а в качестве номера уровня указать листовой уровень дерева (высоту дерева).

Еще одним параметром процедуры ВСТАВКА_ВЕРШИНЫ является флаг, показывающий первый ли раз вызывается процедура вставки в процессе добавления объекта в дерево. Этот флаг влияет на режим работы процедуры и позволяет избежать заикливания из-за повторной вставки объектов, предусмотренных в алгоритме.

Первым шагом процедура ВСТАВКА_ВЕРШИНЫ выбирает узел дерева, в который необходимо вставить переданный элемент. Так как процедура вставки является универсальной и может добавлять не только

конечные объекты в листовые узлы дерева, но и внутренние вершины, то процедура ВЫБОР_ВЕРШИНЫ также должна быть универсальной. Для этого в нее в качестве одного из параметров передается уровень в дереве, на котором необходимо найти наиболее подходящий узел для вставки. Сама процедура ВЫБОР_ВЕРШИНЫ также претерпела изменения в алгоритмах R^* -дерева и будет описана ниже.

Если выбранная вершина V содержит дочерних узлов меньше максимально допустимого числа элементов (меньше принятого предела M), то добавляемая вершина V_{add} просто вставляется как еще один дочерний узел V (шаг 2). Однако если в вершине V достигнут максимум по числу дочерних узлов, то необходимо применить новую тактику. Если текущий уровень является не уровнем корня ($Lev \neq 0$) и процедура вставки вызывается первый раз в процессе добавления некоторого объекта в дерево ($IsFirst = \text{Истина}$), то необходимо попробовать удалить несколько наиболее удаленных от центра объектов из этой вершины и вставить их снова в дерево. При этом вполне вероятно часть объектов попадут в другие вершины, тем самым улучшив структуру дерева и избавив его от необходимости расщеплять узлы (шаг 4).

Если же процедура вставки вызывается не в первый раз ($IsFirst = \text{Ложь}$), то это означает, что повторная вставка не принесла результатов и необходимо разделить переполненный узел (шаг 3). Аналогично нужно поступать, если узел, в который производится вставка, является корнем.

В процедуру деления узла передается все множество дочерних узлов и объект, который необходимо вставить в данную вершину. Процедура деления разбивает это множество на две части и создает две новые вершины для каждой из частей. Алгоритм процедуры ДЕЛЕНИЕ_УЗЛА также изменен в R^* -дереве. Он будет представлен ниже.

После разбиения два новых потомка вставляются в дерево и производится корректировка. При этом изменяются ограничивающие прямоугольники модифицированных узлов, и деление распространяется вверх по дереву (если это необходимо). Для выполнения этого действия вызывается процедура КОРРЕКТИРОВКА_ДЕРЕВА, описанная ранее в параграфе, посвященном R -дереву.

Еще одной процедурой, алгоритм которой необходимо рассмотреть, является процедура ПЕРЕВСТАВКА. В качестве своих параметров она принимает некоторый узел V , который переполнится при вставке в него нового объекта, и все множество объектов, которые должны быть в нем размещены (множество Q , содержащее $(M + 1)$ объект).

Процедура сортирует все элементы множества в порядке уменьшения расстояния их центра от центра вершины V . Затем она выбирает p наиболее удаленные от центра объекты и временно удаляет их из дерева. Значение p может быть произвольным в диапазоне от 1 до M . Более того, можно выбирать разные значения p для листовых и

внутренних узлов. Однако проведенные исследования показали, что наибольшая производительность достигается при p , равном 30% от M (как для листовых, так и для внутренних узлов дерева).

После удаления объектов и корректировки ограничивающего прямоугольника вершины V эти объекты снова добавляются в дерево при помощи все той же процедуры ВСТАВКА_ВЕРШИНЫ. При этом, если они снова все попадут в ту же самую вершину и приведут к ее переполнению, произойдет деление вершины на две части.

Критерии оптимального деления

Еще одним важным элементом алгоритма построения дерева является критерий оптимального деления объектов на группы. В зависимости от того, какое разбиение считать наиболее эффективным, можно разработать абсолютно разные алгоритмы работы.

А. Гуттман в своей работе использовал только один критерий – минимизацию площади потомков. Однако данный критерий является не единственным. Рассмотрим возможные варианты критериев оптимальности.

- *Минимизация площади получающихся потомков.* Во главу этого критерия ставится минимизация площади ограничивающих прямоугольников, получающихся в результате разбиения групп объектов. Преимущества этого подхода однозначны: чем меньше площадь ограничивающих прямоугольников, тем меньше ложных срабатываний при поиске. Из тех вариантов разбиения, которые представлены на рис. 3.17, руководствуясь этим критерием, необходимо выбрать первый, так как площадь ограничивающих прямоугольников у него самая маленькая.

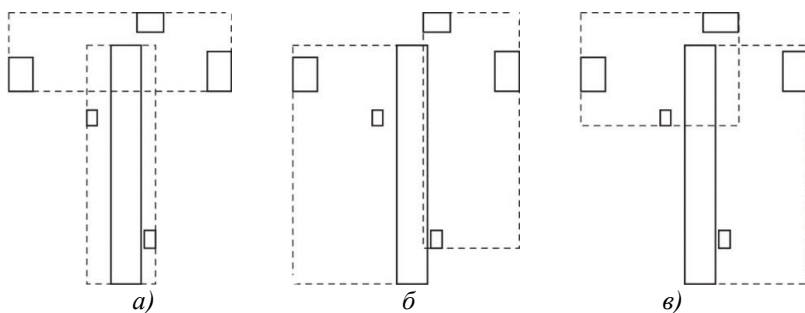


Рис. 3.17. Примеры разбиения объектов на группы: а – минимальная площадь; б – минимальное пересечение; в – минимальный периметр

- *Минимизация перекрытия ограничивающих прямоугольников.* Данный критерий также ставит своей целью уменьшить число «ложных»

путей поиска. Улучшение сводится к тому, что при использовании этого критерия самым важным считается создание структуры, в которой взаимное пересечение узлов дерева минимально. Для этого критерия лучше всего подходит второй вариант разбиения, представленный на рис. 3.17.

- *Минимизация периметра.* Под периметром здесь подразумевается сумма длин всех сторон ограничивающих прямоугольников. Данный критерий очень часто приводит к результатам, похожим на предыдущие разбиения. Однако использование этого критерия ведет к тому, что форма получающихся в результате разбиения групп имеет формы, близкие к квадратным. Как подтверждают многие исследования, существуют поисковые запросы (например запрос на совпадение в двух множествах по пространственным координатам), которые выполняются тем эффективнее, чем ближе форма ограничивающих областей к квадрату. Из вариантов, приведенных на рис. 3.17, данному критерию более всего удовлетворяет третий вариант.

- *Улучшение использования пространства жесткого диска.* Данный критерий позволяет построить дерево, эффективно использующее пространство жесткого диска (сама структура занимает как можно меньше места на диске). Данный критерий не уменьшает число «ложных» срабатываний при поиске, однако он позволяет уменьшить число обращений к жесткому диску. Чем компактнее структура располагается на жестком диске, тем меньше понадобится обращений к нему при поиске.

Алгоритм выбора вершины для вставки нового объекта

В оригинальных *R*-деревьях и их модификациях процедура поиска вершины, в которую будет вставлен новый объект, использовала только критерий минимизации площади, абсолютно не учитывая остальные возможности. Н. Бекман и др. исследователи тестировали влияние выбранного критерия на структуру получающегося в результате дерева. В результате такого тестирования ими был предложен другой алгоритм выбора вершины для вставки объекта (листинг 3.20).

Листинг 3.20

```
//=====
// Выбор вершины для вставки объекта
// Параметры:
//   О – объект, который нужно вставить в дерево
//=====
ВЫБОР_ВЕРШИНЫ(О)
    [1] V = корень дерева
    [2] Если V – лист, то
        Выйти из процедуры и вернуть V
```

```

[3] Если дочерние узлы V являются листьями, то
    // Критерий для листовых узлов (перекрытие)
    V' = первый потомок узла V
    Цикл по всем потомкам V'' вершины V
        Если  $PR(\{V', O\}) - PR(V') > PR(\{V'', O\}) - PR(V'')$ , то
            V' = V''
        Иначе
            // Критерий для внутренних узлов (площадь)
            V' = первый потомок узла V
            Цикл по всем потомкам V'' вершины V
                Если  $MBR(\{V', O\}) - MBR(V') >$ 
                     $MBR(\{V'', O\}) - MBR(V'')$ , то
                    V' = V''
[4] Перейти к шагу 2
Конец ВЫБОР_ВЕРШИНЫ

```

Для выбора наилучшего узла на внутреннем уровне дерева альтернативные критерии, описанные в предыдущем пункте, не дают никакого преимущества по сравнению с оригинальным критерием А. Гуттмана. Поэтому для внутренних узлов дерева выбор остался прежним – по критерию минимизации площади потомков. Однако для листовых узлов критерий минимизации площади перекрывающихся частей дает выигрыш при дальнейшем поиске, поэтому в данном алгоритме он и был применен.

В листинге 3.20 функция вычисления перекрытия некоторого узла V другими узлами обозначена как $PR(V)$. Ее можно реализовать по следующей формуле:

$$PR(V) = \sum_{i=1, V_i \neq V}^N (MBR(V) \cap (MBR(V_i))),$$

где N – число элементов.

Предложенная процедура выбора вершины дает значительные преимущества в следующем случае: распределение данных в пространстве является неоднородным, и в системе наиболее часто используется запрос на поиск небольшой порции данных. Во всех остальных случаях построенное дерево с использованием этой процедуры дает незначительное преимущество по сравнению с оригинальными процедурами А. Гуттмана.

Алгоритм разбиения узла

Процедура разбиения узла в R*-дереве использует метод, дающий значительные преимущества по сравнению с оригинальными. Идея метода похожа на ту, что использовалась в R-дереве Грина. Ограничивающие прямоугольники элементов разделяемой вершины сортируются в порядке возрастания по нижней границе для каждой оси. Для каждого такого отсортированного списка существует $(M-2 \cdot m + 2)$

варианта разбиения объектов на группы. Если обозначить за l число от 0 до $(M-2*m+1)$, то любое из возможных разбиений вдоль заданной оси можно выразить следующей формой: в первую группы помещается $(m+l)$ первых объектов отсортированного списка, а во вторую — все оставшиеся объекты.

Как видно, процедура разбиения зависит от двух важных этапов: каким образом выбрать ось разбиения и как определить значение l , при котором разбиение окажется оптимальным. Наиболее эффективная (в смысле получающейся в результате структуры) процедура деления представлена в листинге 3.21.

Листинг 3.21

```
//=====
// Процедура деления узла
// Параметры:
//   О - объект, который не поместился в вершину
//   V - вершина, которую необходимо разбить на две
//=====
ДЕЛЕНИЕ_УЗЛА(V, O)
    [1] Q = множество всех элементов из V и элемент O
        Удалить все элементы из V
        Создать новую вершину V'
    [2] i = ВЫБОР_ОСИ(Q)
    [3] j = ВЫБОР_ИНДЕКСА_РАЗБИЕНИЯ(Q, i)
    [4] Переместить в V первые (m+j) элементов из Q
        Переместить в V' оставшиеся элементы из Q
    [5] Вернуть V'
Конец ДЕЛЕНИЕ_УЗЛА

//=====
// Выбор оси, по которой будет произведено деление
// Параметры:
//   Q - множество всех объектов
//=====
ВЫБОР_ОСИ(Q)
    [1] Для каждой оси
        Сортировать Q по нижней границе объектов
         $S_i = 0$ 
        Для j, меняющегося от m до (M-m)
             $G_1$  = группа из первых j объектов множества Q
             $G_2$  = группа из оставшихся объектов Q
             $S_i += \text{ПЕРИМЕТР}(G_1) + \text{ПЕРИМЕТР}(G_2)$ 
    [2] i = номер оси, для которой значение  $S_i$  минимально
        Вернуть i
Конец ВЫБОР_ОСИ

//=====
// Выбор номера объекта, по которому производится деление
```

```

// Параметры:
// Q - множество всех объектов
// i - ось, по которой производится деление
//=====
ВЫБОР_ИНДЕКСА_РАЗБИЕНИЯ(Q, i)
  [1] Сортировать объекты из Q по нижней
       границе объектов по оси i
  [2] Для j, меняющегося от 0 до (M-2*m+1)
       G1 = группа первых (m+j) объектов множества Q
       G2 = группа из оставшихся объектов множества Q
       Sj = MBR(G1) ∩ MBR(G2)
  [2] j = номер оси, для которой значение Sj минимально
       Вернуть j
Конец ВЫБОР_ИНДЕКСА_РАЗБИЕНИЯ

```

В процедуре ВЫБОР_ОСИ сортируются объекты по каждой оси координат по нижней координате ограничивающих прямоугольников (в случае равенства нижних границ сортируется по верхней границе). В каждом из отсортированных списков рассматриваются все возможные распределения объектов по описанному принципу. В качестве оси деления выбирается та ось координат, сумма периметров для всех распределений у которой окажется меньше, т. е. на этапе выбора оси распределения применяется критерий минимизации периметра.

Однако в процедуре ВЫБОР_ИНДЕКСА_РАЗБИЕНИЯ применяется другой критерий – минимизация площади перекрытия получившихся групп. Такое применение двух разных критериев позволяет добиться максимально эффективной структуры дерева.

Эффективность процедуры разбиения, кроме всего прочего, также зависит и от величины m . Если m составляет 50% от M , то абсолютно все равно, какие критерии будут выбраны, так как разбиение все равно произойдет на две равные по числу объектов группы. В своих исследованиях авторы R^* -дерева пришли к выводу, что представленная процедура строит дерево с максимально эффективной структурой при m , равном 40% от M .

Выводы

Принудительная повторная вставка элементов, применяемая в R^* -дереве, позволяет уменьшить перекрытие узлов в дереве. Также интересным следствием повторной вставки является улучшение использования пространства, занимаемого индексом на жестком диске.

Авторы структуры провели многогранное тестирование разработанных алгоритмов. Тестирование проводилось на разных наборах данных, среди которых были не только однородное, кластерное, гауссовское и смешанное распределения объектов в пространстве, но и тестирование на реальных картографических данных. Исследования проводились на трех наиболее часто встречающихся типах запросов:

- поиск по области;
- поиск по точке;
- поиск пересекающихся объектов.

При этом измерялось не только время обработки запроса, но и количество обращений к жесткому диску. По всем показателям R^* -дерево превосходит оригинальное R -дерево и R -дерево Грина (по некоторым распределениям и типам запросов превосходство достигает до 400%). Причем R^* -дерево одинаково хорошо подходит как для точечных данных (PAM), так и для пространственных (SAM).

3.2.4. SS-дерево

На практике встречается много приложений, в которых требуется часто выполнять запрос на поиск похожих по некоторым параметрам объектов. Такие запросы называют запросами подобных объектов.

Для выполнения поиска подобных объектов необходимо, чтобы по некоторому набору характеристик была разработана функция похожести, которая будет оценивать, насколько два объекта отличаются друг от друга. Если в качестве характеристик объектов выступают числовые величины, то, приняв эти характеристики за оси в многомерном пространстве, мы получаем геометрическое представление объекта как некоторой точки в N -мерном пространстве (где N – число характеристик объекта). В этом случае в качестве функции похожести объектов можно выбрать функцию евклидова расстояния между точками в многомерном пространстве, соответствующем этим объектам.

Для примера рассмотрим некоторую кредитную организацию. Допустим, в этой организации ведется база данных клиентов, которым были выданы кредиты. Каждый клиент оценивается по трем характеристикам: размер кредита, месячная заработная плата, количество членов семьи. Если эти характеристики принять за оси в трехмерном пространстве, то получится, что каждый клиент является точкой в этом пространстве. При этом если этим точкам сопоставить цвета (например, черный – клиент выплачивает кредит вовремя, серый – с постоянными задержками), то получится некоторая картина, показанная на рис. 3.17.

Имея такое представление о клиентах в целом, организация может принимать решение о том, стоит ли выдавать кредит новому клиенту. Так, в организацию пришел клиент с характеристиками, обозначенными как точка A на рис. 3.17. При этом, оценивая клиентов с похожими характеристиками (тех, что находятся на рисунке ближе всего к точке A – это клиенты B и C), можно сделать вывод, что клиенты с подобными характеристиками являются проблемными, чаще всего не вовремя возвращают деньги и поэтому в кредите стоит отказать.

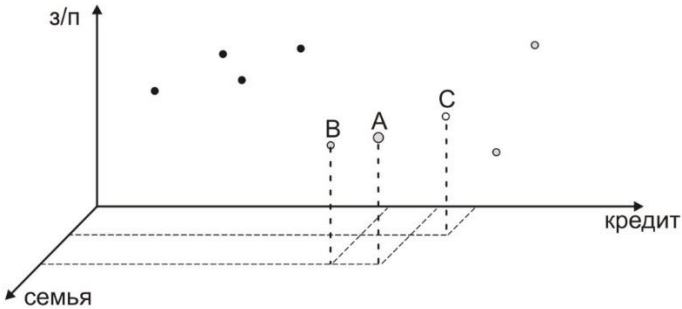


Рис. 3.18. Многомерная база клиентов

Это один из немногих примеров приложений, в которых может потребоваться вести многомерную базу данных с поддержкой запросов поиска похожих объектов. В качестве структуры, индексирующей подобные данные, можно выбрать описанные ранее *K-D-B*-деревья или *R**-дерево. Однако А. Д. Вайт и Р. Джейн предложили новую структуру, имеющие те же принципы, что и *R**-дерево, но специально адаптированную под запросы данного класса. Данная структура была предложена в 1996 году и получила название *SS*-дерево [90].

Структура *SS*-деревя

Главной задачей структуры для индексирования многомерных данных и поддержки запросов подобия является минимизация среднего времени поиска объекта в базе (как и у любой другой индексирующей структуры). Отличие заключается лишь в том, что *SS*-дерево минимизирует среднее время поиска объектов именно для запросов похожих объектов, не обращая внимания на другие типы запросов.

Как было отмечено, чаще всего для определения похожести/непохожести объектов используется евклидово расстояние или одна из его разновидностей – взвешенное евклидово расстояние (подробнее о возможных функциях расстояния и их определении можно посмотреть в разделе 1.3).

Так, пусть дан двумерный случай, показанный на рис. 3.19(а). При этом можно говорить, что точка *A* больше похожа на точку *B*, чем на точку *C*, так как расстояние от *A* до *B* меньше расстояния от *A* до *C*:

$$\sqrt{(x_a - x_b)^2 + (y_a - y_b)^2} < \sqrt{(x_a - x_c)^2 + (y_a - y_c)^2}.$$

Развивая мысль о запросах похожих объектов, можно прийти к такому типу запроса: «необходимо найти все объекты, которые отличаются от данного не больше, чем на величину *d*». Математически это будет выглядеть так:

$$\forall O: \sqrt{(x_a - x_o)^2 + (y_a - y_o)^2} \leq d.$$

Графически же этот тип запроса представлен на рис. 3.19(б), т. е. объекты, попавшие в окружность с радиусом d и центром в точке A будут удовлетворять данному запросу.

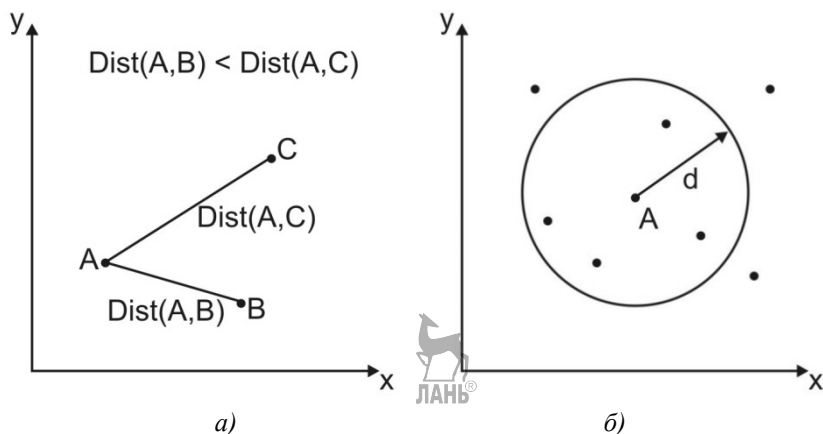


Рис. 3.19. Примеры расстояния в двухмерном случае: а – примеры точек в двухмерном пространстве; б – геометрическая форма евклидова расстояния

На практике часто разные критерии (параметры, оси многомерного пространства) имеют разное влияние на функцию расстояния. Так, в приведенном примере с кредитной организацией отличие в сумме кредита на 10–20 условных единиц не так существенно, как отличие по составу семьи на 2–3 человека. Поэтому при вычислении расстояния необходимо учитывать нормировочные коэффициенты, которые предназначены для выравнивания значимости всех критериев. Для этого функцию расстояния преобразуют к следующему виду (двумерный случай):

$$\text{Dist} = (A, B) = \sqrt{(x_a - x_b)^2 W_1 + (y_a - y_b)^2 W_2}.$$

Функцию расстояния такого вида называют взвешенным евклидовым расстоянием. Коэффициенты W_1 и W_2 для нее выбираются экспертом в данной прикладной области и становятся неотъемлемой частью системы.

Взвешенное евклидово расстояние геометрически не сильно отличается от представленного на рис. 3.19(б). Единственное значимое отличие заключается в том, что при использовании весовых коэффициентов для параметров геометрическое представление расстояния в виде окружности необходимо заменить на представление в

виде овала. Как можно увидеть из представленных примеров, функция расстояния, основанная на формуле Евклида, графически представляет собой окружность (или овал при взвешенном расстоянии). Поэтому и структура, оптимизированная под эту форму областей поиска, будет наиболее эффективна.

Руководствуясь этими критериями, А. Вайт и Р. Джейн предложили изменить структуру R^* -дерева, используя в качестве ограничивающих областей не минимальные ограничивающие прямоугольники (MBR), а минимальные ограничивающие сферы (MBS). Получившуюся в результате структуру было решено назвать SS -деревом [90].

Для наглядного представления различий этих структур на рис. 3.20 представлены два дерева для одного и того же набора данных (а – R^* -дерево, б – SS -дерево).

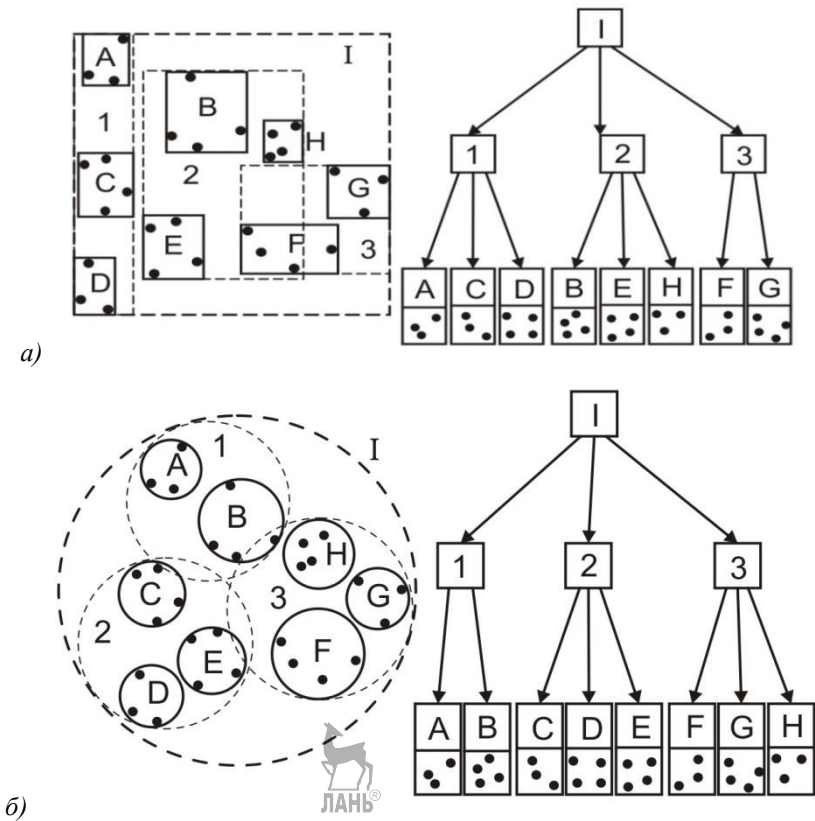


Рис. 3.20. Представление данных в виде деревьев: а – R^* -дерево; б – SS -дерево

Структура узла SS-дерева

SS-дерево по принципам построения и алгоритмам обработки очень похоже на описанное ранее R^* -дерево. Однако, так как в качестве ограничивающих фигур применяются не прямоугольники, а сферы, есть и небольшие различия в структуре узла дерева. Можно придумать разные варианты структуры листовых и внутренних узлов дерева. Один из вариантов представлен в листинге 3.22. В нем в листовых узлах дерева хранятся конечные объекты. Причем в одном листовом узле находится всего один объект. Другие варианты могут предусматривать нахождение сразу группы объектов в одном листовом узле.

Листинг 3.22

```
//=====
// Листовой узел SS-дерева
//=====
Структура ЛИСТ
{
    Точка Центр;
    Число Радиус;
    Байт Данные[];
}

//=====
// Внутренний узел SS-дерева
//=====
Структура УЗЕЛ
{
    Точка Центр;
    Число Радиус;
    Список Дочерние_Узлы;
}
```

Структура листового листа дерева состоит из трех элементов. *Центр* – точка в многомерном пространстве, соответствующая центру объекта, который хранится в данном листовом узле. *Радиус* – минимально возможный радиус окружности, описанной вокруг объекта, соответствующего данному узлу дерева. Если в данном листовом узле хранится точечный объект, то в качестве радиуса можно указать значение 0. Если же в разрабатываемой системе предполагается хранить и обрабатывать только точечные объекты, то данное поле в листовом узле вообще может отсутствовать.

Еще одно поле листового узла – это *данные*. Конкретная реализация этого поля зависит от практической направленности системы. Это поле отвечает за хранение дополнительных данных, связанных с многомерным объектом.

Структура внутреннего узла более разнообразна. В нем нет массива с данными, так как внутренние узлы дерева не содержат объектов. При

этом в нем также сохранились параметры *Центр* и *Радиус*. Однако их смысл немного поменялся. Если в вершине, связанной с объектом, центр являлся геометрическим центром этого самого объекта, то во внутреннем узле дерева *Центр* является центром описанной окружности вокруг всех окружностей дочерних узлов. Соответственно *Радиус* является радиусом этой описанной окружности.

Еще одним отличием внутреннего узла от листового является наличие списка с указателями на дочерние узлы (элемент структуры *Дочерние_Узлы*). В этом списке содержатся просто указатели на структуры дочерних узлов. Причем если следующий уровень является листовым (уровень, в котором хранятся сами многомерные объекты), то в этом списке находятся ссылки на структуры типа ЛИСТ, а во всех остальных случаях – ссылки на структуры типа УЗЕЛ.

Так как в *SS*-дереве перенята большая часть принципов и алгоритмов *R*-деревьев, то к нему также относятся некоторые параметры и ограничения, связанные с этим семейством деревьев. В частности, каждый узел дерева (за исключением корня) должен содержать потомков не больше некоторого числа M и не меньше m . Причем, как было обосновано ранее, M не может быть больше $2m$. Из этого следует, что максимальный размер списка *Дочерние_Узлы* составляет M элементов.

Алгоритм поиска по точному совпадению

Процедура поиска объекта в *SS*-дереве по точному совпадению практически идентична аналогичной процедуре для *R*-деревя. Она точно так же просматривает все дерево, начиная с корня и заканчивая листовыми узлами в поисках вершины, которая могла бы содержать заданный объект. Одна из возможных реализаций такой процедуры представлена в листинге 3.23. Процедура возвращает лист, в котором находится запрошенный объект, или *NULL*, если такого объекта в дереве нет.



Листинг 3.23

```
//=====
// Поиск листа, в котором находится объект O
// Параметры:
//   V – вершина, начиная с которой производится поиск
//   O – объект, который нужно найти.
//=====
ПОИСК_ОБЪЕКТА(V, O)
[1] Если V не является листом, то
    Проверить все записи V', находящиеся в узле V
    Если MBS(V') полностью содержит MBS(O), то
        L = ПОИСК_ОБЪЕКТА(V', O)
    Если L ≠ NULL, то
        Вернуть L
```



```

[2] Если V является листом, то
      O' = объект, находящийся в вершине V
      Если O' = O, то
          Вернуть V
[3] Вернуть NULL
Конец ПОИСК_ОБЪЕКТА

```



Отличие этой процедуры от аналогичной для R -дерева заключается лишь в одном условии шага 1. В R -дереве при определении, надо ли проверять дочерний узел, использовалось условие «полностью ли содержит MBR проверяемого узла V , MBR объекта поиска O ». В данном случае это условие изменилось на «полностью ли содержит MBS проверяемого узла V , MBS объекта поиска O ». В остальном процедура осталась прежней. Представлена на рис. 3.21.

Реализация этого условия для R -дерева может быть следующей (графически она представлена на рис. 3.21(а))

$$V.x_1 \leq O.x_1 \text{ И } V.x_2 \geq O.x_2 \text{ И } V.y_1 \leq O.y_1 \text{ И } V.y_2 \geq O.y_2.$$

Для SS -дерева аналогичная проверка примет вид (графический пример – рис. 3.21(б)):

$$Dist(V.Центр, O.Центр) + O.Радиус \leq V.Радиус.$$

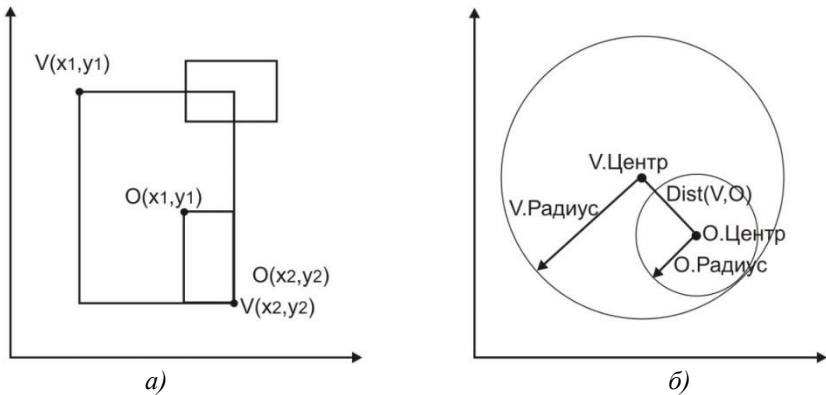


Рис. 3.21. Проверка на вхождение объекта в область узла: а – R -дерево; б – SS -дерево

Алгоритм добавления нового объекта



Алгоритм добавления объекта в SS -дерево тоже не претерпел значительных изменений по сравнению с аналогичным для R^* -деревьев. Общая логика работы осталась прежней (см. листинг 3.19). В нем также используются многие эффективные приемы, разработанные для R -дерева

и его разновидностей (например, принудительные повторные вставки объектов).

Для добавления объекта в дерево для него создают листовую вершину и выбирают внутренний узел дерева, в который эту вершину нужно поместить. Причем в качестве критерия выбора вершины для вставки нового объекта можно принять расстояние от центра вершины до центра вставляемого объекта (т. е. при спуске от корня к листовому уровню в качестве вершины вставки выбирается та вершина, чей центр находится ближе всего к центру вставляемого объекта).

Допустим, в качестве наиболее подходящей вершины для вставки нового объекта была выбрана некоторая вершина V . Если эта вершина содержит потомков меньше, чем максимально возможное число (элементов в списке *Дочерние_Узлы* меньше M), то данный объект просто добавляется в список. После этого корректируются центр описанной окружности и ее радиус у вершины V и всех ее предков, включая корень дерева.

Если же вершина V содержит максимальное число потомков и добавление нового объекта в нее невозможно, то необходимо попробовать удалить из нее часть объектов и вставить их заново в дерево, используя эту же процедуру вставки. Принудительная повторная вставка объектов может привести к тому, что часть объектов перейдет в другие вершины. При этом мы избавляемся от необходимости разбивать данный узел и улучшаем структуру дерева в целом. Выполняя процедуру повторной вставки объектов необходимо учитывать несколько фактов.

- После удаления объектов из вершины и перед их повторной вставкой в дерево необходимо скорректировать центр описанной окружности и ее радиус у вершины V и всех ее предков. Если этого не сделать, то все удаленные объекты снова вернутся в вершину V .

- В качестве объектов, которые необходимо попробовать вставить заново в дерево, выбирают те вершины-потомки узла V , которые наиболее удалены от ее центра. Причем число таких объектов необходимо принять равным 30% от общего числа потомков вершины.

- При выполнении повторных вставок объектов может получиться так, что все объекты вернулись в вершину V , и необходимость ее разбиения осталась. В этом случае алгоритм может заиклиться (снова попытаться принудительно повторно вставить объекты). Чтобы такого не произошло, необходимо каким-то образом пометать вершины, которые уже были переполнены ранее в процессе выполнения алгоритма, и, если в них произошло повторное переполнение, то вместо повторных вставок выполнять их разбиение.

Если повторная вставка объектов не помогла, то вершина разбивается на две. Для этого выбирается измерение, по которому в данной вершине наблюдается наибольшая дисперсия объектов (разброс). После этого вдоль этого измерения разбивается вершина на две части с

сохранением ограничения на заполнение вершин дерева. Образовавшиеся две вершины добавляются в дерево по той же самой логике, что и листовой узел.

Алгоритм построения минимальной описывающей окружности

Почти все алгоритмы работы с SS -деревьями являются небольшими модификациями алгоритмов обработки R^* -деревьев. Эти модификации зачастую являются тривиальными и легко реализуются в практических примерах, поэтому в данной главе они не приводятся. Однако есть ряд алгоритмов, имеющих нетривиальное выполнение. К ним, в частности, относится алгоритм построения минимальной описывающей окружности в пространствах большой размерности. Рассмотрим здесь наиболее популярные и простые алгоритмы для случаев, когда в качестве объектов выступают точки.

Простейший алгоритм. Самый простой алгоритм заключается в нахождении точки пересечения диагоналей минимального ограничивающего прямоугольника. Эта точка и будет центром описывающей окружности (рис. 3.22).

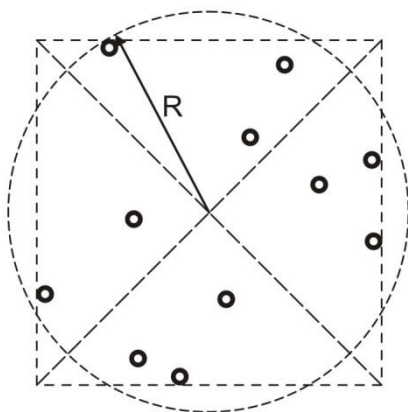


Рис. 3.22. Простейший вариант алгоритма

Радиус же этой окружности рассчитывается как расстояние до самой удаленной точки. Алгоритм программы представлен в листинге 3.24.

Листинг 3.24

```
//=====
// Построение описывающей окружности
// Параметры:
//   Р - список точек, для которых необходимо
//       найти описывающую окружность
//=====
```

ПОСТРОИТЬ_СФЕРА1(P)

[1] Для каждой оси i вычислить

Min_i = минимальное значение по оси i для точек P

Max_i = максимальное значение по оси i для точек P

$C_i = (\text{Max}_i - \text{Min}_i) / 2$

[2] // Центр окружности

C = точка с координатами (C_1, C_2, \dots, C_N)

[3] $R = 0$

Для каждой точки P_i из списка P

Если $R < \text{Dist}(C, P_i)$, то

$R = \text{Dist}(C, P_i)$

[4] Вернуть сферу с центром в точке C радиусом R

Конец ПОИСК_ОБЪЕКТА

Данный алгоритм прост в понимании и реализации. Однако построенная в результате окружность является далеко не минимальной. В практических реализациях лучше использовать более сложные итерационные алгоритмы.

Итерационный алгоритм М. Бадоу и К. Клаксона [19]. Данный алгоритм является итерационным. В результате нескольких итераций ограничивающая сфера получается меньшего размера, что приводит к более эффективной структуре SS -дерева. Однако данный алгоритм более сложен с вычислительной точки зрения. Также к недостаткам этого варианта алгоритма можно отнести то, что результирующая окружность является все же не самой минимальной из всех возможных.

Алгоритм представлен в листинге 3.25, а некоторое пояснение на конкретном примере – рис. 3.23.

Листинг 3.25

```
//=====
// Построение описывающей окружности
// Параметры:
//   P – список точек, для которых необходимо
//       найти описывающую окружность
//=====
ПОСТРОИТЬ_СФЕРА2(P)
[1] C = любая точка из списка P
    Delta = порог точности + 1
    t = 1
[2] Выполнять цикл, пока Delta больше порога точности
    V = самая удаленная точка от C
    R = Dist(C, V)
    C = t*C / (t+1) V / (t+1)
    Delta = V / (t+1)
    t++
[3] Вернуть сферу с центром в точке C
Конец ПОИСК_ОБЪЕКТА
```

Рассмотрим алгоритм программы более подробно. На рис. 3.23 представлен ход выполнения алгоритма по шагам.

На первом шаге алгоритма выбирается любая точка из списка. Допустим, в качестве такой точки была выбрана точка A . Также на этом шаге инициализируется переменная t – номер итерации. Затем алгоритм будет итерационно повторяться до тех пор, пока изменение центра описывающей окружности станет незначительным в рамках поставленной задачи.

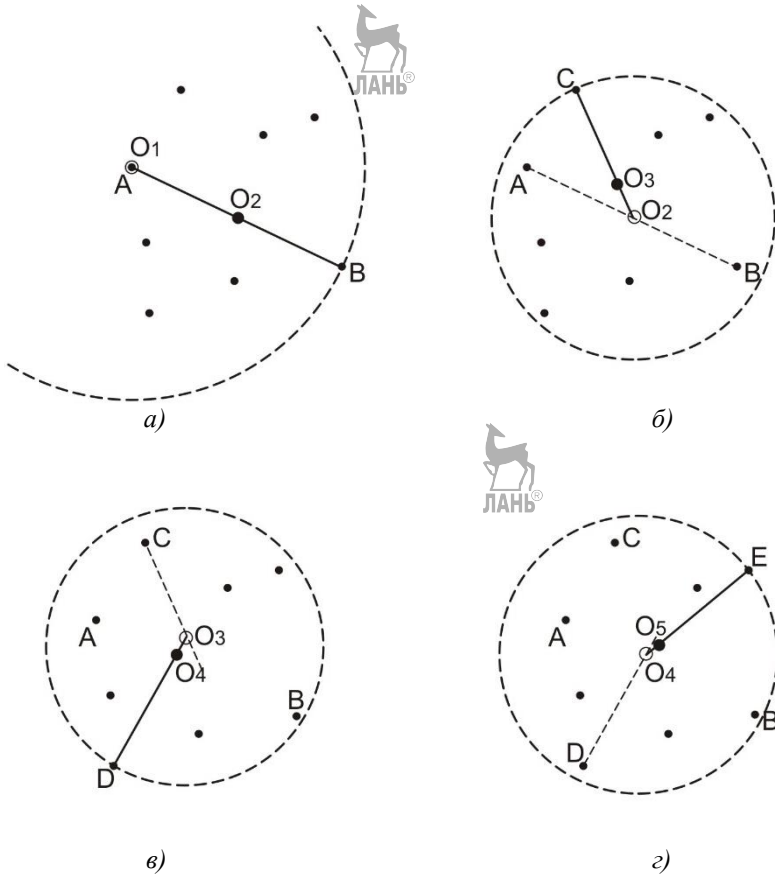


Рис. 3.23. Итерационный алгоритм

На каждом шаге итерационного процесса среди всех точек находится наиболее удаленная от точки, принятой за временный центр. На первой итерации такой точкой будет B . Расстояние до этой точки

будет радиусом описывающей окружности текущей итерации. Следующее приближение центра находится по формуле

$$C = t \cdot C / (t+1) + V / (t+1),$$

где V – наиболее удаленная точка от C ;

t – номер итерации.

Производя вычисления по этой формуле, получаем новое положение центра описывающей окружности — O_2 . На следующей итерации наиболее удаленной точкой станет точка C , и центр при этом переместится в точку O_3 (рис. 3.23(б)). Продолжая подобные вычисления, на четвертом шаге центр сместится в направлении точки E незначительно, поэтому итерационный процесс можно будет прервать. При этом описывающую сферу следует принять с центром в точке O_5 (рис. 3.23(г)).

Эффективность SS-дерева

Проведенные эксперименты показали, что запросы на поиск ближайшего соседа выполняются гораздо быстрее в SS -дереве, чем в различных вариантах R -деревьев. Превосходство становится особенно заметно, если число измерений превосходит 5 (при малом количестве измерений эффективность структур практически одинаковая). Причем это превосходство заметно не только по времени выполнения запроса для структуры в оперативной памяти, но и по числу обращений к жесткому диску, что является немаловажным при организации больших баз данных. Также в SS -дереве более высоким является показатель использования пространства (более 85%).

Однако по другим типам запросов поиска SS -дерево показывает не столь превосходные результаты. По некоторым видам поиска оно даже значительно проигрывает R -деревьям. Это происходит из-за того, что минимальные ограничивающие сферы (MBS), применяемые в SS -деревьях, имеют склонность к значительному перекрытию.

В практических разработках лучше использовать другой вид деревьев — SR -деревья, описанные далее. В них предпринята попытка скрестить характеристики SS - и R -деревьев с целью улучшения их свойств.

3.2.5. SR-дерево

Было отмечено, что применение сферических форм для внутренних узлов дерева дает значительное преимущество по сравнению с прямоугольными, особенно в запросах поиска похожих объектов. SS -дерево, основанное на этой технологии, делит наборы объектов на схожие по местоположению группы. Это достигается с помощью нахождения геометрического центра лежащих в некоторой области объектов.

Тем не менее, сферы имеют ряд недостатков. Самым важным из них является то, что сферические области имеют склонность расти по площади занимаемого пространства и вызывать тем самым значительные перекрытия узлов. Это не может не отразиться, например, на поиске по области.

Чтобы искоренить этот недостаток, Норио Катаяма и Шиничи Сатох в 1997 году предложили новую структуру, в которой предполагалось скрестить R^* -дерево и SS -дерево, взяв из них самые лучшие свойства [54]. Получившаяся структура была названа SR -деревом (*Sphere/Rectangle-Tree*).

Свойства ограничивающих сфер и прямоугольников

Рассмотрим основные положительные и отрицательные стороны использования сфер и прямоугольников в качестве минимальных ограничивающих фигур для узлов дерева. На рис. 3.24 представлены прямоугольник и окружность для некоторого набора точек в двухмерном пространстве.

Преимущества ограничивающих сфер. В запросах на поиск подобных объектов (поиск ближайшего соседа, поиск примера объекта в базе данных для некоторого набора характеристик и т. д.) основная функция, к которой приходится постоянно обращаться, – это расстояние от текущей рассматриваемой точки до группы объектов. Именно анализируя данное расстояние, делается вывод о необходимости проверки объектов, связанных с той или иной вершиной дерева. Если самая близкая точка ограничивающей фигуры находится дальше, чем заданная в запросе поиска дистанция, то отпадает необходимость всяких проверок объектов такой группы.

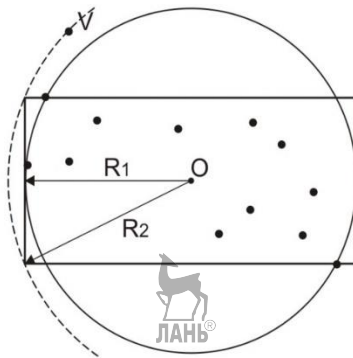


Рис. 3.24. Ограничивающие фигуры – прямоугольник и окружность

В случае сферической формы, ограничивающей группу фигуры, расстояние просто сравнивается с радиусом сферы. Так, пусть

необходимо найти все объекты, находящиеся на расстоянии d от точки V (рис. 3.24). Для определения, стоит ли проверять на соответствие запросу объекты, расположенные в окружности, достаточно проверить условие:

$$\text{Dist}(V, O) \leq R_1 + d,$$

где R_1 – радиус минимальной описывающей окружности.

Если данное условие является истинным, то вполне возможно, что среди объектов, находящихся в заданной окружности, есть те, которые будут удовлетворять запросу поиска. Поэтому если в дереве встречается некоторый внутренний узел, у которого минимальная описывающая окружность удовлетворяет такому условию, то необходимо проверить все его дочерние узлы на соответствие запросу поиска. Подобная логика характерна, например, для *SS*-деревьев.

При использовании прямоугольных областей (например, при реализации *R**-деревьев) аналогичное условие будет выглядеть следующим образом:

$$\text{Dist}(V, O) \leq R_2 + d,$$

где R_2 – половина длины главной диагонали.

Нетрудно заметить, что существуют случаи, когда для прямоугольников заданное условие даст положительный результат и придется выполнить проверку всех объектов, объединенных в данную группу, хотя нужных объектов в ней нет. Это связано с тем, что радиус ограничивающей окружности R_1 практически всегда меньше половины длины главной диагонали ограничивающего прямоугольника R_2 (что и показано на рис. 3.24).

Исходя из подобных размышлений, можно прийти к выводу, что применение сфер дает преимущество в запросах, ориентированных на расстояние, именно за счет меньшего радиуса ограничивающей фигуры. Причем максимальная разница между радиусами зависит от числа измерений (точнее равна корню квадратному из числа измерений). Так, в двухмерном случае максимально возможная относительная разница радиусов может достигать 1,4 (корень из 2), что на практических распределениях может не дать значительного преимущества. Однако при увеличении числа измерений до 16 максимальная относительная разница уже равна 4, а это уже существенно влияет на производительность. Таким образом, использование сфер в качестве группирующих фигур для узлов дерева дает колоссальный прирост производительности в пространствах большой размерности.

Преимущества ограничивающих прямоугольников. Рассмотрим ситуацию на том же самом рисунке, но с точки зрения площади ограничивающей фигуры. ЛАНЬ®

Площадь круга равна πR^2 , а прямоугольника – произведению его сторон. Нетрудно увидеть, что в большинстве случаев площадь круга, описывающего некоторые объекты, будет значительно превосходить площадь прямоугольника, делающего то же самое.

Казалось бы, площадь описывающих фигур, связанных с узлами дерева, прямо не участвует ни в одном из часто используемых запросов поиска. Поэтому данным параметром можно пренебречь. Однако это совсем не так. Если построить несколько описывающих окружностей и прямоугольников для одного и того же набора данных (например, как это сделано на рис. 3.20), то станет совершенно очевиден тот факт, что чем больше площадь ограничивающих фигур, связанных с разными вершинами в дереве, тем больше взаимное пересечение этих фигур. Ситуация усугубляется еще и тем, что окружности сами по себе склонны к взаимному пересечению, в то время как при использовании прямоугольных областей такое пересечение можно значительно минимизировать.

Как было неоднократно описано (в параграфах, посвященных разным вариантам R -деревьев), пересечение минимальных описывающих областей, связанных с разными вершинами в дереве, приводит к ложным срабатываниям в процессе поиска по точному совпадению и поиска по некоторому заданному диапазону. Поэтому можно прийти к выводу, что деревья, использующие прямоугольники в качестве минимальных ограничивающих фигур, будут более эффективными при выполнении поиска подобного плана.

Все изложенное можно обобщить следующим образом.

1. Ограничивающие сферы позволяют объединить объекты в группы маленького диаметра. Тем не менее, они имеют объемы значительно большие, чем ограничивающие прямоугольники.

2. Ограничивающие прямоугольники позволяют объединить точки в области небольшого объема (а значит, и с меньшим взаимным пересечением). Однако они имеют большие диаметры из-за другого поведения их диагональной длины (особенно это заметно в пространствах большой размерности).

Таким образом и ограничивающие сферы, и ограничивающие прямоугольники имеют как достоинства, так и недостатки. Каждая из этих фигур предпочтительнее в каких-то видах поиска. Однако в практических ситуациях необходимо выполнение запросов разных видов, поэтому хорошо бы иметь структуру, учитывающую и небольшой объем, и маленький диаметр.

Структура SR-дерева

Чтобы структура узла дерева имела небольшой объем и при этом маленький диаметр, Н. Катаяма и Ш. Сатох предложили определить ее ограничивающую фигуру как пересечение прямоугольника и сферы [54]. Используя эту идею, они разработали древовидную структуру, получившую название SR -дерево. Общие принципы и методы построения дерева были позаимствованы от R^* -деревьев и соответствуют идее

вложенной иерархии областей. Пример для некоторого набора данных показан на рис. 3.25.

Внутренние и листовые узлы дерева имеют структуру, являющуюся синтезом структур R^* -дерева и SS -дерева. В ней, кроме самих данных и ссылок на дочерние узлы, хранятся координаты минимального ограничивающего прямоугольника (MBR), а также центр и радиус минимальной ограничивающей сферы (MBS), с помощью пересечения которых получается ограничивающая область SR -дерева.

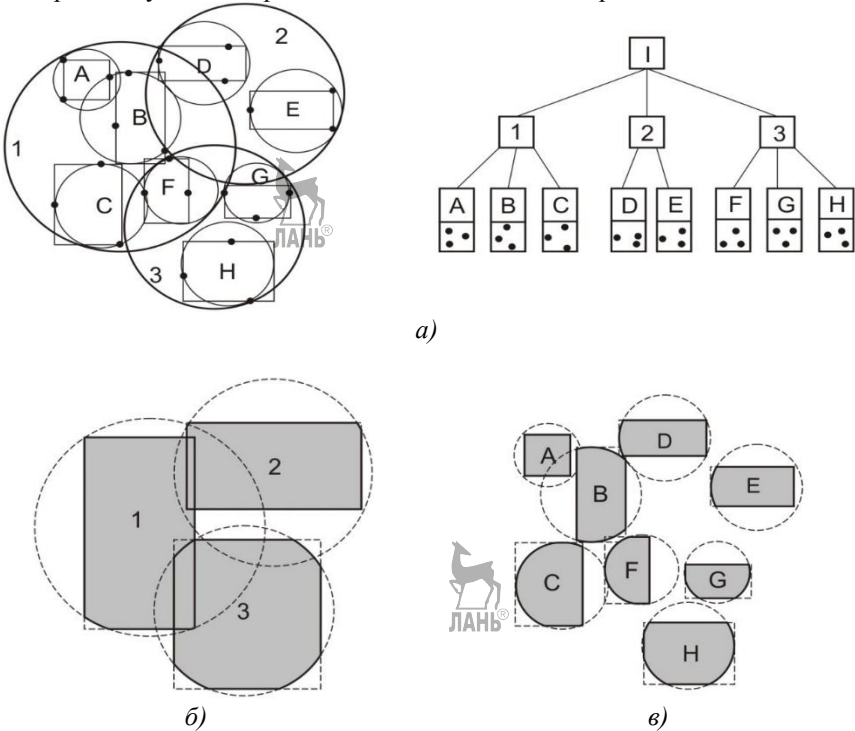


Рис. 3.25. Пример SR -дерева: а – общая структура; б – внутренний уровень; в – листовой уровень

Из всего изложенного следует, что отличие в реализации структуры узла R^* -дерева и SR -дерева заключается во введении в последнюю дополнительных полей для хранения центра и радиуса описывающей окружности. Это увеличивает накладные расходы на само дерево, так как в каждом узле появляются новые $(N+1)$ числовых полей (N компонент координаты центра описывающей окружности в N -мерном пространстве и одно числовое поле – ее радиус).

Алгоритм добавления нового объекта

Алгоритм добавления объекта в *SR*-дерево позаимствован из алгоритмов работы с *SS*-деревьями. Это было сделано из соображений того, что алгоритм построения *SS*-дерева был изначально ориентирован на построение структуры, в которой наиболее эффективно выполняются запросы поиска ближайшего соседа.

Сначала алгоритм добавления определяет наиболее подходящее поддерево по признаку минимизации расстояния от добавляемого объекта до центра вершины. Если выбранная вершина окажется не полностью заполненной, то объект добавляется как очередной дочерний элемент вершины и производится модификация ограничивающего прямоугольника и сферы у данной вершины и всех ее предков.

Если же объект не помещается в выбранную вершину, то, как и в случае с *SS*-деревом, повторно вставляется некоторая часть дочерних элементов, пока не появится возможность обойтись без разбиения узла. Если все же окажется, что разбиения не избежать, то алгоритм рассчитывает разницу центров потомков по всем измерениям. Измерение с наибольшей разницей выбирается как ось, по которой и делится исходный узел на два новых. Более подробно данный алгоритм описан в параграфе, посвященном *SS*-дереву. Единственное отличие алгоритма заключается в том, что после вставки объекта в вершину необходимо обновить не только ограничивающую сферу (*MBS*), но и ограничивающий прямоугольник (*MBR*). Способ обновления *MBR* полностью аналогичен тому, который использовался для *R*-деревьев, однако обновление *MBS* немного отличается от *SS*-дерева.

Центр ограничивающей сферы *SR*-дерева вычисляется следующим образом:

$$x_i = \frac{\sum_{j=1}^k V_j x_i V_j w}{\sum_{j=1}^k V_j w},$$

где j – номер узла потомка ($1 \leq j \leq k$);

i – номер измерения ($1 \leq i \leq N$);

$V_j \cdot x_i$ – i -я координата центра *MBS* j -го потомка;

$V_j \cdot w$ – число объектов, находящихся в поддереве, связанном с вершиной V_j .

В описанном определении участвует величина, которая ранее не присутствовала в *SS*-дереве – число объектов, находящихся в заданном поддереве. Данная величина не играла никакой роли в алгоритмах обработки *SS*-дерева, поэтому она и не вводилась раньше. Однако для *SR*-дерева она является существенной. Поэтому в каждом узле необходимо хранить еще одно поле w . При добавлении/удалении объектов дерева (т. е. при любых модификациях структуры) необходимо предусмотреть пересчет этого поля таким образом, чтобы в нем всегда находилась верная информация.

С помощью указанной формулы рассчитывается центр новой описывающей сферы. При этом радиус окружности можно вычислить следующим образом:

$$r = \min(DistS, DistR);$$

$$DistS = \max(\|x - V_i\| + V_i.r), \text{ где } 1 \leq i \leq k;$$

$$DistR = \max(MaxDist(x, V_i.R)), \text{ где } 1 \leq i \leq k,$$

где $V_i.r$ – радиус *MBS* i -го потомка;

$V_i.R$ – минимальный описывающий прямоугольник (*MBR*) i -го потомка.

В этих формулах *DistS* обозначает максимальное расстояние от центра родительского узла до ограничивающих сфер его потомков, а *DistR* – максимальное расстояние до ограничивающих прямоугольников его потомков.

В *SS*-дереве радиус *MBS* был равен *DistS*. В *SR*-дереве выбирается наименьшее из *DistS* и *DistR*. Это говорит о том, что радиус ограничивающих сфер *SR*-дерева в общем случае будет меньше, чем в *SS*-дереве, а это приведет к более качественной и эффективной структуре в целом.

В описанных формулах используется функция *MaxDist*, которая должна выбирать максимальное расстояние от точки x до прямоугольника R . Она определяется следующей формулой:

$$MaxDist(x, R) = \max(\|x - q\|),$$

где q – любая точка, принадлежащая прямоугольнику R .

Это выражение можно вычислить с небольшими вычислительными затратами, подставляя в него только вершины прямоугольника R , находящиеся наиболее удаленно от точки x .

Алгоритм удаления объекта

Алгоритм удаления объекта из *SR*-дерева ничем не отличается от аналогичного алгоритма в *SS*- и *R**-деревьях. Если при удалении записи из некоторого узла не появилось причин для повторной балансировки дерева (число оставшихся дочерних элементов все еще превышает нижний предел по наполнению узла m), обновляются соответствующие поля узла и его потомка с сохранением общей структуры.

Если же узел дерева после удаления объекта стал содержать недостаточное число потомков, то он удаляется из дерева, а его потомки просто заново вставляются с использованием все той же процедуры добавления объектов.

Данный алгоритм был приведен и описан ранее в соответствующих параграфах, посвященных *SS*- и *R**-деревьям.

Поиск ближайшего соседа

Данный алгоритм был описан ранее. Использование его в *SR*-деревьях принципиально ничем не отличается от использования в других структурах.

Алгоритм представляет собой поиск в глубину. В каждой вершине проверяется каждый дочерний узел на возможность нахождения в нем потенциальных решений. Если эта проверка дает положительный результат, то соответствующее поддерево просматривается в глубину. Сама же проверка заключается в сравнении текущего найденного расстояния и минимального расстояния до ограничивающей фигуры узла.

При просмотре в глубину дочерних узлов некоторой вершины приоритет отдается сначала тем узлам, которые находятся ближе всего к точке поиска. Это позволяет многократно сократить пространство поиска.

Хотя алгоритм поиска ближайшего соседа похож на алгоритмы для *SS*- и *R**-деревьев, но в нем все же есть отличие. Оно заключается в способе вычисления минимального расстояния от точки поиска до области. Поскольку ограничивающая фигура узла *SR*-дерева является пересечением ограничивающей сферы и прямоугольника, минимальное расстояние от точки поиска до этой фигуры вычисляется как большее из минимального расстояния до *MBR* и *MBS*. Его можно определить по следующей формуле:

$$\begin{aligned} Dist &= \max (DistS, DistR), \\ DistS &= \max(0, \|x - V.x\| - V.r), \\ DistR &= \text{MinDist}(x, V.R). \end{aligned}$$

DistS в данном случае является минимальным расстоянием от точки поиска *x* до ограничивающей сферы потомка *V*, а *DistR* – минимальное расстояние от точки поиска до ограничивающего прямоугольника. Это расстояние находится с помощью функции *MinDist*, которую, по аналогии с *MaxDist*, можно определить следующим образом:

$$\text{MinDist}(x, R) = \min(\|x - q\|),$$

где *q* – любая точка, принадлежащая прямоугольнику *R*.

Это выражение также можно вычислить с небольшими вычислительными затратами, если подставлять в него только вершины прямоугольника *R*, находящиеся наиболее близко к точке *x*.

В процедуре поиска ближайшего соседа в *R**-дереве минимальное расстояние определялось по значению величины *DistR*, а в *SS*-дереве – по значению *DistS*. *SR*-дерево выбирает большее из *DistS* и *DistR*. Это обеспечивает лучшую оценку расстояния от точки поиска до ближайшей точки области и улучшает эффективность стратегий сокращения перебора. В результате *SR*-дерево при обходе не посещает многие заведомо неподходящие узлы, которые могут быть проверены в *R**- или



SS-дереве. При больших объемах данных это очень сильно увеличивает эффективность поиска.

Эффективность SR-деревя

Как было показано ранее, *SR*-дерево превосходит по своим параметрам и *R**-дерево, и *SS*-дерево. Это также подтверждают многочисленные эксперименты. Структура дерева получается адаптированной как для запросов ближайшего соседа (диаметры ограничивающих фигур в дереве получаются меньше, чем в других структурах), так и для других видов поиска (суммарная площадь всех потомков и их взаимное перекрытие в *SR*-дереве даже меньше, чем у *R**-деревя).

Благодаря этим свойствам, на реальных данных *SR*-дерево на запросах по области может давать выигрыш по нагрузке на *CPU* до 90% по сравнению с *SS*-деревьями и до 70% - по сравнению с *R*-деревьями.

Однако есть один момент, по которому *SR*-деревья проигрывают своим предшественникам. Для хранения записи об одном узле *R*-деревя необходимо выделить память в размере $2N$ ячеек (где N – размерность пространства). Это необходимо для хранения координат ограничивающего прямоугольника. Как известно, любой прямоугольник можно однозначно идентифицировать по координатам его двух противоположных углов.

Для хранения информации об узле *SS*-деревя необходимо выделить память в размере $(N + 1)$ ячеек. Эта память будет использоваться для хранения координаты центра, описывающей сферы в N -мерном пространстве (N ячеек) и ее радиуса (еще одна ячейка).

В узле *SR*-деревя необходимо хранить и ограничивающий прямоугольник, и ограничивающую сферу. Поэтому накладные расходы составляют $(3N + 1)$. Для пространств большой размерности этот параметр оказывается почти в 3 раза больше, чем для *SS*-деревя и в 1,5 раза – в сравнении с *R*-деревьями.

Размер данных, занимаемых одним узлом дерева, влияет и на число потомков в узле. Обычно для потомков одного узла выделяют блок памяти, кратный по размеру блоку памяти жесткого диска (или другого хранилища). Это сделано для того, чтобы весь узел можно было считать с жесткого диска в память за один раз. Принимая этот факт во внимание, получаем, что максимальное число потомков (параметр M) в *SR*-дереве составляет всего треть по сравнению с *SS*-деревом и две трети – по сравнению с *R*-деревом. При этом число потомков в узле дерева влияет на высоту дерева в целом. Поэтому в общем случае *SR*-дерево будет выше, чем аналогичные структуры.

Данный факт, конечно, влияет на производительность структуры в целом. Однако положительные стороны и характеристики структуры

превосходят негативное влияние высоты дерева, и оно остается предпочтительным для большого спектра систем.

3.2.6. TV-дерево

Все описанные до этого структуры очень хорошо подходят для индексации данных в пространствах небольшой размерности. Самую лучшую производительность они показывают на двух- и трехмерных данных. Но при возрастании числа измерений их производительность резко падает. Это связано с тем, что в пространствах большой размерности очень сложно разделить группы объектов так, чтобы ограничивающие их подобласти не пересекались. Чем больше измерений, тем больше взаимно пересекающихся узлов появляется в дереве. Нетрудно заметить, что такое поведение приведет к лишним просмотрам ветвей дерева и большому перебору при поиске.

В 1994 г. исследователи К. Лин, Г. Джагадиш и К. Фалоутсос предложили новый подход к индексированию, названный *TV-деревом* [61]. Основная идея структуры заключается в уменьшении числа измерений, рассматриваемых в каждом узле дерева. Таким образом, получается сокращение пространства и улучшение характеристик пересечения.

У данной структуры есть как свои достоинства, так и недостатки. Многие исследователи критикуют данную структуру, приводя в качестве аргумента тот факт, что эффективность алгоритмов над ней очень сильно зависит от конкретного применения. Дело в том, что *TV-дерево* требует выделение приоритетов осей пространства. Поэтому практически невозможно написать реализацию, которая была бы универсальной и выполнялась с одинаковой эффективностью в любом применении. В каждом конкретном случае необходимо разрабатывать конкретное *TV-дерево*. Однако в ряде практических ситуаций такой подход вполне оправдан. К тому же идеи, заложенные в *TV-дереве*, впоследствии много раз использовались в других структурах.

Общие принципы TV-дерева

Как было упомянуто, существует ряд приложений, требующих индексирования данных в пространствах большой размерности. При этом *TV-дерево* создавалось для того, чтобы уменьшить негативное воздействие экспоненциального падения производительности по числу измерений.

Решение, которое было предложено исследователями, заключалось в сокращении числа измерений, рассматриваемых в каждом узле дерева. При этом ряд измерений не отбрасывается, а просто учитывается в других узлах. Таким образом происходит динамическое расширение и сокращение набора характеристик в дереве по мере необходимости.

Данный подход применяют люди при классификации объектов. Так, в зоологии все животные объединяются в несколько широких классов (позвоночные и беспозвоночные), затем каждый класс делится в зависимости от других характеристик на теплокровных и холоднокровных и т. д. Таким образом, получаем, что человек работает не со всеми характеристиками сразу, а вводит их в свое рассмотрение только при определенной необходимости.

Именно этот принцип постарались воссоздать исследователи в *TV*-дереве. Вектор характеристик, описывающих объект, динамически сокращается и расширяется в дереве по мере необходимости. При этом все остальные алгоритмы индексирования остались такими же, как в *R*-дереве или *SR*-дереве (в зависимости от реализации).

Все объекты пространства заключаются в некоторый минимальный ограничивающий прямоугольник (*MBR*). Этот *MBR* соотносится с корнем дерева. Далее этот *MBR* в зависимости от определенного набора характеристик разбивается на ряд дочерних *MBR*, содержащих приблизительно равное число объектов (дочерние узлы). Данный процесс продолжается рекурсивно, пока в каждом *MBR* не останется столько объектов, сколько может находиться в листовой вершине дерева. При этом, в отличие от классического *R*-дерева, при формировании *MBR* участвуют не все характеристики, а только некоторое их подмножество. Это позволяет уменьшить взаимное пересечение внутренних узлов дерева и увеличить производительность.

В начальный момент дерево может состоять из нескольких узлов и использовать всего несколько характеристик. При добавлении объектов в дерево и увеличении его размера все большее количество характеристик будет использоваться для индексирования.

В операциях поиска в корне поддерева выбирается только по нескольким, наиболее важным, критериям. Но при движении от корня к листьям рассматривается все большее число характеристик объекта. Этот процесс напоминает телескоп, поэтому дерево и было названо *TV*-деревом (*Telescopic Vector tree*).

Исследователи данной структуры замечают, что вовсе не обязательно использовать в качестве минимальных ограничивающих фигур прямоугольники (*MBR*, как в *R*-дереве). Можно применять сферы (*MBS*, как в *SS*-дереве) или использовать и то, и другое (как в *SR*-дереве). Алгоритмы работы с деревом при этом значительно не изменяются. Необходимо только внести небольшие коррективы в функции поиска (определение ветви дерева при поиске) и функцию расщепления.

Телескопическая функция сокращения

Для работы с *TV*-деревом необходимо ввести телескопическую функцию сокращения. Эта функция должна определять, какие из измерений используются в том или ином уровне дерева, т. е., по сути,

именно эта функция позволяет динамически расширять набор используемых характеристик при движении по дереву. На входе данная функция получает полный список измерений (характеристик объекта) размерностью n и некоторый показатель m (требуемый размер вектора характеристик), а на выходе она возвращает вектор уже требуемой размерности, в котором учитываются только те критерии, которые использовались при индексировании данных в текущем узле.

Математически телескопическую функцию можно описать так. Дан вектор характеристик x , размерностью $n \times 1$, где n – общее число измерений пространства. Имеется также матрица A_m размерностью $m \times n$ ($m \leq n$), называемая *матрицей сокращений*. При умножении вектора x на эту матрицу A_m получаем новый вектор $(A_m x)$, размерность которого уже сокращена до требуемого порога m . Набор матриц A_m , где $m = 1, \dots, n$ как раз и задает телескопическую функцию сокращений числа измерений в дереве. Однако не всякий набор матриц может применяться в TV -дереве. Необходимо чтобы набор матриц сокращений удовлетворял следующему условию: если некоторые сокращения m' двух векторов характеристик x и y равны, то для всех m'' , удовлетворяющих условию $m'' \leq m'$, сокращения m'' этих векторов также равны.

Рассмотрим для примера несколько телескопических функций сжатия. В дальнейшем изложении алгоритмов TV -дерева будет использоваться первая из них – функция сжатия с помощью отсеечения неиспользуемых измерений.

Функция сокращения отсечением последних $(n-m)$ измерений. Это самый простой вариант функции сокращения. В нем в матрице сокращения по главной диагонали расположены 1, а все остальные элементы равны 0. Так, если рассматривается шестимерное пространство ($n = 6$), то матрица сокращений до 3 ($m = 3$) будет иметь следующий вид:

$$A_3 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}.$$

Нетрудно заметить, что при умножении этой матрицы на любой вектор из шести компонентов получится вектор из трех первых компонентов этого вектора. Таким образом, получается просто отсеечение последних $(n-m)$ компонентов вектора.

В реализации данный способ является самым простым. Чтобы не загромождать описание, далее в примерах будет использоваться именно он.

Функция сокращения отсечением произвольных $(n-m)$ измерений. Предыдущий способ является самым простым. Однако он не всегда эффективен. Индексирование объектов в нем использует, в первую очередь, первые характеристики объекта, даже если они являются

не самыми важными. Это может негативно сказаться на процедуре поиска. Конечно, можно сортировать компоненты вектора характеристик по приоритету, но данная операция иногда может оказаться неудобной в реализации.

Если изначально формировать векторы с учетом приоритетов критериев по какой-то причине не получается, можно использовать другую функцию сжатия, которая выбирает любые m критериев из вектора. Матрица сжатия в этом случае должна удовлетворять всего одному условию: в каждой строке и столбце этой матрицы должно быть не более одной 1. При составлении этой матрицы ее столбцы как бы соответствуют критериям, а строки – приоритетам. Так, если критерий с номером 5 имеет наивысший приоритет, то в пятом столбце единица должна стоять в первой строке.

Для примера рассмотрим матрицу сокращений до трех компонентов, которая выбирает 5, 6 и 2 компоненты вектора характеристик, а остальные отбрасывает:

$$A_3 = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

Функция сокращения без отбрасывания компонентов.

Предыдущие функции сокращений удобно использовать в большинстве приложений. Однако иногда хотелось бы использовать такую функцию, которая сокращала вектор без отбрасывания компонентов. Так, если рассматривается пространство из 6 измерений, а $m = 2$, то предыдущие функции отбросят 4 компонента вектора характеристик.

Существует ряд функций, которые совмещают несколько компонентов (например, с помощью операции сложения) и таким образом уменьшают размерность вектора. Рассмотрим одну из таких функций. Матрицу сокращений этой функции без отбрасывания компонентов вектора характеристик зададим граничные условия.

1. Если $m = n$, то матрица сокращений имеет диагональный вид. Так, для размерности 6 матрица будет иметь вид:

$$A_6 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}.$$

2. Если $m \geq n/2$, то необходимо все строки от $(2m-n+1)$ до n предыдущей матрицы попарно сложить (строку $(2m-n+1)$ и $(2m-n+2)$,

строку $(2m-n+3)$ и $(2m-n+4)$, и т. д.). Например, матрица для $m = 4$ примет вид

$$A_4 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix}.$$

В данной матрице ряд компонентов вектора будут складываться при сокращении. При этом ни одна из координат отброшена не будет.

3. Если $m \leq n/2$, то после выполнения шага 2 процедуру попарного сложения строк необходимо повторить, пока не получится матрица сокращения нужного размера. В предельном случае матрица сокращений примет вид:

$$A_1 = [1 \ 1 \ 1 \ 1 \ 1 \ 1].$$

Применяя это правило, можно получить набор матриц сокращения, которые при уменьшении размерности будут учитывать все компоненты вектора и при этом удовлетворять главному свойству матриц сокращения, описанному выше.

Приоритеты и упорядочивание осей пространства в функциях сокращения. Недостаточно продуманное упорядочивание осей пространства в функции сокращения может существенно уменьшить производительность. Если в самом начале будут выбираться критерии, которые являются малоприоритетными или практически неизменными в данном приложении, то индексирование в корневых вершинах приведет к плохому разграничению вариантов и рекурсивному просмотру поддеревьев при поиске, в которых запрошенного элемента нет.

В большинстве приложений можно изначально на основе априорных знаний упорядочить оси пространства или даже их видоизменить так, чтобы многократно повысить производительность структуры. Эти преобразования не имеют прямого отношения к алгоритмам обработки *TV*-дерева, однако их применение бывает очень эффективным.

Одним из таких методов является способ упорядочивания критериев на основе разнообразия изменения их значения – *Karhunen – Loeve (KL)* [39]. Данный метод анализирует ряд объектов с n компонентами (n -мерное пространство). На основе проанализированных данных метод *KL* вырабатывает новые оси пространства и формулы перевода объектов из старого пространства в новое. При этом в новом пространстве оси упорядочены в порядке важности для индексирования данных. На рис. 3.26 представлен пример, иллюстрирующий изменение осей двумерного пространства K_1 и K_2 в другое пространство K_1' и K_2' .

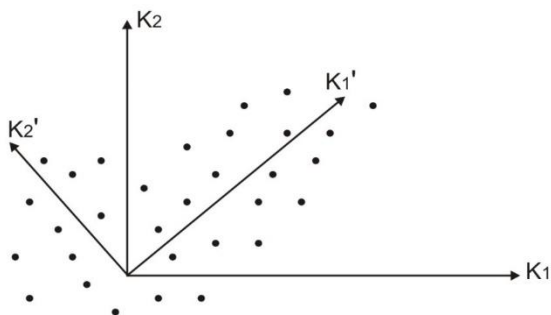


Рис. 3.26. Иллюстрация преобразования Karhunen – Loeve

KL преобразование оптимально, если набор индексируемых данных известен заранее. Это характерно для приложений с редким обновлением данных или статичных наборов (например, различные словари, базы данных на CD и т. д.).

KL преобразование также хорошо работает, если заранее известны не все данные, а только часть данных, которая хорошо описывает все статистические характеристики полного набора. Однако в полностью динамических средах лучше использовать другие методы, такие как дискретные преобразования Фурье, вейвлет преобразования и т. д. В ряде случаев они приведут к лучшим показателям производительности.

Структура узла дерева

Каждый узел в *TV*-дереве представлен минимальным ограничивающим прямоугольником (*MBR*) всех его потомков. Каждый такой прямоугольник определяется координатами двух его противоположных углов. Однако координаты углов *MBR* задаются не в *n*-мерном пространстве, в котором определены индексируемые объекты, а некоторым сокращением, которое получилось после применения телескопической функции сокращения данного узла. Чтобы в процессе реализации алгоритмов обработки знать, какую из матриц сокращения использовать, в узле дополнительно сохраняют ее номер *m*.

В многочисленных экспериментах было замечено, что при удалении от корня к листьям появляется большое число измерений, значение по которым одинаково во всех объектах узла. При этом все равно приходится обрабатывать эти координаты, так как они присутствуют после обработки матрицей сокращения.

Допустим, у нас есть корневой узел, в котором разбиваются объекты по группам в зависимости от критерия K_1 . При этом пусть в левое поддерево попали только объекты, у которых $k_1 = x$, а в правое – объекты, у которых $k_1 = y$. На следующем уровне матрица сокращений может предоставить в рассмотрение уже два критерия – K_1 и K_2 . Однако

использование K_1 при этом является излишним, так как все объекты каждого из поддеревьев имеют одинаковые значения по этому критерию. Чтобы упростить процесс сравнения и увеличить производительность структуры, разработчики ввели понятие *активных* и *неактивных измерений*.

Измерение называется *неактивным*, если все объекты данного узла имеют одинаковые значения по этому измерению. Очевидно, что в этом случае координаты по данным измерениям можно не хранить в каждом узле и не рассматривать при поиске в соответствующем поддереве. Также является очевидным, что все потомки некоторого узла могут отличаться координатами только активных измерений.

Обозначим число активных измерений α . Практические эксперименты с *TV*-деревом показали, что максимальная производительность структуры достигается в том случае, если число активных измерений равно $\alpha = 2$. Поэтому рекомендуется использовать в практических реализациях именно это значение. При меньшем значении активных измерений ($\alpha = 1$) очень мало критериев, по которым можно разделять объекты в пространстве. При большем значении появляются дополнительные расходы производительности вследствие увеличения пересечения областей отдельных узлов.

После применения функции сокращения к некоторому вектору характеристик остается только m используемых критериев (остальные отбрасываются матрицей сокращения), причем только α из них являются активными. Остальные измерения являются фиксированными, и все объекты поддерева должны иметь одинаковые значения по ним. Величина α при этом является некоторым глобальным показателем *TV*-дерева. Иногда конкретную реализацию дерева с определенным α обозначают в самом названии дерева. Так, реализация при $\alpha = 2$ будет называться *TV-2-деревом*.

Введем математическое описание критериев объектов, размещенных в некотором узле дерева. Пусть дан узел дерева V , к которому применяется функция сокращения с параметром m (этот параметр обычно хранится в самом узле и обозначает общее число используемых критериев). При этом само дерево построено с показателем активных измерений, равному α . В этом случае все объекты O , размещенные в данном поддереве, должны удовлетворять следующим условиям:

$$\forall O: O.K_{i,j} = V.K_{i,r}, \quad i = 1, (m - \alpha);$$

$$\forall O: O.K_{i,l} \leq O.K_i < V.K_{i,r}, \quad i = (m - \alpha + 1), m;$$

где $O.K_i - i$ -я координата объекта O ;

$V.K_{i,l}$ – левая граница i -й координаты *MBR* вершины V ;

$V.K_{i,r}$ – правая граница i -й координаты *MBR* вершины V .

Рассмотрим несколько примеров для двумерного пространства (рис. 3.27).

На рис. 3.27(а) представлены вершины для дерева $TV-1$ (с одним активным измерением). При этом вершине V_1 соответствует функция сокращения с $m = 2$. Это означает, что в вершине используются оба измерения, однако всего одно из них будет активным. Все дочерние объекты этой вершины должны иметь координату $K_1 = 10$ (так как это измерение неактивно, то его значение является жестко заданным), а координату K_2 – в диапазоне от 20 до 50.

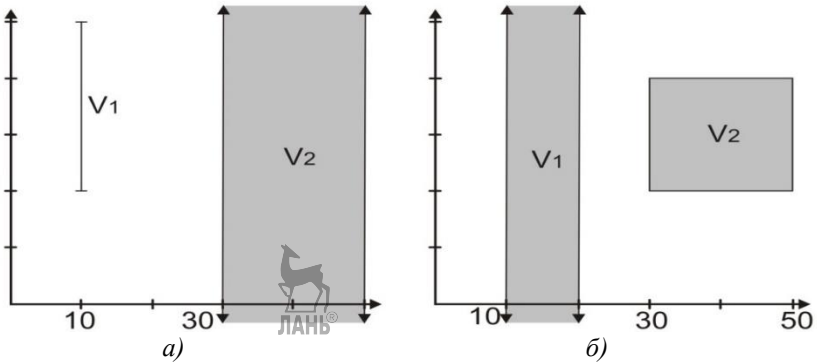


Рис. 3.27. Пример вершин TV -дерева: а – вершины $TV-1$ дерева; б – вершины $TV-2$ дерева

Вершина V_2 на этом же рисунке соответствует случаю, когда у вершины всего одно используемое измерение ($m = 1$). При этом все дочерние объекты этой вершины должны иметь координату K_1 в диапазоне от 30 до 50 (так как первое измерение является активным) и любые значения второй координаты (т. к. второе измерение является неиспользуемым).

На рис. 3.27(б) показан пример вершин $TV-2$ -дерева, у которых число активных измерений $\alpha = 2$. Вершина V_1 соответствует случаю, когда в дереве всего одно используемое измерение ($m = 1$) и оно же будет активным. При этом в данную вершину попадут все объекты, у которых первый ключ находится в диапазоне от 10 до 20, а второй имеет произвольные значения. Если же число используемых измерений будет равно 2 (вершина V_2), то оба эти измерения будут активными и будет задан диапазон значений по каждому из них.

Структура дерева

В общих чертах структура TV -дерева похожа на обычное R -дерево. Каждый узел содержит набор ветвей. Каждая ветвь состоит из MBR (неполной, состоящей только из тех координат, которые являются активными для заданной вершины) и ссылки на дочерний узел. Все

потомки должны находиться в пределах той *MBR*, которая задана в родительской вершине. Однако, как и в *R*-деревьях, *MBR* вершин могут пересекаться друг с другом. Пример *TV*-дерева показан на рис. 3.28.

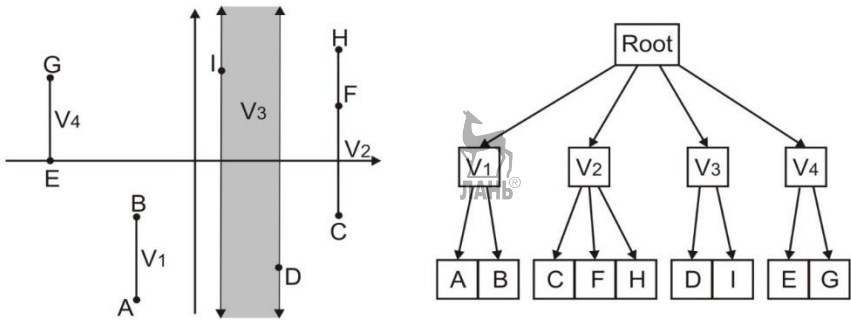


Рис. 3.28. Пример *TV*-1 дерева для двумерного пространства

В *TV*-1 дереве число активных измерений всегда равно 1. Поэтому *MBR* узлов могут принимать форму либо отрезка (вершины V_1 , V_2 , V_4 – если число используемых измерений равно $m = 2$) или прямоугольника, бесконечно расширяющегося вдоль второго измерения (вершина V_3 – число используемых измерений равно $m = 1$).

Алгоритмы поиска

TV-дерево является структурой, у которой каждому узлу соответствует некоторый *MBR*, причем области узлов могут пересекаться. Это не первая структура данного класса. Многие алгоритмы поиска и обработки были позаимствованы из предшествующих структур. Поэтому в данном параграфе псевдокод этих алгоритмов будет опущен.

Поиск на точное совпадение и поиск в области. Ранее уже были рассмотрены эти процедуры для *R*-дерева. В *TV*-дереве все обстоит практически также. Поиск начинается с корня и проверяет каждую ветвь на соответствие запросу. Если *MBR* некоторого узла не соответствует условиям поиска, то все поддерево отбрасывается из рассмотрения, иначе – процедура поиска повторяется для него рекурсивно. Алгоритм заканчивается в листовых вершинах, из которых выбираются объекты для формирования результата поиска.

Единственное отличие от *R*-дерева заключается в том, что в каждой вершине дерева находится не полный *MBR*, а его сокращение, т. е. при проверке соответствия вершины и условий запроса проверяются только те измерения (атрибуты поиска), которые остались после применения матрицы сокращения.

Поиск ближайшего соседа. Аналогично другим процедурам, этот алгоритм также позаимствован из R -деревьев. Поиск начинается с корня, и для каждого его потомка вычисляется минимальная и максимальная граница расстояния до запрошенного объекта. После этого все потомки проверяются в порядке от самого близкого к самому далекому. Причем во время работы процедуры постоянно хранится объект, который является ближайшим среди все просмотренных. Если MBR какой-то вершины окажется дальше этого объекта, соответствующее поддереву отбрасывается и не просматривается в глубину. Иначе – происходит проверка всех его дочерних узлов рекурсивно.

Алгоритм вставки объекта в дерево

Вставка объектов в TV -дерево в общих чертах ничем не отличается от тех процедур, которые были разработаны для R^* -дерева. Чтобы вставить объект в дерево, необходимо выбрать наиболее подходящий узел для размещения нового объекта. Для этого происходит перемещение от корня к листовому уровню и оценивается каждое поддерево по определенным критериям. Как только будет достигнут листовой узел, объект вставляется в найденную вершину. Если при этом произошло переполнение, то сначала процедура вставки пытается избежать деления узла с помощью удаления и повторной вставки в дерево наиболее удаленных объектов переполненной вершины. Если такой подход не помогает и некоторый лист все равно остается переполненным, то вызывается процедура деления узла. На самом последнем шаге происходит корректировка MBR всех вершин, которые так или иначе были модифицированы в процессе вставки объекта. Псевдокод данных процедур в данном параграфе дублироваться не будет (см. листинг 3.19).

Отличия процедур TV -дерева и R^* -дерева заключаются не в самом алгоритме вставки, а в алгоритмах поиска подходящего узла дерева, в который будет произведена вставка (**ВЫБОР_ВЕРШИНЫ**), и процедуре расщепления (**ДЕЛЕНИЕ_УЗЛА**). Эти алгоритмы мы рассмотрим более подробно.

Выбор вершины для вставки. Процедура выбора узла дерева, в котором будет размещен вставляемый объект, несколько отличается от аналогичных процедур для рассмотренных ранее структур. Эти отличия сосредоточены прежде всего на критериях выбора. В частности, появляются новые критерии, связанные с нововведением TV -дерева, – сокращением рассматриваемых измерений. На рис. 3.29 представлены примеры, иллюстрирующие эти критерии.

Рассмотрим критерии выбора узла в порядке убывания их приоритета.

1. *Минимальное увеличение числа пересекающихся MBR узлов.* Согласно этому критерию, необходимо выбирать тот узел, который после изменения его MBR не приведет к новым пересечениям областей узлов

(или увеличение числа пересекающихся областей будет минимальным). Так, на рис. 3.29(а) для вставки объекта P необходимо выбрать узел $R1$, так как расширение $R2$ или $R3$ приведет к новым парам пересекающихся областей (на рисунке ограничивающие области узлов имеют форму круга).

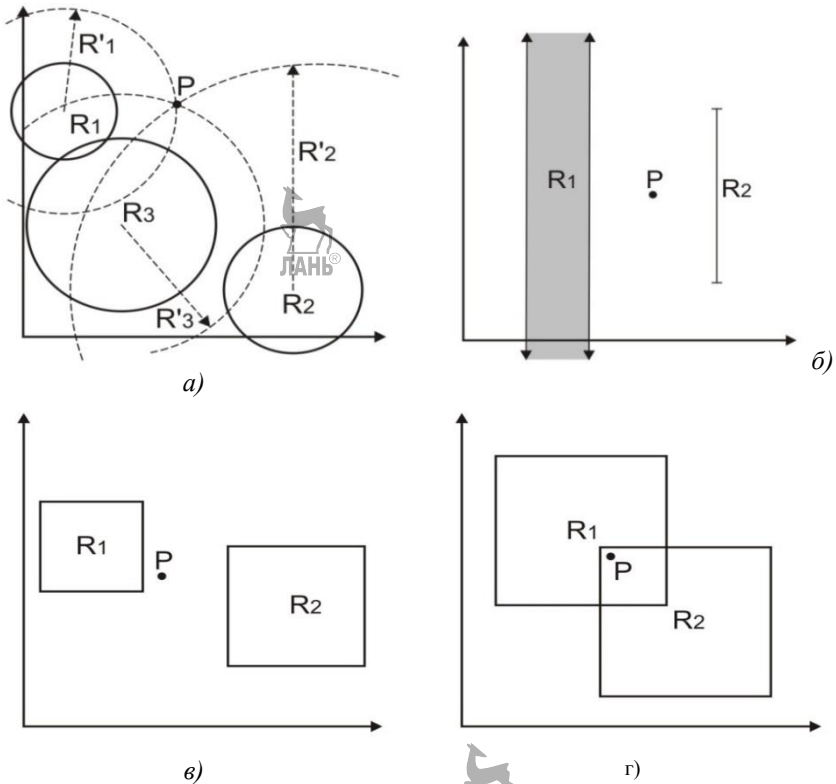


Рис. 3.29. Критерии выбора узла для вставки: а – минимальное число пересекающихся областей; б – минимальное сокращение используемых измерений; в – минимальное увеличение MBR; г – минимальное расстояние до центра MBR

2. *Минимальное уменьшение в размерности.* При вставке объекта часть его измерений может быть не согласована с тем сокращением, которое принято в вершине.

Так, рассмотрим ситуацию, представленную на рис. 3.29(б). В вершине $R2$ используются оба измерения, причем одно из них (первое) является неактивным. Все дочерние объекты этой вершины имеют строго заданную первую координату. Это значительно ускоряет процедуры поиска. При этом если объект P попытаться вставить в вершину $R2$, то

необходимо будет сократить число используемых измерений до 1 (объект P имеет совершенно другую координату по этому измерению). Таких ситуаций стоит избегать. Необходимо выбирать те вершины, которые при вставке как можно меньше изменяют число используемых измерений. Поэтому в представленном примере следует выбрать для вставки вершину $R1$.

3. *Минимальное увеличение MBR*. Если несколько вершин по предыдущим критериям оказались равноправны, то необходимо выбрать ту из них, которая при вставке увеличит свою MBR как можно меньше. На рис. 3.29(в) представлен такой случай. При этом вершина $R1$ является более предпочтительной.

4. *Минимальное расстояние от центра ограничивающей области до вставляемого объекта*. На рис. 3.29(г) представлен случай, в котором ни один из перечисленных критериев не может однозначно показать, какой из узлов более предпочтителен. В этом случае необходимо выбрать узел $R1$, т. к. объект P находится ближе к его центру, чем к центру $R2$.

Расщепление узла при переполнении. Переполнение узла – это еще один важный момент, по которому TV -дерево отличается от своих предшественников. Причем переполняться могут не только листовые вершины TV -дерева, но и внутренние вершины. При этом применяется разная стратегия обработки переполнения для этих узлов.

- Переполнения листового узла начинают обрабатываться так же, как в R^* -дереве. Выбирается некоторое число наиболее удаленных от центра объектов узла, они исключаются из данной вершины и повторно вставляются в дерево. Как было описано ранее, это позволяет значительно повысить производительность структуры путем более эффективной группировки размещения объектов. Если данная операция не помогла и в процессе повторных вставок вновь возникает переполнение некоторой вершины, то вызывается расщепление переполненной вершины.

- Переполнение внутренней вершины TV -дерева всегда приводит к непосредственному расщеплению узла.

Самым важным из рассмотренных шагов обработки переполнения является процедура расщепления. Целью расщепления является перераспределить объекты переполненной вершины на две группы так, чтобы дальнейшая обработка этих групп была как можно более эффективной.

Существует несколько вариантов процедуры расщепления. Самый простой – метод кластеризации, который группирует объекты по степени их схожести. Для этого из всего набора выбираются два самых отличающихся объекта – $R1$ и $R2$, и они становятся центрами новых групп. Далее каждый из оставшихся объектов добавляется к одной из групп, к центру которой он оказался ближе. Данный алгоритм уже приводился при описании R -дерева.

Более эффективным может оказаться алгоритм, целью которого является уменьшение области, которую покрывают *MBR*. Он дает несколько иное распределение и позволяет увеличить число используемых измерений в узле.

Расширение функции сокращения. Наиболее важным моментом работы всего алгоритма вставки нового объекта является корректировка *MBR*. Особенно это становится заметным после расщепления узла. При расчете новых *MBR* может оказаться, что ряд измерений имеют одинаковые значения для всех объектов узла (см. рис. 3.30).

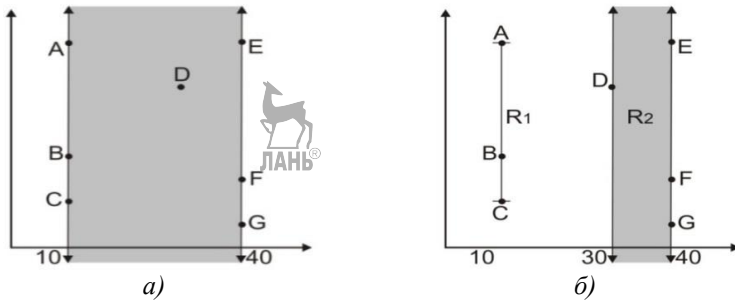


Рис. 3.30. Расширение функции сокращения: а – до расщепления; б – после расщепления

В представленном примере функция сокращения для данного узла расширяется и добавляет новые используемые измерения. Так, на рис. 3.30(а) до расщепления использовалось всего одно измерение ($m = 1$), которое и было активным. После расщепления новый узел *R1* может расширить функцию сокращения до двух используемых измерений ($m = 2$), при этом активных измерений останется, как и раньше, одно. Расширение функции сокращения и добавление новых используемых измерений является особенностью *TV*-дерева и многократно увеличивает производительность структуры в целом.

Рассмотрим процедуру корректировки *MBR* и функции сокращения для случая, когда функция сокращения просто отсекает последние измерения. Пример реализации показан с помощью псевдокода в листинге 3.26.



Листинг 3.26

```
//=====
// Пересчет новой функции сокращения и MBR узла
// Параметры:
// V – узел дерева, для которого производится пересчет
```

```

//=====
КОРРЕКТИРОВКА_УЗЛА(V)
[1] // Вычисление количества неактивных измерений
    n = общее количество измерений
    Выполнить цикл по i от 1 до n
        K = i-я координата первого потомка вершины V
        Выполнить цикл по всем объектам V', дочерних V
            Если K ≠ i-й координате V', то
                i--
        Прервать оба цикла
    m = i // количество неактивных измерений
[2] // Вычисление MBR по активным измерениям
    Выполнить цикл по i от (m+1) до min(m+α, n)
    Kmin = минимальная i-я координата потомков V
    Kmax = максимальная i-я координата потомков V
    i-я граница V.MBR = [Kmin, Kmax]
[3] // Корректировка используемых измерений
    V.m = min(m+α, n)
Конец КОРРЕКТИРОВКА_УЗЛА

```

Алгоритм удаления объекта из дерева

Процедура удаления объекта из дерева является самой простой. Объект по описанным правилам ищется в дереве и удаляется из него. Однако необходимо помнить несколько важных замечаний.

1. После удаления объекта из вершины дерева эта вершина может оказаться редко заполненной или пустой. Такие вершины удаляются из дерева, а оставшиеся в них объекты заново вставляются в дерево.

2. Удаление вершин может рекурсивно распространяться вверх, если после удаления пустой листовой вершины оказывается редко заполненный ее родительский узел.

3. После удаления объекта из вершины дерева необходимо корректировать *MBR* этой вершины. При этом особенно важно не забывать проверять возможность расширения функции сокращения, рассмотренной ранее. Удаление объектов из дерева является еще одним источником ситуаций, при которых могут появляться условия увеличения числа используемых измерений, а это и есть отличительная черта *TV*-дерева от других структур.

3.3. Структуры с разделением объектов

Одной из самых серьезных проблем структур, рассмотренных в этой главе ранее, является пересечение областей узлов. Основа этих структур – иерархическая группировка близко расположенных объектов. При этом

минимальные ограничивающие прямоугольники различных узлов могут пересекаться. Если алгоритм формирования дерева и последовательность вставок были удачно подобраны, то площадь пересечения в большинстве случаев является незначительной. Однако при неблагоприятном стечении обстоятельств пересечение *MBR* могут значительно повлиять на производительность структуры в целом. Это связано с тем, что для поиска в тех областях, где находится пересечение *MBR* узлов, появляется необходимость проверки не одного, а нескольких поддеревьев.

Исследователи постоянно работают над совершенствованием методов доступа. Одним из таких совершенствований является новый подход к формированию структуры, не допускающей пересечения узлов дерева. Однако, как было замечено ранее, существуют случаи, при которых нельзя избежать пересечения *MBR* двух объектов. В данной главе предлагаются методы, основанные на разделении объектов на части. Если объект попадает на границу двух узлов, то он разбивается на части так, чтобы каждая из них размещалась в отдельном узле дерева и при этом все узлы продолжали иметь нулевое пересечение друг с другом.

Отсутствие пересечений в дереве приводит к уменьшению числа проверок при поиске. Так, при поиске по точному совпадению в каждом узле дерева можно однозначно выбрать потомка, соответствующего запросу. При этом проверка нескольких веток дерева становится излишней.

В настоящее время разработаны модификации большинства популярных структур, в которых реализован принцип деления объектов. Так, для *K-D*-дерева был разработан вариант расширенного *K-D*-дерева [65] и *BSP*-дерева [35]. Для *R*-дерева модификация с делением объектов называется *R+*-дерево [84, 83]. Более подробно эти структуры будут рассмотрены далее.

Структуры с делением объектов позволяют избавиться от главного недостатка деревьев, рассмотренных ранее в этой главе. Однако и эти структуры имеют свои недостатки. Главные из них связаны с добавлением и удалением объектов в индекс.

Первый недостаток связан с делением объектов при вставке. Если новый объект попадает на границу нескольких узлов, то его приходится разбивать на части. Причем число частей зависит от того, на границе скольких узлов расположен объект. Поэтому в общем случае в процессе вставки одного объекта в дерево может появиться более одного узла. Для того чтобы в процессе обработки все эти узлы соотносились с одним объектом, необходимо добавлять в каждый из них специальный идентификатор, характеризующий, какому именно объекту принадлежит данная часть.

Такое дробление и размещение объекта в нескольких узлах одновременно само по себе не сильно влияет на производительность большинства запросов поиска. Оно несет только сложности в

программировании. Однако все-таки есть косвенный негативный эффект в подобном дроблении. Полученная структура имеет большее число узлов, чем число проиндексированных объектов. А это уже влияет на размер дерева, его высоту и скорость обработки.

Вторым недостатком является обработка процесса переполнения. При вставке объектов в листовую узел (или отдельных частей объектов, в случае их деления) рано или поздно произойдет переполнение этого узла. При этом появится необходимость его расщепления. Как было показано в параграфах, посвященных *R*-деревьям, достаточно редко возникает ситуация, при которой можно разделить все объекты на две непересекающиеся группы. А так как в дереве не может быть пересекающихся узлов, то приходится еще раз дробить объекты, находящиеся в переполненном узле.

При распространении расщепления вверх может произойти переполнение внутренних узлов. При этом и они не всегда делятся на непересекающиеся группы. Поэтому переполнение внутренних узлов приходится обрабатывать расщеплением дочерних узлов вниз, которые не всегда заполнены полностью. Такой процесс может приводить к полупустым узлам дерева.

Третья проблема, связанная с делением объектов, появилась после многократного экспериментирования и исследования полученных результатов. Оказалось, что многократные деления объектов при вставке и расщеплении узлов могут в некоторых случаях приводить к значительной фрагментации объектов, разбиению их на маленькие кусочки, которые дают дополнительную нагрузку на операции обработки. Данная проблема очень хорошо описана в работе О. Гюнтера и Г. Нольтемеера [39].

Несмотря на описанные недостатки, структуры данного класса очень широко используются на практике. Существует ряд приемов, позволяющих минимизировать негативные влияния описанных недостатков. Некоторые из них будут рассмотрены далее.

3.3.1. BSP-дерево

Рассмотрение структур с разделением объектов начнем с *BSP*-дерева. Как уже было упомянуто, *BSP*-дерево является одной из разновидностей *K-D*-дерева. Данная структура появилась очень давно, и в настоящее время существует большое число модификаций и улучшений базовых алгоритмов в зависимости от конкретных применений.

Впервые *BSP*-дерево (*Binary Space Partitioning*) было описано Г. Фачем, З. Кедемом и В. Найлором в 1980 году [35]. Но уже через несколько лет появилось множество его модификаций [34]. Со временем структура получила большую популярность для двух- и трехмерных пространств. В настоящее время существует как программная реализация

этих алгоритмов, так и аппаратная (например, в видеокarte, где с помощью *BSP*-дерева сортируются объекты сцен).

Основной областью, в которой позиции *BSP*-дерева со временем не только не уменьшились, но и укрепились, можно считать компьютерную графику. Описание этой структуры часто можно встретить в учебных пособиях по вычислительной геометрии и компьютерной графике. *BSP*-дерево описывается как один из самых эффективных алгоритмов представления сцен и отсеечения ненужных объектов при движении камеры в пространстве. Это связано с особенностью структуры дерева, в котором, в отличие от большинства рассмотренных структур, разрешается делить пространство на две части произвольной плоскостью.

Рассмотрим основные принципы построения и обработки этой структуры.

Общие принципы построения BSP-дерева

Во второй главе этого издания было подробно описано *K-D*-дерево. Основная идея оригинального *K-D*-дерева и всех структур, основанных на нем, заключается в рекурсивном делении пространства на две части. В качестве плоскости деления может выступать любая плоскость, перпендикулярная одной из осей пространства. Различные варианты *K-D*-дерева отличаются принципом выбора таких плоскостей. Так, в оригинальном дереве оси, которым плоскость деления перпендикулярна, должны чередоваться, в других реализациях – номер оси может сохраняться в самом узле дерева.

Как было отмечено, деление плоскостью объектов на две группы возможно только в том случае, если рассматриваемые объекты являются *n*-мерными точками. Для пространственных объектов такой подход не годится, так как на практике не всегда возможно выбрать такую ось, которая бы разделила объекты на две непересекающиеся группы. Поэтому для модификации под пространственные объекты в структуру был внесен ряд изменений.

Первое изменение, которое добавили в оригинальное *K-D*-дерево – это возможность пересечения плоскостью деления объектов. Каждый узел по-прежнему разбивает все пространство на две части с помощью некоторой гиперплоскости. Однако эта плоскость может пересекать объекты пространства, разбивая их на две части. Каждая из частей первоначального объекта помещается в соответствующее поддерево и рассматривается в дальнейшем обособленно. Структура данных, которая отличается от оригинального *K-D*-дерева только этим принципом, получила название расширенного *K-D*-дерева (*Extended K-D-tree*) [65].

Второе изменение коснулось выбора гиперплоскостей деления. В оригинальном *K-D*-дерево гиперплоскость выбиралась по определенному алгоритму, и она обязательно должна была быть

перпендикулярной одной из осей пространства. В *BSP*-дереве было решено отказаться от этого принципа и выбирать гиперплоскость произвольно, в зависимости от тех нужд, которые предъявляет прикладная область, или в зависимости от конкретного набора данных. Это позволяет более гибко приспосабливаться под данные и строить более эффективные структуры. Однако такой подход имеет и недостатки. Так как гиперплоскость деления пространства вычислить автоматически нельзя, то появляется необходимость в каждом узле дерева хранить ее координаты, по которым в любой момент можно ее рассчитать. Это влечет некоторые накладные расходы, которые иногда могут иметь такой же размер, как и сами данные.

Рассмотрим пример *BSP*-дерева на двумерном пространстве (рис. 3.31). Слева на рисунке изображен набор объектов, а справа – построенное *BSP*-дерево.

Процесс построения дерева начинается с корня. Все пространство разбивается на две части некоторой гиперплоскостью. В случае двумерных данных гиперплоскость деления будет иметь вид прямой. На рис. 3.31 в качестве такой прямой выбрана прямая l_1 . Она помещается в корень дерева. Все объекты, которые расположены с одной стороны этой прямой, помещаются в одно поддереву, а с другой – в другое.

Далее процесс деления продолжается рекурсивно в каждой из полученных подобластей до тех пор, пока полученная после очередного деления часть пространства не станет содержать нужное число объектов. В нашем примере таковым ограничением являются два объекта. Для поддержания этого свойства левая часть пространства разбивается еще одной прямой l_2 , а затем его левая часть – прямой l_4 . В результате мы получаем левое поддерево, показанное на рис. 3.31.

Если в результате деления пространства некоторой прямой произойдет пересечение объекта (например, как в случае с прямой l_3), то этот объект разбивается на две части, каждая из которых попадает в соответствующее поддерево.

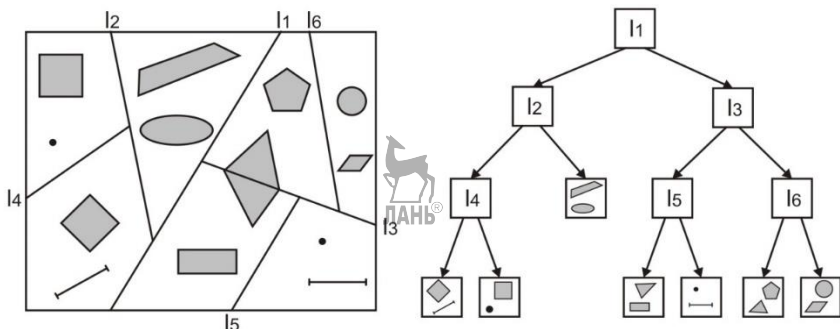


Рис. 3.31. Пример двумерного *BSP*-дерева

Как видно из рисунка, результирующее дерево может получиться несбалансированным. Балансировка и высота дерева зависят от эффективности алгоритма выбора плоскости деления пространства и последовательности операций вставок. В большинстве реализаций динамическая балансировка не предусматривается, так как при этом необходимо перестроить большую часть индекса, что не всегда приемлемо.

Алгоритм поиска

Рассмотрим алгоритм поиска объектов в *BSP*-дереве. Этот алгоритм полностью определяется структурой дерева и несколько похож на поиск в *K-D*-дереве. Запрос по области имеет практически такую же реализацию, поэтому он не будет рассматриваться здесь. Однако для запроса по точному совпадению есть существенные отличия, связанные с тем, что объекты в *BSP*-дереве могут делиться и размещаться в нескольких поддеревьях. Процедура поиска с помощью псевдокода представлена в листинге 3.27.

Листинг 3.27

```
//=====
// Поиск объекта в BSP-дереве
// Параметры:
//   O - объект, который необходимо найти
//   V - текущая просматриваемая вершина (первоначально
//       передается корень дерева)
//=====
ПОИСК(O, V)
[1] // Проверка листовой вершины
    Если V - листовая вершина, то
        Для всех объектов O' вершины V проверяем
            Если O'=O, то
                Выйти из процедуры и вернуть ИСТИНУ
            Выйти из процедуры и вернуть ЛОЖЬ
[2] // Проверка внутреннего узла
    Если V - внутренняя вершина, то
        l - плоскость, соответствующая вершине V
        Если O находится слева от l, то
            Вернуть результат ПОИСК(O, Left(V))
        Иначе Если O находится справа от l, то
            Вернуть результат ПОИСК(O, Right(V))
        Иначе
            Разбиваем O на две части O' и O'' прямой l
            Res=ПОИСК(O', Left(V)) И ПОИСК(O'', Right(V))
            Вернуть в качестве результата Res
Конец ВСТАВКА
```

BSP-дерево, в отличие от *K-D*-дерева, содержит вершины двух типов. Внутренние узлы предназначены только для деления пространства на части и не содержат индексируемых объектов. Листовые же узлы напротив предназначены для хранения объектов. Учитывая эту особенность, нужно процедуру поиска составить из двух частей. Первая часть предназначена для листовых узлов, вторая – для внутренних.

Поиск объекта начинается в корне. Каждая внутренняя вершина просматривается в дереве и определяется местоположением объекта поиска относительно нее. Если объект целиком расположен слева или справа от плоскости деления пространства, то происходит переход по дереву к соответствующему потомку. Однако если объект пересекается линией, то процедура существенно усложняется. Объект поиска разбивается на две части, и каждая из этих частей ищется в разных потомках. Результат поиска считается успешным только в том случае, если обе части были найдены в соответствующих поддеревьях (именно поэтому в листинге результат поиска в обоих поддеревьях совмещен с помощью операции «логического И»). Если хотя бы одна из частей найдена не будет, то это будет означать, что объект находится в дереве не полностью и результат процедуры является неудачным.

Процесс поиска заканчивается в листовых узлах. Листовая вершина содержит конечный набор объектов, который проверяется на соответствие объекту поиска (или той его части, которая была получена при перемещении до данного узла).

Описанный алгоритм с небольшими изменениями может легко использоваться не только для поиска по точному совпадению, но и для поиска по области или пространственному поиску ближайшего соседа некоторой точки.

В заключение необходимо пояснить один момент: как в процедуре определяется положение объекта относительно плоскости деления. Если плоскость деления всегда перпендикулярна некоторой оси пространства (как, например, в *K-D*-дереве), то положение объекта (слева или справа от плоскости) определяется по его координатам. Но в *BSP*-дереве плоскость деления может быть произвольной. Поэтому встает вопрос, что считать левой, а что считать правой стороной для этой плоскости? Ответ на данный вопрос зависит от конкретной реализации. В большинстве случаев для самого дерева это является непринципиальным. Главное, чтобы определение левой и правой сторон не отличалось при выполнении вставки объектов и при их последующем поиске. Чаще всего определение сторон происходит по нормали плоскости (по тому, как она определена в вершине). Алгоритмы геометрического определения положения объектов выходят за рамки данного издания. Их можно посмотреть в любом пособии по компьютерной графике.

Алгоритм вставки объекта

Рассмотрим еще один важный алгоритм – вставка объекта в существующее дерево. Данный алгоритм в общих чертах похож на рассмотренный алгоритм поиска по точному совпадению. Псевдокод процедуры поиска представлен в листинге 3.28.

Листинг 3.28

```
//=====
// Вставка объекта в BSP-дерево
// Параметры:
//   О – объект, который необходимо вставить в дерево
//   V – текущая просматриваемая вершина (первоначально
//       передается корень дерева)
//=====
ВСТАВКА(О, V)
    [1] // Проверка, достигли ли листовой вершины
        Если V – листовая вершина, то
            Вставить объект О в вершину V
        Если количество объектов в V меньше предела, то
            Выйти из процедуры
        Иначе
            РАЗБИТЬ_УЗЕЛ(V)
    [2] // Если текущий узел V – внутренний узел
        Если V – внутренняя вершина, то
            l – плоскость, соответствующая вершине V
            Если О находится слева от l, то
                ВСТАВКА(О, Left(V))
            Иначе Если О находится справа от l, то
                ВСТАВКА(О, Right(V))
        Иначе
            Разбиваем О на две части О' и О'' прямой l
            ВСТАВКА(О', Left(V))
            ВСТАВКА(О'', Right(V))
Конец ВСТАВКА
```

В процедуру вставки объекта передаются вставляемый объект и вершина, которая проверяется на данном шаге на возможность размещения объекта в ней. В процедуре предполагается, что к моменту вызова процедуры дерево существует (оно состоит хотя бы из одной вершины – корня). Если это не так и есть вероятность вызова процедуры для пустого дерева, необходимо добавить соответствующую проверку в нее (при отсутствии дерева оно создается, и в него помещается добавляемый объект).

Процедура добавления объекта в дерево, как и процедура поиска, состоит из двух частей. Каждая из этих частей обрабатывает один тип

узлов. В представленном листинге первый шаг процедуры предназначен для листовых узлов, второй – для внутренних вершин дерева.

Если процесс вставки дошел до листового узла, объект размещается в нем. При этом следует помнить, что каждый узел дерева имеет вместимость – максимальное число объектов, которые могут быть размещены в нем. Данный параметр является глобальным параметром дерева и выбирается на этапе его проектирования и реализации алгоритмов. Если вставка объекта в листовую вершину привела к ее переполнению, необходимо поделить узел. Алгоритм деления будет подробнее рассмотрен далее при описании процедуры начального построения дерева. Стоит только заметить, что процедура деления затрагивает только данный узел и не требует какой-либо реорганизации дерева или его части. Это очень важная особенность, положительно влияющая на производительность и позволяющая параллельно выполнять несколько запросов поиска и вставки в дерево.

Второй шаг алгоритма обрабатывает внутренние вершины. Этот шаг очень похож на аналогичные действия процедуры поиска. Процедура проверяет положение объекта относительно плоскости деления узла, и если объект находится с какой-то из ее сторон, то просто переходит к соответствующему поддереву. Если же объект пересекается плоскостью, он разбивается на две части, каждая из которых независимо друг от друга вставляется в дерево.

Алгоритм удаления объекта

Удаление объекта из *BSP*-дерева похоже на процедуру поиска и имеет такую же самую структуру. Сначала происходит определение положения объекта или всех его частей в дереве с помощью процедуры, представленной в листинге 3.27. После нахождения каждой из его частей она удаляется из листовой вершины. Единственный нюанс, который стоит помнить при выполнении процедуры удаления, это то, что после удаления объекта листовой узел может оказаться пустым. В этом случае он просто удаляется из дерева, а также удаляется его предок, так как плоскость деления пространства, соответствующая предку пустого узла, является ненужной в дереве.

Алгоритм начального построения дерева

В большинстве приложений большая часть пространственных данных известна заранее. К таким приложениям можно отнести, прежде всего, компьютерные игры, в которых генерация игровых сцен происходит заранее, на этапе проектирования. При этом в процессе исполнения игры сцена остается статичной или претерпевает незначительные изменения.

Используя этот факт, можно заранее построить *BSP*-дерево оптимальной, сбалансированной структуры (или близкое к

сбалансированному), что даст многократное повышение производительности. Для этого используются алгоритмы начального построения дерева.

Рассмотрим построение *BSP*-дерева на примере. Допустим, у нас имеется набор двумерных отрезков, показанных на рис. 3.32. Необходимо для этого набора построить дерево.

На первом шаге алгоритма происходит выбор плоскости (в двумерном случае это прямая), которая разобьет все пространство на две части и разделит индексируемые объекты на группы. Выбор оптимальной плоскости деления является достаточно сложной задачей. С одной стороны, необходимо минимизировать количество делений объектов выбранной плоскостью, поскольку это увеличивает число узлов в дереве, и соответственно объектов, которые затем будут участвовать при использовании его на практике. Во многих источниках именно минимизация количества делений и снижение количества узлов ставится во главу угла при создании оптимального *BSP*-дерева.

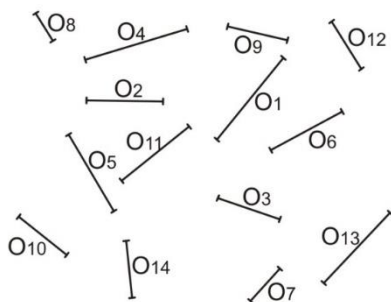


Рис. 3.32. Набор двумерных отрезков

В то же время возникает необходимость построения сбалансированного дерева, под которым в данном контексте понимается его первоначальное значение – по возможности равное число узлов по обе стороны от корневого узла. Кроме того, это же правило справедливо и для узлов-потомков, так как они также являются корневыми узлами для своих потомков.

Стоит заметить, что далеко не во всех приложениях требуется идеальная балансировка дерева. Так, если *BSP*-дерево используется только для рендеринга полных сцен без каких-либо модификаций самого алгоритма, то балансировка будет не сильно влиять на производительность. При отрисовке сцены важны именно пространственные характеристики объектов, их взаимное пересечение и положение. При этом в большинстве случаев все равно приходится обходить все дерево целиком (в нужном порядке). Поэтому, является ли

оно сбалансированным или нет, становится непринципиальным, главное, чтобы оно учитывало пространственное расположение отдельных объектов.

Существует достаточно много рекомендаций и общих принципов реализации процедуры деления. Рассмотрим один из самых популярных вариантов – перебор всех граней индексируемых объектов.

Одним из самых простых и в то же время самых эффективных с точки зрения полученного дерева является перебор всех возможных вариантов деления пространства. Для этого рассматривают все возможные варианты деления объектов на две группы и плоскости, которыми можно достичь такой группировки. Каждый из полученных вариантов анализируется и сравнивается с другими по числу пересечений объектов и равномерности деления (числу объектов в каждой группе).

Совершенно очевидно, что в любом пространстве можно провести бесконечное число плоскостей. Поэтому полный перебор теоретически становится невозможным. Однако можно значительно уменьшить число рассматриваемых вариантов и сделать задачу решаемой, если использовать в переборе только те плоскости, которые содержат в себе какую-либо из граней индексируемых объектов. Так, в рассматриваемом примере необходимо перебрать все отрезки разделяемой части пространства, продлить их до прямой и проверить качество деления пространства этой прямой. Именно этот алгоритм чаще всего используется в практических реализациях.

Согласно выбранному принципу, лучшим кандидатом для первоначального деления является отрезок O_1 . Прямая, проходящая через него, разделит все отрезки на две равные группы, и при этом удастся избежать пересечения. Поэтому именно эту прямую следует выбрать в качестве плоскости деления пространства для корневой вершины.

Если среди индексируемых объектов присутствуют точки или отрезки (как в рассматриваемом случае), то может возникнуть ситуация, при которой объект находится на самой плоскости деления пространства (не слева или справа, а именно совпадает с ней). Подобные ситуации можно обрабатывать по-разному. В ряде приложений рекомендуют подобные объекты относить всегда к левому поддереву, в других – помещать в самую внутреннюю вершину. В большинстве случаев первый подход является более предпочтительным, так как вносит ясность и упрощает процедуры обработки. Однако если дерево строится только для индексирования отрезков, то второй подход может оказаться более производительным. Так как в рассматриваемом примере пространство содержит только отрезки, воспользуемся им и поместим отрезок O_1 в корневую вершину (рис. 3.33).

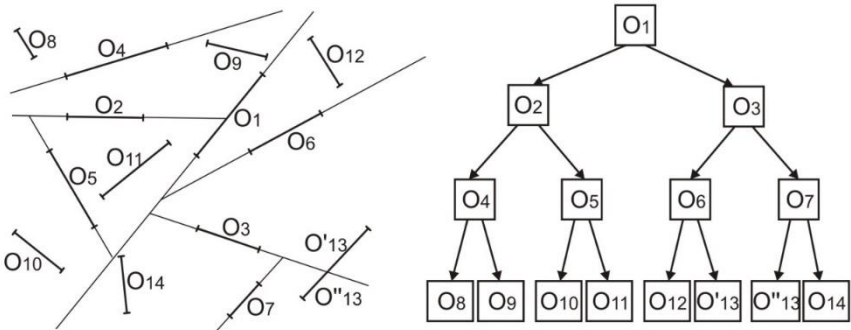


Рис. 3.33. BSP-дерево отрезков

Далее по аналогичному алгоритму рассматривается каждое из подпространств отдельно и на основе их деления строится левое и правое поддереву. Особый интерес может вызвать правое поддерево. При делении пространства в него попали 6 отрезков – O_3 , O_6 , O_7 , O_{12} , O_{13} , O_{14} . Однако ни один из них не обеспечивает дальнейшее деление пространства, при котором не нужно было бы делить отрезки на части и при этом дерево оказалось бы сбалансированным. Некоторые из них требуют большого числа расщеплений, другие – создают группы разного размера. Поэтому при выборе новой вершины идут на компромисс – выбирают тот из отрезков, который позволит обойтись как можно меньшим числом делений, но при этом получившиеся при делении группы будут приблизительно равны по числу объектов в них. Таковым стал отрезок O_3 .

Процесс деления пространства продолжается рекурсивно до тех пор, пока в каждой из его частей не останется по одному отрезку. Результирующее дерево показано на рис. 3.33.

Выше был описан один из самых популярных алгоритмов построения дерева. Однако он является не единственным. В зависимости от конкретного применения и назначения разработано большое число модификаций этого метода.

3.3.2. R+-дерево

Рассмотрим еще одну структуру с делением объектов – R^+ -дерево. Данную структуру предложили Т. Селис, Н. Русополос и К. Фалоутсос в 1987 году [83]. Она является модификацией обычного R -дерева, на которое распространили идеи деления объектов. Этот прием позволил избавиться от целого ряда недостатков оригинальной структуры, но, в свою очередь, привел к некоторым усложнениям и появлению новых нюансов, рассмотренных далее. Как заявляют создатели, R^+ -дерево позволяет уменьшить число обращений к жесткому диску при

выполнении операций поиска до 50%. Однако появление дробления объектов усложняет алгоритмы обработки.

Общие принципы построения R^+ -дерева

R^+ -дерево – это идеально сбалансированное по высоте, сильноветвящееся дерево, которое в общих чертах имеет приблизительно такую же структуру, как оригинальное R -дерево. Оно состоит из узлов двух типов. Внутренние узлы предназначены для деления пространства на прямоугольные области, которые, в свою очередь, делятся на подобласти и т. д. Листовые вершины содержат конечные наборы пространственных объектов. Точно так же, как и в R -дереве, каждый узел характеризуется минимальным ограничивающим прямоугольником MBR , который описывает все его дочерние объекты, и имеет ограничение на максимальное число элементов в нем M .

В параграфе, посвященном оригинальному R -дереву, было рассмотрено несколько технологий построения этого дерева. Каждая из них ориентировалась на свой базовый набор критериев, главными из которых являлись либо уменьшение площади пространства, занимаемого отдельными узлами дерева, либо уменьшение области взаимного пересечения MBR узлов. Оба эти фактора очень сильно влияют на производительность структуры в целом.

Ориентация на уменьшение площади пространства узлов приводит к тому, что MBR этих узлов охватывают меньше пустого места, не содержащего объектов. Это позволяет уменьшить время выполнения неудачных запросов на точное совпадение (запросов отсутствующих в индексе объектов). Также эта технология позволяет ускорить пространственные запросы, такие, как запросы по области и запросы ближайшего соседа (в этом случае сокращение просматриваемых ветвей дерева при выполнении запроса является более эффективным).

С другой стороны, уменьшение области взаимного пересечения MBR отдельных узлов также приводит к увеличению скорости работы структуры. Причем данный критерий позволяет увеличить скорость работы абсолютно всех операций. Это достигается вследствие того, что пересекающиеся области в дереве приходится обрабатывать несколько раз (по одному разу для каждой ветви дерева, участвующей в пересечении).

Таким образом получается, что эти два критерия имеют особое значение при построении дерева. Однако создать набор правил и ограничений, учитывающих одновременно оба критерия для R -дерева, достаточно сложно. Это связано с тем, что очень часто ориентирование на каждый из этих критериев приводит к совершенно противоположным результатам, что и было показано в параграфах, посвященных R -дереву и его разновидностям.

В R^+ -дереве MBR различных узлов запрещено пересекаться. Это достигается благодаря тому, что объекты, которые должны вызвать такое пересечение, разбиваются на части, каждая из которых индексируется отдельно. Такое изменение позволяет учесть оба описанных критерия при построении дерева и создать структуру, сбалансированную по обоим показателям. С одной стороны, R^+ -дерево имеет нулевое пересечение внутренних узлов, а с другой – в процедуре построения и деления узлов можно выстроить набор правил, ориентирующихся на уменьшение площади этих узлов. На рис. 3.34 представлен пример построенного R^+ -дерева для набора объектов, используемых ранее для построения R -дерева.

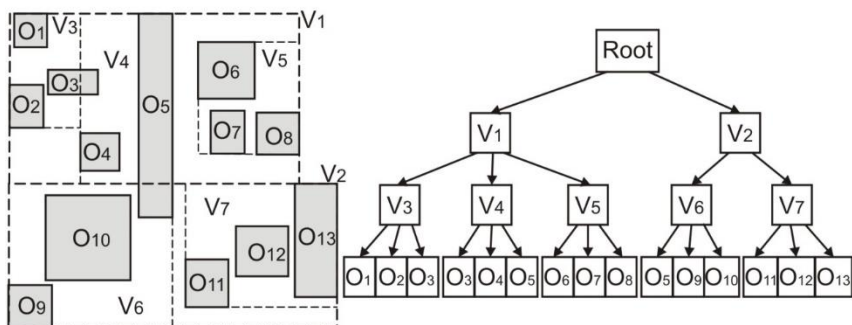


Рис. 3.34. R^+ -дерево

Рассмотрим основные свойства и ограничения, которые вводятся для R^+ -деревьев.

1. Для каждого внутреннего узла дерева вычисляется MBR . Дочерний объект может находиться в данном узле тогда и только тогда, когда его MBR полностью входит в MBR предка.

2. Исключением из свойства 1 являются листовые вершины дерева. MBR конечных объектов, находящихся в них, могут не полностью принадлежать прямоугольнику узла (но при этом обязательно иметь с ним непустое пересечение). Если часть объекта выходит за границы MBR листового узла, то это означает, что его оставшая часть принадлежит другому узлу дерева. Не существует ни одной части объекта, которая бы не принадлежала одному и только одному листовому узлу дерева.

3. Для любых двух узлов дерева (в том числе и листовых) пересечение их MBR равно нулю.

4. Корневая вершина должна содержать как минимум двух потомков. Исключением может являться случай, при котором корневая вершина является листом дерева (дерево состоит всего из одной вершины).

5. Все листья дерева находятся на одном уровне, т. е. дерево является идеально сбалансированным.

Выполнение всех этих свойств поддерживается особыми процедурами вставки, деления и удаления объектов, рассмотренными в данном параграфе далее.

Стоит также вернуться к показателю использования памяти структурой. Этот показатель зависит от максимального и минимального числа потомков у каждого узла дерева. Как было указано, каждый узел R^+ -дерева, как и узел оригинального R -дерева, ограничен M элементами в нем. Это означает, что листовая вершина не может содержать более M объектов, а внутренние вершины дерева содержат не более M потомков. Таким образом получается, что M является степенью ветвления дерева.

Однако, в отличие от R -деревьев, R^+ -деревья из-за объективных причин не могут поддерживать минимальный предел наполнения узла $m = M/2$. Это связано с тем, что в процессе расщепления узлов, деление может распространяться как вверх по дереву, так и вниз (подробнее этот процесс будет рассмотрен далее). Поэтому в дереве могут появляться вершины с неконтролируемой нижней границей наполнения. Конечно, такое поведение должно негативно сказаться на общем проценте использования памяти структурой. Однако практические эксперименты показали, что на реальных распределениях коэффициент использования памяти падает не более чем на 10% [83].

Алгоритм поиска объекта в R^+ -дереве

Алгоритм поиска объектов в R^+ -дереве аналогичен алгоритму поиска по R -дереву. Однако в R -дереве узлы могли пересекаться и индексировать одну и ту же часть пространства. Поэтому при поиске появлялась необходимость проверки всех дочерних вершин некоторого внутреннего узла. В R^+ -дереве MBR узлов не пересекаются, и этот факт позволяет однозначно выбрать всего одну ветвь дерева для поиска, отбросив все остальные. Выбор происходит по принципу ненулевого пересечения MBR узла и объекта поиска. Такое пересечение гарантирует, что если объект поиска присутствует в дереве, то либо он целиком, либо какая-либо из его частей находятся в выбранной ветке. Причем, в отличие от BSP -деревьев, R^+ -деревья в листьях всегда содержат объект целиком, даже если при вставке он был разбит на части (в некоторых реализациях в листьях содержится только реальный MBR объекта и его идентификатор или ссылка на него, а сам объект находится в другом файле). Именно поэтому можно искать в любой из ветвей дерева, которая имеет ненулевое пересечение с объектом, а не просматривать всех потомков, как это было в R -деревьях или BSP -деревьях.

Еще одним интересным замечанием является тот факт, что процедура поиска объекта является линейной и может быть реализована не только рекурсивно, но и с помощью обычных циклов и операторов

перехода. Один из вариантов реализации поиска объекта по точному совпадению без использования рекурсии представлен в листинге 3.29.

Листинг 3.29

```
//=====
// Поиск объекта О в дереве по точному совпадению
// Параметры:
//   О - объект поиска
//=====
ПОИСК_ОБЪЕКТА(О)
[1] // Начальная инициализация переменных
    V = корень дерева
[2] // Проверка внутренних вершин
    Если V является внутренней вершиной дерева, то
        Для всех потомков V' вершины V проверить
            Если  $MBR(V') \cap MBR(O) \neq \emptyset$ , то
                V = V'
                Перейти к шагу 2
            Вернуть ЛОЖЬ
[3] // Проверка листовых вершин дерева
    Если V является листом, то
        Проверить все объекты O' узла V
        Если O' = O, то
            Вернуть ИСТИНУ
        Вернуть ЛОЖЬ
Конец ПОИСК_ОБЪЕКТА
```

Сложность этого алгоритма в общем случае является логарифмической по числу объектов в дереве. Для нахождения объекта или определения факта его отсутствия в худшем случае необходимо сделать такое число шагов, какова высота дерева. Это значительно превосходит аналогичный показатель для R -дерева, в котором при поиске необходимо проверять несколько ветвей.

Процедура поиска по области и поиска ближайшего соседа в R -дерева также выполняются быстрее. Однако с алгоритмической точки зрения они практически ничем не отличаются от таких же самых процедур, рассмотренных ранее для R -деревьев и их разновидностей. Единственное, что следует учитывать, так это то, что при поиске по области один и тот же объект может попасть в список результатов несколько раз. Это произойдет, если при индексировании объект оказался на границе нескольких узлов дерева, а при поиске именно этот участок вошел в запрашиваемую область. Если для приложения критично однократное нахождение каждого объекта в списке результатов, то при формировании этого списка необходимо следить за дублированием объектов в нем.

Алгоритм добавления нового объекта в R^+ -дерево

Алгоритм вставки объектов является одним из центральных алгоритмов, который отличает R^+ -дерево от R -дерева. Именно алгоритм вставки позволяет поддерживать все те свойства, которые были описаны.

Вставка объекта начинается с поиска листовой вершины, в которой этот объект будет размещен. Однако уже на этом шаге появляются отличия от аналогичных процедур R -дерева. Объект в R^+ -дереве может быть размещен в нескольких листовых вершинах одновременно, если он попал на границу раздела этих вершин. Поэтому поиск производится не лучшей для размещения вершины, а всех вершин, с которыми вставляемый объект имеет ненулевое пересечение.

После получения списка пересекающихся листовых вершин объект помещается в каждый из них. Причем если какой-либо узел при вставке объекта окажется переполненным, он расщепляется на две части.

Деление листовых узлов дерева вызывает увеличение числа элементов их предков. Это, в свою очередь, также может привести к переполнению. Таким образом, при вставке объектов деление вершин может распространяться вверх по дереву от листового уровня к корню. Однако по аналогии с рассмотренным ранее $K-D-B$ -деревом, деление внутренних узлов не всегда возможно без дополнительных делений дочерних. Поэтому при распространении процесса деления вверх возможно деление соседних узлов (распространение деления вниз). Подробнее процесс деления будет рассмотрен далее в этом параграфе.

Именно такой алгоритм был предложен создателями структуры. К сожалению, в данном алгоритме есть небольшой изъян, который не позволяет его без модификации использовать на практике. В этом алгоритме совсем не учитывается вставка объектов в дерево, при которой область, занимаемая вставляемым объектом, не принадлежит ни одному из узлов дерева. В этом случае алгоритм выдаст неправильный результат. Это было замечено и описано в последующих работах других исследователей [37, 71].

Алгоритм вставки объекта в дерево в общем виде с помощью псевдокода представлен в листинге 3.30. В этом алгоритме учтен и исправлен первоначальный недостаток R^+ -дерева.

Листинг 3.30

```
//=====
// Вставка объекта в дерево
// Параметры:
//   O - вставляемый объект
//   V - текущая просматриваемая вершина
//=====
ВСТАВКА(O, V)
    [1] // Поиск листовых вершин для вставки
```

Если V – внутренний узел дерева, то
 Q = набор дочерних элементов вершины V
 Сортировать Q по приоритету включения O
 $B = MBR(O)$

- [1.1] V' = первая вершина в списке Q
 Исключить V' из Q
 Расширить MBR вершины V' для включения области B
 (но без нарушения свойств R^+ -деревьев)
 ВСТАВКА(O, V')
 $B = B \setminus MBR(V')$
 Если $B \neq \emptyset$, то
 Выйти из процедуры вставки
 Если $Q = \emptyset$, то
 E = объект
 Повторить столько раз, какова высота уровня V
 Создать пустую вершину V''
 $MBR(V'') = B$
 Добавить E в вершину V''
 $E = V''$
 Вставить E в V
 Если количество объектов в $V >$ предела M , то
 ДЕЛЕНИЕ_УЗЛА(V, m)
 Выйти из процедуры вставки
 Перейти к шагу 1.1
- [2] // Вставка объектов в лист
 Если V – листовый узел дерева, то
 Добавить O в V
 Скорректировать MBR узла V
 Если количество объектов в $V >$ предела M , то
 ДЕЛЕНИЕ_УЗЛА(V, m)
- Конец ВСТАВКА

Алгоритм вставки является рекурсивным. Он проходит по дереву, спускаясь от корня к листьям. При достижении листового уровня, объект вставляется в соответствующую вершину. Этим занимается второй шаг представленного алгоритма. Вставка в листовую вершину является тривиальной и не требует каких-либо серьезных затрат. Единственное что стоит учитывать, так это то, что при вставке листовая вершина может оказаться переполненной. В этом случае вызывается процедура ДЕЛЕНИЕ_УЗЛА, которая будет рассмотрена далее.

В отличие от листовых вершин обработка внутренних вершин дерева требует особого внимания. Ее реализует первый шаг алгоритма в представленном листинге. Как уже отмечалось, первоначально создатели алгоритма предполагали, что достаточно будет просмотреть все ветви дерева и выбрать те из них, которые пересекаются с указанным объектом. На рис. 3.35(а) показан случай, при котором данный подход будет некорректным. Добавляемый объект O этого рисунка не пересекается ни

с одним узлом V_i , а следовательно, если безукоризненно следовать описанному подходу, он не сможет быть размещен в дереве.

Чтобы обойти указанный недостаток, в представленном листинге предлагается вставлять объект в любую вершину, которую можно расширить до вмещения всего объекта O или какой-либо из его частей (на рис. 3.35(а) такой вершиной будет V_2). Для этого все узлы кандидаты сортируются в последовательности увеличения приоритета на вмещение объекта O (список Q в листинге 3.30). Приоритет вершины необходимо считать тем большим, чем на меньшую площадь придется увеличить его MBR при вставке объекта.

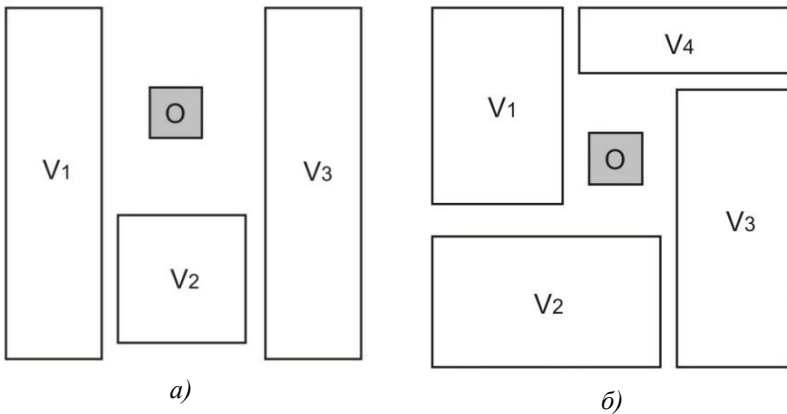


Рис. 3.35. Примеры плохого расположения вставляемого объекта: а – объект не пересекается ни с одним узлом дерева; б – ни один узел дерева не может включить объект

После сортировки вершин их циклически извлекают из списка и пытаются вставить в них объект. Для этого MBR вершины корректируют так, чтобы она вмещала в себя как можно большую часть вставляемого объекта (в идеале – весь объект целиком). При этом, однако, стоит следить за тем, чтобы MBR отдельных узлов не пересекались. После выполнения процедуры рассчитывается та часть объекта, которая осталась не охваченной вершиной при вставке. Для этого в листинге используется переменная B .

Если на определенном шаге окажется, что весь объект целиком размещен в дереве ($B = \emptyset$), процедуру вставки заканчивают. Однако существует вероятность появления ситуации, при которой даже после попытки вставить объект во все вершины останется некоторая его часть, не размещенная в них. Пример такой ситуации показан на рис. 3.35(б). В листинге 3.30 этой ситуации соответствует ветка, при которой список Q оказался пуст ($Q = \emptyset$), но переменная B соответствует непустой

области. В этом случае предлагается создать новую вершину, которая сможет вместить в себя оставшуюся часть объекта. Единственное, что стоит при этом учитывать, так это тот факт, что дерево должно остаться идеально сбалансированным. Поэтому необходимо создать не просто вершину для объекта, а целое поддерево соответствующей высоты.

Как видно из представленного листинга, процедура вставки объекта в R^+ -дерево является более сложной и требует больших вычислительных ресурсов, чем аналогичная процедура для оригинального R -дерева. Однако в большинстве случаев операция вставки вызывается сравнительно редко, поэтому она не очень сильно влияет на эффективность структуры в целом.

Алгоритм деления переполненных вершин дерева

Во время вставки может произойти переполнение листовых вершин. Как было показано, такой случай должен обрабатываться с помощью процедуры деления. При делении один узел разбивается на два новых и его объекты перераспределяются на две группы. Существует несколько алгоритмов деления узла. Однако, как заявляют разработчики структуры, наилучшим является деление по геометрическому принципу, аналогичное тому, которое использовалось в R -дереве Грина. Процедура деления узла представлена в листинге 3.31.

Листинг 3.31

```
//=====
// Деление переполненного узла дерева
// Параметры:
//   V - переполненный узел
//   m - минимальное количество элементов в потомках
//=====
ДЕЛЕНИЕ_УЗЛА(V, m)
[1] // Поиск оси деления
    Q = множество элементов вершины V
    N - количество элементов в вершине V
    Для каждой i-й оси пространства выполнить
        Сортировать элементы Q по i-й координате
        Для всех j от m до (N-m) выполнить
             $K_{i,j}$  = правая граница j-го элемента множества Q
             $S_{i,j}$  = эффективность деления по координате  $K_{i,j}$ 
        Выбрать i и j, для которых  $S_{i,j}$  наилучший
        i = измерение деления
        K = правая граница выбранного объекта
[2] // Деление узла на две части
    {V', V''} = РАСПРЕДЕЛИТЬ_ЭЛЕМЕНТЫ(V, i, K)
[3] // Корректирование предков узла
    Если V - корень дерева, то
        Vnew - новый пустой узел
```

```

        Добавить V' и V'' как дочерние в Vnew
        Рассчитать MBR узла Vnew
        Vnew - новый корень дерева
    Иначе
        Vparent = Parent(V)
        Удалить V из Vparent
        Добавить V', V'' в Vparent
        Если Vparent переполнился, то
            ДЕЛЕНИЕ_УЗЛА(Vparent, m)
Конец ДЕЛЕНИЕ_УЗЛА

//=====
// Распределение объектов на две группы
// Параметры:
//   V - переполненный узел
//   i - номер оси деления
//   K - граница деления
//=====
РАСПРЕДЕЛИТЬ_ЭЛЕМЕНТЫ(V, i, K)
[1] // Создать два новых пустых узла
    Создать два пустых узла V' и V''
[2] // Распределить элементы по новым узлам
    Для всех элементов E, дочерних для V, выполнить
        Если i-я правая граница E ≤ K, то
            Поместить E в V'
        Иначе, Если i-я левая граница E ≥ K, то
            Поместить E в V''
    Иначе
        Если V - листовая вершина, то
            Поместить E в V'
            Поместить E в V''
        Иначе
            {Eleft, Eright} = РАСПРЕДЕЛИТЬ_ЭЛЕМЕНТЫ(E, i, K)
            Поместить Eleft в V'
            Поместить Eright в V''
            Удалить вершину E
[3] // Рассчитать MBR узлов
    Рассчитать MBR для V'
    Рассчитать MBR для V''
    Вернуть пару узлов {V', V''}
Конец РАСПРЕДЕЛИТЬ_ЭЛЕМЕНТЫ

```

Процедура деления узла начинается с выбора оси пространства, по которой пройдет граница между новыми узлами (первый шаг алгоритма). Для этого все множество элементов переполненной вершины сортируется вдоль каждой оси. Сортировку необходимо проводить по большей координате *MBR* элементов.

В отсортированном наборе выбирается элемент в середине так, чтобы полученные при делении по его границе группы были не меньше

m (m – минимальное число элементов в полученных после деления узлах). Величина m передается в процедуру деления, как второй параметр. Конечно, можно выбрать $m = \lfloor M/2 \rfloor$, но подобный шаг вынудит рассматривать всего единственно возможный вариант деления по каждой оси, что приведет к не очень хорошим результатам деления.

Каждый из полученных вариантов деления оценивается с помощью некоторой процедуры эффективности. Данный шаг не был подробно расписан в листинге ввиду разнообразия его вариантов. В зависимости от практического применения или конкретных предпочтений можно выбрать один из следующих критериев эффективности (или использовать комбинацию этих критериев).

1. *Уменьшение площади MBR узлов.* В данном случае в качестве показателя эффективности выступает площадь полученных в результате такого деления узлов. Для каждого из потенциальных узлов вычисляется MBR и рассчитывается его площадь. Чем сумма площадей узлов меньше, тем более эффективным считается деление. Данный критерий позволяет уменьшить пустое пространство, индексируемое узлами, что влечет к повышению общей производительности структуры.

2. *Увеличение расстояния между узлами.* Для вычисления данного критерия не нужно рассчитывать MBR потенциальных узлов. Функция эффективности в этом случае основана на расстоянии между узлами, которое получится при делении по границе выбранного элемента. Для оценки этого расстояния необходимо сравнить пространство между границами текущего элемента и следующего за ним (следующего в отсортированном наборе Q). Чем это расстояние больше, тем лучше считается разбиение, так как меньшее пустого пространства будет проиндексировано в результате деления. В общем случае данный критерий выполняет ту же самую функцию, что и предыдущий, с той лишь разницей, что он проще в реализации.

3. *Уменьшение числа делений элементов.* Данный критерий уменьшает число объектов, которое необходимо разбить при делении на группы. Для его расчета подсчитывают число элементов, попавших на границу деления узлов. Чем это число меньше, тем более эффективным считается деление. В отличие от предыдущих критериев, которые ориентировались прежде всего на уменьшение пустого пространства, данный критерий пытается улучшить внутренние характеристики самого дерева, такие, как число дублирования объектов и процент использования памяти.

4. *Ориентация на более равномерное деление.* В данном критерии разбиение считается более удачным в том случае, если оно позволяет разбить узел на две равные по числу потомков группы. Этот критерий также улучшает внутренние характеристики дерева. Однако он практически никогда не используется обособленно. Лучше всего его

применять в паре с каким-либо другим критерием в качестве дополнительного фактора.

После выбора оси, по которой произойдет деление узла, и границы деления (в качестве нее выступает правая граница выбранного элемента) происходит непосредственно процесс распределения элементов на две группы (второй шаг алгоритма). Если процедура деления была вызвана для внутреннего узла дерева, то распределение объектов на две группы может потребовать рекурсивное деление дочерних узлов. Поэтому оно выполнено в виде отдельной процедуры РАСПРЕДЕЛИТЬ_ЭЛЕМЕНТЫ (см. листинг 3.31). Случай, в котором без рекурсивного деления обойтись нельзя, представлен на рис. 3.36.

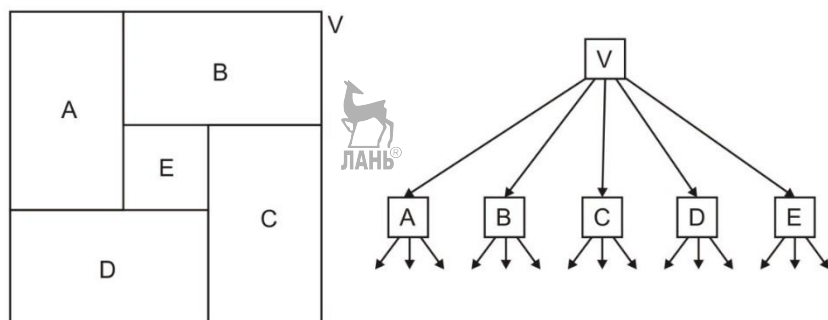


Рис. 3.36. Ситуация неизбежного деления потомков внутреннего узла

На рисунке вершина V содержит пять потомков (допустим, в рассматриваемом примере ограничение на максимальное число элементов в узле равно четырем). При этом, какая бы ось ни была выбрана для деления, все равно появляется необходимость деления потомков (при условии, что вершина V является внутренней вершиной дерева). Это вытекает из первого свойства R^+ -дерева, представленного выше – MBR всех потомков внутренней вершины должны полностью вмещаться в MBR вершины-предка.

Процедуре распределения объектов на две группы передается рассчитанная граница деления и измерение, вдоль которого это разбиение должно произойти. В качестве результата процедура возвращает набор из двух новых вершин, которые вмещают в себя все объекты переполненной вершины.

Работа процедуры распределения достаточно проста. Она перебирает все элементы переданной вершины и проверяет выполнение одного из трех условий.

1. Элемент полностью находится левее границы раздела. В этом случае он помещается в первую вершину V' .

2. Элемент полностью находится правее границы раздела. В этом случае он помещается во вторую вершину V'' .

3. Элемент пересекается границей раздела. Этот случай обрабатывается по-разному в зависимости от типа элементов. Если переполненная вершина является листовой, а распределяемые элементы – это конечные объекты, то элемент помещается в обе новые вершины – V'' и V''' . Если же переполненная вершина является внутренней вершиной дерева, а распределяемые объекты – это другие вершины дерева, необходимо поделить такой элемент на две части границей раздела и поместить каждую из получившихся частей в соответствующие узлы.

Таким образом, получается, что деление внутренних вершин может рекурсивно распространяться вниз. Данный эффект не очень хорошо сказывается на производительности структуры, так как из-за него появляются вершины дерева с маленьким коэффициентом использования памяти (полупустые вершины).

Заключительным шагом процедуры деления узла (третий шаг в листинге 3.31) является корректировка родительских узлов расщепленной вершины. Причем данная корректировка отличается для корневой вершины (у которой предка нет) и всех остальных вершин. Если переполненная вершина является корнем, то в процедуре создается новый корень дерева, потомками которого становятся две новые вершины. Если переполненная вершина является внутренней вершиной дерева, то из нее удаляется старая вершина и добавляются две новые. Причем обязательно при этом необходимо предусмотреть тот факт, что добавление новых вершин может вызвать переполнение вершины-предка и появится необходимость распространить расщепление вверх по дереву.

Процедура деления, описанная в этом параграфе, не является идеальной и содержит ряд изъянов. Так, если в системе возможно добавление одного и того же объекта несколько раз (или объектов с полностью совпадающими *MBR*, что для структуры рассматривается как идентичные объекты), то процедура деления не всегда сможет произвести деления на два непустых узла. Поэтому в подобных случаях ситуации переполнения нужно обрабатывать дополнительно и предусматривать вариант реакции на невозможность деления переполненного узла.

Алгоритм удаления объекта из R^+ -дерева

Операция удаления начинается с поиска всех вхождений заданного объекта в дерево. Сама процедура поиска аналогична той, которая выполнялась в *BSP*-дереве, или той, которая использовалась при вставке в R^+ -дерево. В результате поиска получаем список листовых вершин, в которых упоминается удаляемый узел. Из каждой этой вершины объект удаляется и проверяется возможность объединения такой вершины с

соседней. Это делается для того, чтобы повысить процент использования памяти и уменьшить число пустых и полупустых вершин.

Процедура удаления объекта является достаточно простой, и поэтому не будет рассматриваться с помощью псевдокода в этом параграфе.

3.4. Многослойные структуры

Многослойные структуры похожи на структуры с перекрытием регионов. Причем чаще всего они являются модификациями структур с перекрытием. Точно так же в многослойных структурах находятся объекты, сгруппированные в определенные узлы. Причем в целом по структуре эти узлы могут накладываться и пересекаться.

Однако группа многослойных структур имеет ряд существенных отличий, позволивших выделить их в отдельную категорию. Каждая из структур категории имеет свой определенный набор характеристик и нововведений, отличающих ее от других. Перечислим здесь наиболее общие характеристики, свойственные большинству структур данной категории.

1. В целом структура рассматривается как набор слоев, в каждом из которых поддерживается свой определенный порядок.

2. Чаще всего набор слоев образует иерархию, в которой есть наиболее детализированные (нижние) и более крупные (верхние) слои. Поддерживается эта иерархия общими принципами размещения объектов. Если объект при добавлении может быть размещен на более низком уровне, он туда и помещается.

3. Каждый слой индексирует пространство независимо от остальных слоев, т. е. в нем строится полноценная самостоятельная структура некоторого типа.

4. Узлы (ячейки, блоки, группы) внутри слоя не могут пересекаться (т. е. внутри слоя организуется структура без перекрытия регионов).

5. При индексировании объекты могут свободно перемещаться между слоями при нарушении какого-либо свойства структуры в одном из слоев.

Благодаря своим свойствам, многослойные структуры обладают одним очень важным преимуществом по сравнению со структурами с перекрытием регионов. Ограничение на невозможность пересечения внутренних узлов в пределах одного слоя позволяет многократно повысить производительность операций поиска. При этом, в отличие от структур с делением объектов, рассмотренных в предыдущем параграфе, в многослойных структурах объект размещается всего в одном узле одного слоя и не разбивается на части при вставке. Это также позволяет

выполнять различные оптимизации пространственных запросов, в том числе и диапазонных.

Однако есть и недостатки у структур данной категории. Первым из них можно считать то, что пространство как бы фрагментируется между слоями. Часть объектов помещается в одном слое, часть в другом. Для некоторых практических применений такое поведение может оказаться неприемлемым.

Вторым недостатком является то, что операции обработки данных должны работать не с одним индексом, а с несколькими индексами, расположенными в разных слоях. Таким образом, получается, что для выполнения того же самого поиска необходимо учитывать и производить поиск в нескольких слоях, а результат поиска в каждом слое объединять в общий набор результата. Это приводит к некоторым дополнительным расходам. Однако в большинстве случаев более оптимальная структура индексов каждого из слоев по сравнению с аналогичной структурой без использования слоев, позволяет компенсировать этот недостаток. Так, в многослойном файле-решетке, благодаря использованию техники слоев, удается избежать ряда делений и построить более эффективную структуру.

В данном параграфе будет рассмотрен пример одной из таких структур, в которой применение многослойности действительно дает выигрыш в производительности.

3.4.1. Многослойный файл-решетка

Одной из структур, для которой существует несколько попыток создать многослойный вариант, является файл-решетка. Файл-решетка – одна из простых, но очень эффективных структур индексирования точечных данных. Эта структура была подробно описана в гл. 2.

Файл-решетка первоначально создавалась для индексирования точечных данных. Это наложило ряд ограничений на нее. В частности, для точек всегда можно найти разбиение пространства на n непересекающихся частей с помощью некоторой пространственной решетки. Однако в случае объемных пространственных объектов это не всегда возможно. Какой бы ни была выбрана решетка, в индексе практически всегда найдутся объекты, попавшие на границы деления и, следовательно, их невозможно будет разместить всего в одной ячейке.

Первая попытка адаптировать файл-решетку для пространственных объектов заключалась в применении техники деления объектов. Если некоторый объект накладывается на границу решетки, он размещается в нескольких ячейках этой решетки. Этот подход был подробно рассмотрен в предыдущем параграфе, посвященном R^+ -деревьям. Файл-решетка с делением объектов позволил использовать структуру для индексирования пространственных объектов. Однако структура индекса с большим числом объектов имела каталог очень большого размера и

была не самой эффективной с точки зрения производительности запросов поиска.

В 1988 году Г. Сикс и П. Видмаер разработали новый многослойный вариант файлов-решеток (*multi-layer grid file*) [85]. Их структура состояла из нескольких самостоятельных решеток (слоев), каждая из которых образовывала свой каталог. На всех слоях, кроме самого верхнего, было запрещено деление объектов. Таким образом, получился более эффективный и быстродействующий вариант.

Не стоит путать многослойный файл-решетку с многоуровневым файлом-решеткой [91] (*multilevel grid file*). Многоуровневый файл также использует технику слоев, но не для индексирования пространственных объектов, а для иерархической организации своего каталога. Как было отмечено, каталог файла-решетки имеет большой размер и хранится на жестком диске. При этом, чтобы увеличить скорость обработки и поиска записей, можно использовать многоуровневое формирование каталога. Пример с двухуровневым каталогом [42] был рассмотрен ранее во второй главе.

Еще одним многослойным вариантом этой структуры является двойной файл-решетка (*twin grid files*) [45]. В этой модификации используются две самостоятельные решетки, которые вместе образуют два слоя структуры. Однако эта структура не будет рассматриваться в данном параграфе. Двойной файл-решетка разрабатывался не для пространственных, а для точечных данных. Появление в нем второго слоя вызвано вовсе не пересечением объектов при индексировании, а необходимостью уменьшить размеры каталога. Как показали создатели структуры, использование двух слоев и двух решеток позволяет уменьшить общие затраты на хранение каталога, что и было предложено ими в данной модификации.

Общие принципы построения многослойного файла-решетки

Многослойный файл-решетка представляет собой упорядоченный набор структур, рассматриваемых как слои. Каждый такой слой образует самостоятельную решетку со своим каталогом и своим набором гиперплоскостей. Деление пространства гиперплоскостями не обязано совпадать в разных слоях, хотя подобное совпадение и не запрещено структурой. Пример двухмерного двухслойного файла-решетки показан на рис. 3.37.

На рисунке изображен файл-решетка с вместимостью ячеек 3 (в каждой ячейке может находиться не более трех объектов). Как видно из примера, большая часть объектов при индексировании попала в первый слой. Однако объекты O_{18} и O_{19} являются настолько большими, что при попытке размещения их в первом слое они займут несколько ячеек. Так как в многослойном файле-решетке запрещено такое поведение, эти

объекты были размещены в верхнем слое, в котором они не пересекаются решеткой.

В то же время объекты O_{16} и O_{17} являются достаточно малыми, чтобы разместиться в ячейках нижнего слоя. Однако они точно так же были перенесены во второй слой. Это произошло потому, что эти объекты попали на границу ячеек нижнего слоя. Отнести их к какой-либо одной ячейке стало невозможно, и они были перенесены во второй слой.

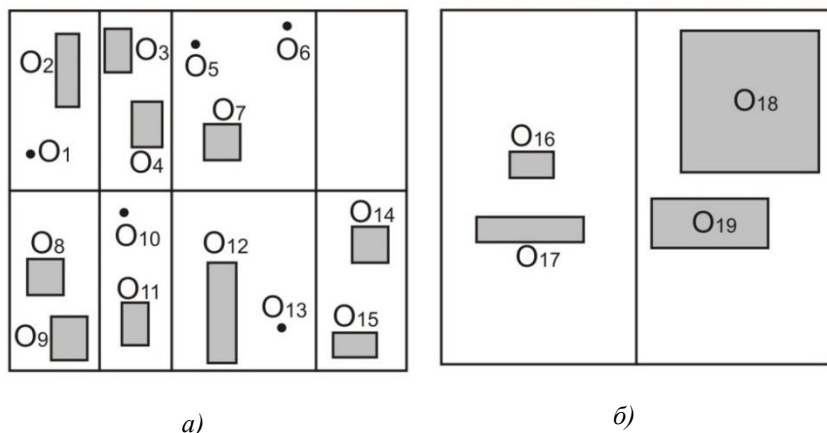


Рис. 3.37. Пример двухмерного двухслойного файла-решетки: а – первый слой (нижний); б – второй слой (верхний)

В общем случае многослойный файл-решетка должен удовлетворять следующим свойствам.

1. Файл-решетка состоит из N слоев. В практических реализациях N не может быть бесконечным, так как при этом теряется производительность структуры в целом. Поэтому чаще всего N ограничено небольшим значением.

2. На каждом слое построена своя решетка с отдельным каталогом. Решетки разных слоев не зависят друг от друга. Принцип формирования линейных шкал и каталога решетки полностью совпадает с тем, который использовался в оригинальном виде структуры.

3. Объекты в слоях с первого по $(N-1)$ не пересекаются с гиперплоскостями деления решетки. Поэтому размещение объектов во внешних блоках строго определено структурой самой решетки.

4. Слой с номером N включает объекты по технике деления. Если объект попадает на границу деления ячеек, он размещается в обеих ячейках.

За поддержание этих свойств отвечает процедура вставки, рассмотренная далее в этом параграфе.

Алгоритм поиска объекта в многослойном файле-решетке

Рассмотрим два наиболее важных типа запросов – поиск по точному совпадению объектов и поиск по области. Именно эти два вида запросов раскрывают все основные свойства многослойного файла-решетки. Остальные виды запросов реализуются аналогичным образом.

Запрос объекта по точному совпадению. В этом запросе в процедуру поиска передается некоторый объект O , наличие которого необходимо определить в индексе. Алгоритм процедуры поиска представлен в листинге 3.32.

Листинг 3.32

```
//=====
// Поиск объекта O по точному совпадению
// Параметры:
//   O – объект поиска
//=====
ПОИСК_ОБЪЕКТА(O)
[1] // Определение границ объекта O
    P1 = нижняя левая точка MBR объекта O
    P2 = верхняя правая точка MBR объекта O
[2] // Поиск объекта
    Цикл по слоям с первого по N
        C1 = координаты ячейки текущего слоя для P1
        C2 = координаты ячейки текущего слоя для P2
        Если C1=C2 или номер слоя равен N, то
            В = внешний блок для ячейки C1
            Для всех объектов O' блока В выполнить
                Если O' = O, то
                    Вернуть ИСТИНУ
    Вернуть ЛОЖЬ
Конец ПОИСК_ОБЪЕКТА
```

Как показано в листинге, операция поиска начинается с нижнего слоя и охватывает все слои структуры. Однако слой проверяется только в том случае, если объект поиска не пересекается с его решеткой (исключение составляет самый верхний слой, в котором допускается пересечение с решеткой).

Для проверки пересечения с решеткой вычисляется номер ячейки, содержащей нижний левый (C_1) и правый верхний (C_2) углы запрошенного объекта. В случае двухмерного пространства C является парой чисел (i_1, i_2) – номер ячейки по вертикали и горизонтали в каталоге решетки данного слоя. Эти номера определяются с помощью линейных шкал решетки (подробнее этот процесс был описан во второй главе). Так как линейные шкалы имеют маленький размер и хранятся обычно в оперативной памяти, то определение C_1 и C_2 не требует обращений к жесткому диску и выполняется очень быстро.

Если ячейки C_1 и C_2 не совпадают, то слой нужно пропустить. Объект пересекается с решеткой и ни одна ячейка не может его содержать. Исключение составляет самый верхний слой, в котором разрешено деление объектов.

Таким образом, для реализации поиска на точное совпадение необходимо проверять не все слои структуры, а только те из них, которые действительно могут содержать запрошенный объект. Причем, как показывают эксперименты, обычно объект расположен в самом нижнем из слоев, который может его вместить. В этом случае для его поиска понадобится проверить всего один слой. Если учесть, что каждый слой многомерного файла-решетки имеет более удобную и эффективную структуру, чем аналогичные каталоги файлов-решеток с делением объектов, то получается, что данный алгоритм дает выигрыш при выполнении операций поиска.

Запрос объекта по области. В процедуру поиска по области передается граница некоторой области R . Необходимо найти все объекты, имеющие ненулевое пресечение с этой областью.

Данный запрос является более сложным в вычислительном плане. Алгоритм процедуры для поиска по области показан в листинге 3.33.

Листинг 3.33

```
//=====
// Поиск по области
// Параметры:
//   R - область поиска
//=====
ПОИСК_ПО_ОБЛАСТИ(R)
[1] // Начальная инициализация
    P1 = нижняя левая точка области R
    P2 = верхняя правая точка области R
    Res = пустое множество для результата поиска
[2] // Поиск по слоям
    Цикл по слоям с первого по N
        C1 = координаты ячейки текущего слоя для P1
        C2 = координаты ячейки текущего слоя для P2
        Для всех ячеек диапазона от C1 до C2 выполнить
            В = внешний блок для выбранной ячейки
            Для всех объектов O' блока В выполнить
                Если  $MBR(O) \cap R \neq \emptyset$ , то
                    Добавить O' в множество результата Res
[3] // Вернуть множество результата
    Вернуть Res
Конец ПОИСК_ПО_ОБЛАСТИ
```

Для выполнения поиска по области необходимо проверить ячейки всех слоев, которые пересекаются с областью поиска. При этом

отбросить слой по каким-либо критериям нельзя. Поэтому в общем случае процедура поиска по области является более ресурсоемкой. Однако создатели структуры показали, что в целом поиск по многослойному файлу-решетке более выгоден, чем поиск по аналогичному файлу-решетке с разделением объектов [85].

Алгоритм добавления объекта

Вставка объекта начинается с выбора слоя. Для этого пытаются разместить объект в каждом из слоев, начиная с первого (самого нижнего). Как только будет найден слой, способный разместить объект без нарушения свойств структуры, в него добавляется объект. В подобной последовательности действий заключается главное отличие многослойного файла-решетки от двойного файла-решетки, в котором объекты могут свободно перемещаться между слоями. В двойном файле-решетке объект может быть расположен абсолютно в любом из слоев, в котором его размещение не вызовет переполнение ячеек.

Описанный алгоритм вставки часто приводит к неявной сортировке объектов по их размеру. Маленькие объекты и многомерные точки оседают в самом нижнем слое, ячейки которого имеют самый маленький размер. Большие объекты, наоборот, поднимаются в верхние слои.

Если при вставке объекта не был найден слой, в котором можно было бы поместить объект без пересечения с решеткой, создают еще один пустой слой и в него добавляют объект. Однако бесконечное увеличение числа слоев бывает невыгодно на практике. Большое количество самостоятельных решеток, соответствующих слоям, приводит к значительному увеличению накладных расходов на использование памяти под каталоги и уменьшению производительности операций поиска. Поэтому создатели структуры предложили ограничить максимальное число слоев некоторым пределом N . Параметр N выбирается в момент проектирования структуры и не может быть изменен в дальнейшем без полной переиндексации.

Ограничив число слоев, проектировщик сталкивается с проблемой: что делать с объектами, которые не могут быть помещены ни в один из N слоев без пересечения с решеткой? Разработчики решили эту проблему просто. Они предложили организовывать решетку без пересечения объектов только на первых $(N-1)$ слоях, а на последнем слое строить обычный вариант файла-решетки с делением объектов (если объект при вставке на последнем слое попадает на границу двух или более ячеек, он добавляется в каждую из этих ячеек). Подобный алгоритм вставки показан в листинге 3.34.

Листинг 3.34

```
//=====
// Вставка объекта
```

```

// Параметры:
//   О - новый объект
//=====
ВСТАВКА(О)
  [1] // Определение границ объекта О
      Р1 = нижняя левая точка MBR объекта О
      Р2 = верхняя правая точка MBR объекта О
  [2] // Вставка объекта
      Цикл по слоям i с первого по N
      С1 = координаты ячейки текущего слоя для Р1
      С2 = координаты ячейки текущего слоя для Р2
      Если С1 = С2, то
          В = внешний блок для ячейки С1
          Добавить О в блок В
          Если блок В переполнился, то
              ДЕЛЕНИЕ_СЛОЯ(i, С1)
          Выйти из процедуры вставки
      Иначе, Если номер слоя i=N, то
          Для всех ячеек С' диапазона от С1 до С2
              В = внешний блок для ячейки С'
              Добавить О в блок В
              Если блок В переполнился, то
                  ДЕЛЕНИЕ_СЛОЯ(i, С')
          Выйти из процедуры вставки
Конец ВСТАВКА

//=====
// Деление слоя в переполненной ячейки
// Параметры:
//   i - номер слоя, в котором произошло переполнение
//   С - координаты переполненной ячейки
//=====
ДЕЛЕНИЕ_СЛОЯ(i, С)
  [1] // Деление решетки
      Выбор оси j и координаты Кj деления решетки
      Добавление в i-ю решетку новой гиперплоскости
  [2] // Восстановление свойств структуры
      Если i ≠ N, то
          Для всех ячеек С', j-я граница которых содержит Кj
              В = внешний блок для ячейки С'
          Для всех объектов О' блока В выполнить
              Если j-я граница О' содержит Кj, то
                  Удалить О' из блока В
              ВСТАВКА(О')
Конец ДЕЛЕНИЕ_СЛОЯ

```

В приведенном листинге предполагается, что все N слоев структуры созданы заранее (допускается, что в начальный момент они могут быть пустыми). Такой подход оправдан, так как в практических реализациях

число объектов таково, что все слои при их размещении будут задействованы.

Стоит остановиться на реализации деления решетки некоторого слоя (процедура ДЕЛЕНИЕ_СЛОЯ листинга 3.34). Во многом она идентична той, которая была рассмотрена для деления решетки во второй главе. На первом шаге вдоль некоторого измерения пространства выбирается гиперплоскость, которая разобьет переполненную ячейку на две части (алгоритм выбора измерения и координаты деления подробно в листинге не рассматриваются, так как описывались ранее в соответствующих разделах для оригинальных файлов-решеток). После выбора новой гиперплоскости она добавляется в линейные шкалы решетки, и корректируется каталог слоя.

Однако в отличие от оригинальных файлов-решеток, в данном алгоритме можно прийти к нарушению свойств структуры. Для восстановления всех характеристик и корректной работы выполняется второй шаг алгоритма. Если текущий слой не является самым верхним (в котором возможны пересечения объектов с решеткой), необходимо проверить объекты всех ячеек, попавших на новую гиперплоскость решетки. Все объекты, которые не удовлетворяют свойствам структуры, удаляются из текущего слоя и вставляются в индекс заново с помощью уже рассмотренной процедуры вставки.

Стоит также заметить, что для упрощения представления алгоритма, в процедуре деления убрана проверка на объединение ячейки в группы. Однако в практических реализациях этого делать не стоит, так как подобное поведение приведет к значительному ухудшению производительности и уменьшению коэффициента использования памяти (который в этом варианте и так является достаточно низким из-за наличия нескольких каталогов).

Алгоритм удаления объекта

Операция удаления начинается с выполнения процедуры поиска объекта по точному совпадению (листинг 3.32). Как только объект на каком-то из слоев будет определен, он удаляется из внешнего блока. В заключение проверяется этот блок на число объектов в нем. Если после удаления блок оказался полупустым или даже пустым, он объединяется с соседним в группу. Это позволяет повысить коэффициент использования памяти.

Если файл-решетка не предполагает постоянного сокращения индекса, то можно не предусматривать процедуру сокращения решетки. Ситуация возникновения предпосылок для нее маловероятна, а трудоемкость подобной операции велика. Однако если вариант с постоянным сокращением возможен, то после объединения ячеек в группы проверяют наличие в решетке неиспользуемых строк и столбцов и в случае их обнаружения – корректируют решетку. После модификации

решетки можно попытаться перенести объекты с более высоких уровней, что позволит повысить эффективность операций поиска.



Резюме

В последнее время информационные системы, такие как системы картографии и навигации, становятся все более и более популярными. Подобные сервисы появляются не только в больших и мощных системах, но они внедряются в приложениях для настольных ПК и веб-сервисов. При этом обойтись без структур индексирования попросту нельзя. В связи с этим проблемы, затронутые в данной главе, являются очень актуальными.

По данному направлению рассмотрены все наиболее развитые многомерные структуры. Особое внимание уделяется структурам с перекрытием областей и структурам с разделением объектов на части.

Многомерные структуры – это новое и очень интересное направление. Его еще предстоит исследовать.



ГЛАВА 4. МНОГОМЕРНАЯ ПИРАМИДА

Объекты с многочисленными параметрами далеко не всегда можно свести к одному параметру с помощью свертки либо уменьшения размерности объекта [4]. Отношения между данными существенно влияют на время выполнения основных операций над ними. Если алгоритм часто использует операции вставить, найти максимум\минимум, удалить максимум\минимум, удалить заданный элемент, изменить заданный элемент и некоторые другие, то здесь весьма удобным являются пирамидальные структуры данных [11]. Они позволяют выполнять большинство операций за один шаг, т. е. практически не зависят от размерности задачи, либо имеют логарифмическую сложность в худшем случае. Одной из таких структур является многомерная пирамида.

Для эффективного представления многомерной приоритетной очереди используется K - D -пирамида [27]. Базовая форма K - D -пирамиды не использует дополнительного пространства, требует линейное время на построение и обеспечивает постоянную сложность доступа к элементам, содержащим минимальный ключ любого измерения, логарифмическое время для вставки, удаления или изменения любого элемента очереди. Более того, структура может быть расширена до многомерной двусторонней приоритетной очереди. Для некоторых приложений оптимальным вариантом будут многомерные приоритетные очереди.

После разработки единой структуры отпала необходимость в дополнительной памяти для хранения связей, дополнительных операциях для поддержки различных приоритетных отношений. При использовании неявного представления K - D -пирамида не требует дополнительной памяти для указателей. Более того, K - D -пирамида может поддерживать некоторые или все операции в любом количестве измерений, содержать ранее изученные структуры, типа двусторонних очередей, в особых случаях.

Эта пирамида связана с K - D -деревьями, которые расширяют деревья бинарного поиска до нескольких ключей. Основным недостатком K - D -деревьев являются трудные операции удаления и балансировки дерева. Представленная K - D -пирамида автоматически балансируется и более проста в представлении.

Детальный анализ сложности операций над K - D -пирамидой можно найти в [26].

Минимальная K - D -пирамида H отражает множество элементов, каждый из которых содержит k ключей $key_1, key_2, \dots, key_k$, где key_i соответствует полностью упорядоченному множеству K_i .

Пирамида H является бинарным деревом, удовлетворяющим следующим условиям:

1. H – полное бинарное дерево;
2. H поддерживает основное свойство K - D -пирамиды:
 - элемент в корне имеет наименьшее значение ключа key_1 в дереве;
 - каждый уровень дерева, кратный соответствующему ключу key_i , содержит наименьший ключ key_i в своем поддереве. Другими словами, в K - D -пирамиде узел уровня i имеет наименьший ключ $key_{\text{mod}(i, k)+1}$ в его поддереве.

Определим уровень узла как число узлов на пути от корня к данному узлу. Уровень $i = \lceil \log n \rceil$, где n – номер элемента внутри пирамиды. Высота пирамиды равна наибольшему уровню ее узлов. На практике высота пирамиды обычно значительно больше, чем число измерений. Мы будем называть k размерностью пирамиды. Например, 2- d пирамида показана на рис. 4.1.

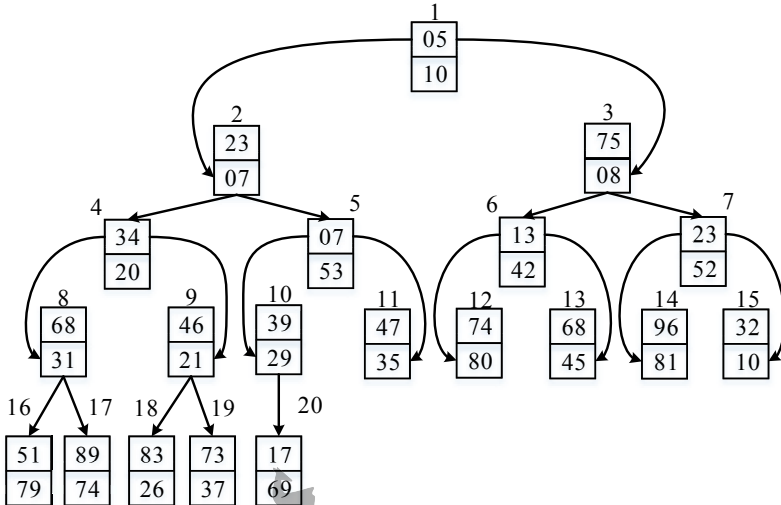


Рис. 4.1. 2- d пирамида из 20 элементов

Очевидно, что K - D -пирамида может быть представлена как явно, так и неявно (в виде массива). Если не сказано иное, мы будем рассматривать явное представление. Легко заметить, что одномерная пирамида является

бинарным деревом, а двумерная с $key_1 = -key_2$ для всех узлов – *min-max* пирамидой [18].

Рассмотрим операции на примере 2-*d* пирамид, затем обобщим сказанное до *K-D*-пирамид.

Базовые операции на 2-*d* пирамиде. Самой важной операцией является восстановление основного свойства пирамиды. Предположим, что в 2-*d* пирамиде один узел изменен.

Алгоритм ВСПЛЫТИЕ (узел): для каждого предка узла, начиная с корня, проверять основное свойство пирамиды между предком и узлом; если он нарушен, то поменять местами предка и узел. На рис. 4.1(а) изображена 2-*d* пирамида с измененным (неправильным) узлом (5). Рядом с элементом стоит его номер, внутри ключи: key_1 (сверху) и key_2 (снизу). На рис. 4.1(б) – результат работы процедуры *ВСПЛЫТИЕ* ().

Процедура *ВСПЛЫТИЕ* в рассмотренном примере состоит из двух шагов:

шаг 1. Сравняются элементы 5 и 1 по key_1 . Основное свойство пирамиды выполняется. Элементы местами не меняем.

шаг 2. Сравняются элементы 5 и 2 по key_2 . Основное свойство пирамиды не выполняется. Меняем элементы местами.

После такой процедуры все узлы, кроме одного, будут соответствовать основному свойству пирамиды, поэтому нам нужно всего лишь восстановить порядок в поддереве, корнем которого является неправильный узел. Это делает следующий рекурсивный алгоритм.

Алгоритм ПОГРУЖЕНИЕ (узел): для каждого потомка рассматриваемого узла проверяем основное свойство минимальной пирамиды. В случае нарушения этого свойства элементы меняются местами и процедура *ПОГРУЖЕНИЕ* выполняется рекурсивно.

Процедура *ПОГРУЖЕНИЕ* в рассмотренном примере состоит из двух шагов:

шаг 1. Сравняются элементы 5 и 20 по key_1 . Основное свойство пирамиды не выполняется. Меняем элементы местами.

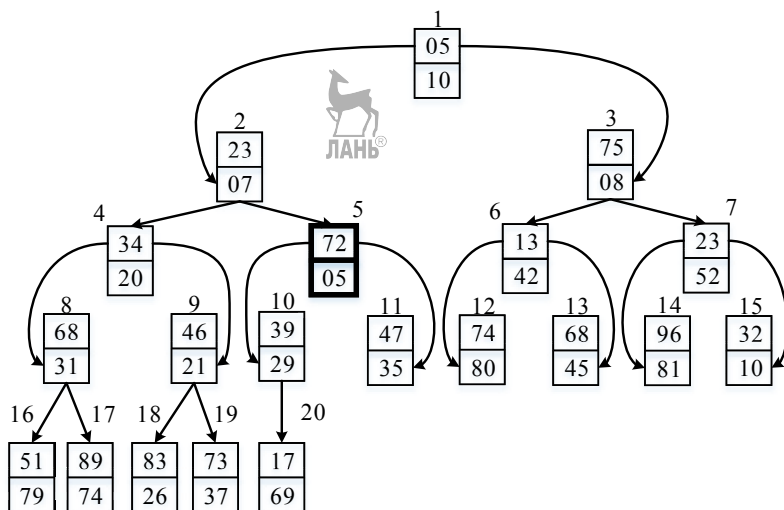
шаг 2. Сравняются элементы 10 и 20 по key_2 . Основное свойство пирамиды не выполняется. Меняем элементы местами.

Эта процедура показана на рис. 4.2(в), где неправильный узел на рис. 4.2(б) восстанавливается относительно его потомков. Процедура может быть представлена как процедура проталкивания неправильного узла вниз по пирамиде.

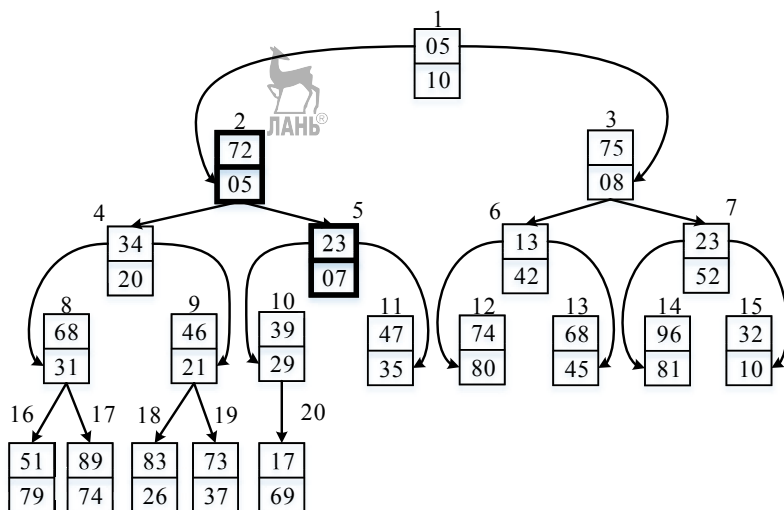
Теперь процедуру восстановления основного свойства пирамиды можно представить в два шага: вызвать *ВСПЛЫТИЕ* (узел) и затем вызвать *ПОГРУЖЕНИЕ* (узел).

Базирующиеся на этих алгоритмах операции над 2-*d* пирамидой довольно просты.

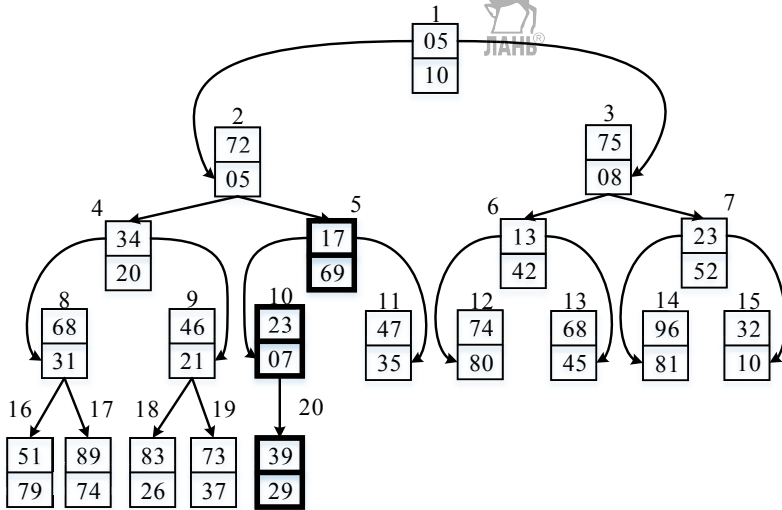
Для вставки элемента нам нужно добавить его в конец пирамиды и восстановить ее основное свойство (пример дан на рис. 4.2).



a)



б)

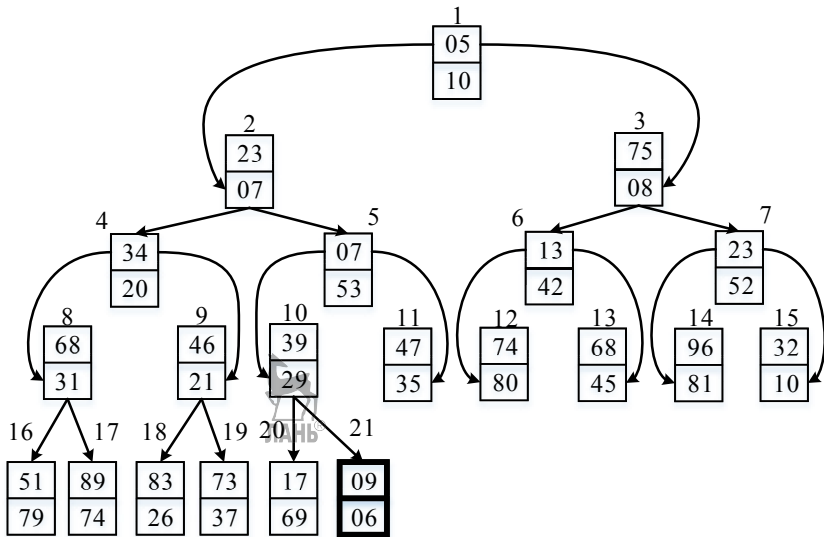


в)

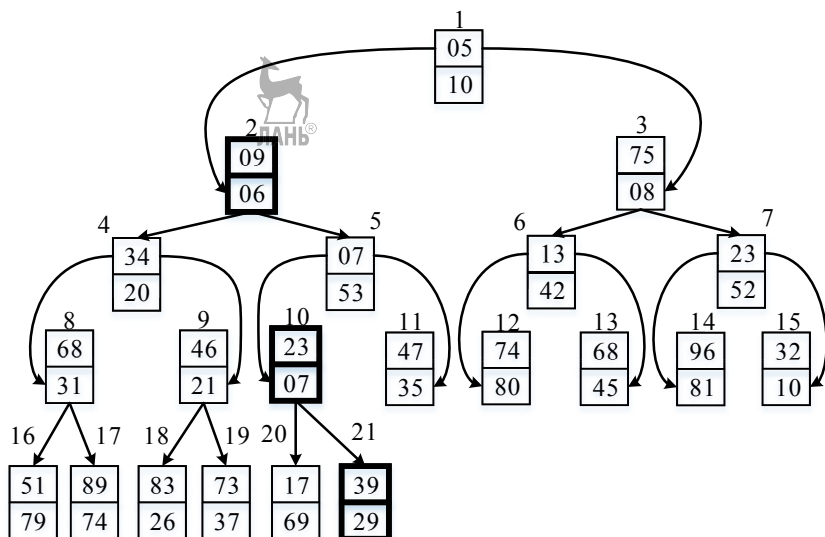
Рис. 4.2. Пример восстановления основного свойства 2-d пирамиды:

а) исходное положение пирамиды с измененным элементом;

б) ВСПЛЫТИЕ; в) ПОГРУЖЕНИЕ



а)



б)

Рис. 4.3. Вставка элемента в 2-d пирамиду: а) вставляется 21-ый элемент с ключами $key_1 = 09$, $key_2 = 06$, основное свойство пирамиды нарушено; б) ВСПЛЫТИЕ

В данном примере алгоритм ВСПЛЫТИЕ состоит из следующих четырех шагов:

шаг 1. Сравняются элементы 21 и 1 по key_1 . Основное свойство пирамиды выполняется. Элементы местами не меняем.

шаг 2. Сравняются элементы 21 и 2 по key_2 . Основное свойство пирамиды не выполняется. Меняем элементы местами.

шаг 3. Сравняются элементы 21 и 5 по key_1 . Основное свойство пирамиды выполняется.

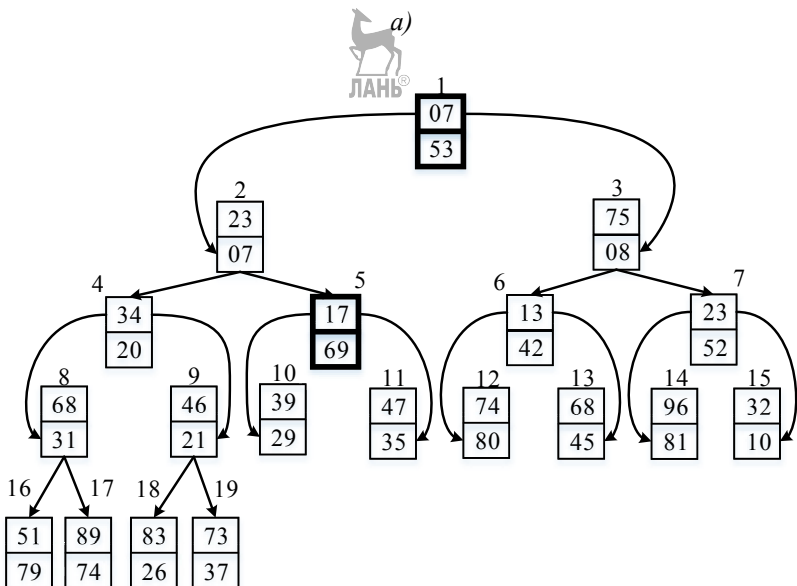
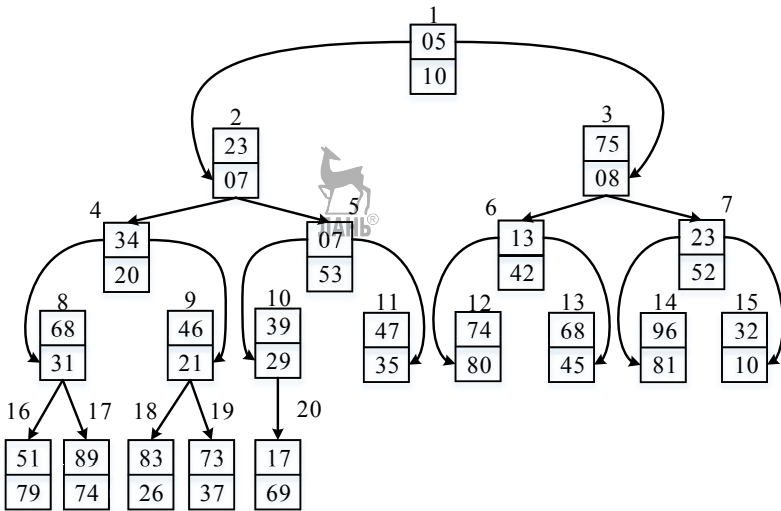
шаг 4. Сравняются элементы 21 и 10 по key_2 . Основное свойство пирамиды не выполняется. Меняем элементы местами.

Для удаления минимального элемента по ключу key_1 , мы должны поменять местами корень с этим элементом и восстановить порядок пирамиды. В этом случае необходимо выполнить только погружение. Пример дан на рис. 4.4(б) и рис. 4.4(в) для пирамиды с рис. 4.4(а).

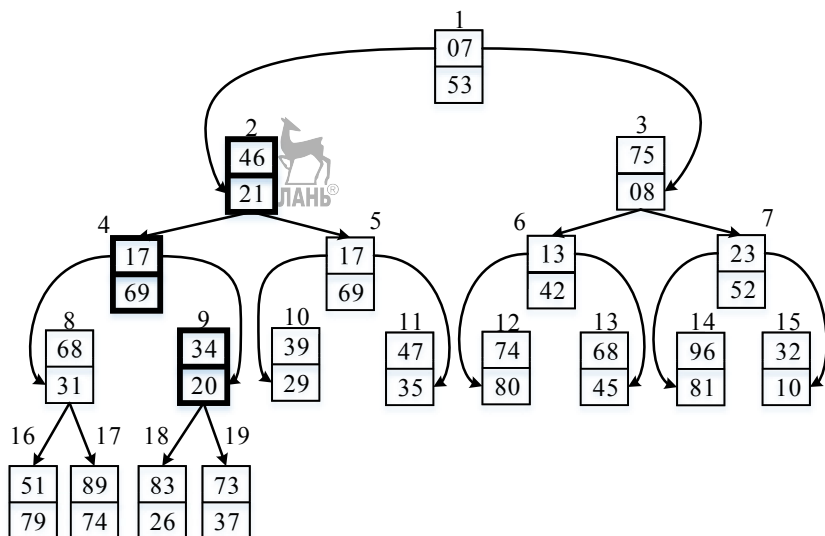
При удалении минимального элемента по ключу key_2 , мы сначала располагаем его среди первых трех элементов пирамиды. Затем мы меняем его местами с последним элементом пирамиды и восстанавливаем основное свойство пирамиды.

Для удаления произвольного элемента (с известной позицией) мы просто меняем его местами с последним элементом в пирамиде и

восстанавливаем основное свойство. Чтобы изменить произвольный элемент, нужно просто выполнить изменение элемента, а затем — восстановить основное свойство пирамиды.



б)



в)

Рис. 4.4. Пример удаления элемента с минимальными ключами из 2-d пирамиды: а) исходное положение 2-d пирамиды; б) удаляется элемент $\min key_1$; в) удаление $\min key_2$

Создание 2-d пирамиды похоже на создание бинарной пирамиды, т. е. оно выполняется снизу вверх. Мы описываем его в рекурсивной форме, хотя рекурсия не является необходимой — она упрощает объяснение (так как имеет параметр, показывающий ключ уровня). Этот алгоритм состоит из 3 шагов. Первый вызывает процедуру создания поддерева СОЗДАТЬ (левое_поддерево, $key_{\text{mod}(i, 2)+1}$), а после этого процедуру СОЗДАТЬ (правое_поддерево, $key_{\text{mod}(i, 2)+1}$) и в конце ПОГРУЖЕНИЕ (корень).

В представленном примере удаление элемента $\min key_1$ происходит следующим образом (см. рис. 4.4(б)):

шаг 1. Удаляется элемент 1, на его место вставляется элемент 20. Сравниваются элементы 1 и 5 по key_1 . Основное свойство пирамиды не выполняется. Меняем элементы местами.

шаг 2. Сравниваются элементы 10 и 5 по key_1 . Основное свойство пирамиды выполняется.

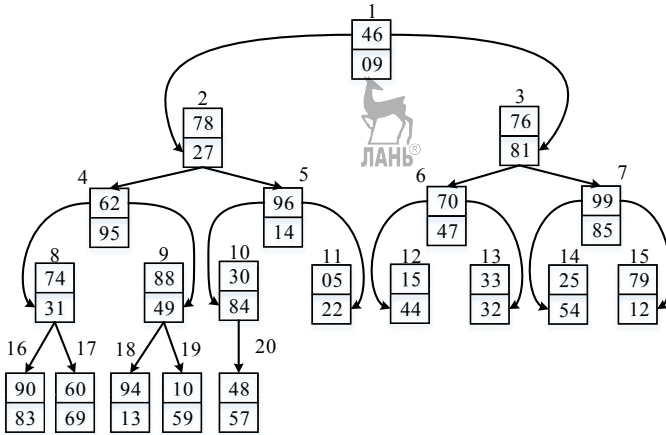
В представленном примере удаление элемента $\min key_2$ происходит следующим образом (см. рис. 4.4 (в)):

шаг 1. Удаляется элемент 2, на его место вставляется элемент 20. Сравниваются элементы 2 и 9 по key_2 . Основное свойство пирамиды не выполняется. Меняем элементы местами.

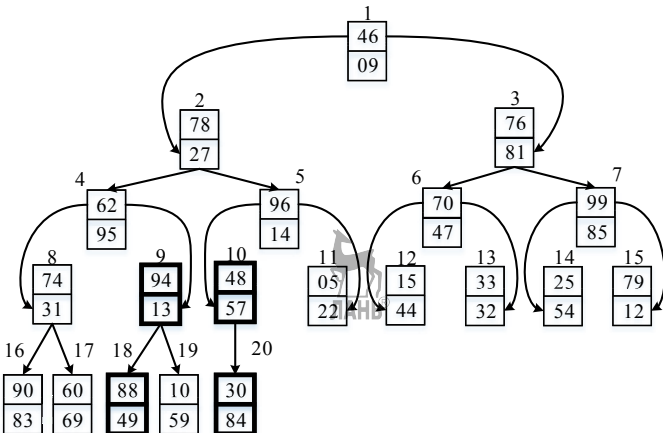
шаг 2. Сравниваются элементы 4 и 9 по key_1 . Основное свойство пирамиды не выполняется. Меняем элементы местами.

шаг 3. Сравниваются элементы 18 и 9 по key_2 . Основное свойство пирамиды выполняется.

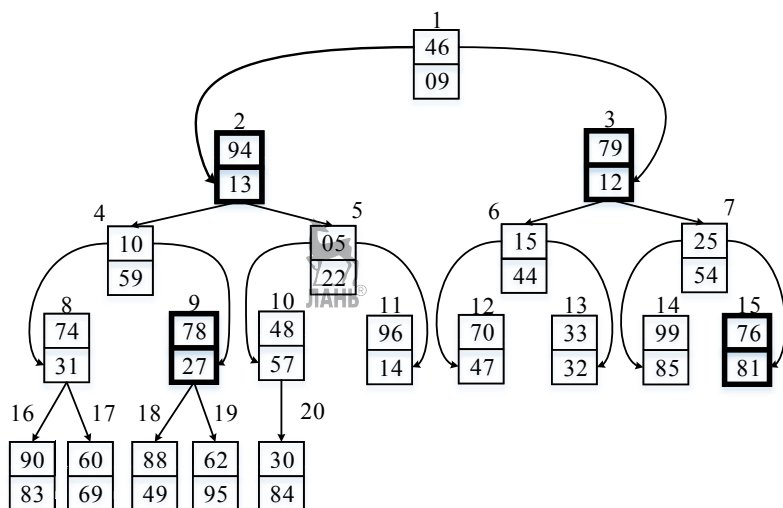
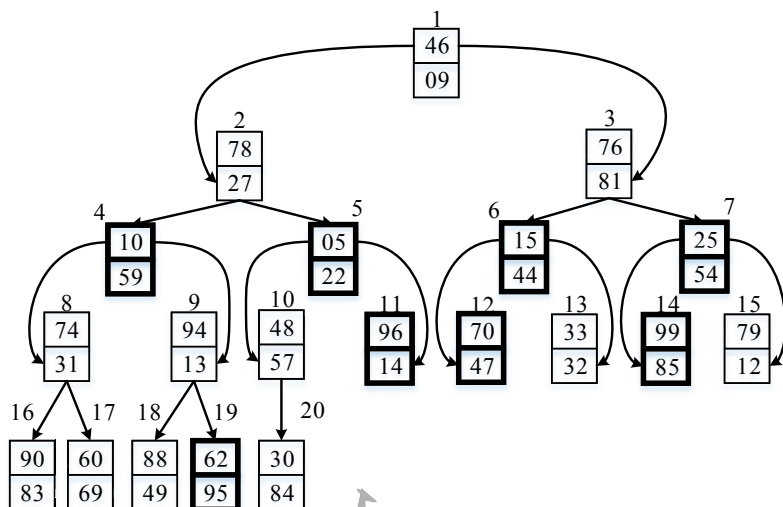
Пример показан на рис. 4.5.



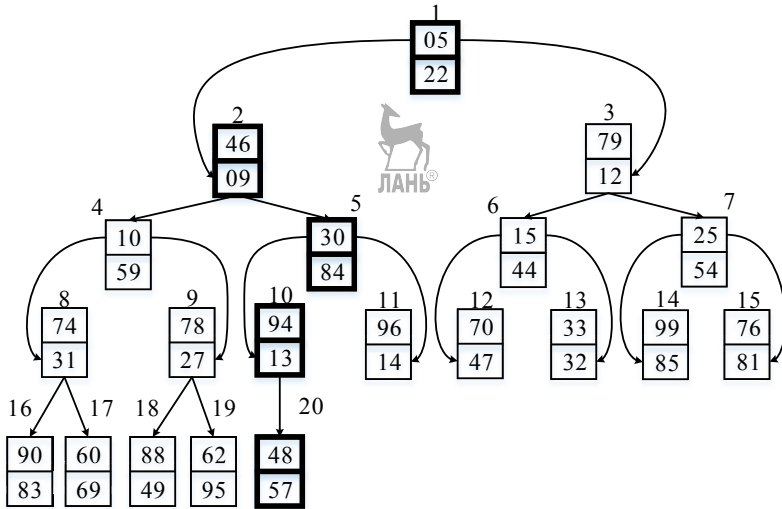
a)



б)



2)



д)

Рис. 4.5. Создание 2-d пирамиды: а) бинарное дерево, представляющее множество из 20 2-d элементов; б) восстановление основного свойства пирамиды на 4-ом уровне по key_2 ; в) восстановление основного свойства пирамиды на 3-ем уровне по key_1 ; г) восстановление основного свойства пирамиды на 2-ом уровне по key_2 ;

д) восстановление основного свойства пирамиды на 1-ом уровне по key_1 ;

2-d пирамида из n элементов может быть создана за время $O(n)$. В такой пирамиде поиск элемента с минимальным значением любого ключа имеет сложность $O(1)$. Вставка, удаление элемента с минимальным значением любого ключа, удаление любого элемента, позиция которого известна, и изменение известного элемента имеет сложность $O(\log n)$ в худшем случае.

2-d пирамида может быть представлена как *min-max* пирамида [18], если мы установим $key_2 = -key_1$ для всех узлов, где key_1 – исходный минимальный ключ, а key_2 – новый максимальный ключ. Однако, операции для 2-d пирамиды были разработаны так, что key_1 и key_2 могут быть полностью не связаны. Для этого случая более эффективно будет использование операций, специфичных для *min-max* пирамид. Для случая, в котором узел нормализован по отношению к потомкам, в общей 2-d пирамиде минимальный ключ может быть в любом из его детей и внуков. Однако, в *min-max* пирамиде, если в ней имеется какой-либо внук, минимальный ключ будет в одном из внуков. Поэтому функция погружения может лишь по очереди проверять внуков, пока они не

кончатся. Точно также функция всплытия может использовать перевернутый подход и идти только через минимальные и максимальные уровни. Это снизит число сравнений на 50%. В общем, операции на 2- d пирамиде могут быть улучшены, если результаты некоторых сравнений могут содержаться в результатах некоторых других сравнений, базирующихся на известных отношениях между двумя приоритетами.

Базовые операции над K - D -пирамидами. Выводы из ранее сказанного могут быть легко обобщены для случая K - D -пирамиды при любом постоянном k . Извлечение минимального элемента по ключу key_i представлено поиском среди постоянного числа узлов первых i уровней, поэтому требует $O(1)$ времени. Если извлечение часто используется, позиции могут быть сохранены в памяти для обеспечения быстрого доступа. Функция всплытия остается такой же (независимо от k), функция погружения может быть представлена следующим образом, который является обобщением случая с 2- d пирамидой. Пусть key_i будет ключом уровня узла. Найти узел V с наименьшим ключом key_i среди потомков узла в пределах k уровней.

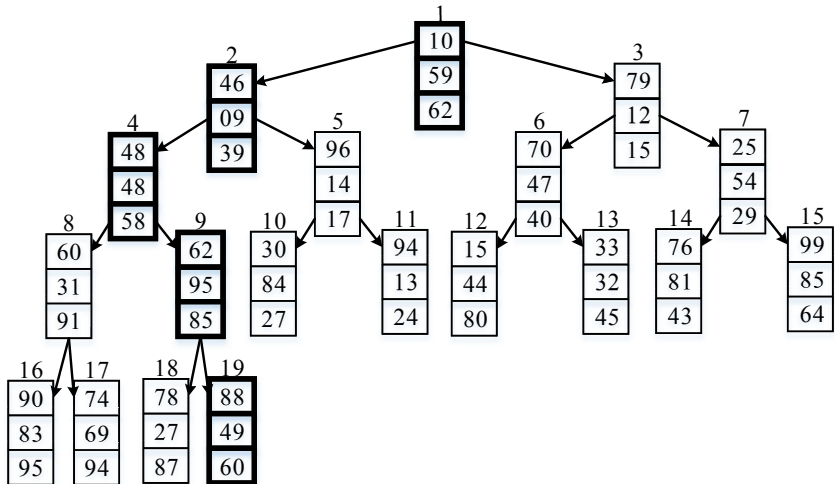
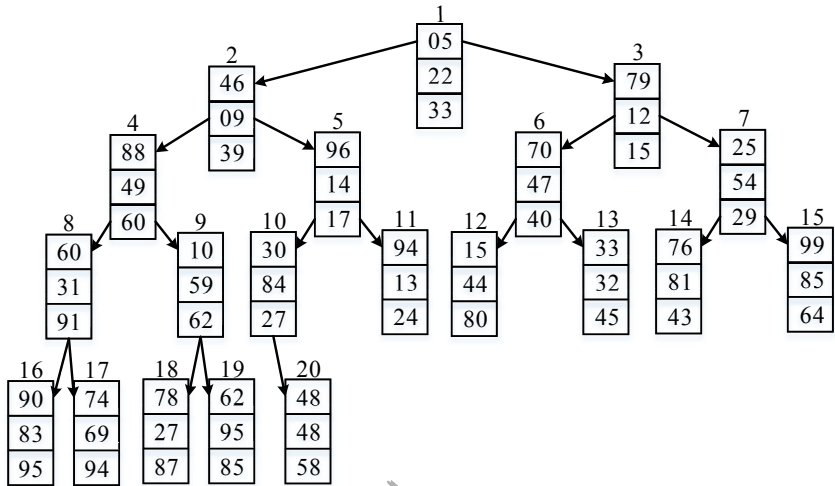
Листинг 4.1

```
//=====
// Процедура погружения
// Параметры:
//   V – погружаемая вершина
//   S – область поиска
//   Res – множество результатов поиска
//=====
ПОГРУЖЕНИЕ (V)
  [1] Если V имеет меньший ключ  $key_i$ , чем  $V'$ , то
        остановка
        иначе
        поменять местами V и  $V'$ 
  [2] ВСПЛЫТИЕ ( $V'$ ) для поддерева, корнем которого V
  [3] ПОГРУЖЕНИЕ ( $V'$ )
Конец ПОГРУЖЕНИЕ
```



Создание K - D -пирамиды аналогично случаю с 2- d пирамидой. Отличие лишь в том, что любой уровень описан по отношению к модулю k вместо 2. Поэтому K - D -пирамида из n элементов, где k константа, может быть создана за время $O(n)$. При этом требуется $O(1)$ времени для извлечения элемента с минимальным значением любого из k ключей, для вставки, удаления элемента с минимальным значением любого из k ключей, и удаления или изменения известного элемента требуется время $O(\log n)$.

На рис. 4.6 показана 3- d пирамида и результат удаления корня.



б)

Рис. 4.6. Удаление элемента с минимальным key_1 в 3-d пирамиде:
а) исходная пирамида; б) результирующая пирамида



Пошагово процедура выполняется следующим образом:

шаг 1. Удаляется элемент 1, на его место вставляется элемент 20. Сравниваются элементы 1 и 9 по key_1 . Основное свойство пирамиды не выполняется. Меняем элементы местами.

шаг 2. Сравниваются элементы 2 и 9 по key_2 . Основное свойство пирамиды выполняется.

шаг 3. Сравниваются элементы 4 и 9 по key_3 . Основное свойство пирамиды не выполняется. Меняем элементы местами.

шаг 4. Сравниваются элементы 9 и 19 по key_1 . Основное свойство пирамиды не выполняется. Меняем элементы местами.

Следует заметить, что скрытая константа при удалении экспоненциальна по k , что делает операции неэффективными, когда значение k слишком велико. (С другой стороны, трудоемкость операции вставки не зависит от k , а трудоемкость создания пирамиды пропорциональна k^2 [27]). Далее показано, что экспоненциальная зависимость от константы k может быть устранена после легкой модификации структуры.

Базовые возможности K - D -пирамиды могут быть расширены для поддержки других операций. Мы кратко представим некоторые из них.

Ранее было показано, что 2 - d пирамида может использоваться для представления min - max пирамид. В общем, если каждый элемент имеет n ключей $key_1, key_2, \dots, key_k$, мы можем (концептуально) включить еще k ключей $key_1, key_2, \dots, key_k$, где $key_i = -key_i$, где знак минус представляет общее функциональное отображение. Так, $2K$ - D -пирамида становится двусторонней k -мерной пирамидой. Заметим, что новые ключи не надо хранить. Вместо этого отображение может быть закодировано в операциях, поэтому не требуется дополнительной памяти.

Объединение базовых K - D -пирамид является сложной задачей. В особых случаях, бинарные пирамиды (1 - d пирамиды) требуют $O(\log^2 n)$ времени для объединения, а min - max пирамиды (особый тип 2 - d пирамид) – минимум $O(n)$.

С другой стороны, в [2] показано, что введение некоторого числа упрощений в основное свойство min - max пирамиды позволяет получить структуру, похожую на биномиальную очередь, объединение которой можно провести за $O(\log n)$. Особенно, если пирамида разобрана на набор блоков, каждый из которых содержит 2^i элементов с уникальными i , в форме полного бинарного дерева плюс единичный элемент. Для нечетных i полное бинарное дерево является упрощенной min - max (max - min) пирамидой, где слово «упрощенной» означает, что некоторые узлы не подчиняются основному свойству пирамиды. Уже было показано, как два узла одинакового размера были объединены за постоянное время, поэтому вся пирамида объединяется за логарифмическое время. Эта техника может быть обобщена до K - D -пирамид простой декомпозицией, и для полного бинарного дерева

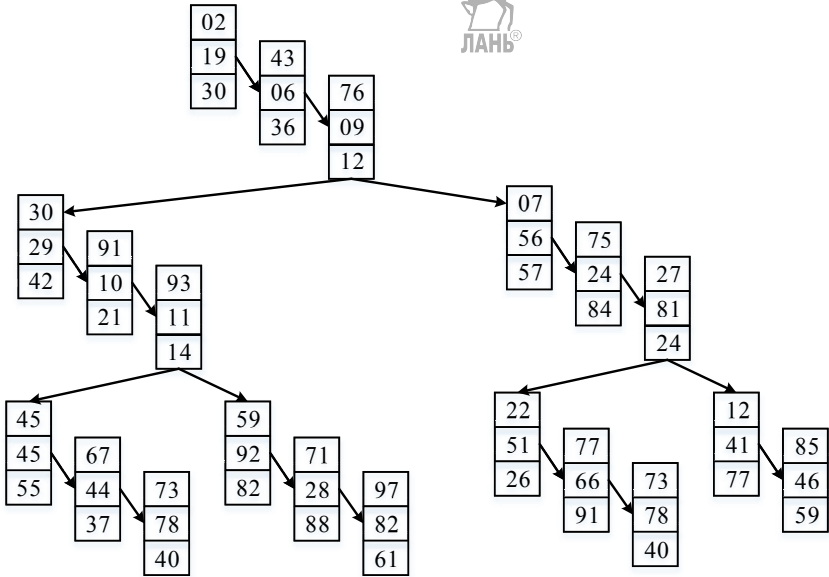
высоты i , корень будет иметь ключ $key_{mod(i, k)+1}$ в упрощенном порядке. Операции будут более сложными, нежели в случае с упрощенными *min-max* пирамидами, но общая сложность будет такой же, например, объединение (двусторонней) K - D -пирамиды точно также занимает логарифмическое время.

По определению, пирамида поддерживает эффективный доступ к элементу с наибольшим (и/или наименьшим) ключом. Часто это становится причиной ситуации, когда функция в домене из одного или более приоритетных отношений формирует новое приоритетное отношение и на основе него обращается к элементам. *Min-max* пирамида является частным случаем. В общем случае, все подобные предопределенные, функционально сформированные приоритеты могут быть закодированы в операции, и неявное хранение соответствующих «ключей» не является необходимостью. В этом случае, K - D -пирамиды могут быть использованы для представления обобщенных приоритетных очередей. Двусторонние пирамиды являются частным случаем обобщения.

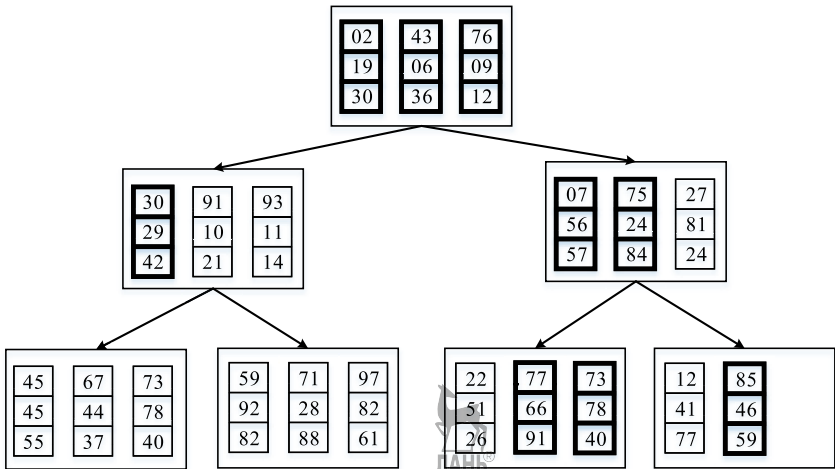
Пирамиды, которые мы недавно обсуждали, имеют очень малый порядок отношений, поэтому в любой части пирамиды, при нахождении минимального ключа в корне, второй наименьший ключ может находиться где угодно, среди первых k уровней потомков. Результатом этого является требование экспоненциального времени для удаления элемента. Когда k большое, в случае с двусторонними представлениями, этот недочет должен быть исправлен для того, чтобы данные структуры данных можно было применять на практике. Кратко опишем способ решения этой проблемы.

Вместо построения полного бинарного дерева для пирамиды, мы позволим каждому узлу на уровне i ($mod(i, k) \neq 0$) иметь не более одного потомка, в то время как узлы на уровнях i ($mod(i, k) = 0$) все еще могут иметь двух. Рис. 4.7(а) показывает модифицированную структуру для 3- d пирамиды, показанной на рис. 4.6(а). В случае возможности эффективного неявного представления каждая цепочка из k узлов может быть упакована в один узел, поэтому основное бинарное дерево все еще остается полным, как показано на рис. 4.7(б).

Очевидно, что операции погружения теперь требуется проверить последние $2k$ потомков для выбора минимума. С другой стороны, высота пирамиды почти в два раза меньше, поэтому время, затрачиваемое на операцию удаления, снизилось до $O(k^2)$. Стоимость операции вставки, однако, увеличилась до k . Также можно заметить небольшое увеличение времени для операции создания пирамиды (но все еще $O(k^2n)$).



a)



б)

Рис. 4.7. Улучшенная структура для больших значений k и ее представление

Хотя рис. 4.7 показывает модификацию для $k = 3$, такая структура предпочтительна только при больших k . На практике малые значения k (например, 2 или 3) встречаются чаще, а в таком случае оригинальная структура более эффективна.

Резюме

Мы представили K - D -пирамиду, структуру данных, эффективно представляющую многомерную приоритетную очередь, построенную без использования дополнительной памяти. Одна форма поддерживает вставку, удаление любого минимума, создание пирамиды за $O(\log n)$, $O(2^k \log n)$ и $O(k^2 n)$, соответственно, с особой простотой операций для $k = 2$, в то время как другие формы дают времена $O(k * \log n)$, $O(k^2 \log n)$ и $O(k^2 n)$. (Заметим, что для типичных значений k разница между 2^k и k^2 невелика). Более того, K - D -пирамида может быть расширена для поддержки двусторонних операций и операций слияния.

Представление K - D -пирамиды является очень простым. Мы закодировали основные операции на C , что заняло около 120 строк кода. Поэтому данная структура очень практична. Полное представление со сравнением производительности представлено в [26].

Остаются открытыми несколько связанных со структурой проблем. Можно улучшить ограничения для вставки и удаления минимального элемента до $O(k * \log n)$, используя только $O(n)$ дополнительного пространства. Мы не знаем, как этого можно достичь, используя небольшое дополнительное пространство.

Другой интересной проблемой является проблема использования межприоритетных отношений для улучшения эффективности операций над K - D -пирамидой. В случае двусторонней приоритетной очереди, мы увидели, что такие отношения дают значительное снижение сложности. Воздействие на другие представленные отношения также достойны изучения.



ЗАКЛЮЧЕНИЕ

В результате проведенной авторами работы впервые были систематизированы знания по многомерным структурам данных. Проведены исследования и дан подробный анализ различных многомерных структур в зависимости от условий решаемых задач. Рассмотрены многомерные структуры для работы в оперативной памяти и доступ к многомерным данным во внешней памяти. В отличие от классических структур, рассмотрено применение многомерных структур к различным видам объектов. Здесь также проведены исследования эффективности различных алгоритмов и даны рекомендации по их применению.

Рассмотренные многомерные структуры позволили существенно развить теоретические методы поиска информации в базах данных и различных поисковых и информационных системах.

Предполагается, что дальнейшее использование рассмотренных многомерных структур в различных системах достаточно перспективно и позволит создавать эффективное программное обеспечение на новом уровне.

Авторы надеются, что разработчики программного обеспечения более осознанно, а соответственно и более эффективно будут применять рассмотренные в работе многомерные структуры данных и алгоритмы их обработки. Научные работники и аспиранты, опираясь на эти знания, предложат новые подходы либо более эффективные структуры. Для студентов это прекрасная возможность сразу получить современные базовые знания по структурам данных и оценить их роль в решении задач автоматизации. Для преподавателей эта книга будет очень хорошим подспорьем при чтении различных курсов компьютерного направления.



СПИСОК ЛИТЕРАТУРЫ



1. Аверченков, В.И. Мониторинг и системный анализ информации в сети Интернет: монография / В.И. Аверченков, С.М. Рошин. – Брянск: БГТУ, 2006. – 160 с.
2. Вирт, Н. Алгоритмы и структуры данных. – М.: Мир, 1989.
3. Гулаков, В.К. Использование многомерных деревьев для обработки многомерной информации / В.К. Гулаков, А.О. Трубаков, Е.О. Трубаков // Вестник БГТУ № 3 (15), 2007. – С. 46–54.
4. Гулаков, В. К. Сокращение размерности данных методом сингулярного разложения / В.К. Гулаков, В.Н. Матюшин // Информационные технологии. Радиоэлектроника. Телекоммуникации (ITRT-2012) : сб. ст. II международной заочной научно-технической конференции. Ч. 1 / Поволжский гос. ун-т сервиса. – Тольятти : Изд-во ПВГУС, 2012. – С. 415–422.
5. Гулаков, В.К. Использование многомерного анализа изображений для систем определения брака на производстве / В.К. Гулаков, А.О. Трубаков // Тезисы международной научно-практической конференции «Состояние, проблемы и перспективы автоматизации технической подготовки производства на промышленных предприятиях». – Брянск, 2009. – С. 37.
6. Гулаков, В.К. Использование многомерных структур в темпоральных базах данных / В.К. Гулаков, А.О. Трубаков, П.В. Марченко // Тезисы международной конференции «Информационные технологии в образовании, технике и медицине». – Волгоград, 2009. – С. 58.
7. Касьянов, В. Н. Графы в программировании – обработка, визуализация и применение / В. Н. Касьянов, В.А. Евстигнеев. – СПб.: БХВ-Петербург, 2003. – 1104 с.
8. Кастельс, М. Информационная эпоха: экономика, общество и культура: [пер. с англ.] / М. Кастельс. – М.: ГУ ВШЭ, 2000.
9. Климов, С.М. Интеллектуальные ресурсы общества / С.М. Климов. – СПб: ИВЭСЭМ, Знание, 2002.
10. Кнут, Д. Искусство программирования. – Т. 3. Сортировка и поиск. – 2-е изд. : [пер. с англ.] / Д.Кнут. – М.: Издат. дом «Вильямс», 2000. – 832 с.

11. Кормен, Т. Х. Алгоритмы: построение и анализ. / Лейзерсон, Ч. И., Ривест, Р. Л., Штайн, К. // 2-е издание. : Пер. с англ. – М. : Издат. дом «Вильямс», 2005. – 1296 с.
12. Сковорцов, А.В. Глобальные алгоритмы построения R-деревьев // Геоинформатика: Теория и практика. Томск, 1998. – Вып. 1. – С. 67–83.
13. Трубаков, А.О. Многомерный подход при решении проблемы контекстного поиска изображений на основе гистограммного подхода / А.О. Трубаков // Тезисы конференции «Наука и производство». – Брянск, 2009. – С. 181–183.
14. Трубаков, А.О. Многомерная модель поиска изображений в хранилищах данных / А.О. Трубаков // Тезисы конференции «Информационные системы и технологии 2009». – Обнинск, 2009. – С. 166–167.
15. Уоррен, Г. Алгоритмические трюки для программистов : [пер. с англ.] / Г. Уоррен. – М.: Вильямс, 2004. – 288 с.
16. An Introduction to Multidimensional Database Technology. – Kenan Systems Corporation, 1995.
17. Abel, D.J. A data structure and algorithm based on a linear key for a rectangle retrieval problem / D.J. Abel, J.L. Smith // Computer Vision 24, 1983. – P. 1–13.
18. Atkinson, M. Min-Max Heaps and Generalized Priority Queues / M. Atkinson, J. Sack, N. Santoro, T. Strothotte // Comm. ACM, Vol. 29, 1986. – P. 996–1000.
19. Bentley, J.L. Multidimensional binary search tree used for associative searching // Communications of the ACM 18(9), 1975 – P. 509–517.
20. Beckmann, N. The R*-tree: An efficient and robust access method for points and rectangles / N. Beckmann, H.-P. Kriegel, R. Schneider, B. Seeger // ACM SIGMOD International Conference on Management of Data, – 1990. – P. 322–331.
21. Bentley, J.L. Quad trees: A data structure for retrieval of composite keys. / J.L. Bentley, R. Finkel // Acta Informatica 4(1), 1974. – P. 1–9.
22. Bentley, J.L. Data structures for range searching / J. L. Bentley, J. H. Friedman // ACM Computing Survey Engineering 4(5), 1979. – P. 397–409.
23. Chung, K. Efficient algorithms for coding Hilbert curve of arbitrary-sized image and application to window query / K. Chung, Y. Huang, Y. Liu // Information Sciences 177, 2007.
24. Cédric du Mouza, Witold Litwin, Philippe Rigaux. SDR-tree: A Scalable Distributed R-tree // ICDE, 2007. – P. 296–305.
25. Ding, Y., Weiss, M. The Relaxed Min-Max Heap: A Mergeable Double-Ended Priority Queue // Acta Informatica, Vol.30 (1993), to appear.

26. Ding, Y. Efficient Implementations of Multi-dimensional Priority Queues. / Y. Ding, M. A. Weiss // School of Computer Science Technical Report, Florida International University, Feb. 1993.
27. Ding, Y. The K-D heap: An efficient multi-dimensional priority queue / Y. Ding, M. A. Weiss // In: Proc. 3rd Workshop on Algorithms and Data Structures. Lecture Notes in Computer Science, vol. 709. – P. 302–313. Springer (1993).
28. Faloutsos, C. Multiattribute hashing using Gray-codes / C. Faloutsos // In Proc. ACM SIGMOD Int. Conf. on Management of Data, 1986 – P. 227–238.
29. Faloutsos, C. Gray-codes for partial match and range queries / C. Faloutsos // IEEE Trans. Software Eng. 14, 1988 – P. 1381–1393.
30. Fukunaga, K. Introduction to Statistical Pattern Recognition / K. Fukunaga // New York: Academic Press, 1990.
31. Fuchs, H. Near real time shaded display of rigid objects / H. Fuchs, G. D. Abram, E.D. Grant // Computer Graphics. Vol. 17 (3), 1983 – P. 65–72.
32. Finkel, Quad-trees: A data structure for retrieval of composite keys / R. Finkel, J. L. Bentley // Acta Informatica. Vol. 4(1), 1974 – P. 1–9.
33. Faloutsos, C. Fractals for secondary key retrieval / C. Faloutsos, S. Roseman // In Proc. 8th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems, 1989 – P. 247–252.
34. Faloutsos, C. DOT: A spatial access method using fractals / C. Faloutsos, Y. Rong // In Proc. 7th IEEE Int. Conf. on Data Eng., 1991 – P. 152–159.
35. Fukunaga, K. Introduction to Statistical Pattern Recognition / K. Fukunaga // New York: Academic Press, 1990.
36. Guttman, A. R-trees: A dynamic index structure for spatial searching // ACM SIGMOD International Conference on Management of Data, 1984 – P. 47–54.
37. Gunther, O. Efficient Structures for Geometric Data Management / O. Gunther // Number 337 in LNCS. Berlin/Heidelberg/New York: Springer-Verlag, 1988.
38. Greene, D. An implementation and performance analysis of spatial data access methods // 5th IEEE International Conference on Data Engineering, 1989 – P. 606–615.
39. Gunther, O. Spatial database indices for large extended objects / O. Gunther, H. Noltemeier // In Proc. 7th IEEE Int. Conf. on Data Eng., 1991 – P. 520–526.
40. Gaede, V. Spatial access methods and query processing in the object-oriented GIS / V. Gaede, W.-F. Rieker // GODOT. In Proc. of the AGDM'94 Workshop, Delft, The Netherlands, 1994 – P. 40–52.
41. Guéting R.H. Multidimensional D-tree: an efficient dynamic file structure for exact match queries / R. H. Guéting, H. P. Kriegel // Proc. 1 Oth G 1

- Annual Conf., Informatic Fachberichte. – Springer Verl., 1980. – P. 375–388.
42. Hinrichs, K. Implementation of the grid file: Design concepts and experience / K. Hinrichs // BIT 25, 1985 – P. 569–592.
 43. Hilbert, D. Ueber die stetige Abbildung einer Linie auf ein Flächenstück / D. Hilbert // Mathematischen Annalen, 1891 – P. 459–460.
 44. Henrich, A. Adapting the Transformation Technique to Maintain Multi-Dimensional Non-Point Objects in k-d-Tree Based Access Structures / A. Henrich // In Proc. 3rd ACM Int. Workshop on Advances in Geographic Information Systems (ACM-GIS'95), Baltimore, Maryland, USA. ACM Press, 1995.
 45. Hutflesz, A. Twin grid files: Space optimizing access schemes / A. Hutflesz, H.-W. Six, P. Widmayer // ACM SIGMOD International Conference on Management of Data, 1988 – P. 183–190.
 46. Henrich, A. The LSD tree: Spatial access to multidimensional point and non-point objects / A. Henrich, H.-W. Six, P. Widmayer // Fifteenth International Conference on Very Large Data Bases, 1989 – P. 45–53.
 47. Jagadish, H.V. Linear clustering of objects with multiple attributes / H.V. Jagadish // In Proc. ACM SIGMOD Int. Conf. on Management of Data, 1990 – P. 332–342.
 48. Jagadish, H.V. A retrieval technique for similar shapes. / H.V. Jagadish // Proceedings of the ACM SIGMOD Conference, Denver, CO, 1991.
 49. Kronacker, M. High-concurrency locking in R-trees / M. Kronacker, D. Banks // 21th International Conference on Very Large Data Bases, 1995 – P. 134–145.
 50. Kamel, I. Hilbert R-tree: An improved R-tree using fractals / I. Kamel, C. Faloutsos // Twentieth International Conference on Very Large Data Bases, 1994 – P. 500–509.
 51. Kriegel, H.-P. Multidimensional order preserving linear hashing with partial expansions / H.-P. Kriegel, B. Seeger // In Proceedings of the International Conference on Database Theory, LNCS 243, 1986.
 52. Kriegel, H.-P. Multidimensional quantile hashing is very efficient for non-uniform record distributions / H.-P. Kriegel, B. Seeger // In Proceedings of the Third IEEE International Conference on Data Engineering, 1987 – P. 10–17.
 53. Kriegel, H.-P. PLOP-hashing: A grid file without directory / H.-P. Kriegel, B. Seeger // In Proceedings of the Fourth IEEE International Conference on Data Engineering, 1988 – P. 369–376.
 54. Katayama, N. The sr-tree: an index structure for highdimensional nearest neighbor queries / N. Katayama, S. Satoh // ACM SIGMOD international conference on Management of data, 1997 – P. 369–380.
 55. Iyengar S.S. a. e. Multidimensional data structures: review and outlook // Adv. Comput. – 1988. – Vol. 27. – P. 69–119.

56. Kwong, Y.S. r-trees: a variant of D-trees / Y.S. Kwong, D. Wood // . – Corp. Sei. Tech. Rep. 78-CS-1 8, McMaster Univ., Ontario, 1978.
57. Maruyama K. Index structures for virtual memory — comparison between B-trees and M-trees//IBM Res. Rep. RC 5258, 1975.
58. Litwin, W. Linear hashing: a new tool for file and table addressing / W. Litwin // 6th International Conference on VLDB, 1980 – P. 212–223.
59. Larson, P.-A. Linear hashing with partial expansions / P.-A. Larson // 6th International Conference on VLDB, 1980 – P. 224–232.
60. Lawder, J. K. Using State Diagrams for Hilbert Curve Mappings / J. K. Lawder // Technical Report JL2/00, Birkbeck College, University of London, 2000.
61. Lin, K. I. The tv-tree: an index structure for highdimensional data / K. I. Lin, H. V. Jagadish, C. Faloutsos // The VLDB Journal, 3(4), 1994 – P. 517–542.
62. Lawder, J.K. Querying Multi-Dimensional Data Indexed Using the Hilbert Space-Filling Curve / J.K. Lawder, P.J.H. King // ACM SIGMOD Record, Vol. 30, No. 1, March, 2001 – P. 19–24.
63. Lomet, D. B. The hB-tree: A robust multiattribute search structure / D. B. Lomet, B. Salzberg // 5th IEEE International Conference on Data Engineering, 1989 – P. 296–304.
64. Mokbel, M. F. Spatio-temporal Access Methods / M. F. Mokbel, T. M. Ghanem, W. G. Aref // IEEE Data Engineering Bulletin. Vol. 26(2), 2003 – P. 40–49.
65. Matsuyama, T. A file organization for geographic information systems based on spatial proximity / T. Matsuyama, L.V. Hao, M. Nagao // Int. J. Comp. Vision, Graphics and Image Processing. Vol. 26 (3), 1984 – P. 303–318.
66. Niblack, W. The qbic project: Querying images by content using color, texture, and shape / W. Niblack, R. Barber, W. Equitz, M. Flickner, E. Glasman, D. Petkovic, P. Yanker, C. Faloutsos, G. Taubin // SPIE 1993 International Symposium on Electronic Imaging: Science and Technology Conference 1908, Storage and Retrieval for Image and Video Databases, San Jose, CA, 1993.
67. Ng, V. Concurrent accesses to R-trees / V. Ng, T. Kameda // Advances in Spatial Databases, 1993 – P. 142–161.
68. Nievergelt, J. The grid file: An adaptable, symmetric multikey file structure. / J. Nievergelt, H. Hinterberger, K. C. Sevcik // 3rd ECI Conference, 1981 – P. 236–251.
69. Otoo, E.J. A mapping function for the directory of a multidimensional extendible hashing / E.J. Otoo // 10th International Conference on VLDB, 1984 – P. 491–506.
70. Oosterom, P. The Reactive data structures for geographic information systems // Ph.D. Thesis, University of Leiden, 1990.

71. Ooi, B.C. Efficient Query Processing in Geographic Information Systems / B.C. Ooi // Number 471 in LNCS. Berlin/Heidelberg/New York: Springer-Verlag, 1990.
72. Orenstein, J. A class of data structures for associative searching / J. Orenstein, T. H. Merrett // In Proc. 3rd ACM SIGACT-SIGMOD Symp. on Principles of Database Systems, 1984 – P. 181–190.
73. Pagel, B. U. The transformation technique for spatial objects revisited / B. U. Pagel, H.-W. Six, H. Toben // Advances in Spatial Databases, Number 692 in LNCS, Berlin/Heidelberg/New York, 1993 – P. 73–88.
74. Robinson, J.T. The K-D-B-tree: A search structure for large multidimensional dynamic indexes // ACM SIGMOD International Conference on Management of Data, 1981 – P. 10–18.
75. Roussopoulos, N. Direct spatial search on pictorial databases using packed R-trees / N. Roussopoulos, D. Leifker // ACM SIGMOD International Conference on Management of Data, 1985 – P. 17–31.
76. Samet, H. The design and analysis of spatial data structures / H. Samet // Reading, MA: Addison-Wesley, 1989.
77. Samet, H. Applications of Spatial Data Structures. // Reading, MA: Addison-Wesley, 1990.
78. Foundations of Multidimensional and Metric Data Structures. Samet, Hanan / Imprint: MORGAN KAUFFMAN, 2006. – 1024 p.
79. Sagan, H. Space-Filling Curves / H. Segan // Berlin/Heidelberg/New York: Springer-Verlag, 1994.
80. Seeger, B. Techniques for design and implementation of spatial access methods / B. Seeger, H.-P. Kriegel // In Proc. 14th Int. Conf. on Very Large Data Bases, 1988 – P. 360–371.
81. Seeger, B. and H.-P. Kriegel. The buddy-tree: An efficient and robust access method for spatial data base systems / B. Seeger, H.-P. Kriegel // 16th International Conference on Very Large Data Bases, 1990 – P. 590–601.
82. Shekhar, S. CCAM: A connectivity-clustered access method for aggregate queries on transportation networks: A summary of results / S. Shekhar, D.-R. Liu // In Proc. 11th IEEE Int. Conf. on Data Eng., 1995 – P. 410–419.
83. Sellis, T. The R+-tree: A dynamic index for multi-dimensional objects / T. Sellis, N. Roussopoulos, C. Faloutsos // In Proc. 13th Int. Conf. on Very Large Data Bases, 1987 – P. 507–518.
84. Stonebraker, M. An analysis of rule indexing implementations in data base systems / M. Stonebraker, T. Sellis, E. Hanson // In Proc. 1st Int. Conf. on Expert Data Base Systems, 1986.
85. Six, H. Spatial searching in geometric databases / H. Six, P. Widmayer // In Proc. 4th IEEE Int. Conf. on Data Eng., 1988 – P. 496–503.
86. Tamminen, M. The extendible cell method for closest point problems / M. Tamminen // BIT 22, 1982 – P. 27–41.

87. Tamminen, M. Comment on quad- and octrees // Communications of the ACM 30(3), 1984 – P. 204–212.
88. Vaishnavi V.K. Multidimensional height-balanced trees / AEEE Trans. Connut. — 1984. — Vol. C-33, N 4. — P. 334–343.
89. White, M. N-trees: Large ordered indexes for multi-dimensional space / M. White // Technical report, Application Mathematics Research Staff, Statistical Research Division, US Bureau of the Census, 1981.
90. White, D.A. Similarity indexing with the ss-tree / D. A. White, R. Jain // Twelfth International Conference on Data Engineering, 1996 – P. 516–523.
91. Whang, K.-Y. Multilevel grid files / K.-Y. Whang, R. Krishnamurthy // Yorktown Heights, NY: IBM Research Laboratory, 1985.
92. Bayer, F. Organization and maintenance of large ordered indexes / F. Bayer, E. McCreight // Acta Informatica 1, 3, 1972 – P. 173–189.



*Василий Константинович ГУЛАКОВ,
Андрей Олегович ТРУБАКОВ,
Евгений Олегович ТРУБАКОВ*

СТРУКТУРЫ И АЛГОРИТМЫ ОБРАБОТКИ МНОГОМЕРНЫХ ДАННЫХ

Монография

Издание второе, стереотипное

Зав. редакцией литературы по информационным
технологиям и системам связи *О. Е. Гайнутдинова*



ЛР № 065466 от 21.10.97
Гигиенический сертификат 78.01.10.953.П.1028
от 14.04.2016 г., выдан ЦГСЭН в СПб

Издательство «ЛАНЬ»
lan@lanbook.ru; www.lanbook.com;
196105, Санкт-Петербург, пр. Юрия Гагарина, 1, лит. А.
Тел.: (812) 412-92-72, 336-25-09.
Бесплатный звонок по России: 8-800-700-40-71

Подписано в печать 02.04.21.
Бумага офсетная. Гарнитура Школьная. Формат 60×90¹/₁₆.
Печать офсетная. Усл. п. л. 22,25. Тираж 30 экз.

Заказ № 385-21.

Отпечатано в полном соответствии
с качеством предоставленного оригинал-макета
в АО «Т8 Издательские Технологии».
109316, г. Москва, Волгоградский пр., д. 42, к. 5.