УДК 004(075.8)
ББК 32.81я73
М99

Рецензенты:
И.Ю. Бринк, зав. кафедрой «Моделирование, конструирование и дизайн» института сферы обслуживания и предпринимательства (филиала) Донского государственного технического университета в г. Шахты, почетный работник науки и техники РФ, президент некоммерческого партнерства «Инновационно-технологический центр «ИнТех-Дон», д-р техн. наук, проф.,
А.В. Кузнецова, канд. техн. наук, доц.

Мясникова, Нелли Александровна.
М99 Алгоритмы и структуры данных : учебное пособие / Н.А. Мясникова. — Москва : КНОРУС, 2021. — 186 с. — (Бакалавриат).

ISBN 978-5-406-04167-3

Представлены основные положения и типовые решения по конструированию, созданию сложных структур данных (линейных и древовидных), необходимость в которых возникает при решении различных практических задач. Приведены классические алгоритмы обработки данных (сортировка и поиск) и примеры работы этих алгоритмов, что позволяет получить практические навыки по использованию основных алгоритмов обработки данных.

Соответствует ФГОС ВО последнего поколения.

Для студентов высших учебных заведений, обучающихся по направлениям «Информатика и вычислительная техника», «Программная инженерия», «Прикладная информатика». Может быть полезно для разработчиков программного обеспечения вычислительной техники и автоматизированных систем.

УДК 004(075.8)
ББК 32.81я73

Мясникова Нелли Александровна
АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ

Изд. № 586326. Формат 60×90/16. Усл. печ. л. 12,0.

ООО «Издательство «КноРус».
117218, г. Москва, ул. Кедрова, д. 14, корп. 2.
Тел.: +7 (495) 741-46-28.
E-mail: welcome@knorus.ru www.knorus.ru

Отпечатано в АО «Т8 Издательские Технологии».
109316, г. Москва, Волгоградский проспект, д. 42, корп. 5.
Тел.: +7 (495) 221-89-80.

ISBN 978-5-406-04167-3

© Мясникова Н.А., 2021
© ООО «Издательство «КноРус», 2021

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Без понимания структур данных и алгоритмов невозможно создать серьезный программный продукт. Знание теории структур, методов представления данных на логическом и физическом уровнях, а также эффективных алгоритмов обработки различных структур данных, умение проводить сравнительный анализ и оценку эффективности применяемых алгоритмов совершенно необходимы для высококвалифицированных программистов.

Решение любой задачи сводится к обработке множества данных, которые представляют собой абстракцию некоторой части реального мира. Для решения конкретной задачи с использованием вычислительной техники необходимо, во-первых, выбрать подходящий уровень абстрагирования, т.е. определить множество данных, представляющих реальную ситуацию и относящуюся к реальной задаче. И, во-вторых, необходимо выбрать способ представления этих данных с учетом возможностей языка программирования и вычислительной техники.

При выборе представления данных необходимо учитывать, какие алгоритмы обработки будут к ним применяться. В то же время выбор алгоритма, в свою очередь, существенно зависит от строения данных. В связи с этим структура данных и структура программы связаны между собой, причем данные предшествуют алгоритму, так как прежде чем выполнять какие-либо операции, нужно определить объекты, к которым они применяются.

Решение одной и той же задачи можно получить с помощью различных алгоритмов, поэтому при разработке программных продуктов важное место занимает оценка эффективности применяемых алгоритмов. Это требует умения проводить сравнительный анализ алгоритмов и правильно выбирать из них наиболее эффективные алгоритмы для решения поставленной задачи.

Таким образом, знание структур данных, методов представления данных на логическом и физическом уровнях, а также эффективных алгоритмов обработки различных структур данных совершенно необходимо для высококвалифицированных прикладных и системных программистов, разработчиков информационных систем различной степени сложности, автоматизированных систем управления, трансляторов, операционных систем, систем управления базами данных, систем искусственного интеллекта и др.

Материал учебного пособия соответствует требованиям государственного образовательного стандарта к дисциплине «Структуры и алгоритмы компьютерной обработки данных» и включает шесть глав, которые охватывают все изучаемые темы. В Главе 1 излагаются общие сведения о структурах данных и алгоритмах, приводится классификации структур данных. Глава 2 содержит материалы по линейным связанным структурам (стеки, очереди, циклические списки). Рассмотрены вопросы представления этих структур, операции над ними и алгоритмы их реализации. В Главе 3 рассматриваются такие нелинейные

ВВЕДЕНИЕ

Без понимания структур данных и алгоритмов невозможно создать серьезный программный продукт. Знание теории структур, методов представления данных на логическом и физическом уровнях, а также эффективных алгоритмов обработки различных структур данных, умение проводить сравнительный анализ и оценку эффективности применяемых алгоритмов совершенно необходимы для высококвалифицированных программистов.

Решение любой задачи сводится к обработке множества данных, которые представляют собой абстракцию некоторой части реального мира. Для решения конкретной задачи с использованием вычислительной техники необходимо, во-первых, выбрать подходящий уровень абстрагирования, т.е. определить множество данных, представляющих реальную ситуацию и относящуюся к реальной задаче. И, во-вторых, необходимо выбрать способ представления этих данных с учетом возможностей языка программирования и вычислительной техники.

При выборе представления данных необходимо учитывать, какие алгоритмы обработки будут к ним применяться. В то же время выбор алгоритма, в свою очередь, существенно зависит от строения данных. В связи с этим структура данных и структура программы связаны между собой, причем данные предшествуют алгоритму, так как прежде чем выполнять какие-либо операции, нужно определить объекты, к которым они применяются.

Решение одной и той же задачи можно получить с помощью различных алгоритмов, поэтому при разработке программных продуктов важное место занимает оценка эффективности применяемых алгоритмов. Это требует умения проводить сравнительный анализ алгоритмов и правильно выбирать из них наиболее эффективные алгоритмы для решения поставленной задачи.

Таким образом, знание структур данных, методов представления данных на логическом и физическом уровнях, а также эффективных алгоритмов обработки различных структур данных совершенно необходимо для высококвалифицированных прикладных и системных программистов, разработчиков информационных систем различной степени сложности, автоматизированных систем управления, трансляторов, операционных систем, систем управления базами данных, систем искусственного интеллекта и др.

Материал учебного пособия соответствует требованиям государственного образовательного стандарта к дисциплине «Структуры и алгоритмы компьютерной обработки данных» и включает шесть глав, которые охватывают все изучаемые темы. В Главе 1 излагаются общие сведения о структурах данных и алгоритмах, приводится классификации структур данных. Глава 2 содержит материалы по линейным связанным структурам (стеки, очереди, циклические списки). Рассмотрены вопросы представления этих структур, операции над ними и алгоритмы их реализации. В Главе 3 рассматриваются такие нелинейные

Толковый словарь исходных терминов

Алгоритм – конечная последовательность предписаний, исполнение которых позволяет получить решение некоторой задачи.

Атрибут – отдельная специфическая существенная характеристика объекта.

Данные – информация, представленная в виде, пригодном для обработки автоматическими средствами при возможном участии человека.

Идентификатор объекта – элемент данных, однозначно определяющий объект внутри системы.

Информация – сведения, являющиеся объектом хранения, передачи и обработки.

Информационная система – система, предназначенная для решения задач поиска (информационно-поисковая система) или логической обработки информации (информационно-логическая система).

Ключ – данные, определяющие возможность доступа к другим данным.

Код – набор знаков в совокупности со схемой кодирования для представления информации в виде данных.

Множество – совокупность каких-либо объектов, обладающих характеристическим свойством, общим для всех этих объектов.

Объект данных – любой элемент данных, хранимый в базе данных и дающий информацию о конкретных объектах или явлениях реального мира. Связи между объектами данных в базе данных определены отношениями.

Поиск данных – операция обнаружения местоположения данных, осуществляемая посредством анализа каждого объекта по определенным критериям.

Рекурсия – способ описания процессов через самих себя.

Сортировка – упорядочение совокупности объектов в соответствии с заданным отношением порядка.

Список (списковая структура) – набор элементов данных, связанных друг с другом таким образом, что каждый элемент содержит адрес последующего элемента.

Ссылка – часть объекта, предназначенная для указания местонахождения другого объекта.

Структура – совокупность связей между компонентами системы, объекта, программы, обеспечивающих их целостность.

Структура данных – множество элементов данных, упорядоченных одним из принятых способов.

Тип данных – множество значений, которые могут принимать переменная или выражение в совокупности с множеством операций, допустимыми над этими значениями.

Файл – набор данных на логическом уровне рассмотрения.

Язык программирования – искусственный язык для представления программ.

ГЛАВА 1
СТРУКТУРЫ ДАННЫХ И АЛГОРИТМЫ

1.1. Понятие структур данных и алгоритмов

Структуры данных и алгоритмы служат теми материалами, из которых строятся программы. Более того, сам компьютер состоит из структур данных и алгоритмов. Встроенные структуры данных представлены теми регистрами и словами памяти, где хранятся двоичные величины. Заложенные в конструкцию аппаратуры алгоритмы – это воплощенные в электронных логических цепях жесткие правила, по которым занесенные в память данные интерпретируются как команды, подлежащие исполнению. Поэтому в основе работы всякого компьютера лежит умение оперировать только с одним видом данных – с отдельными битами, или двоичными цифрами. Работает же с этими данными компьютер только в соответствии с неизменным набором алгоритмов, которые определяются системой команд центрального процессора.

Задачи, которые решаются с помощью компьютера, редко выражаются на языке битов. Как правило, данные имеют форму чисел, литер, текстов, символов и более сложных структур типа последовательностей, списков и деревьев. Еще разнообразнее алгоритмы, применяемые для решения различных задач; фактически алгоритмов не меньше чем вычислительных задач.

Для точного описания абстрактных структур данных и алгоритмов программ используются такие системы формальных обозначений, называемые языками программирования, в которых смысл всякого предложения определяется точно и однозначно. Среди средств, представляемых почти всеми языками программирования, имеется возможность ссылаться на элемент данных, пользуясь присвоенным ему именем (идентификатором). Одни именованные величины являются константами, которые сохраняют постоянное значение в той части программы, где они определены, другие – переменными, которым с помощью оператора в программе может быть присвоено любое новое значение. Но до тех пор, пока программа не начала выполняться, их значение не определено.

Имя константы или переменной помогает программисту, но компьютеру оно ни о чем не говорит. Компилятор же, транслирующий текст программы в двоичный код, связывает каждый идентификатор с определенным адресом памяти. Но для того чтобы компилятор смог это выполнить, нужно сообщить о «типе» каждой именованной величины. Человек, решающий какую-нибудь за-

дачу «вручную», обладает интуитивной способностью быстро разобраться в типах данных и тех операциях, которые для каждого типа справедливы. Так, например, нельзя извлечь квадратный корень из слова или написать число с заглавной буквы. Одна из причин, позволяющих легко провести такое распознавание, состоит в том, что слова, числа и другие обозначения выглядят поразному. Однако, для компьютера все типы данных сводятся в конечном счете к последовательности битов, поэтому различие в типах следует делать явным.

Типы данных, принятые в языках программирования, включают натуральные и целые числа, вещественные (действительные) числа (в виде приближенных десятичных дробей), литеры, строки и т.п.

В некоторых языках программирования тип каждой константы или переменной определяется компилятором по записи присваиваемого значения; наличие десятичной точки, например, может служить признаком вещественного числа. В других языках требуется, чтобы программист явно задал тип каждой переменной, и это дает одно важное преимущество. Хотя при выполнении программы значение переменной может многократно меняться, тип ее меняться не должен никогда; это значит, что компилятор может проверить операции, выполняемые над этой переменной, и убедиться в том, что все они согласуются с описанием типа переменной. Такая проверка может быть проведена путем анализа всего текста программы, и в этом случае она охватит все возможные действия, определяемые данной программой.

В зависимости от назначения языка программирования защита типов, осуществляемая на этапе компиляции, может быть более или менее жесткой. Так, например, язык PASCAL, изначально являвшийся прежде всего инструментом для иллюстрирования структур данных и алгоритмов, сохраняет от своего первоначального назначения весьма строгую защиту типов. PASCAL-компилятор в большинстве случаев расценивает смешение в одном выражении данных разных типов или применение к типу данных несвойственных ему операций как фатальную ошибку. Напротив, язык С, предназначенный, прежде всего для системного программирования, является языком с весьма слабой защитой типов. С-компиляторы в таких случаях лишь выдают предупреждения. Отсутствие жесткой защиты типов дает системному программисту, разрабатывающему программу на языке С, дополнительные возможности, но такой программист сам отвечает за правильность своих действий.

Структура данных относится, по существу, к «пространственным» понятиям: ее можно свести к схеме организации информации в памяти компьютера. Алгоритм же является соответствующим процедурным элементом в структуре программы – он служит рецептом расчета.

Первые алгоритмы были придуманы для решения численных задач типа умножения чисел, нахождения наибольшего общего делителя, вычисления тригонометрических функций и других. Сегодня в равной степени важны и нечисленные алгоритмы; они разработаны для таких задач, как, например, поиск в тексте заданного слова, планирование событий, сортировка данных в

указанном порядке и т.п. Нечисленные алгоритмы оперируют с данными, которые не обязательно являются числами; более того, не нужны никакие глубокие математические понятия, чтобы их конструировать или понимать. Из этого, однако, вовсе не следует, что в изучении таких алгоритмов математике нет места; напротив, точные, математические методы необходимы при поиске наилучших решений нечисленных задач при доказательстве правильности этих решений.

Структуры данных, применяемые в алгоритмах, могут быть чрезвычайно сложными. В результате выбор правильного представления данных часто служит ключом к удачному программированию и может в большей степени сказываться на производительности программы, чем детали используемого алгоритма. Вряд ли когда-нибудь появится общая теория выбора структур данных. Самое лучшее, что можно сделать, – это разобраться во всех базовых «кирпичиках» и в собранных из них структурах. Способность приложить эти знания к конструированию больших систем – это прежде всего дело инженерного мастерства и практики.

1.2. Информация и ее представление в памяти

Начиная изучение структур данных или информационных структур, необходимо ясно установить, что понимается под информацией, как информация передается и как она физически размещается в памяти вычислительной машины.

1.2.1. Природа информации

Можно сказать, что решение каждой задачи с помощью вычислительной машины включает запись информации в память, извлечение информации из памяти и манипулирование информацией. Можно ли измерить информацию?

В теоретико-информационном смысле информация рассматривается как мера разрешения неопределенности. Предположим, что имеется n возможных состояний какой-нибудь системы, в которой каждое состояние имеет вероятность появления p , причем все вероятности независимы. Тогда неопределенность этой системы определяется в виде:

$$H = -\sum_{i=1}^{n} (p(i) * \log_2(p(i))) \qquad (1.1)$$

Для измерения неопределенности системы выбрана специальная единица, называемая битом. Бит является мерой неопределенности (или информации), связанной с наличием всего двух возможных состояний, таких, как, например, истинно/ложно или да/нет. Бит используется для измерения как неопределенности, так и информации. Это вполне объяснимо, поскольку количество полученной информации равно количеству неопределенности, устраненному в результате получения информации.

1.2.2. Хранение информации

В цифровых вычислительных машинах можно выделить три основных вида запоминающих устройств: сверхоперативная, оперативная и внешняя память. Обычно сверхоперативная память строится на регистрах. Регистры используются для временного хранения и преобразования информации.

Некоторые из наиболее важных регистров содержатся в центральном процессоре компьютера. Центральный процессор содержит регистры (иногда называемые аккумуляторами), в которые помещаются аргументы (т.е. операнды) арифметических операций. Сложение, вычитание, умножение и деление занесенной в аккумуляторы информации выполняется с помощью очень сложных логических схем. Кроме того, с целью проверки необходимости изменения нормальной последовательности передачи управления, в аккумуляторах могут анализироваться отдельные биты. Кроме запоминания операндов и результатов арифметических операций, регистры используются также для временного хранения команд программы и управляющей информации о номере следующей выполняемой команды.

Оперативная память предназначена для запоминания более постоянной по своей природе информации. Важнейшим свойством оперативной памяти является адресуемость. Это означает, что каждая ячейка памяти имеет свой идентификатор, однозначно идентифицирующий ее в общем массиве ячеек памяти. Этот идентификатор называется адресом. Адреса ячеек являются операндами тех машинных команд, которые обращаются к оперативной памяти. В подавляющем большинстве современных вычислительных систем единицей адресации является байт – ячейка, состоящая из 8 двоичных разрядов. Определенная ячейка оперативной памяти или множество ячеек могут быть связаны с конкретной переменной в программе. Однако для выполнения арифметических вычислений, в которых участвует переменная, необходимо, чтобы до начала вычислений значение переменной было перенесено из ячейки памяти в регистр. Если результат вычисления должен быть присвоен переменной, то результирующая величина снова должна быть перенесена из соответствующего регистра в связанную с этой переменной ячейку оперативной памяти.

Во время выполнения программы ее команды и данные в основном размещаются в ячейках оперативной памяти. Полное множество элементов оперативной памяти часто называют основной памятью.

Внешняя память служит прежде всего для долговременного хранения данных. Характерным для данных на внешней памяти является то, что они могут сохраняться там даже после завершения создавшей их программы и могут быть впоследствии многократно использованы той же программой при повторных ее запусках или другими программами. Внешняя память используется также для хранения самих программ, когда они не выполняются. Поскольку стоимость внешней памяти значительно меньше оперативной, а объем значительно больше, то еще одно назначение внешней памяти – временное хранение тех кодов и

данных выполняемой программы, которые не используются на данном этапе ее выполнения. Активные коды выполняемой программы и обрабатываемые ею на данном этапе данные должны обязательно быть размещены в оперативной памяти, так как прямой обмен между внешней памятью и операционными устройствами (регистрами) невозможен.

Как хранилище данных, внешняя память обладает в основном теми же свойствами, что и оперативная, в том числе и свойством адресуемости. Поэтому в принципе структуры данных на внешней памяти могут быть теми же, что и в оперативной, и алгоритмы их обработки могут быть одинаковыми. Но внешняя память имеет совершенно иную физическую природу, для нее применяются (на физическом уровне) иные методы доступа, и этот доступ имеет другие временные характеристики. Это приводит к тому, что структуры и алгоритмы, эффективные для оперативной памяти, не оказываются таковыми для внешней памяти.

1.3. Классификация структур данных

Теперь можно дать более конкретное определение данного на машинном уровне представления информации.

Независимо от содержания и сложности любые данные в памяти ЭВМ представляются последовательностью двоичных разрядов, или битов, а их значениями являются соответствующие двоичные числа. Данные, рассматриваемые в виде последовательности битов, имеют очень простую организацию или, другими словами, слабо структурированы. Для человека описывать и исследовать сколько-нибудь сложные данные в терминах последовательностей битов весьма неудобно. Более крупные и содержательные, нежели бит, «строительные блоки» для организации произвольных данных получаются на основе понятия «структуры данного».

Под *структурой данных* в общем случае понимают множество элементов данных и множество связей между ними. Такое определение охватывает все возможные подходы к структуризации данных, но в каждой конкретной задаче используются те или иные его аспекты. Поэтому вводится дополнительная классификация структур данных, направления которой соответствуют различным аспектам их рассмотрения. Прежде чем приступать к изучению конкретных структур данных, дадим их общую классификацию по нескольким признакам.

Понятие *«физическая структура данных»* отражает способ физического представления данных в памяти машины и называется еще структурой хранения, внутренней структурой или структурой памяти.

Рассмотрение структуры данных без учета ее представления в машинной памяти называется *абстрактной* или *логической* структурой. В общем случае между логической и соответствующей ей физической структурами существует различие, степень которого зависит от самой структуры и особенностей той среды, в которой она должна быть отражена. Вследствие этого различия суще-

ствуют процедуры, осуществляющие отображение логической структуры в физическую и, наоборот, физической структуры в логическую. Эти процедуры обеспечивают, кроме того, доступ к физическим структурам и выполнение над ними различных операций, причем каждая операция рассматривается применительно к логической или физической структуре данных.

Различаются *простые* (базовые, примитивные) структуры (типы) данных и *интегрированные* (структурированные, композитные, сложные). Простыми называются такие структуры данных, которые не могут быть расчленены на составные части, большие, чем биты. С точки зрения физической структуры важным является то обстоятельство, что в данной машинной архитектуре, в данной системе программирования мы всегда можем заранее сказать, каков будет размер данного простого типа и какова структура его размещения в памяти. С логической точки зрения простые данные являются неделимыми единицами. Интегрированными называются такие структуры данных, составными частями которых являются другие структуры данных – простые или в свою очередь интегрированные. Интегрированные структуры данных конструируются программистом с использованием средств интеграции данных, предоставляемых языками программирования.

В зависимости от отсутствия или наличия явно заданных связей между элементами данных следует различать *несвязные* структуры (векторы, массивы, строки, стеки, очереди) и *связные* структуры (связные списки).

Весьма важный признак структуры данных – ее изменчивость, изменение числа элементов и (или) связей между элементами структуры. В определении изменчивости структуры не отражен факт изменения значений элементов данных, поскольку в этом случае все структуры данных имели бы свойство изменчивости. По признаку изменчивости различают структуры *статические, полустатические, динамические* [8]. Классификация структур данных по признаку изменчивости приведена на рис. 1.1. Базовые структуры данных, статические, полустатические и динамические характерны для оперативной памяти и часто называются оперативными структурами. Файловые структуры соответствуют структурам данных для внешней памяти.

Важный признак структуры данных – характер упорядоченности ее элементов. По этому признаку структуры можно делить на ***линейные и нелинейные*** структуры.

В зависимости от характера взаимного расположения элементов в памяти линейные структуры можно разделить на структуры с ***последовательным*** распределением элементов в памяти (векторы, строки, массивы, стеки, очереди) и структуры с *произвольным связным* распределением элементов в памяти (односвязные, двусвязные списки). Пример нелинейных структур – многосвязные списки, деревья, графы.

В языках программирования понятие «структуры данных» тесно связано с понятием «типы данных». Любые данные, т.е. константы, переменные, значения функций или выражения, характеризуются своими типами.

Рис. 1.1. Классификация структур данных

Информация по каждому типу однозначно определяет:

1) структуру хранения данных указанного типа, т.е. выделение памяти и представление данных в ней, с одной стороны, и интерпретирование двоичного представления, с другой;

2) множество допустимых значений, которые может иметь тот или иной объект описываемого типа;

3) множество допустимых операций, которые применимы к объекту описываемого типа.

В последующих главах данного пособия рассматриваются структуры данных и соответствующие им типы данных.

1.4. Операции над структурами данных

Над любыми структурами данных могут выполняться четыре общие операции: создание, уничтожение, выбор (доступ), обновление.

Операция создания заключается в выделении памяти для структуры данных. Память может выделяться в процессе выполнения программы или на этапе компиляции. В ряде языков (например, в С) для структурированных данных, конструируемых программистом, операция создания включает в себя также установку начальных значений параметров, создаваемой структуры.

Например, в PASCAL оператор I:integer приведет к выделению адресного пространства для переменной I, в С (int I) в результате описания типа будет выделена память для соответствующих переменных. Для структур данных, объяв-

ленных в программе, память выделяется автоматически средствами систем программирования либо на этапе компиляции, либо при активизации процедурного блока, в котором объявляются соответствующие переменные. Программист может и сам выделять память для структур данных, используя имеющиеся в системе программирования процедуры/функции выделения/освобождения памяти. В объектно-ориентированных языках программирования при разработке нового объекта для него должны быть определены процедуры создания и уничтожения.

Главное заключается в том, что независимо от используемого языка программирования, имеющиеся в программе структуры данных не появляются «из ничего», а явно или неявно объявляются операторами создания структур. В результате этого всем экземплярам структур в программе выделяется память для их размещения.

Операция уничтожения структур данных противоположна по своему действию операции создания. Некоторые языки, такие как BASIC, не дают возможности программисту уничтожать созданные структуры данных. В языках C, PASCAL структуры данных, имеющиеся внутри блока, уничтожаются в процессе выполнения программы при выходе из этого блока. Операция уничтожения помогает эффективно использовать память.

Операция выбора используется программистами для доступа к данным внутри самой структуры. Форма операции доступа зависит от типа структуры данных, к которой осуществляется обращение. Метод доступа – один из наиболее важных свойств структур, особенно в связи с тем, что это свойство имеет непосредственное отношение к выбору конкретной структуры данных.

Операция обновления позволяет изменить значения данных в структуре данных. Примером операции обновления является операция присваивания, или, более сложная форма – передача параметров.

Вышеуказанные четыре операции обязательны для всех структур и типов данных. Помимо этих общих операций для каждой структуры данных могут быть определены операции специфические, работающие только с данными данного типа (данной структуры). Специфические операции рассматриваются при рассмотрении каждой конкретной структуры данных.

1.5. Структурность данных и технология программирования

Большинство авторов публикаций, посвященных структурам и организации данных, делают основной акцент на том, что знание структуры данных позволяет организовать их хранение и обработку максимально эффективным образом – с точки зрения минимизации затрат как памяти, так и процессорного времени. Другим не менее, а может быть, и более важным преимуществом, которое обеспечивается структурным подходом к данным, является возможность

структурирования сложного программного изделия. Современные промышленно выпускаемые программные пакеты – изделия чрезвычайно сложные, объем которых исчисляется тысячами и миллионами строк кода, а трудоемкость разработки – сотнями человек. Естественно, что разработать такое программное изделие «все сразу» невозможно, оно должно быть представлено в виде какой-то структуры – составных частей и связей между ними. Правильное структурирование изделия дает возможность на каждом этапе разработки сосредоточить внимание разработчика на одной обозримой части изделия или поручить реализацию разных его частей разным исполнителям.

При структурировании больших программных изделий возможно применение подхода, основанного на структуризации алгоритмов и известного, как «нисходящее» проектирование или «программирование сверху вниз», или подхода, основанного на структуризации данных и известного, как «восходящее» проектирование или «программирование снизу вверх».

В первом случае структурируют прежде всего действия, которые должна выполнять программа. Большую и сложную задачу, стоящую перед проектируемым программным изделием, представляют в виде нескольких подзадач меньшего объема. Таким образом, модуль самого верхнего уровня, отвечающий за решение всей задачи в целом, получается достаточно простым и обеспечивает только последовательность обращений к модулям, реализующим подзадачи. На первом этапе проектирования модули подзадач выполняются в виде «заглушек». Затем каждая подзадача в свою очередь подвергается декомпозиции по тем же правилам. Процесс дробления на подзадачи продолжается до тех пор, пока на очередном уровне декомпозиции получают подзадачу, реализация которой будет вполне обозримой. В предельном случае декомпозиция может быть доведена до того, что подзадачи самого нижнего уровня могут быть решены элементарными инструментальными средствами (например, одним оператором выбранного языка программирования).

Другой подход к структуризации основывается на данных. Программисту, который хочет, чтобы его программа имела реальное применение в некоторой прикладной области, не следует забывать о том, что программирование – это обработка данных. В программах можно изобретать сколь угодно замысловатые и изощренные алгоритмы, но у реального программного изделия всегда есть Заказчик. У Заказчика есть входные данные, и он хочет, чтобы по ним были получены выходные данные, а какими средствами это обеспечивается – его не интересует. Таким образом, задачей любого программного изделия является преобразование входных данных в выходные. Инструментальные средства программирования предоставляют набор базовых (простых, примитивных) типов данных и операции над ними. Интегрируя базовые типы, программист создает более сложные типы данных и определяет новые операции над сложными типами. Можно здесь провести аналогию со строительными работами: базовые типы – «кирпичики», из которых создаются сложные типы – «строительные блоки». Полученные на первом шаге композиции «строительные блоки» ис-

пользуются в качестве базового набора для следующего шага, результатом которого будут еще более сложные конструкции данных и еще более мощные операции над ними и т.д. В идеале последний шаг композиции дает типы данных, соответствующие входным и выходным данным задачи, а операции над этими типами реализуют в полном объеме задачу проекта.

Программисты, поверхностно понимающие структурное программирование, часто противопоставляют нисходящее проектирование восходящему, придерживаясь одного выбранного ими подхода. Реализация любого реального проекта всегда ведется встречными путями, причем, с постоянной коррекцией структур алгоритмов по результатам разработки структур данных и наоборот.

Еще одним чрезвычайно продуктивным технологическим приемом, связанным со структуризацией данных является инкапсуляция. Смысл ее состоит в том, что сконструированный новый тип данных – «строительный блок» – оформляется таким образом, что его внутренняя структура становится недоступной для программиста – пользователя этого типа. Программист, использующий этот тип данных в своей программе (в модуле более высокого уровня), может оперировать с данными этого типа только через вызовы процедур, определенных для этого типа. Новый тип данных представляется для него в виде «черного ящика» для которого известны входы и выходы, но содержимое – неизвестно и недоступно.

Инкапсуляция чрезвычайно полезна и как средство преодоления сложности, и как средство защиты от ошибок. Первая цель достигается за счет того, что сложность внутренней структуры нового типа данных и алгоритмов выполнения операций над ним исключается из поля зрения программиста-пользователя. Вторая цель достигается тем, что возможности доступа пользователя ограничиваются лишь заведомо корректными входными точками, следовательно, снижается и вероятность ошибок.

Современные языки программирования блочного типа (PASCAL, C) обладают достаточно развитыми возможностями построения программ с модульной структурой и управления доступом модулей к данным и процедурам. Расширения же языков дополнительными возможностями конструирования типов и их инкапсуляции делает язык объектно-ориентированным. Сконструированные и полностью закрытые типы данных представляют собой объекты, а процедуры, работающие с их внутренней структурой – методы работы с объектами. При этом в значительной степени меняется и сама концепция программирования. Программист, оперирующий объектами, указывает в программе ЧТО нужно сделать с объектом, а не КАК это надо делать.

Технология баз данных развивалась параллельно с технологией языков программирования и не всегда согласованно с ней. Отчасти этим, а отчасти и объективными различиями в природе задач, решаемых системами управления базами данных (СУБД) и системами программирования, вызваны некоторые терминологические и понятийные различия в подходе к данным в этих двух сферах. Ключевым понятием в СУБД является понятие модели данных, в ос-

новном тождественное понятию логической структуры данных. Отметим, что физическая структура данных в СУБД не рассматривается вообще. Но сами СУБД являются программными пакетами, выполняющими отображение физической структуры в логическую (в модель данных). Для реализации этих пакетов используются те или иные системы программирования, разработчики СУБД, следовательно, имеют дело со структурами данных в терминах систем программирования. Для пользователя же внутренняя структура СУБД и физическая структура данных совершенно прозрачна; он имеет дело только с моделью данных и с другими понятиями логического уровня.

Вопросы для самоконтроля

1. Приведите классификацию структур данных.
2. В чем различие между абстрактной и физической структурами данных?
3. Назовите основные операции, выполняемые над любыми структурами данных.
4. Что такое нисходящее и восходящее проектирование?

ГЛАВА 2
ТИПЫ ДАННЫХ

2.1. Концепция типа данных

Концепция типов данных является одной из центральных в любом языке программирования. С типом величины связаны три ее свойства: форма внутреннего представления, множество принимаемых значений и множество допустимых операций. Автор языка Паскаль Н. Вирт придавал большое значение разнообразию типов данных в языке программирования. В своей книге «Алгоритмы и структуры данных» он подчеркивает зависимость алгоритма решения задачи от способа организации данных в программе[1]. Удачно выбранный способ организации данных упрощает алгоритм решения задачи.

Концепция типа данных появилась в языках программирования высокого уровня как естественное отражение того факта, что обрабатываемые программой данные могут иметь различные множества допустимых значений, храниться в памяти компьютера различным образом, занимать различные объемы памяти и обрабатываться с помощью различных команд процессора.

Тип данных – фундаментальное понятие теории программирования. *Тип данных* определяет множество значений, набор операций, которые можно применять к таким значениям, и способ реализации хранения значений и выполнения операций. Любые данные, которыми оперируют программы, относятся к определенным типам.

Тип – относительно устойчивая и независимая совокупность элементов, которую можно выделить во всем рассматриваемом множестве (предметной области).

Полиморфный тип – представление набора типов как единственного типа.

Современные языки программирования (включая Ассемблер) поддерживают оба способа задания типа. Так, в C++ тип enum является примером задания типа через набор значений. Определение класса (если рассматривать класс как тип данных) фактически является определением предиката типа, причем возможна проверка предиката как на этапе компиляции (проверка соответствия типов), так и на этапе выполнения (полиморфизм очень тесно связано с полиморфными типами). Для базовых типов подобные предикаты заданы создателями языка изначально.

Теоретически не может существовать языков, в которых отсутствуют типы (включая полиморфные). Это следует из того, что все языки основаны на машине Тьюринга или на лямбда-исчислении. И в том, и в другом случае необходимо оперировать как минимум одним типом данных – хранящимся на

ленте (машина Тьюринга) или передаваемым и возвращаемым из функции (лямбда-исчисление).

Ниже перечислены языки программирования по способу определения типов данных:

1) Языки с полиморфным типом данных. Одни языки не связывают переменные, константы, формальные параметры и возвращаемые значения функций с определенными типами, поддерживая единственный полиморфный тип данных. В чистом виде таких языков не встречается, но близкие примеры – MS Visual Basic, Delphi – тип variant, Пролог, Лисп – списки. В этих языках переменная может принимать значение любого типа, в параметры функции можно передавать значения любых типов, и вернуть функция также может значение любого типа. Сопоставление типов значений переменных и параметров с применяемыми к ним операциями производится непосредственно при выполнении этих операций. Например, выражение **a+b**, может трактоваться как сложение чисел, если a и b имеют числовые значения, конкатенация строк, если a и b имеют строковые значения, и как недопустимая (ошибочная) операция, если типы значений a и b несовместимы. Такой порядок называют «динамической типизацией« (соответствует понятию полиморфизм в ООП, полиморфный тип в теории типов). Языки, поддерживающие только динамическую типизацию, называют иногда «бестиповыми». Это название не следует понимать как признак отсутствия понятия типов в языке – типы данных все равно есть.

2) Языки с неявным определением типов. Казалось бы, BASIC является примером языка без типов. Однако это строго типизированый язык: в нем различаютсястроковые типы (добавляется символ $), массивы (добавляется []) и числовые типы (ничего не добавляется).

3) Языки с типом, определяемым пользователем. Также хорошо известны языки, в которых типы данных определяются автоматически, а не задаются пользователем. Каждой переменной, параметру, функции приписывается определенный тип данных. В этом случае для любого выражения возможность его выполнения и тип полученного значения могут быть определены без исполнения программы. Такой подход называют «статической типизацией». При этом правила обращения с переменными, выражениями и параметрами разных типов могут быть как очень строгими (C++), так и весьма либеральными (Си). Например, в классическом языке Си практически все типы данных совместимы – их можно применять совместно в любых выражениях, присваивать значение переменной одного типа переменной другого почти без ограничений. При таких операциях компилятор генерирует код, обеспечивающий преобразование типов, а логическая корректность такого преобразования остается на совести программиста. Подобные языки называют «языками со слабой типизацией». Противоположнось им – «языки с сильной типизацией», такие как Ада. В них каждая операция требует операндов строго заданных типов. Никакие автоматические преобразования типов не поддерживаются – их можно выполнить только

явно, с помощью соответствующих функций и операций. Сильная типизация делает процесс программирования более трудоемким, но дает в результате программы, содержащие заметно меньше трудно обнаруживаемых ошибок.

На практике, современные языки программирования поддерживают несколько моделей определения типов одновременно. Каждый язык программирования поддерживает один или несколько *встроенных типов данных* (базовых типов), кроме того, развитые языки программирования предоставляют программисту возможность описывать собственные типы данных, комбинируя или расширяя существующие.

Преимущества от использования типов данных

Надежность. Типы данных защищают от трех видов ошибок.

1. Некорректное присваивание. Пусть переменная объявлена как имеющая числовой тип. Тогда попытка присвоить ей символьное или какое-либо другое значение в случае статической типизации приведет к ошибке компиляции и не даст такой программе запуститься. В случае динамической типизации интерпретатор программы перед выполнением потенциально опасного действия сравнит типы данных переменной и значения и также выдаст ошибку. Все это позволяет избежать неправильной работы и «падения» программы.

2. Некорректная операция. Позволяет избежать попыток применения выражений вида «Hello world» + 1. Поскольку, как уже говорилось, все переменные в памяти хранятся как наборы битов, то при отсутствии типов подобная операция была выполнима (и могла дать результат вроде «ello worldЁ»). С использованием типов (см. далее «Контроль типов») такие ошибки отсекаются опять же на этапе компиляции.

3. Некорректная передача параметров. Если функция »синус« ожидает, что ей будет передан числовой аргумент, то передача ей в качестве параметра строки «Hello world» может иметь непредсказуемые последствия. При помощи контроля типов такие ошибки также отсекаются на этапе компиляции.

Стандартизация. Благодаря соглашениям о типах, поддерживаемых большинством систем программирования, сложилась ситуация, когда программисты могут быстро менять свои рабочие инструменты, а программы не требуют больших переделок при переносе исходных текстов в другую среду. Стандартизации по универсальным типам данных еще есть куда развиваться.

Документация. Использование того или другого типа данных объясняет намерения программиста. Например, **`enum`** и **`bool`** в языке не обязательны и вместо них может быть **`int`** – но оба они означают, что перед нами не целая величина, а одно из нескольких предопределенных значений.

2.2. Типы данных

Все данные, используемые в программе, можно разделить на две группы: константы и переменные. К первой группе относятся данные, не изменяющие своего значения в ходе выполнения программы, данные второй группы могут изменять свое значение. Как константы, так и переменные могут быть различных типов, которые определяют их структуру, набор допустимых значений, правила использования и способ представления в ЭВМ.

Тип определяет:

- возможные значения переменных, констант, функций, выражений, принадлежащих к данному типу;
- внутреннюю форму представления данных в ЭВМ;
- операции и функции, которые могут выполняться над величинами, принадлежащими к данному типу.

Иерархию типов данных можно представить следующей схемой.

Простые (скалярные) типы:

- целые;
- вещественные;
- символьные;
- указатели;
- перечислимый тип.

Составные (структурированные) типы:

- массив;
- запись (структура);
- множество;
- файл.

Переменная простого (скалярного) типа в любой момент времени хранит только одно значение. В отличие от простых переменных, переменные составного (структурированного) типа одновременно хранят несколько значений.

Целые и вещественные переменные предназначены для хранения чисел, символьные переменные – это также числовые переменные, они хранят ASCII коды символов.

Перечислимый тип представляет собой набор целочисленных констант, используемых обычно для организаций разветвлений в программе.

Указатель – это переменная, значением которой является адрес объекта (обычно другой переменной) в памяти компьютера. Таким образом, если одна переменная содержит адрес другой переменной, то говорят, что первая переменная указывает (ссылается) на вторую.

Массив – это группа элементов одинакового типа (double, float, int и т. п.). Из объявления массива компилятор должен получить информацию о типе элементов массива и их количестве.

Структура – это совокупность элементов, объединенных под одним именем. Структура представляет собой составной объект, в который могут входить элементы различных типов. Для каждого элемента выделяется своя область памяти.

Множество – совокупность каких-либо объектов, обладающих характеристическим свойством, общим для всех этих объектов.

Файл – поименованная совокупность элементов данных, однотипных по структуре, оформленная как единое целое средствами языка программирования и хранимая во внешней памяти.

Обязательное описание типа приводит к избыточности в тексте программ, но такая избыточность является важным вспомогательным средством разработки программ и рассматривается как необходимое свойство современных алгоритмических языков высокого уровня.

Можно все многообразие типов данных классифицировать по трем критериям:

- по признаку стандартности (стандартные и нестандартные);
- по структурной организации (простые и сложные или структурированные);
- по признаку счетности множества значений (порядковые и непорядковые).

По структурной организации выделены следующие простые типы:

- целый;
- вещественный;
- литерный;
- перечисляемый;
- интервальный.

В простых типах данных выделяются порядковые типы, в которых каждому значению из множества ставится в соответствие целочисленный порядковый номер. Порядковыми типами являются:

- целый;
- литерный;
- перечисляемый;
- интервальный.

К структурированным типам относятся типы:

- массив;
- запись;
- множество;
- файл.

Типы данных бывают следующие.

Простые

Числовые. Хранятся числа. Могут применяться обычные арифметические операции.

Целочисленные: со знаком, то есть могут принимать как положительные, так и отрицательные значения; и без знака, то есть могут принимать только неотрицательные значения.

Вещественные: с запятой (то есть хранятся знак и цифры целой и дробной частей) и с плавающей запятой (то есть число приводится к виду $m*b^c$, где m – мантисса, b – основание показательной функции, e – показатель степени (порядок) (в англоязычной литературе экспонента), причем в *нормальной форме* $0<=m<b$, а в *нормализованной форме* $1<=m<b$, e – целое число и хранятся знак и числа m и e).

Числа произвольной точности, обращение с которыми происходит посредством длинной арифметики. Примером языка с встроенной поддержкой таких типов является UBASIC, часто применяемый среди криптографов.

Символьный тип. Хранит один символ. Могут использоваться различные кодировки.

Логический тип. Имеет два значения: истина и ложь, при троичной логике может иметь и третье значение – «не определено» (или «неизвестно»). Могут применяться логические операции. Используется в операторах ветвления и циклах. В некоторых языках является подтипом числового типа, при этом ложь=0, истина=1.

Перечисляемый тип. Может хранить только те значения, которые прямо указаны в его описании.

Строковый тип. Хранит строку символов. Аналогом сложения в строковой алгебре является конкатенация (прибавление одной строки в конец другой строки). В языках, близких к бинарному представлению данных, чаще рассматривается как массив символов, в языках более высокой абстракции зачастую выделяется в качестве простого.

Указатель. Хранит адрес в памяти компьютера, указывающий на какую-либо информацию, как правило – указатель на переменную.

Структурированные (составные, сложные)

Массив. Является индексированным набором элементов одного типа. Наиболее популярны: одномерный массив – вектор (в случае чисел) или строковый тип (в случае символов), двумерный массив – матрица.

Запись (структура). Набор различных элементов (полей записи), хранимый как единое целое. Возможен доступ к отдельным полям записи. Например, struct в C или record в Pascal.

Множество. В основном совпадает с обычным математическим понятием множества. Допустимы стандартные операции с множествами и проверка на принадлежность элемента множеству. В некоторых языках рассматривается как составной тип

Файл. Хранит только однотипные значения, доступ к которым осуществляется только последовательно (файл с произвольным доступом, включенный в

некоторые системы программирования, фактически является неявным массивом).

Класс. Определяемый пользователем тип данных, для которого предварительно специфицирован ряд определяемых пользователем функций.

Другие типы данных. Если описанные выше типы данных представляли какие-либо объекты реального мира, то рассматриваемые здесь типы данных представляют объекты компьютерного мира, то есть являются исключительно компьютерными терминами.

Контроль типов и системы типизации

Процесс проверки и накладывания ограничений типов – контроля типов, может выполняться во время компилирования (статическая проверка) или во время выполнения (динамическая проверка).

Статическая типизация – контроль типов осуществляется при компиляции.

Динамическая типизация – контроль типов осуществляется во время выполнения.

Контроль типов, также, может быть *строгим* и *слабым*.

Строгая типизация – совместимость типов автоматически контролируется транслятором:

Номинативная типизация (англ. *nominative type system*) – совместимость должна быть явно указана (наследована) при определении типа.

Структурная типизация (англ. *structural type system*) – совместимость определяется структурой самого типа (типами элементов, из которых построен составной тип).

Слабая типизация – совместимость типов никак транслятором не контролируется. В языках со слабой типизацией обычно используется подход под названием «утиная типизация» – когда совместимость определяется и реализуется общим интерфейсом доступа к данным типа.

2.3. Основные простые типы данных в C++

Основные (стандартные, простые) типы данных часто называют арифметическими, поскольку их можно использовать в арифметических операциях. Для описания основных типов определены следующие ключевые слова:

1. int (целый);
2. char (символьный);
3. wchar_t (расширенный символьный);
4. bool (логический);
5. float (вещественный);
6. double (вещественный с двойной точностью).

Первые четыре типа называют целочисленными (целыми), последние два – типами с плавающей точкой. Код, который формирует компилятор для обработки целых величин, отличается от кода для величин с плавающей точкой.

Существует четыре спецификатора типа, уточняющих внутреннее представление и диапазон значений стандартных типов:

- short (короткий);
- long (длинный);
- signed (знаковый);
- unsigned (беззнаковый).

Целый тип (int)

Размер типа int не определяется стандартом, а зависит от компьютера и компилятора. Для 16-разрядного процессора под величины этого типа отводится 2 байта, для 32-разрядного – 4 байта.

Спецификатор short перед именем типа указывает компилятору, что под число требуется отвести 2 байта независимо от разрядности процессора. Спецификатор long означает, что целая величина будет занимать 4 байта. Таким образом, на 16-разрядном компьютере эквиваленты int и short int, а на 32-разрядном – int и long int.

Внутреннее представление величины целого типа – целое число в двоичном коде. При использовании спецификатора signed старший бит числа интерпретируется как знаковый (0 – положительное число, 1 – отрицательное). Спецификатор unsigned позволяет представлять только положительные числа, поскольку старший разряд рассматривается как часть кода числа. Таким образом, диапазон значений типа int зависит от спецификаторов. Диапазоны значений величин целого типа с различными спецификаторами для IBM PC-совместимых компьютеров приведены в таблице «Диапазоны значений простых типов данных» в конце главы.

По умолчанию все целочисленные типы считаются знаковыми, то есть спецификатор signed можно опускать. Константам, встречающимся в программе, приписывается тот или иной тип в соответствии с их видом. Если этот тип по каким-либо причинам не устраивает программиста, он может явно указать требуемый тип с помощью суффиксов L, l (long) и U, u (unsigned). Например, константа 32L будет иметь тип long и занимать 4 байта. Можно использовать суффиксы L и U одновременно, например, 0x22UL или 05Lu.

Примечание: Типы short int, long int, signed int и unsigned int можно сокращать до short, long, signed и unsigned соответственно.

Символьный тип (char)

Под величину символьного типа отводится количество байт, достаточное для размещения любого символа из набора символов для данного компьютера, что и определило название типа. Как правило, это 1 байт. Тип char, как и другие целые типы, может быть со знаком или без знака. В величинах со знаком можно

хранить значения в диапазоне от –128 до 127. При использовании спецификатора unsigned значения могут находиться в пределах от 0 до 255. Этого достаточно для хранения любого символа из 256-символьного набора ASCII. Величины типа char применяются также для хранения целых чисел, не превышающих границы указанных диапазонов.

Расширенный символьный тип (wchar_t)

Тип wchar_t предназначен для работы с набором символов, для кодировки которых недостаточно 1 байта, например, Unicode. Размер этого типа зависит от реализации; как правило, он соответствует типу short. Строковые константы типа wchar_t записываются с префиксом L, например, L»Gates».

Логический тип (bool)

Величины логического типа могут принимать только значения true и false, являющиеся зарезервированными словами. Внутренняя форма представления значения false – 0 (нуль). Любое другое значение интерпретируется как true. При преобразовании к целому типу true имеет значение 1.

Типы с плавающей точкой (float, double и long double)

Стандарт C++ определяет три типа данных для хранения вещественных значений: float, double и long double. Типы данных с плавающей точкой хранятся в памяти компьютера иначе, чем целочисленные. Внутреннее представление вещественного числа состоит из двух частей – мантиссы и порядка. В IBM PC-совместимых компьютерах величины типа float занимают 4 байта, из которых один двоичный разряд отводится под знак мантиссы, 8 разрядов под порядок и 23 под мантиссу. Мантисса – это число, большее 1.0, но меньшее 2.0. Поскольку старшая цифра мантиссы всегда равна 1, она не хранится.

Для величин типа double, занимающих 8 байт, под порядок и мантиссу отводится 11 и 52 разряда соответственно. Длина мантиссы определяет точность числа, а длина порядка – его диапазон. Как можно видеть из таблицы в конце записи, при одинаковом количестве байт, отводимом под величины типа float и long int, диапазоны их допустимых значений сильно различаются из-за внутренней формы представления. Спецификатор long перед именем типа double указывает, что под его величину отводится 10 байт.

Константы с плавающей точкой имеют по умолчанию тип double. Можно явно указать тип константы с помощью суффиксов F, f (float) и L, l (long). Например, константа 2E+6L будет иметь тип long double, а константа 1.82f – тип float.

Для написания переносимых на различные платформы программ нельзя делать предположений о размере типа int. Для его получения необходимо пользоваться операцией sizeof, результатом которой является размер типа в байтах.

В стандарте ANSI диапазоны значений для основных типов не задаются, определяются только соотношения между их размерами, например:

sizeof(float) ≤ slzeof(double) ≤ sizeof(long double)
sizeof(char) ≤ slzeof(short) ≤ sizeof(int) ≤ sizeof(long)

Примечание: Минимальные и максимальные допустимые значения для целых типов зависят от реализации и приведены в заголовочном файле <limits.h> (<climits>), характеристики вещественных типов – в файле <float.h> (<cfloat>), а также в шаблоне класса numeric_limits

Тип void

Кроме перечисленных, к основным типам языка относится тип void, но множество значений этого типа пусто. Он используется для определения функций, которые не возвращают значения, для указания пустого списка аргументов функции, как базовый тип для указателей и в операции приведения типов.

Различные виды целых и вещественных типов, различающиеся диапазоном и точностью представления данных, введены для того, чтобы дать программисту возможность наиболее эффективно использовать возможности конкретной аппаратуры, поскольку от выбора типа зависит скорость вычислений и объем памяти. Но оптимизированная для компьютеров какого-либо одного типа программа может стать не переносимой на другие платформы, поэтому в общем случае следует избегать зависимостей от конкретных характеристик типов данных.

Для вещественных типов в таблице приведены абсолютные величины минимальных и максимальных значений.

Диапазоны значений числовых типов данных языка С представлены в табл. 2.1.

Объявление переменных числового типа

Одним из отличий языка С от ряда других языков программирования является отсутствие принципа умолчания, что приводит к необходимости явного объявления всех переменных, используемых в программе, вместе с указанием соответствующих им типов.

Объявление переменной имеет следующий формат:
[спецификатор_класса_памяти] спецификатор_типа идентификатор [=инициатор].

Спецификатор класса памяти определяется одним из 4 ключевых слов языка С: auto,extern,register,static и указывает, во-первых, каким образом будет распределяться память под объявляемую переменную и, во-вторых, область обратиться видимости этой переменной, т.е. из каких частей программы можно к ней.

Спецификатор типа – одно или несколько ключевых слов, определяющих тип объявляемой переменной. Инициатор задает начальное значение или список начальных значений, присваиваемых переменной при объявлении.

Таблица 2.1

Тип данных	Размер памяти, бит	Диапазон значений
char (символьный)	8	от –128 до 127
signed char (знаковый символьный)	8	от –128 до 127
unsigned char (беззнаковый символьный)	8	от 0 до 255
short int (короткое целое)	16	от –32768 до 32767
unsigned int (беззнаковое целое)	16	от 0 до 65535 (16-битная платформа) от 0 до 4294967295 (32-битная платформа)
int (целое)	16 32	от –32768 до 32767 (16-битная платформа) от –2147483648 до 2147483647 (32-битная платформа)
long (длинное целое)	32	от –2147483648 до 2147483647
unsigned long (длинное целое без знака)	32	от 0 до 4294967295
long long int (C99)	64	от $-(2^{63}-1)$ до $2^{63}-1$
unsigned long long int (C99)	64	от 0 до $2^{64}-1$
float (вещественное)	32	от 3.4E–38 до 3.4E38
double (двойное вещественное)	64	от 1.7E–308 до 1.7E308
long double (длинное вещественное)	80	от 3.4E–4932 до 3.4E4932
_Bool (C99)	8	true(1), false(0)
bool (C++)	8	true(1), false(0)

Примеры инициализации переменных:
int i=5; float f=12.35; char ch='a';
int k=0, b=5, d=7;

Переменная любого типа может быть объявлена немодифицируемой, что достигается добавлением ключевого слова const к спецификатору типа. Объекты с типом const представляют собой данные, используемые только для чтения, т.е. этой переменной не может быть присвоено новое значение: например, const int a=5. Отметим, что если после слова const отсутствует спецификатор типа, то подразумевается спецификатор типа int. Если ключевое слово const стоит перед объявлением составных типов (массив, структура, объединение), то это приводит к тому, что каждый элемент также должен являться немодифицируемым, т.е. значение ему может быть присвоено только один раз. Ключевое слово void означает отсутствие типа.

Данные целого типа

Для определения данных целого типа используются ключевые слова char,int,short,long, которые определяют диапазон значений и размер области

памяти, выделяемой под переменные (табл. 2.1). Так, переменная типа char занимает в памяти 1 байт, short – 2 байта, long – 4 байта. Размер переменной типа int определяется типом процессора (аппаратной платформой). При объявлении целых типов можно использовать ключевые слова signed и unsigned, которые указывают, как интерпретируется старший бит объявляемой переменной. Если указано ключевое слово unsigned, то старший бит интерпретируется как часть числа, в противном случае старший бит интерпретируется как знаковый [6]. В случае отсутствия ключевого слова unsigned целая переменная считается знаковой. В том случае, если спецификатор типа состоит из ключевого типа signed или unsigned и далее следует идентификатор переменной, то она будет рассматриваться как переменная типа int. Отметим, что ключевые слова signed и unsigned не обязательны.
В памяти данные хранятся в двоичном коде. На рис. 2.1 изображено внутреннее представление данных целого типа.

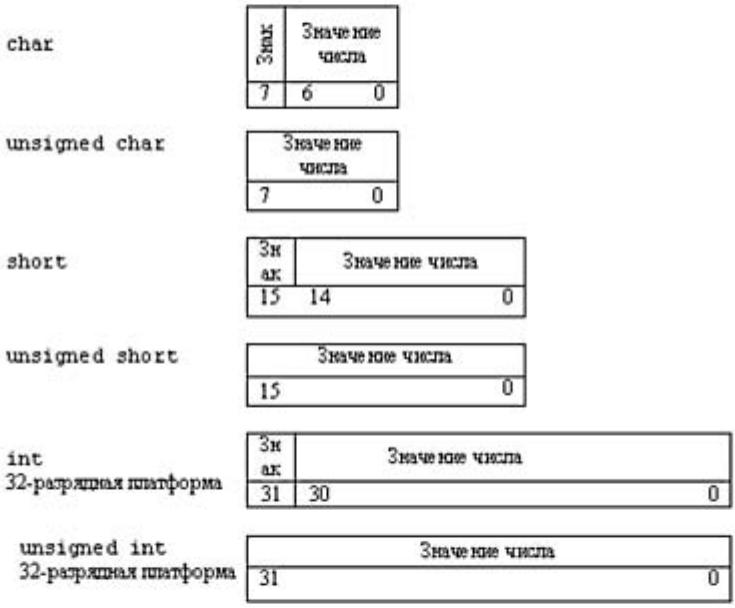


Рис. 2.1. Внутреннее представление данных целых типов

Переменная типа char(signed char) занимает в памяти 1 байт, при этом старший бит хранит информацию о знаке числа: 0 соответствует положительному числу, 1 – отрицательному. Биты с 0-го по 6-й используются для записи значения числа. Запись в каждый из этих битов значения 1 соответствует наибольшему положительному числу, равному 127, при этом старший бит установлен в 0. Такое представление целых чисел называется прямым кодом. Для хранения отрицательных чисел используется представление чисел, называемое

дополнительным кодом. Получить дополнительный код отрицательного числа можно по следующему правилу:

- в биты, предназначенные для хранения значения числа, записывается – модуль отрицательного числа в прямом коде;
- в старший (знаковый) бит помещается 1;
- в битах, предназначенные для хранения значения числа, формируется
- обратный код, т.е. 1 заменяется на 0, а 0 на 1;
- к обратному коду числа прибавляется 1.

Рассмотрим дополнительный код для числа –1 (рис. 2.2).

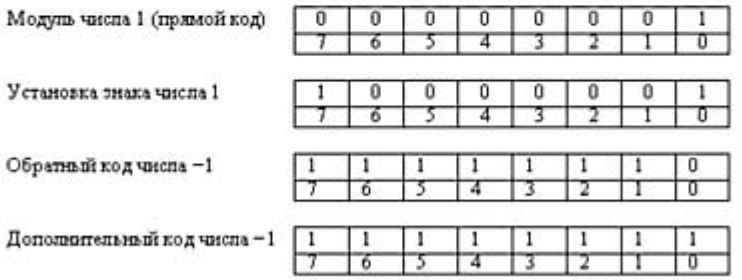


Рис. 2.2. Дополнительный код отрицательного числа

Как видно из рис. 2.2, минимальное по модулю отрицательное число представлено единицами во всех двоичных разрядах, предназначенных для хранения числа. Если в эти биты записать нули, получится наибольшее по модулю отрицательное число. Для переменной типа char это значение равно минус 128. Переменная типа unsigned char хранит целые положительные значения, при этом все 8 бит используются для записи числа. Такое внутреннее представление числа позволяет записывать в переменную значения в диапазоне от 0 до 255.

Мы рассмотрели примеры представления целых чисел для переменной типа char, так как это самый компактный тип, но то же самое представление имеют и остальные целые типы, различия только в размере памяти, занимаемой переменной. Однако тип char по сравнению с другими целыми типами имеет особое назначение: он используется для представления символа или объявления строковых литералов. Следует помнить, что по умолчанию тип char или signed char интерпретируется как однобайтовая целая величина со знаком и с диапазоном значений от минус 128 до 127, хотя только значения в диапазоне 0 – 127 имеют символьные эквиваленты. Для представления символов русского алфавита модификатор типа идентификатора данных имеет вид unsigned char, так как коды русских букв превышают величину 127.

Данные вещественного типа

Для определения данных, представляющих число с плавающей точкой, используются ключевые слова float,double,long double. Величина типа float занимает в памяти 4 байта, из которых 1 бит отводится для знака, 8 битов для порядка и 23 бита для мантиссы [6]. Порядок хранится в виде двоичного положительного числа, которое получается при сложении величины порядка и числа 127 (1111111_2). Мантисса хранится в нормализованном виде, т.е. ее значение должно быть в диапазоне от 1 до 2, а старший бит равен 1. Если нормализация мантиссы нарушается, выполняется сдвиг мантиссы до тех пор, пока старшей цифрой не окажется единица, одновременно при каждой операции сдвига мантиссы происходит изменение величины порядка. Так как старшая цифра в нормализованной мантиссе – единица, она отбрасывается. Отброшенная единица называется неявной единицей. Диапазон значений переменной с плавающей точкой приблизительно равен от 3.4E–38 до 3.4E+38 , а точность (количество значащих цифр) равна 7.

Рассмотрим, каким образом в ячейках памяти, выделенных под переменную типа float, хранится конкретное число.

Пусть float x=5.375; в двоичной системе $x_2=101.011$ или $x_2=1.01011\cdot2^{10+111111}$, тогда с учетом того, что в нормализованной мантиссе первая цифра всегда 1, и эта единица отбрасывается, а порядок сдвигается на 127, получаем внутреннее представление числа x.

x	0	1	0	0	0	0	0	0	1	0	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Знак Порядок Мантисса

Рис. 2.3. Внутреннее представление вещественного числа

Величина типа double занимает 8 байт в памяти, формат которой аналогичен формату float. Биты памяти распределяются следующим образом: 1 бит – знак, 11 битов – порядок и 52 бита – мантисса. Порядок сдвигается на величину 1023. С учетом опущенного старшего бита мантиссы диапазон значений равен от 1.7E–308 до 1.7E+308, а точность составляет 15 десятичных значащих цифр.

2.4. Составные типы данных в C++

Массивы

Массивы – это группа элементов одинакового типа (double, float, int и т.п.). Из объявления массива компилятор должен получить информацию о типе элементов массива и их количестве. Объявление массива имеет два формата: спецификатор-типа описатель [константное – выражение];

спецификатор-типа описатель [ ];

Описатель – это идентификатор массива.

Спецификатор-типа задает тип элементов объявляемого массива. Элементами массива не могут быть функции и элементы типа void.

Константное-выражение в квадратных скобках задает количество элементов массива. Константное-выражение при объявлении массива может быть опущено в следующих случаях:

- при объявлении массив инициализируется,
- массив объявлен как формальный параметр функции,
- массив объявлен как ссылка на массив, явно определенный в другом файле.

В языке СИ определены только одномерные массивы, но поскольку элементом массива может быть массив, можно определить и многомерные массивы. Они формализуются списком константных-выражений следующих за идентификатором массива, причем каждое константное-выражение заключается в свои квадратные скобки. Каждое константное-выражение в квадратных скобках определяет число элементов по данному измерению массива, так что объявление двухмерного массива содержит два константных-выражения, трехмерного – три и т.д. Отметим, что в языке СИ первый элемент массива имеет индекс равный 0.

Примеры

```
int a[2][3]; /* представлено в виде матрицы
          a[0][0] a[0][1] a[0][2]
           a[1][0] a[1][1] a[1][2]   */
double b[10]; /* вектор из 10 элементов имеющих тип double */
   int w[3][3] = { { 2, 3, 4 },
           { 3, 4, 8 },
           { 1, 0, 9 } };
```

В последнем примере объявлен массив w[3][3]. Списки, выделенные в фигурные скобки, соответствуют строкам массива, в случае отсутствия скобок инициализация будет выполнена неправильно.

Структуры

Структуры – это составной объект, в который входят элементы любых типов, за исключением функций. В отличие от массива, который является однородным объектом, структура может быть неоднородной. Тип структуры определяется записью вида:

struct { список определений }

В структуре обязательно должен быть указан хотя бы один компонент. Определение структур имеет следующий вид:

тип-данных описатель;

где тип-данных указывает тип структуры для объектов, определяемых в описателях. В простейшей форме описатели представляют собой идентификаторы или массивы.

Пример.

```
struct { double x,y; } s1, s2, sm[9];
     struct { int  year;
              char moth, day; } date1, date2;
```

Переменные s1, s2 определяются как структуры, каждая из которых состоит из двух компонент x и y. Переменная sm определяется как массив из девяти структур. Каждая из двух переменных date1, date2 состоит из трех компонентов year, moth, day. >p>Существует и другой способ ассоциирования имени с типом структуры, он основан на использовании тега структуры. Тег структуры аналогичен тегу перечислимого типа. Тег структуры определяется следующим образом:

```
struct тег { список описаний; };
```

где тег является идентификатором.

В приведенном ниже примере идентификатор student описывается как тег структуры:

```
struct student { char name[25];
                 int id, age;
                 char prp;      };
```

Тег структуры используется для последующего объявления структур данного вида в форме:

```
struct тег список-идентификаторов;
```

Пример.

```
struct studeut st1,st2;
```

Использование тегов структуры необходимо для описания рекурсивных структур. Ниже рассматривается использование рекурсивных тегов структуры.

```
struct node { int data;
              struct node * next; }  st1_node;
```

Тег структуры node действительно является рекурсивным, так как он используется в своем собственном описании, т.е. в формализации указателя next. Структуры не могут быть прямо рекурсивными, т.е. структура node не может содержать компоненту, являющуюся структурой node, но любая структура может иметь компоненту, являющуюся указателем на свой тип, как и сделано в приведенном примере.

Доступ к компонентам структуры осуществляется с помощью указания имени структуры и следующего через точку имени выделенного компонента, например:

```
st1.name=«Иванов»;
st2.id=st1.id;
st1_node.data=st1.age;
```

2.5. Абстрактный тип данных

Наличие перечисляемых, уточняемых и конструируемых типов данных в сочетании со средствами выделения динамической памяти позволяет конструировать и использовать структуры данных, достаточные для создания произвольно сложных программ. Ограниченность этих средств состоит в том, что при определении типов и создании структур невозможно зафиксировать правила их использования.

Абстракция – это способ отвлечься от неважных деталей и, таким образом, выбрать наиболее важные признаки для рассмотрения. В процессе создания программы разработчик строит программную модель решаемой задачи и в процессе построения программной модели оперирует элементами этой модели. Программный код структурируется соответствующим образом. Для выделения программных сущностей в коде программы естественно использовать механизм абстракции, так называемый механизм *поведенческой абстракции (функциональной абстракции)*.

Функциональная абстракция подразумевает выделение набора операций (функций) для работы с элементами программной модели. Таким образом, сущности программной модели представляются с помощью набора операций. Так осуществляется поведенческая абстракция сущности в программном коде. Концепция абстрактных типов данных (АТД) была сформулирована в 1974 году. Сами авторы использовали термин «operational cluster» [24], т.е. набор операций, и назвали такой набор операций *абстрактным типом данных* (АТД).

В настоящее время абстрактные типы данных играют важную роль в теории программирования и являются одним из ключевых понятий в программировании.

Абстрактные типы данных используются как метод абстракции данных в программном коде, а также как метод выделения объектов в программной модели. Для описания абстрактных типов данных используется т.н. поведенческая абстракция. Объекты рассматриваются как черные ящики, доступ к которым обеспечивается посредством т.н. кластера операций. Это позволяет подменять реализации абстрактного типа данных без необходимости что-то менять в коде взаимодействия.

Для абстрактных типов данных имеется строгое математическое описание – теория сигнатур многосортных алгебраических систем. Абстрактный тип данных описывается как сигнатура Σ-алгебры с конечным набором аксиом в

виде равенств. Для каждого класса Σ-алгебр имеется т.н. инициальная Σ-алгебра, из которой имеется в каждую алгебру этого класса единственный гомоморфизм. Инициальную Σ-алгебру можно построить как множество термов, построенных из операций сигнатуры. При этом роль атомарных термов играют нульарные операции, называемые также константами. Инициальные алгебры позволяют производить вычисления на сигнатуре Σ-алгебры: получать ответы на вопросы, проверять корректность задания АТД и т.д,

Концепция АТД является, наряду с объектно-ориентированным подходом, наиболее популярной в настоящее время методологией для составления программ. В процессе декомпозиции программы на составляющие компоненты доступ к ним организуется посредством т.н. *кластера операций*, который представляет собой конечный список операций, которые могут быть использованы для модификации данных, предоставляемых данным компонентом.

Отличительной особенностью абстрактных типов данных как механизма абстракции является тот факт, что функциональность компонента программы, описываемая кластером операций, может быть реализована различными способами. Различные реализации абстрактных типов данных взаимозаменяемы благодаря механизму абстракции АТД, позволяющему скрыть детали реализации с помощью набора предопределенных операций.

Концепция абстрактных типов данных хорошо описывается с помощью математической теории алгебраических систем. Алгебраическую систему или, проще говоря, алгебру (абстрактную алгебру), неформально можно определить как множество с набором операций, действующих на элементах данного множества. Операции реализуются как функции от одного или более параметров, действующие на элементах данного множества (для операции с одним аргументом) или на декартовых произведениях множества (для операций с несколькими аргументами). Описание операций, включающее в себя описание типов аргументов и возвращаемых значений, называется *сигнатурой алгебраической системы*. Сигнатуры, очевидно, представляют математическую модель абстрактного типа данных. Это обстоятельство дает возможность описывать программные сущности, заданные посредством АТД, как алгебраические системы.

Основной идеей ***АТД*** является то, что при его определении специфицируется не только структура значений типа, но и набор допустимых операций над переменными и значениями этого типа. В наиболее сильном случае доступ к внутренней структуре типа доступен только через его операции. В число операций обязаны входить один или несколько конструкторов значений типа.

Имеется много разновидностей языков с ***АТД***, языковые средства которых весьма различаются. Кроме того, к этому семейству языков примыкают языки объектно-ориентированного программирования. По поводу них одни авторы полагают, что для них термин ***«язык объектно-ориентированного программирования»*** является модной заменой старого термина ***«язык с АТД»***. Другие находят между этими языковыми семействами много тонких отличий, часть которых считают серьезными.

2.5.1. Представление типа

При программировании с использованием *АТД* возможны три подхода (они могут быть смешаны).

1. Перед началом написания основной программы полностью определить все требуемые типы данных;

2. Определить только те характеристики *АТД*, которые требуются для написания программы и проверки ее синтаксической корректности;

3. Воспользоваться готовыми библиотечными определениями. В каждом из этих подходов имеются свои достоинства и недостатки, но их объединяет то, что при написании программы известны по меньшей мере внешние характеристики всех типов данных. В некотором смысле это означает, что расширен язык программирования.

Подобная внешняя характеристика *АТД* называется его представлением или спецификацией. Представление включает имя *АТД* и набор спецификаций доступных пользователю операций со значениями этого типа. Со своей стороны, спецификация операции состоит из имени и типов параметров (в последнее время такие спецификации принято называть сигнатурами операций). Для однозначного определения компилятором того, какая реально функция или процедура должна быть вызвана при обращении к операции, обычно требуют, чтобы сигнатуры всех операций всех *АТД*, используемых в программе, были различны (мы еще вернемся к этой теме ниже при обсуждении возможностей полиморфизма).

Переменные, используемые для внутреннего представления значений типа называются переменными состояния, а их совокупность состоянием значений.

Иногда, исходя из соображений эффективности допускается прямой доступ к некоторым переменным состояния значений типа (путем использования обычных предопределенных операций чтения и записи). В этом случае переменные состояния, доступные в таком режиме, также специфицируются во внешнем представлении типа.

2.5.2. Реализация типа

Реализация типа представляет собой многовходовой программный модуль, точки входа которого соответствуют набору операций реализуемого типа. Естественно, должно иметься полное соответствие реализации типа его спецификации. Набор статических переменных (в смысле языков Си/Си++) этого модуля образует структуру данных, используемую для представления значений типа. Такой же структурой обладает любая переменная данного абстрактного типа.

Иногда для целей реализации типа бывает полезно иметь в составе его операций такие, которые недоступны для внешнего использования и носят служебный характер. Такие функции и/или процедуры специальным образом помечаются в реализации типа (например, как приватные), и их сигнатуры не включаются во внешнюю спецификацию типа. Переменные состояния, которые

должны быть прямо доступны для внешнего использования, также помечаются специальным образом.

В большинстве современных языков программирования основной концепцией, используемой для описания абстракций в программном коде, является объектно-ориентированный подход. Объектно-ориентированное программирование (ООП) также, как и подход к программированию на основе АТД, является, в некоторой степени, развитием идей о модульном программировании. В ООП результатом абстракции являются *объекты*, представляющие собой абстрактные машины (устройства), обладающие внутренним состоянием, которое можно изменять посылкой сообщений через методы объекта. Объекты обмениваются сообщениями друг с другом и, таким образом, реализуется алгоритм программной модели. В этом смысле объект есть почти то, что подразумевал Парнас в своей работе, когда рассуждал о модульном программировании. Одно из наиболее значимых отличий АТД от ООП заключается в том, что абстрактный тип не содержит данных и, тем более, каких либо состояний этих данных. Свойство абстракции АТД как раз и заключается в том, что абстрактный тип представляет собой поведенческую абстракцию, т.е. кластер операций, посредством которых производится взаимодействие с данными. Поведенческая абстракция целиком и полностью описывается спецификацией абстрактного типа данных. Реализация АТД должна соответствовать спецификации данного абстрактного типа, в этом и заключается абстракция на основе АТД.

На практике, в языках программирования, которые поддерживают концепцию ООП, часто также реализуется и концепция АТД. Для реализации АТД используются средства ООП. Для этого абстрактный тип определяют как класс с определенным интерфейсом. После чего в качестве реализаций данного абстрактного типа выступают его объекты-наследники (в смысле наследования ООП). Работа с реализациями АТД ведется через интерфейс базового объекта, т.е. через операции данного АТД. Для этого обычно используется принцип подстановки, согласно которому объекты-наследники можно подставлять в аргументы, имеющие базовый тип.

Рассмотрим в качестве примера, как АТД реализуются в языке программирования C++. C++ является языком, поддерживающим ООП на основе т.н. механизма классов. Абстрактный тип данных определяется как класс, предоставляющий так называемые чисто виртуальные методы. Для иллюстрации рассмотрим пример реализации АТД стека целочисленных значений на C++:

```
class Stack
{
 public:
  virtual void push(unsigned int elem) = 0;
  virtual unsigned int top() = 0;
  virtual void pop() = 0;
  virtual bool empty() = 0;
};
```

Здесь, ввиду особенностей реализации объектов на C++, объекты АТД Stack передаются в операции и возвращаются из них неявно, в виде скрытых параметров. Поэтому аргументы операций почти ничем не отличаются от тех, что описаны в спецификации АТД «стек» выше. В языке C++ любой класс, объявляющий чисто виртуальный метод (т.е. приравненный нулю при объявлении), называется «абстрактным типом». Это, конечно, еще не абстрактный тип данных, но уже нечто методологически на него похожее. Экземпляры абстрактных типов создавать нельзя. Таким образом, абстрактные типы используются в C++ для определения интерфейсов (кластеров операций), т.е. реализуют в несколько усеченном виде концепцию АТД. Для задания реализации абстрактных типов в языке C++ используется механизм наследования и переопределения, объявленных в базовом классе методов-операций. Вот, например, как будет выглядеть реализация стека на основе массива:

```
class ArrayStack : public Stack
{
 // Массив реализуется на основе типа std::vector стандартной библиотеки C++
 std::vector<unsigned int> array_;
 public:
  void push(unsigned int elem)
  {
   array_.push_back(elem);
  }

 unsigned int top()
 {
  if (array_.empty())
  {
   throw std::invalid_argument("the stack is empty");
  }
  return array_.back();
 }

 void pop()
 {
  if (array_.empty())
  {
   throw std::invalid_argument("the stack is empty");
  }
  array_.pop_back();
 }

 bool empty()
 {
  return array_.empty();
 }
};
```

В реализации используется тип std::vector стандартной библиотеки языка C++. Это параметризованный тип, реализующий концепцию «динамического массива значений». Тип предоставляет методы для вставки/удаления значений в конец массива. Также имеются методы для доступа к последнему элементу массива и проверки массива на непустоту. При невыполнении предусловий на операции pop и top генерируется исключение.

Вопросы для самоконтроля

1. Назовите категории типов данных и их особенности.
2. Опишите абстрактный тип данных.
3. Какие вы знаете разновидности целого, вещественного типов данных?
4. Какой тип данных является базовым для массива?
5. Приведите описание структурного типа на примере.

ГЛАВА 3
ЛИНЕЙНЫЕ СПИСКИ

3.1. Линейные списки – понятия и определения

Практически всегда данные, обрабатываемые на ЭВМ, находятся между собой в определенных структурных отношениях, т.е. логически связаны. Стандартные средства языков программирования позволяют в некоторых случаях отразить эту связь, используя данные составных (сложных) типов:

1. Массивы (здесь существуют отношения следования).
2. Множества (здесь существуют отношения принадлежности).
3. Записи (физическое объединение разнотипных данных).

Однако в реальном мире связи между объектами могут быть весьма сложными: древовидными, многосвязными и пр. Поэтому для эффективного моделирования реального мира на ЭВМ необходимо знать способы представления различных структур в памяти ЭВМ и методы работы с ними.

Для обозначения структур наиболее общего вида используется термин список (или списковая структура).

Список – совокупность связанных узлов.

На рис. 3.1 приведен пример списковой структуры.

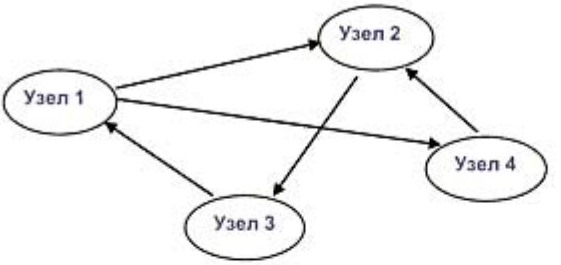


Рис. 3.1. Списковая структура

Линейный список (рис. 3.2) – это частный случай списка с $n>=0$ узлами x_1, x_2,..., x_n, структурные свойства которого характеризуются следующими утверждениями:

а) Если $n>0$, то x_1 – первый узел (т.е. ему не предшествует никакой другой узел);

б) Если $1<k<n$, то узлу x_k предшествует узел x_{k-1} , а за ним следует узел x_{k+1} ;

в) x_n является последним узлом.

Рис. 3.2. Линейный список

К линейным спискам применим специальный набор операций:

1) получить доступ к k – ому узлу списка, чтобы проанализировать или изменить значения его компонент;
2) включить новый узел непосредственно перед k – тым узлом.
3) исключить k – й узел;
4) объединить два или более линейных списка в один список;
5) разбить линейный список на два или более списка;
6) сделать копию линейного списка;
7) определить количество узлов в списке;
8) выполнить сортировку узлов списка в возрастающем порядке, по значениям некоторых компонент узлов;
9) найти в списке узлы с заданным значением определенной компоненты[3].

Все перечисленные операции редко требуется выполнять одновременно, поэтому удобно классифицировать списки по набору применяемых к ним операций.

Так, очень часто встречаются линейные списки, в которых включение, исключение или доступ к значениям почти всегда производятся в первом или последнем узлах. Такие списки имеют специальные названия.

Стек – линейный список, в котором все включения, исключения и обычно всякий доступ осуществляются в одном конце списка.

Очередь – линейный список, в котором все включения проводятся на одном конце списка, а все исключения и обычно всякий доступ – на другом его конце.

Дек (очередь с двумя концами) – линейный список, в котором все включения, исключения и обычно всякий доступ делаются на обоих концах списка.

Стек – список типа LIFO (Last In First Out) – последний пришел, первый ушел (рис. 3.3).

Рис. 3.2. Линейный список

К линейным спискам применим специальный набор операций:

1) получить доступ к k – ому узлу списка, чтобы проанализировать или изменить значения его компонент;
2) включить новый узел непосредственно перед k – тым узлом.
3) исключить k – й узел;
4) объединить два или более линейных списка в один список;
5) разбить линейный список на два или более списка;
6) сделать копию линейного списка;
7) определить количество узлов в списке;
8) выполнить сортировку узлов списка в возрастающем порядке, по значениям некоторых компонент узлов;
9) найти в списке узлы с заданным значением определенной компоненты[3].

Все перечисленные операции редко требуется выполнять одновременно, поэтому удобно классифицировать списки по набору применяемых к ним операций.

Так, очень часто встречаются линейные списки, в которых включение, исключение или доступ к значениям почти всегда производятся в первом или последнем узлах. Такие списки имеют специальные названия.

Стек – линейный список, в котором все включения, исключения и обычно всякий доступ осуществляются в одном конце списка.

Очередь – линейный список, в котором все включения проводятся на одном конце списка, а все исключения и обычно всякий доступ – на другом его конце.

Дек (очередь с двумя концами) – линейный список, в котором все включения, исключения и обычно всякий доступ делаются на обоих концах списка.

Стек – список типа LIFO (Last In First Out) – последний пришел, первый ушел (рис. 3.3).

3.2. Последовательное размещение узлов линейного списка в памяти

При таком размещении соседние узлы списка занимают смежные участки памяти и для их представления в языках программирования удобно использовать массивы. Сам массив используется как "хранилище данных", а для работы со списком необходимо иметь еще один или два указателя, отличающих:

а) верх – для стека;
б) начало – конец – для очереди;
в) левый, правый конец – для дека.

Сконструируем набор процедур для стека (статическое размещение в памяти)

```
#include <iostream>
#include <cstdlib>
using namespace std;

#define MAX 100

char *stack[MAX]; // Массив для стека
int tos = 0; // Для вершины стека

void Push(char *i) // Функция добавления элемента в стек
{
  if(tos >= MAX) // Проверка на заполненность стека
  {
    cout << «Стек полон!» << endl;
    return;
  }
  strcpy(stack[tos],i); // Записываем строку в стек
  tos = tos + 1; // Увеличиваем вершину стека
}

int Size() // Для определения размера стека
{
  if (tos >= 0)
          return tos;// Если больше нуля, то возвращаем текущее
значение вершины стека
  else
          return 0; // иначе возвращаем ноль
}

char* Pop(void) // Функция вытаскивания элемента из головы
{
  char *str = new char[30];
  tos = tos - 1; // Сдвигаемся от головы вниз
  if(tos < 0) // Если вершина меньше нуля, то стек пустой
```

```
  {
    cout << «Стек пуст!» << endl;
    return 0;
  }
  strcpy(str, stack[tos]); //иначе в буфер помещаем текущее значе-
ние вершины
  return str; // возвращаем это значение
}

void DeleteStack(char *element) // Функция удаления
{
  int razmer = Size(); // Получаем размерность стека
  char *buf[MAX]; // Для буфера, используется при удалении
  for(int i = 0;i<MAX;i++) // Выделение памяти
  {
    buf[i] = new char[30];
  }
  int c = 0; // Счетчики
  int ravn = 0;
  for(int i = 0;i <= Size();i++) // Цикл для удаления
  {
    /*Проверяем,если не равны элемент из
    стека и элемент введенный с клавиатуры,
    то копируем в буфер
    увеличиваем счетчик
    и удаляем полностью данную ячейку из стека*/
    if (strcmp(element,stack[i])!=0)
    {
      strcpy(buf[c],stack[i]);
      c = c + 1;
      delete stack[i];
    }
    else //иначе просто удаляем полностью ячейку из стека
    {
      delete stack[i];
      ravn = ravn + 1;
    }
  }
  tos = 0;
  /*Выделяем память для стека, так как она была
  удалена
  и кладем в стек значения*/
  for (int i=0;i<=(razmer-ravn);i++)
  {
    stack[i] = new char[30];
    Push(buf[i]);
  }
}

int main()
{
```

```
  for (int i = 0; i < 4; i++) // Выделение памяти для элементов
стек
  {
    stack[i] = new char[30];
  }

     for (int i = 0; i < 4; i++)
          Push(«1»);

     for (int i = 0; i < 4; i++)
          cout << stack[i] << « «;

     Pop();
     cout << endl;

     for (int i = 0; i < 3; i++)
          cout << stack[i] << « «;

     cout << endl;

     system(«PAUSE»);
     return 0;
}
```

3.3. Связанное хранение узлов линейного списка в памяти

При таком размещении узлы списка занимают несмежные участки памяти, и для их связывания используется ссылочный механизм.

Набор процедур для работы со связанным стеком

Реализация стека и пример работы с ним

```
#include <iostream>
#include <cassert> // для assert
#include <iomanip> // для setw
using namespace std;

template <typename T>
class Stack
{
private:
  T *stackPtr;            // указатель на стек
  const int size;           // максимальное количество элементов в
стеке
  int top;              // номер текущего элемента стека
public:
```

```
  Stack(int = 10);          // по умолчанию размер стека равен 10
элементам
  Stack(const Stack<T> &);      // конструктор копирования
  ~Stack();              // деструктор

  inline void push(const T &);    // поместить элемент в вершину
стека
  inline void pop();         // удалить элемент из вершины стека и
вернуть его
  inline void printStack();      // вывод стека на экран
  inline int getStackSize() const; // получить размер стека
};

// конструктор Стека
template <typename T>
Stack<T>::Stack(int maxSize) :
  size(maxSize) // инициализация константы
{
  stackPtr = new T[size]; // выделить память под стек
  top = 0; // инициализируем текущий элемент нулем;
}

// деструктор Стека
template <typename T>
Stack<T>::~Stack()
{
  delete [] stackPtr; // удаляем стек
}

// функция добавления элемента в стек
template <typename T>
inline void Stack<T>::push(const T &value)
{
  // проверяем размер стека
  assert(top < size); // номер текущего элемента должен быть мень-
ше размера стека
  stackPtr[top++] = value; // помещаем элемент в стек
}

// функция удаления элемента из стека
template <typename T>
inline void Stack<T>::pop()
{
  // проверяем размер стека
  assert(top > 0); // номер текущего элемента должен быть больше 0
  stackPtr[--top]; // удаляем элемент из стека
}

// вывод стека на экран
template <typename T>
inline void Stack<T>::printStack()
```

```
{
  for (int ix = top - 1; ix >= 0; ix--)
    cout << «|» << setw(4) << stackPtr[ix] << endl;
}

// вернуть размер стека
template <typename T>
inline int Stack<T>::getStackSize() const
{
  return size;
}

int main()
{
  Stack<char> stackSymbol(5);
  int ct = 0;
  char ch;

  while (ct++ < 5)
  {
          cout << «Введите новый элемент:» << endl;
    cin >> ch;
    stackSymbol.push(ch); // помещаем элементы в стек
  }

  cout << endl;

  stackSymbol.printStack(); // печать стека

  cout << «\n\nУдалим элемент из стека\n»;
  stackSymbol.pop();

  stackSymbol.printStack(); // печать стека

     system(«PAUSE»);
     return 0;
}
```

Набор процедур для работы со связанной очередью

Реализация очереди и пример работы с ней

```
#include <cassert>
#include <iostream>
#include <cassert> // для assert

using namespace std;

template<typename T>
class Queue
```

```
{
private:
  T *queuePtr;    // указатель на очередь
  const int size; // максимальное количество элементов в очереди
  int begin,      // начало очереди
    end;      // конец очереди
  int elemCT;    // счетчик элементов
public:
  Queue(int = 10);      // конструктор по умолчанию
  ~Queue();         // деструктор
  void enqueue(const T &); // добавить элемент в очередь
  void dequeue(); // удалить элемент из очереди
  void printQueue();
};

// конструктор по умолчанию
template<typename T> Queue<T>::Queue(int sizeQueue) :
  size(sizeQueue), // инициализация константы
  begin(0), end(0), elemCT(0)
{
  // дополнительная позиция поможет нам различать конец и начало
очереди
  queuePtr = new T[size + 1];
}

// деструктор класса Queue
template<typename T> Queue<T>::~Queue()
{
  delete [] queuePtr;
}

// функция добавления элемента в очередь
template<typename T>
void Queue<T>::enqueue(const T &newElem)
{
  // проверяем, ести ли свободное место в очереди
  assert(elemCT < size);

  queuePtr[end++] = newElem;

  elemCT++;

  // проверка кругового заполнения очереди
  if (end > size)
    end -= size + 1; // возвращаем end на начало очереди
}

// функция удаления элемента из очереди
template<typename T>
void Queue<T>::dequeue()
{
```

```
  // проверяем, есть ли в очереди элементы
  assert(elemCT > 0);

  T returnValue = queuePtr[begin++];
  elemCT--;

  // проверка кругового заполнения очереди
  if (begin > size)
    begin -= size + 1; // возвращаем behin на начало очереди
}
template<typename T>
void Queue<T>::printQueue()
{
  cout << «Очередь: «;

  if (end == 0 && begin == 0)
    cout << « пустая\n»;
  else
  {
    for (int ix = end; ix >= begin; ix--)
      cout << queuePtr[ix] << « «;
    cout << endl;
  }
}

int main ()
{
  Queue<char> myQueue(14);

  myQueue.printQueue(); // вывод очереди

  int ct = 1;
  char ch;

  // добавление элементов в очередь
  while (ct++ < 14)
  {
    cin >> ch;
    myQueue.enqueue(ch);
  }

  myQueue.printQueue(); // вывод очереди

  // удаление элемента из очереди
  myQueue.dequeue();
  myQueue.dequeue();
  myQueue.dequeue();

  myQueue.printQueue(); // вывод очереди

     system(«PAUSE»);
     return 0;
}
```

3.4. Циклические списки

Циклический список не имеет первого и последнего элементов. Необходимо, следовательно, ввести такие элементы. Удобно использовать внешний указатель, указывающий на *последний* элемент, что автоматически делает следующий за ним элемент первым (рис. 3.6).

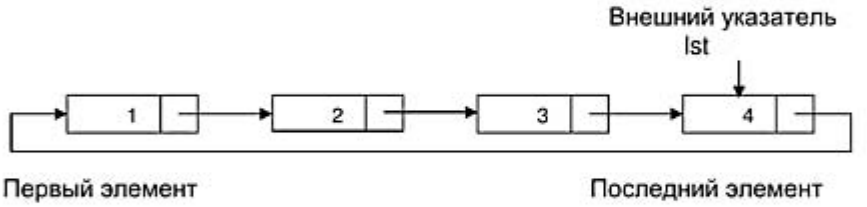


Рис. 3.6. Циклический список

Можно также ввести соглашение, по которому нулевой указатель представляет пустой циклический список. Циклический список может быть использован для реализации стека или очереди.

Голова (заголовок) циклического списка – специальный узел, который распознается по специальному коду, задаваемому в поле данных. Просмотр циклического списка заканчивается по достижению элемента заголовка. Голова помещается в конце или в начале списка.

Указатель циклического списка удобно адресовать к *последнему* узлу, т.к. в этом случае достаточно просто получить и адрес *первого* узла, так что циклический список может быть использован как обычная очередь (рис. 2.7). Необходимо лишь скорректировать процедуры включения и исключения элемента.

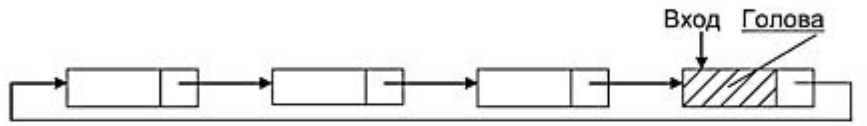


Рис. 3.7. Циклический список с заголовком

Реализация циклического списка и пример работы с ним

```
#include <iostream>
using namespace std;

template <class T>
class LoopList // Шаблон циклического списка
{
```

```
private:
     struct node          // структура, представляющая единичный
элемент списка
  {
          T data;         // переменная необходимая для хранения
данных в элементе списка
          node *next; // указатель на следующий элемент списка
          node *prev; // указатель на предыдущий элемент списка
  };
  node *head;         // указатель первый добавленный элемент
  node *curr;         // указатель на текущий элемент
public:
     int length;          // количество элементов в списке

  LoopList();         // конструктор
  LoopList (T x);     // конструктор с параметром
  ~LoopList(); // деструктор

  void Init(); // текущий элемент ссылается на первый
  void Push(T data); // добавляем новый элемент в список
  T Pop();            // извлекаем текущий элемент
  void Print();       // печать списка
  void Clear();       // очистить весь список
  int isNoEmpty();// проверка состояния списка (если список не
пуст)

     void next();     // перейти к следующему элементу
  void prev(); // перейти к предыдущему элементу
};

// конструктор кольцевого списка
template <class T>
LoopList<T>::LoopList()
{
   head = NULL; // обнуляем указатель на первый элемент
   curr = NULL; // обнуляем указатель на текущий элемент
   length = 0; // обнуляем количество элементов в списке
}

// деструктор кольцевого списка
template <class T>
LoopList<T>::~LoopList()
{
   Clear(); // очистить список
}

// приведение списка к состоянию текущий элемент - первый не уда-
ленный
template <class T>
void LoopList<T>::Init()
{
```

```
   curr = head; // устанавливаем текущий элемент на первый не удаленный
}

// реализация перехода к следующему элементу
template <class T>
void LoopList<T>::next()
{
     if(isNoEmpty()) // если список не пуст
    curr = curr->next; // установить текущую позицию на следующую
}

// реализация перехода к предыдущему элементу
template <class T>
void LoopList<T>::prev()
{
  if(isEmpty()) // если список не пуст
    curr = curr->prev; // установить текущую позицию на предыдущую
}

// реализация проверки списка на наличие элементов
template <class T>
int LoopList<T>::isNoEmpty()
{
   if (curr == NULL) // если текущая позиция имеет нулевое значение
      return 0; // вернуть «ложь»
   else // иначе
      return 1; // вернуть «истина»
}

// реализация добавления элемента
template <class T>
void LoopList<T>::Push(T data)
{
  node *inserted; // создать новый указатель на элемент
  inserted = new node;   // выделить память под элемент
  inserted->data = data; // установить входной параметр в поле данных элемента
  if (!isNoEmpty()) // если список пуст
  {
    head = inserted; // установить указатель первого элемента на новый элемент
    curr = inserted; // установить указатель текущего элемента на новый элемент
    curr->next = inserted; // установить указатель следующего элемента на новый элемент
    curr->prev = inserted; // установить указатель предыдущего элемента на новый элемент
  }
  else // если список не пуст
```

```
  {
    inserted->next = curr->next; // перенаправляем указатель следующего элемента в добавляемом
    inserted->next->prev = inserted;    // перенаправляем указатель следующего элемента на добавляемый
    curr->next = inserted; // перенаправляем следующий указатель на добавляемый
    inserted->prev = curr; // перенаправляем предыдущий добавляемого на текущий
  }
  length++; // увеличиваем количество элементов в списке
  curr = inserted; // устанавливаем текущий указатель на добавленный
}

// реализация извлечения текущего элемента
template <class T>
T LoopList<T>::Pop()
{
  T tag; // переменная под возвращаемое значение
  if (!isNoEmpty()) return 0; // если список пуст вернуть «ложь»
  node *temp = curr; // сохраняем указатель на текущий элемент
  tag = temp->data; // присваиваем переменной значение данных текущего элемента
  if (length == 1) // если элемент единственный в списке
  {
    head = NULL; // обнулить значение первого элемента
    curr = NULL; // обнулить значение текущего элемента
  }
  else // если элемент не единственный
    {
    curr->next->prev = curr->prev; // связываем следующий и предыдущий
    curr->prev->next = curr->next; // связываем предыдущий и следующий
    curr = curr->next; // перенаправляем текущий элемент на следующий
    }
  if(temp == head) // если удаляемый элемент - первый добавленный
    head = head->next; // перенаправить первый на следующий
  length--; // уменьшить количество элементов
  delete temp; // удалить предыдущий текущий элемент
  return tag; // вернуть данные удаленного элемента
}

// реализация вывода всех элементов списка
template <class T>
void LoopList<T>::Print()
{
  if (isNoEmpty()) // если список не пуст
  {
```

```
    node *tempCar = head; // сохраняем указатель на текущий эле-
мент
    for (int i = 0; i < length; i++) // заводим цикл на количество
элементов списка
    {
    cout << tempCar->data << « «; // выводим данные текущего эле-
мента
    tempCar = tempCar->next ; // переходим к следующему элементу
    }
    cout << endl;
  }
  else // если список пуст
    cout<< «Список пуст» << endl;
}

// реализация очистки всего списка
template <class T>
void LoopList<T>::Clear()
{
  for(int i = 0; i < length;) // заводим цикл на количество эле-
ментов списка
    Pop(); // извлекаем текущий элемент
}

int main()
{
  LoopList<int> *L = new LoopList<int>();

     for(int i = 0; i < 10; i++)
          L->Push(i*i); // заполняем кольцевой список квадратами
чисел

     L->Print(); // печать кольцевого списка
     cout << endl;

     L->Pop(); // извлекаем элемент из кольцевого списка

     L->Print(); // печать кольцевого списка
     cout << endl;

     L->Clear(); // очищаем кольцевой список
     L->Print(); // печать кольцевого списка
     cout << endl;

  system(«PAUSE»);
  return 0;
}
```

3.5. Двунаправленные связанные списки

Хотя циклический список имеет свои преимущества перед линейным, он имеет также ряд недостатков:

- циклический список нельзя просматривать в обратном направлении;
- располагая только значением указателя для данного элемента, удалить последний невозможно (только следующий за ним).

При необходимости иметь такую возможность можно, воспользовавшись соответствующей структурой данных, называемой двунаправленным связанным списком.

Двунаправленный связанный список – это линейный список, в котором каждый элемент содержит два указателя: один указатель указывает на предшествующий элемент, а другой – на последующий.

Двунаправленные связанные списки могут быть линейными и циклическими, могут содержать или не содержать элемент заголовка (рис.3.8, 3.9, 3.10).

Рис. 3.8. Двунаправленный связанный линейный список

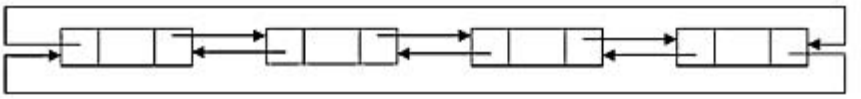


Рис. 3.9. Двунаправленный циклический список без заголовка

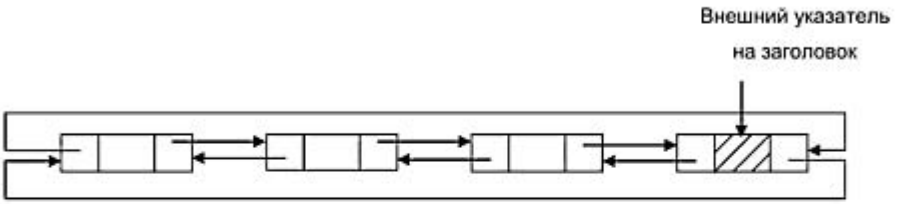
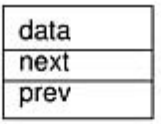


Рис. 3.10. Двунаправленный циклический список с заголовком

Будем считать, что элементы двунаправленного связанного списка содержат три поля:

- data – поле данных, содержащее информацию, хранимую в элементе;
- next – ссылка на следующий элемент списка
- prev – ссылка на предыдущий элемент списка.

Описание узла:




```
struct node
{
 T data;
node* next;
node* prev;
 };
```

Преимущества двунаправленных списков:

1) список можно просматривать в любом направлении;
2) можно исключать узел по его ссылке, а не по ссылке на соседний элемент;
3) можно включать новые узлы как слева, так и справа от заданного узла.

Реализация двунаправленного списка и пример работы с ним

```
#include <iostream>
using namespace std;

template <typename T>
class List // Шаблон двунаправленного списка
{
private:
  struct node // структура, представляющая единичный элемент спи-
ска
  {
    T data; // переменная необходимая для хранения данных в эле-
менте списка
    node* next; // указатель на следующий элемент списка
    node* prev; // указатель на предыдущий элемент списка
  };
  node* head; // указатель первый добавленный элемент
public:
     List(); // конструктор
     ~List(); // деструктор

  void PushBack(T); // добавляем новый элемент в конец списка
  void PushFront(T); // добавляем новый элемент в начало списка
  void PopBack(); // извлекаем текущий элемент из конца
  void PopFront(); // извлекаем текущий элемент из начала
  void PrintList(); // печать списка
  void PrintReverse(); // печать списка в обратном направлении
};

// конструктор двунаправленного списка
template <typename T>
List<T>::List()
{
     head = new (node); // создаем новый узел
     head->next = NULL; // обнуляем указатель на следующий элемент
}

// деструктор двунаправленного списка
```

```
template <typename T>
List<T>::~List()
{
  node *p, *p1;
  p = head; // принять значение первого элемента
  p1 = p->next; // перенаправить на следующий

     while(p1 != NULL) // обход списка
  {
    p = p1; // присвоить следующее значение
    p1 = p1->next; // перенаправить на следующий
    delete p;
  }
}

// реализация добавления элемента в конец
template <typename T>
void List<T>::PushBack(T el)
{
  node* p,*p1; // создание временных узлов
  p = head; // взять значение первого
  while(p->next != NULL) // пока список не пуст
          p = p->next; // перенаправить на следующий

  p1 = new (node); // создать новый узел
  p1->data = el; // взять значение добавляемого элемента
  p1->next = NULL; // ссылку на следующий обнулить

  p1->prev = p; // предыдущая ссылка берет значение бывшей текущей
  p->next = p1; // следующая ссылка у бывшей текущей становится
добавляемым значением
}

// реализация добавления элемента в начало
template <typename T>
void List<T>::PushFront(T el)
{
  node* p; // создание временных узлов
  p = new (node); // создать новый узел
  p->data = el; // взять значение добавляемого элемента
  if(head->next == NULL) // если список не пуст
  {
    p->next = head->next; // добавляемый элемент становится первым
    head->next = p;
    p->prev = head;
  }
  else // если список пуст
  {
    p->next = head->next;
    head->next->prev = p;
    head->next = p;
```

```
    p->prev = head;
  }
}

// реализация извлечения элемента из конца
template <typename T>
void List<T>::PopBack()
{
  node* p,* p1; // создание временных узлов
  p = head; // взять значение первого
  p1 = p->next; // перенаправить на следующий
  while(p1->next != NULL) // пока список не пуст
  {
    p = p1; // взять значение предыдущего элемента
    p1 = p1->next; // перенаправить на следующий
  }
  p->next = NULL; // ссылку на следующий обнулить
  delete p1; // удалить элемент
}

// реализация извлечения элемента из начала
template <typename T>
void List<T>::PopFront()
{
  node* p; // создание временного узла
  p = head->next; // принять значение второго элемента
  head->next = p->next; // обмен ссылками втрого и первого элемен-
та
  p->next->prev = head;
  delete p; // удалить элемент
}

// реализация вывода всех элементов списка
template <typename T>
void List<T>::PrintList()
{
  node* p; // создать временный узел
  p = head->next;
  while(p != NULL) // пока список не пуст
  {
    cout << p->data << " "; // напечатать элемент
    p = p->next; // перенаправить на следующий
  }
}

int main()
{
  List<int> *L = new List<int>();

     for(int i = 0; i < 10; i++)
```

```
        L->PushBack(i*i); // заполняем двунаправленный список
квадратами чисел

    L->PrintList(); // печать двунаправленного списка
    cout << endl;

    L->PopBack(); // извлечь элемент из конца
    L->PopFront(); // извлечь элемент из начала

    L->PrintList(); // печать двунаправленного списка
    cout << endl;

  system("PAUSE");
  return 0;
}
```

3.6. Многосвязные линейные списки

Иногда в инженерной практике возникают задачи, в которых необходимо использовать разреженные матрицы. Разреженная матрица – это матрица высокого порядка, в которой большинство элементов равно нулю. Целесообразно оперировать матрицей так, будто матрица представлена в памяти вся, но для экономии не отводить место на нулевые элементы, то есть хранить только значащие элементы (ненулевые).

Рассмотрим такое представление, которое состоит из циклически связанных списков для каждой строки и каждого столбца. Каждый узел матрицы представим в виде записи, содержащей пять полей.

Влево	Индекс строки	Индекс столбца	Значение	Вверх

«Влево», «вверх» – связи со следующим ненулевым элементом слева в строке или сверху в столбце.

Пример. Использование двухсвязных двунаправленных списков для представления разреженных матриц.

$$\begin{pmatrix} 14 & 0 & 0 & 0 \\ 10 & 0 & 13 & 0 \\ 0 & 0 & 0 & 0 \\ -15 & 0 & -21 & 2 \end{pmatrix}$$ – разреженная матрица.

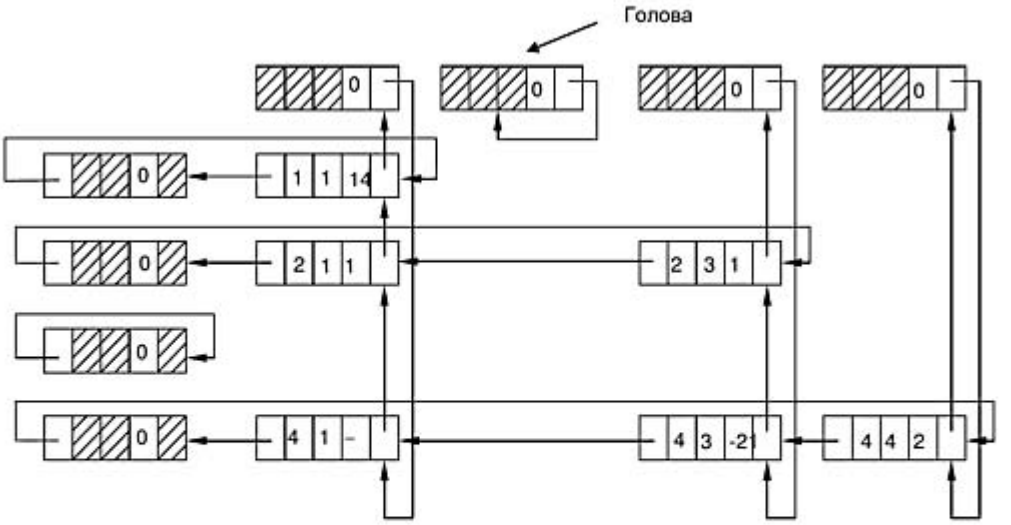


Рис. 3.11. Представление разреженной матрицы в виде многосвязного линейного списка

Связь «влево» в головах горизонтальных списков является адресом самого правого значения в строке, а связь «вверх» в головах вертикальных списков является адресом самого нижнего значения в столбце.

При последовательном распределении памяти матрица размера 200x200 заняла бы 40 000 слов, в то же время разреженная матрица 200x200 с большим числом нулевых элементов может быть представлена в вышеописанной форме в памяти, содержащей приблизительно 4000 слов (в 10 раз меньше расход памяти).

Разреженные матрицы применяются в алгоритмах решения линейных уравнений, обращения матриц и линейного программирования (симплекс – метод).

Вопросы для самоконтроля

1. Назовите операции, применяемые к линейным спискам.
2. Какие существуют способы хранения линейных списков в памяти?
2. Что такое «заголовок» циклического списка?
4. Где находится вход в циклический список?
5. Перечислите наиболее часто используемые операции при работе со списковыми структурами.
6. Назовите разновидности двунаправленных связанных списков.
7. Что такое разреженная матрица, когда она используется?

ГЛАВА 4
ДРЕВОВИДНЫЕ СТРУКТУРЫ

4.1. Основные понятия и определения

В рассмотренных ранее линейных списках каждый узел был связан лишь с одним узлом (в двунаправленных с двумя, но это не меняет сути дела). Перейдем к рассмотрению структур, в которых каждый узел может иметь несколько связей.

Древовидная *структура* – это конечное множество, содержащее один или более узлов (n) такое, что:

- имеется один специально обозначенный узел, называемый *корнем* данного дерева;
- остальные узлы содержатся в m>= 0 попарно непересекающихся множествах T_1, T_2, … , T_m, каждое из которых в свою очередь является деревом. Деревья T_1, T_2, … , T_m называются поддеревьями данного корня.

Это *рекурсивное* определение (*рекурсия* – способ описания функций или процессов через самих себя), здесь дерево из n узлов определено через совокупность деревьев с меньшим числом узлов. *Список* есть *древовидная* структура, у которой каждый узел имеет не более *одного* «*поддерева*». Поэтому последовательность (список) называется также *вырожденным деревом*.

Любой узел может вырасти в поддерево, а узлы поддерева в свою очередь также могут разрастаться аналогично тому, как в природе почки молодого дерева вырастают в ветви, имеющие собственные почки, которые в свою очередь дают новые ветви, и т.д.

Дерево представляет собой иерархию элементов, называемых *узлами*. На самом верхнем уровне иерархии имеется только один узел – *корень. Каждый узел, кроме корня,* связан *с одним* узлом на более высоком уровне, называемым *исходным узлом* для данного узла. Каждый элемент может быть связан с одним или несколькими элементами на более низком уровне, которые называются *порожденными.* Элементы, расположенные в конце ветви, то есть не имеющие порожденных, называются *листьями*[3]. Дерево обычно изображается в перевернутом виде с *корнем вверху, листьями внизу* (рис. 4.1).

Дерево может быть определено как иерархия узлов с парными связями, в которой:

- самый верхний уровень иерархии имеет один узел, называемый *корнем;*
- все узлы, кроме корня, связываются с одним и только одним узлом на более высоком уровне по отношению к ним самим.

Степень узла – число непосредственных потомков внутреннего узла.

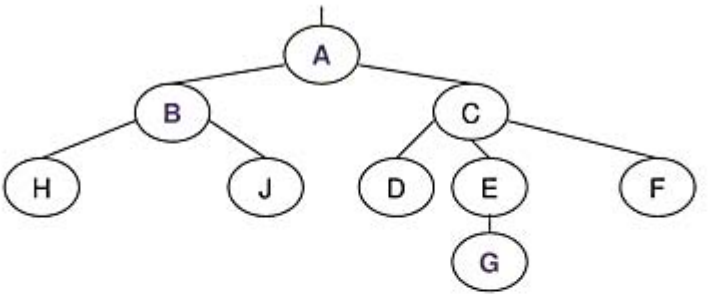


Рис. 4.1. Древовидная структура

В примере (рис. 4.1) степень узлов: A,B – 2, C – 3, E – 1; H,J,D,G,F – 0.

Из-за рекурсивности определения каждый узел дерева можно считать корнем некоторого поддерева. При этом число поддеревьев (порожденных узлов) данного узла называется его степенью.

Если x находится на уровне i, то говорим, что y на уровне i+1. Узел y, который находится непосредственно под узлом x, называется *непосредственным потомком* x (или сыном, или подчиненным). Узел x называется непосредственным предком y (или исходным, или отцом). Считается, что корень дерева находится на уровне 1 (рис. 4.2).

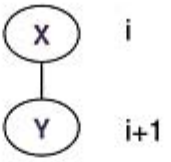


Рис. 4.2. Фрагмент дерева

Уровень узла по отношению к дереву T определяется следующим образом:

- корень дерева имеет уровень 1;
- корни поддеревьев, непосредственно входящих в дерево, имеют уровни 2,3, и т.д.

Глубина (высота) дерева – максимальный уровень узла дерева (в примере дерево имеет глубину, равную четырем) или число уровней в дереве.

Степень дерева – максимальная степень его узлов (в примере дерево имеет степень = 3, т.к. максимальная степень узла = 3).

Лист (концевой узел, *терминальный элемент*) – элемент, не имеющий потомков; это узел степени 0 (на рис. 4.1 H,J,D,G,F – листья).

Внутренний узел (или узел *разветвления*) – это узел, не являющийся концевым (т.е. не являющийся листом).

Момент – число узлов (в примере – 9).

Вес – число листьев (в примере – 5).

Основание – число корней (в примере – 1).

Длина пути x – число ветвей (ребер), которые необходимо пройти, чтобы продвинуться от корня к узлу x. Корень имеет длину пути 1, его непосредственный потомок – длину пути 2 и т.д. Вообще узел на уровне i имеет длину пути i.

Длина пути дерева – сумма длин путей всех его узлов. Она также называется *длиной внутреннего пути*.

Средняя длина пути P_i есть:

$P_i = \frac{1}{n}\sum_i n_i * i$, где n_i – число узлов на уровне i.

Лес – множество (обычно упорядоченное), состоящее из некоторого (может быть равного 0) числа непересекающихся деревьев.

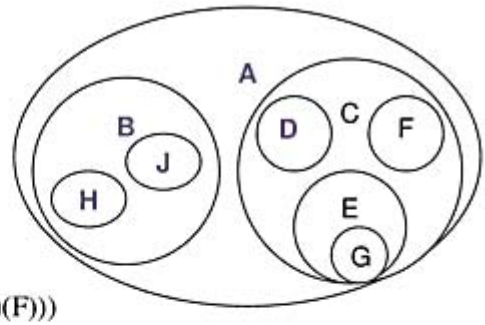
Дерево *сбалансировано*, если для каждого его узла высота левого и правого поддеревьев различается не более, чем на 1.

Высота сбалансированного дерева $h \le \log_2 n$, где n – количество узлов.

4.2. Способы изображения древовидных структур

Рассмотрим 4 способа изображения деревьев.

1. В виде вложенных множеств:



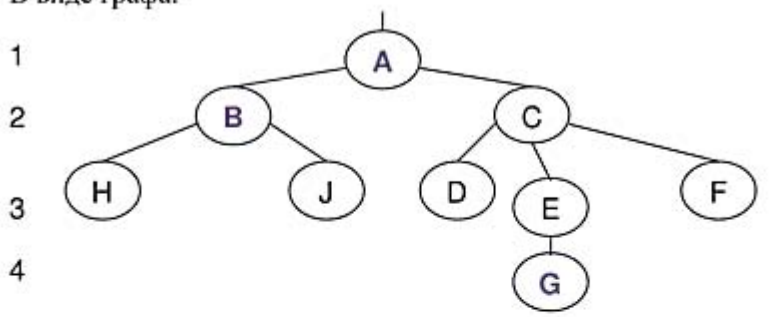
2. В виде вложенных скобок:

(A(B(H)(J))(C(D)(E(G))(F)))

3. В виде ступенчатой записи:

A	B C	H J DE	G

4. В виде графа:



4.3. Упорядоченные и ориентированные деревья

Если в структуре дерева имеет значение относительный порядок поддеревьев, то такое дерево называется упорядоченным.

Упорядоченное дерево – дерево, у которого ветви каждого узла упорядочены.

Ориентированные деревья – это деревья, отличающиеся друг от друга только относительным порядком поддеревьев узлов. В этом случае эти деревья не считают различными[4].

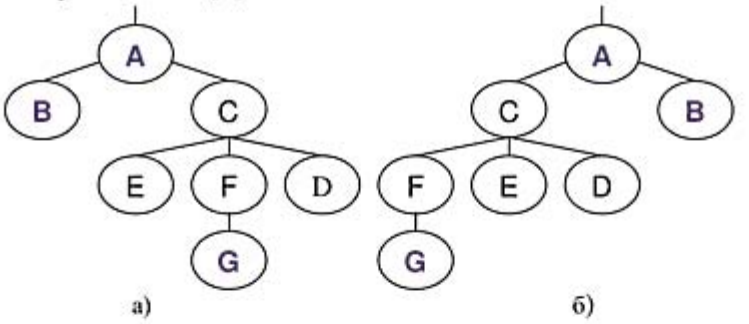


Рис. 4.3. Упорядоченное и ориентированное деревья

Деревья, изображенные на рис. 4.3 (а, б) различаются как упорядоченные, хотя как ориентированные деревья они считались бы одинаковыми.

Бинарное дерево – конечное множество узлов, которое или пусто, или состоит из корня и двух непересекающихся бинарных деревьев, называемых левым и правым поддеревьями данного корня. Сама природа представления данных в компьютере устанавливает *точный порядок* для всякого дерева, и поэтому в большинстве случаев наибольший интерес представляют упорядоченные деревья.

Бинарное дерево – упорядоченное дерево степени 2. Бинарные деревья, изображенные на рис. 4.4, различны между собой. В одном случае корень имеет пустое правое поддерево, а в другом – левое поддерево пусто, хотя как *деревья* они одинаковы.

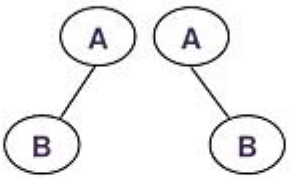


Рис. 4.4. Бинарные деревья

4.4. Построение сбалансированного дерева

Прежде чем обсудить, как лучше использовать деревья и как выполнять операции с деревьями, покажем на примере, как программа может строить дерево[1].

Требуется построить дерево с n узлами и наименьшей высотой. Значением узлов будут n чисел, прочитанных из входного файла. Чтобы достичь минимальной высоты при данном числе узлов n, нужно распределить узлы равномерно слева и справа от корневого узла. Такое распределение удобно записать в виде рекурсивного алгоритма:

1. Взять первое число и поместить его в узел.
2. Из очередных nl = n div 2 чисел построить левое поддерево (тем же способом, что и все дерево).
3. Из оставшихся nr = n – nl – 1 чисел построить правое поддерево.

Узел бинарного дерева удобно представить в виде записи:

key	ключ узла дерева
left	ссылка на левое поддерево
right	ссылка на правое поддерево

Описание структуры узла бинарного дерева (на языке C++):

```
struct node
{
 int key;
 node *left;
 node *right;
 int count;
};
```

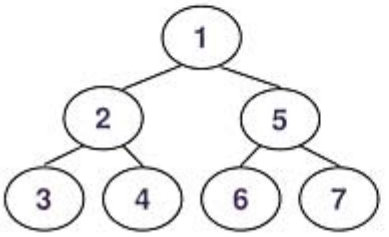
Программа построения и печати сбалансированного дерева (на языке C++)

```
#include<stdafx.h>
#include<iostream>
#include<conio.h>
#include<math.h>
using namespace std;
     struct node
     {
     int key;
     node *left,*right;
     };
//Построение бинарного дерева
     node *tree(int n)
      {
           node *z;
           int x=0;
           int nl=0, nr=0;
           if (n==0)
      return NULL;
           else
      {
           nl=n/2;
           nr=n-nl-1;
           cout<<«Введите ключ» ;
           cin>>x;
           z=NULL;
           z= new node();
           z->key=x;
           z->left=tree(nl);
           z-> right=tree(nr);
           return z;
      }
      }
//Печать бинарного дерева
     void printtree(node *p, int h)
      {
           int i;
           if (p!=NULL)
           {
           printtree(p->left, h+1); //Печать левого поддерева
          for(i=1; i<=h; i++)
           cout<<" ";
           cout<<p->key<<«\n»;
           printtree(p->right,h+1); //Печать правого поддерева

           }
      }
//Основная программа
void main()
     {
```

```
node *tr;
tr=NULL;
tr=new node();
int n, x=0, h=0;
system("cls");
cout<<<«Введите число узлов дерева» ;
cin>>n;
tr=NULL;
tr=tree(n);
system("cls");
cout<<<«Печать дерева» <<endl;
printtree(tr,0);
getch();
}
```

Например, при n= 7 (ключи 1,2,3,4,5,6,7) этой программой будет построено дерево и распечатано в виде ступенчатой записи:




```
_   _   3
_   2
_       4
1   _
_   _   6
_   5
_       7
_   _
```

4.5. Прохождение бинарных деревьев

В ЭВМ обычные деревья представляются в виде некоторых бинарных деревьев. Разберем свойства бинарных деревьев. Из определения бинарного дерева вытекает естественный способ представления бинарных деревьев в ЭВМ. Для этого достаточно иметь две связи right(правая) и left(левая) в каждом узле и переменную связи T, которая является «указателем на это дерево». Если дерево пусто, то T = NULL, в противном случае T – адрес корня этого дерева, а left(T) и right(T) – указатели соответственно на левое и правое поддеревья этого корня[1,3].

Эти правила рекурсивно определяют представление всякого бинарного дерева в памяти ЭВМ, например, дерево (рис. 4.5) будет представлено в памяти ЭВМ.

Такое простое и естественное представление в памяти ЭВМ объясняет особую важность структур типа бинарных деревьев. Помимо того, что произвольные деревья можно легко представить с помощью бинарных деревьев, многие деревья, возникающие в прикладных вопросах, по своему существу явля-

ются *бинарными*, поэтому бинарные деревья вызывают такой повышенный интерес.

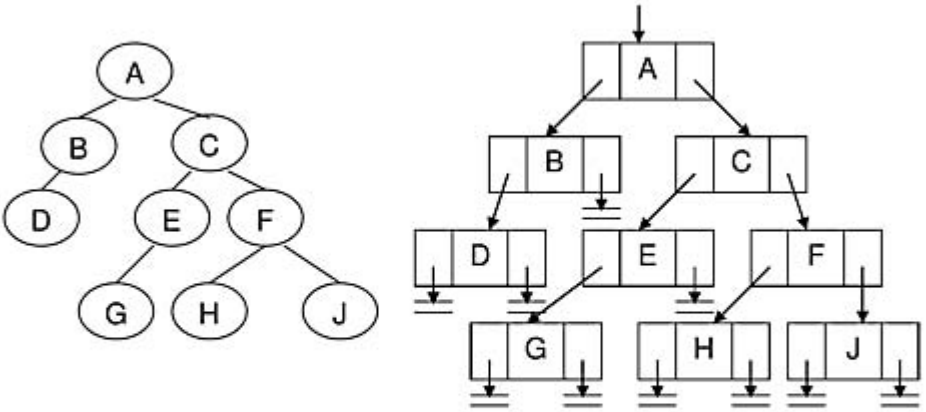


Рис. 4.5. Представление бинарного дерева в памяти ЭВМ

Для работы с древовидными структурами имеется множество алгоритмов, и во всех этих алгоритмах встречается одна и та же *идея*, а именно идея *прохождения* или *обхода* дерева. *Обход* – способ некоторого исследования узлов дерева, при котором каждый узел проходится точно один раз. Полное прохождение дерева дает линейную расстановку узлов.

Для прохождения бинарного дерева можно воспользоваться тремя способами прохождения узлов[3].

1. Прямой порядок (preorder (префиксный), сверху – вниз).
2. Обратный порядок (postorder (инфиксный), слева – направо).
3. Концевой порядок (endorder (постфиксный), снизу – вверх).

Алгоритмы прохождения деревьев

1. *Прямой порядок* обхода (каждый корень проходится раньше своего левого поддерева):

1.1. Попасть в корень.

1.2. Пройти левое поддерево.

1.3. Пройти правое поддерево.

2. *Обратный* (каждый корень проходится после своего левого поддерева):

2.1. Пройти левое поддерево.

2.2. Пройти корень.

2.3. Пройти правое поддерево.

3. *Концевой* (каждый корень проходится после левого и правого поддеревьев):

3.1. Пройти левое поддерево.

3.2. Пройти правое поддерево.

3.3. Пройти корень.

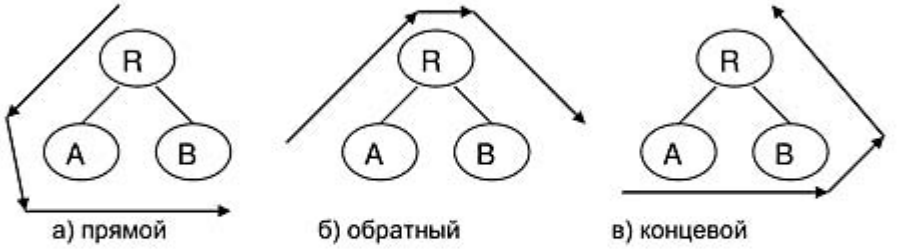


Рис. 4.6. Обходы дерева

Применяя эти операции к бинарному дереву (рис. 4.6), находим, что узлы располагаются следующим образом:

1. A B D C E G F H J – при прохождении в прямом порядке.
2. D B A G E C H F J – при прохождении в обратном порядке .
3. D B G E H J F C A – при прохождении в концевом порядке.

Эти три способа расстановки узлов бинарного дерева в последовательность чрезвычайно важны, поскольку они используются во многих методах, применимых к деревьям.

Эти три метода легко сформировать в виде рекурсивных процедур. Они вновь служат примером того, что действия с рекурсивно определенными структурами данных лучше всего описываются рекурсивными алгоритмами.

1. Префиксный порядок:

```
//Прямой обход дерева
     void pr(node *t) //t - ссылка на корень дерева
      {
           if (t!=NULL)
           {
             P(t); //Операция с узлами дерева (чтение, модификация,удаление и т. д.)
             pr(t->left);
             pr(t->right);
           }
      }
```

2. Инфиксный порядок:

```
//Обратный обход дерева
     void obr(node *t) //t - ссылка на корень дерева
      {
```

```
        if (t!=NULL)
        {
          obr(t->left);
          P(t); //Операция с узлами дерева (чтение, модифика-
      ция,удаление и т. д.)
          obr(t->right);
        }
}
```

3. Постфиксный порядок:

```
//Концевой обход дерева
    void konc (node *t) //t - ссылка на корень дерева
     {
        if (t!=NULL)
        {
          konc (t->left);
          konc (t->right);
          P(t);   //Операция с узлами дерева (чтение, модифика-
    ция,удаление и т. д.)
        }
     }
```

Отметим, что ссылка t передается как *параметр – значение*. Это отражает тот факт, что здесь существует сама *ссылка* (указатель) на рассматриваемое дерево, а не переменная, значение которой есть эта ссылка и которая могла бы изменить значение, если бы t передавался как параметр – переменная.

4.6. Поиск по дереву с включением

Бинарные деревья часто используются для представления множеств данных, элементы которых имеют *ключи* – некоторые значения (числовые или символьные), по которым один элемент отличается от другого.

Если дерево организовано таким образом, что для каждого узла T_i все ключи в левом поддереве *меньше* ключа узла T_i, а ключи в правом поддереве *больше* ключа узла T_i, то это дерево называется *деревом поиска*.

В дереве поиска можно найти место каждого ключа, двигаясь, начиная от корня, и переходя на левое или правое поддерево каждого узла в зависимости от значения его ключа. N элементов можно организовать в бинарное дерево с высотой $h <= \log_2 n$, поэтому для поиска среди n элементов может потребоваться не более $\log_2 n$ сравнений, если дерево идеально сбалансировано.

Рассмотрим пример размещения последовательности целых чисел в виде дерева поиска. Будем считать, что в последовательности могут встречаться одинаковые элементы, поэтому в узле будем хранить ключ (само число) и счетчик.

key	ключ узла дерева
left	ссылка на левое поддерево
right	ссылка на правое поддерево
count	счетчик числа одинаковых узлов

Описание структуры узла дерева поиска (на языке C++):

```
struct node
{
 int key;
 node *left;
 node *right;
 int count;
};
```

Процесс поиска представим в виде рекурсивной процедуры. Параметр процедуры P передается как параметр-переменная, а не как параметр-значение. В случае *включения* переменной должно присваиваться новое значение ссылки, которая перед этим имела значение nil.

Алгоритм поиска по дереву с включением очень часто применяется в программах работы с базами данных (в СУБД) и в трансляторах для организации объектов, которые нужно хранить и искать в памяти.

Процедура «Поиск по дереву с включением»(на языке C++)

```
#include<stdafx.h>
#include <conio.h>
#include <iostream>
using namespace std;
struct node
{
 int key;
 node *left;
 node *right;
 int count;
};
//Процедура поиска с включением
node * search(int x, node *p)
{
 node *t;
 if (p==NULL)
 {
 p=new node;
 p->key=x;          //заполнение полей структуры узла дерева
 p->left=NULL;
 p->right=NULL;
 p->count=1;
```

key	ключ узла дерева
left	ссылка на левое поддерево
right	ссылка на правое поддерево
count	счетчик числа одинаковых узлов

Описание структуры узла дерева поиска (на языке C++):

```
struct node
{
 int key;
 node *left;
 node *right;
 int count;
};
```

Процесс поиска представим в виде рекурсивной процедуры. Параметр процедуры P передается как параметр-переменная, а не как параметр-значение. В случае *включения* переменной должно присваиваться новое значение ссылки, которая перед этим имела значение nil.

Алгоритм поиска по дереву с включением очень часто применяется в программах работы с базами данных (в СУБД) и в трансляторах для организации объектов, которые нужно хранить и искать в памяти.

Процедура «Поиск по дереву с включением»(на языке C++)

```
#include<stdafx.h>
#include <conio.h>
#include <iostream>
using namespace std;
struct node
{
 int key;
 node *left;
 node *right;
 int count;
};
//Процедура поиска с включением
node * search(int x, node *p)
{
 node *t;
 if (p==NULL)
 {
 p=new node;
 p->key=x;          //заполнение полей структуры узла дерева
 p->left=NULL;
 p->right=NULL;
 p->count=1;
```

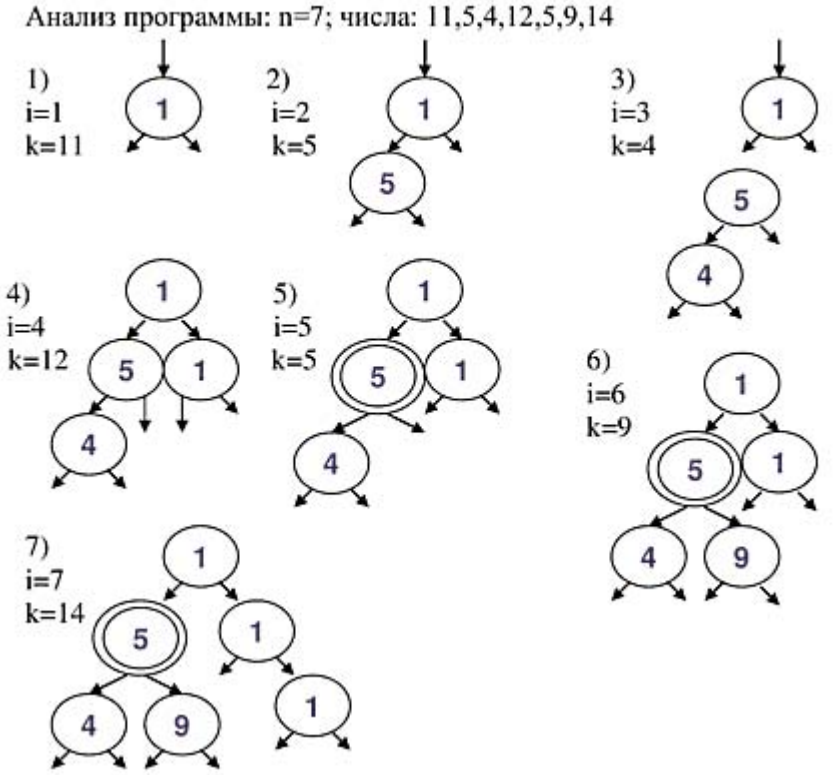


Рис. 4.7. Построение дерева поиска при n = 7. Иллюстрация работы программы

4.7. Удаление узлов из двоичного дерева

Принципиально удаление заключается в переприсвоении ссылок и освобождении памяти, однако здесь следует различать три случая:

1. У удаляемого узла отсутствует хотя бы один непосредственный потомок. В этом случае ссылка на него из вышестоящего узла заменяется на ссылку данного узла на существующий потомок, а память, которую занимал узел, освобождается.

2. Хотя бы один из потомков удаляемого узла является концевым узлом. В этом случае такой потомок занимает место удаленного узла.

3. Ни один из непосредственных потомков удаляемого узла не является концевым. В этом случае удаляемый узел нужно заменить:

- либо на самый правый элемент его левого поддерева;

- либо на самый левый элемент его правого поддерева (т.к. это самые близкие к нему элементы по величине).

Пример.

Дано дерево (рис. 4.8 (1). Удалить последовательно узлы 13,10,5,10.

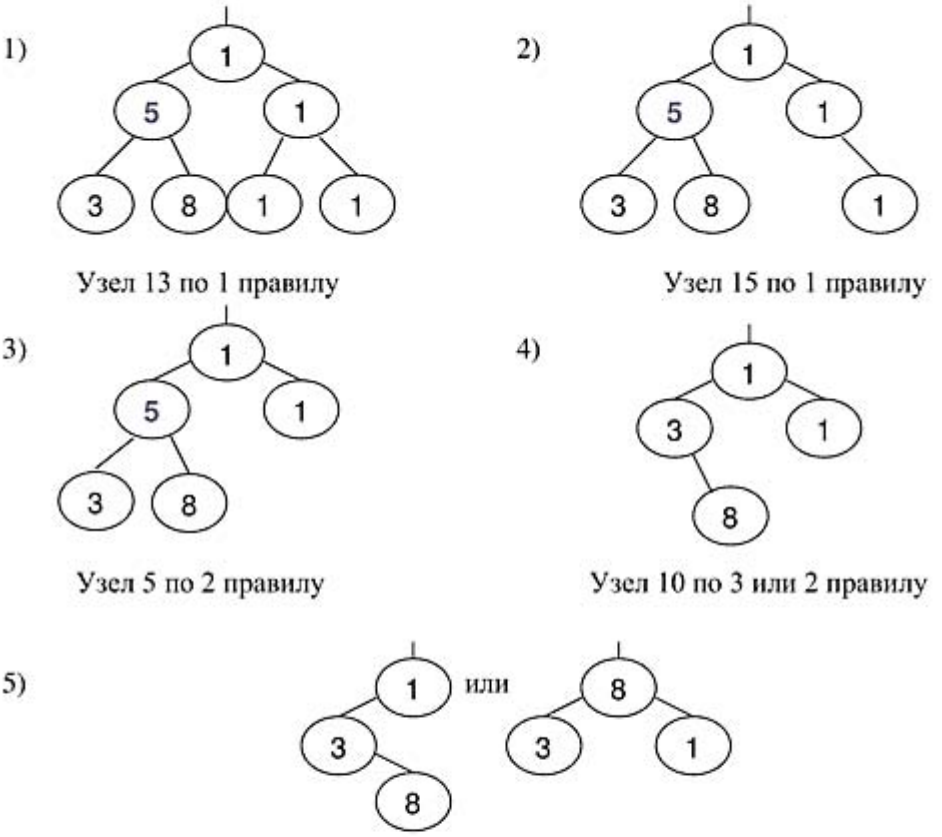


Рис. 4.8. Иллюстрация работы процедуры «Удаление из двоичного дерева»

4.8. Сбалансированные деревья. Включение в сбалансированное дерево. Разработка алгоритма балансировки дерева

Идеально сбалансированное дерево – это дерево, у которого для каждого узла высота его двух поддеревьев одинакова. Основное достоинство сбалансированных деревьев – минимальная высота h: $h \le \log_2 n + 1$.

Максимальное число узлов в дереве заданной высоты h достигается в случае, когда все узлы имеют d поддеревьев, кроме узлов уровня h, не имеющих ни одного. Тогда в дереве степени d содержится узлов:

$$N_d(h) = 1 + d + d^2 + \ldots + d^{h-1} = \sum_{i=0}^{h-1} d^i \ .$$

При d = 2 получаем $N_2(h) = \sum_{i=0}^{h-1} 2^i = 2^h - 1$. Отсюда $h \le \log_2 n + 1$. Следовательно, время поиска данных в таком дереве минимально.

Однако с течением времени свойство идеальной сбалансированности будет нарушаться из-за возможных включений и удалений элементов. В принципе, можно выполнять балансировку дерева после каждого включения или исключения, однако это очень «дорого», т.е. может потребоваться существенная перестройка структур. Задачу можно упростить, если принять менее строгое определение сбалансированности.

Определение: дерево является сбалансированным тогда, когда для каждого узла высота его двух поддеревьев различается не более, чем на 1 (по «строгому определению» – для каждого узла высота его двух поддеревьев одинакова).

Такие деревья (по менее строгому определению) были впервые предложены и изучены учеными Адельсоном – Вельским и Ландисом в 1962 г. и получили название АВЛ – деревья (по фамилиям их изобретателей). Преимуществом таких деревьев является их достаточно простая балансировка при средней длине поиска информации примерно такой же, как у идеально сбалансированных деревьев. *Идеально сбалансированные* деревья являются также АВЛ – сбалансированными.

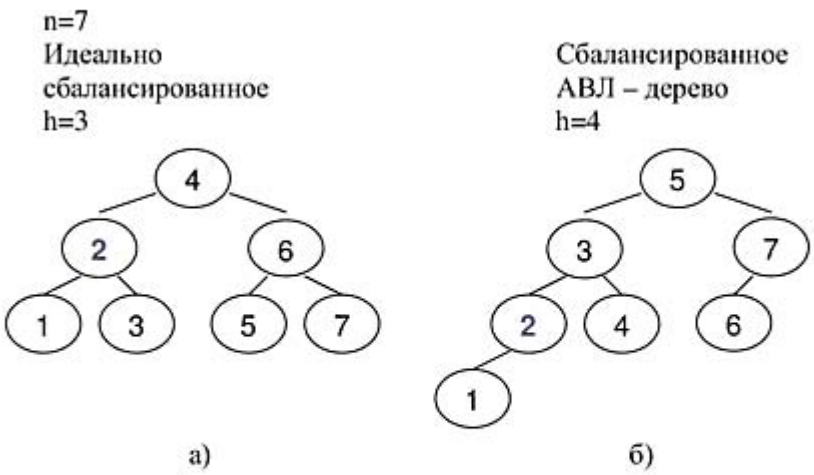


Рис. 4.9. Идеально сбалансированное (а) и АВЛ сбалансированное (б) деревья

Возьмем сбалансированное дерево и посмотрим, что произойдет, когда в сбалансированное дерево включается «новый узел». Пусть дан корень R с левым (A) и правым (B) поддеревом (рис. 4.10). Предположим, что в A включается новый узел вызывая увеличение его высоты на 1.

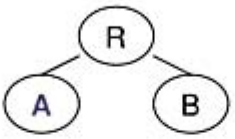


Рис. 4.10. Сбалансированное дерево

До разработки программы необходимо выбрать способ хранения информации о сбалансированности дерева.

Показатель сбалансированности (баланс) вершины будем интерпретировать как разность между высотой правого и левого поддерева, а сам алгоритм будет основан на описании типа и определении узла, расширяющегося до:

key – ключ узла дерева;
left – ссылка на левое поддерево;
right – ссылка на правое поддерево;
count – счетчик числа одинаковых узлов;
balance – показатель сбалансированности узла (принимает значения: -1, 0, 1).

Показатель сбалансированности узла – высота его правого поддерева минус высота его левого поддерева (т.е. баланс = $h_b - h_a$) может принимать значения 1, 0, -1. Алгоритм включения и балансировки полностью определяется способом хранения информации о сбалансированности дерева. Одно из решений – хранить показатель сбалансированности в каждом узле дерева.

Очевидно, что показатель сбалансированности дерева будет указан для каждого узла и должен характеризовать три случая:

1. Высота левого поддерева больше высоты правого:
   баланс = -1 (т.к. $h_b - h_a = -1$).
2. Левое и правое поддеревья имеют одинаковые высоты:
   баланс = 0 ($h_b - h_a = 0$).
3. Высота левого поддерева меньше высоты правого:
   баланс = 1 ($h_b - h_a = 1$).

Возможны три ситуации в зависимости от высоты деревьев перед включением:

1. До включения $h_a = h_b$; (p->balance=0). A и B становятся неравной высоты, но критерий сбалансированности не нарушается (вес склонится влево). После включения $h_a = h_b + 1$.

2. До включения $h_a < h_b$; (p->balance =1). A и B приобретают равную высоту, т.е. сбалансированность даже улучшается, после включения $h_a = h_b$.

3. До включения $h_a > h_b$; т.е. $h_a = h_b + 1$; (p->balance = -1). Критерий сбалансированности нарушается, и дерево нужно перестраивать, т.е. необходима балансировка. После включения $h_a=h_b+2$, т.е. необходима балансировка.

Процесс включения вершины фактически состоит из следующих трех последовательно выполняемых частей:

1. Прохода по пути поиска, пока не убедимся, что ключа в дереве нет (если есть ключ – не включаем).
2. Включение новой вершины и определение результирующего показателя сбалансированности.
3. «Отступления» по пути поиска и проверки в каждой вершине показателя сбалансированности. Если необходимо – балансировка.

Эта процедура реализуется простым расширением уже готовой процедуры *поиска с включением*. В этой процедуре описываются действия, связанные с поиском в каждой вершине, и в силу рекурсивной природы самой процедуры ее легко приспособить для выполнения дополнительных действий при возвращении вдоль пути поиска. Информация, которую нужно передавать на каждом шаге, указывает, увеличилась или нет высота поддерева, где произошло включение. Поэтому расширим список параметров процедур и добавим булевское значение f (высота дерева увеличена). До разработки программы необходимо выбрать способ хранения информации о сбалансированности дерева. Очевидно, что показатель сбалансированности дерева будет указан для каждого узла и должен характеризовать три случая.

На основе процедуры «Поиск по дереву с включением» можно написать новую процедуру «Поиск по дереву с включением и балансировкой». Необходимые операции балансировки полностью заключаются в обмене значениями ссылок. Фактически ссылки обмениваются значениями по кругу, что приводит к однократному или двукратному «повороту» двух или трех узлов. Виды поворотов: LL-однократный левый, RR-однократный правый,

LR-двукратный налево-направо, RL-двукратный направо-налево. Кроме «вращения» ссылок следует также изменить соответствующие показатели сбалансированности узлов.

Эмпирические проверки оправдывают предположение, что ожидаемая высота сбалансированного дерева, которое строится в программе (процедуре *поиск* с балансировкой) равна H = Log n + c (где c – малая константа; c приблизительно равно 0.25).

Это значит, что на практике АВЛ – сбалансированные деревья ведут себя также, как идеально сбалансированные, хотя с ними намного легче работать.

Эмпирически можно предполагать, что в среднем балансировка необходима приблизительно *один раз на каждые два включения*. При этом однократный и двукратный повороты одинаково вероятны.

Последующий пример был тщательно подобран, чтобы показать как можно больше поворотов при минимальном числе включений.

Пример. Включение узлов в сбалансированное дерево с последующей балансировкой.

Включаются узлы с ключами 4,5,7,2,1,3,6

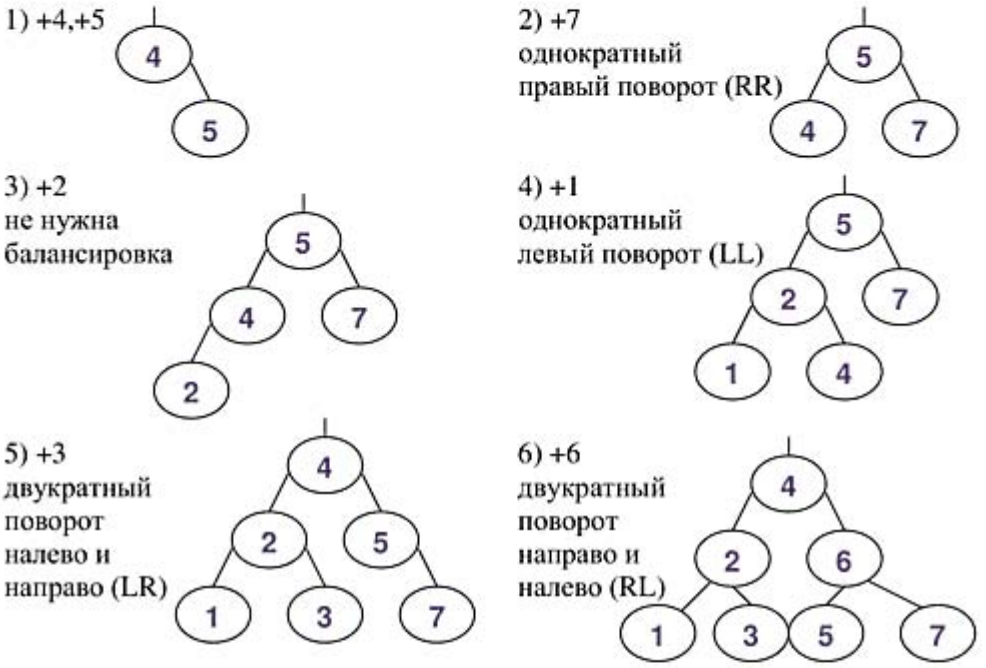


Рис. 4.11. Повороты деревьев

Из-за сложности операций балансировки считается, что сбалансированные деревья следует использовать лишь в том случае, когда поиск информации происходит значительно чаще, чем включение.

4.9. B–деревья

До сих пор мы ограничивали наши рассуждения деревьями, в которых каждый узел имеет самое большее двух потомков.

Сильно ветвящиеся деревья – это деревья, в которых каждый узел может иметь более двух потомков. Имеется практически очень важная область применения сильно ветвящихся деревьев, которые представляют общий интерес. Это формирование и использование крупномасштабных деревьев поиска, в которых необходимы и включения, и удаления, но для которых ОП недостаточно велика

или слишком дорогостояща, чтобы использовать ее для долговременного хранения.

Предположим, что узлы дерева должны храниться на внешнем ЗУ, таком как диск (новое здесь то, что ссылки представляют адреса на диске, а не адреса ОП). Если множество данных, состоящее, например, из миллиона (10^6) элементов, хранится в виде бинарного дерева, то для поиска элемента потребуется в среднем около $\log_2 10^6$, приблизительно 20 шагов поиска. Поскольку теперь каждый шаг включает обращение к диску, будет весьма желательна организация памяти, требующая меньше обращений. Сильно ветвящееся дерево является идеальным решением этой проблемы.

Если происходит обращение к некоторому одиночному элементу, расположенному на ВЗУ, то без больших дополнительных затрат можно обращаться к целой группе элементов. Отсюда следует, что дерево нужно разделить на поддеревья, считая, что все эти поддеревья одновременно полностью доступны.

Будем называть такие поддеревья *страницами*. На рис. 4.12 изображено бинарное дерево, разделенное на страницы, каждая из которых состоит из 3 узлов.

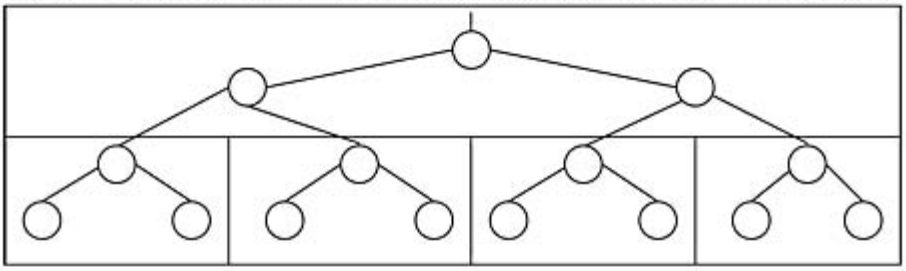


Рис. 4.12. Бинарное дерево, разделенное на страницы

Уменьшение количества обращений к диску (а теперь обращение к каждой странице предполагает обращение к диску) может быть значительным. Предположим, что на каждой странице размещается 100 узлов, тогда дерево поиска, содержащее 10^6 элементов, потребует в среднем $\log_{100} 10^6 = 3$ обращения к страницам в место 20.

При хранении дерева на диске соблюдать сбалансированность (а тем более идеальную) очень дорого. Для этого случая очень разумный критерий был сформулирован Р. Бэйером: каждая страница содержит от n до 2n узлов при заданном постоянном n. Следовательно, в дереве с N элементами и максимальным размером страницы 2n узлов наихудший случай потребует $\log_n N$ обращений к страницам. А обращения к страницам составляют, как известно, основную часть затрат на поиск. Кроме того, коэффициент использования памяти составляет не менее 50%, т.к. страницы заполнены хотя бы наполовину. При всех этих преимуществах данная схема требует сравнительно простых алгоритмов поиска, включения и удаления.

Свойства B–деревьев

Рассматриваемые структуры данных называются B–деревьями и имеют следующие свойства[4]:

1. Каждая страница содержит не более 2n элементов (ключей).
2. Каждая страница, кроме корневой, содержит не менее n элементов (n – порядок B – дерева).
3. Каждая страница является либо *листом*, т.е. не имеет потомков, либо имеет m + 1 потомок, где m – число находящихся на ней ключей.
4. Все листья находятся на одном и том же уровне.

На рис. 4.13 показано B–дерево порядка 2 с 3-мя уровнями. Все страницы содержат 2, 3 или 4 элемента. Исключением является корень, которому разрешается содержать только один элемент. Все листья находятся на уровне 3.

Ключи расположены в возрастающем порядке слева направо, если спроецировать дерево на один уровень, вставляя потомков между ключами, находящимися на странице – предке.

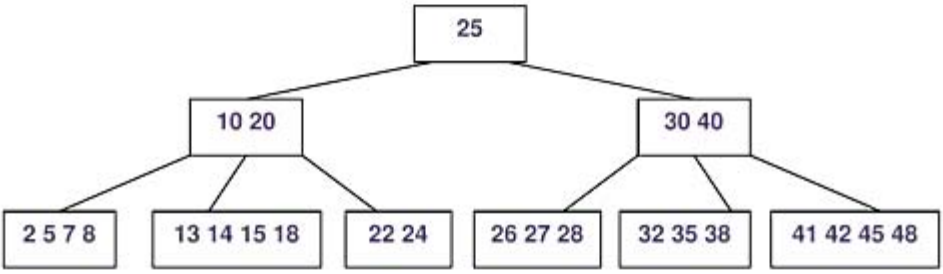


Рис. 4.13. B–дерево порядка 2

Пример. Включение в B–дерево

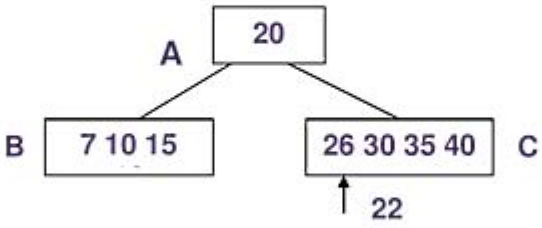


Рис. 4.14. Включение в B–дерево порядка 2

Нужно включить в это дерево элемент с ключом 22. Включение состоит из этапов:

1. Выяснение, что ключ 22 отсутствует (включение в страницу C невозможно, т.к. она уже заполнена).
2. Страница C расщепляется на две страницы, т.е. размещается новая страница D.
3. Количество m + 1 ключей поровну распределяется на C и D, а средний ключ перемещается на один уровень вверх, на страницу предок A.

На рис. 4.15 изображено B–дерево после включения элемента с ключом 22.

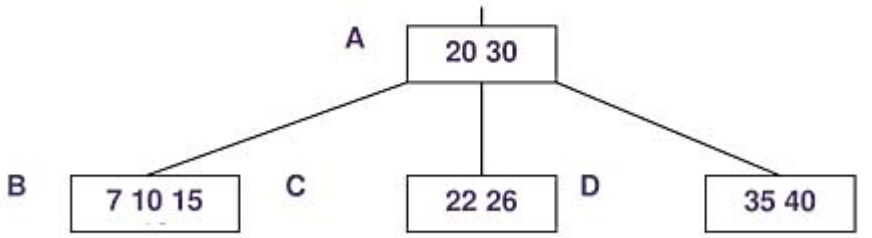


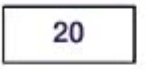
Рис. 4.15. B–дерево после включения элемента

Пример. Построение B–дерева порядка 2 (n=2) с последовательностью вставляемых ключей:

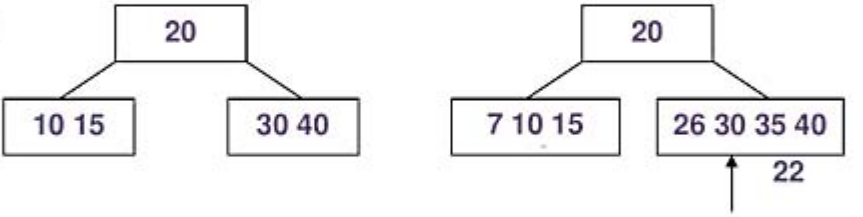
20; 40 10 30 15 35 7 26 18 22; 5; 42 13 46 27 8 32; 38 24 45 25;

(точки с запятой указывают моменты размещения новых страниц).

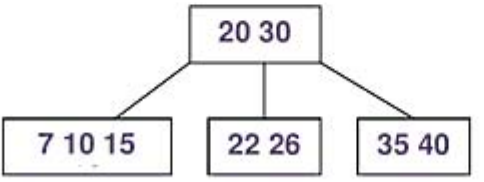
1)



2)



3)



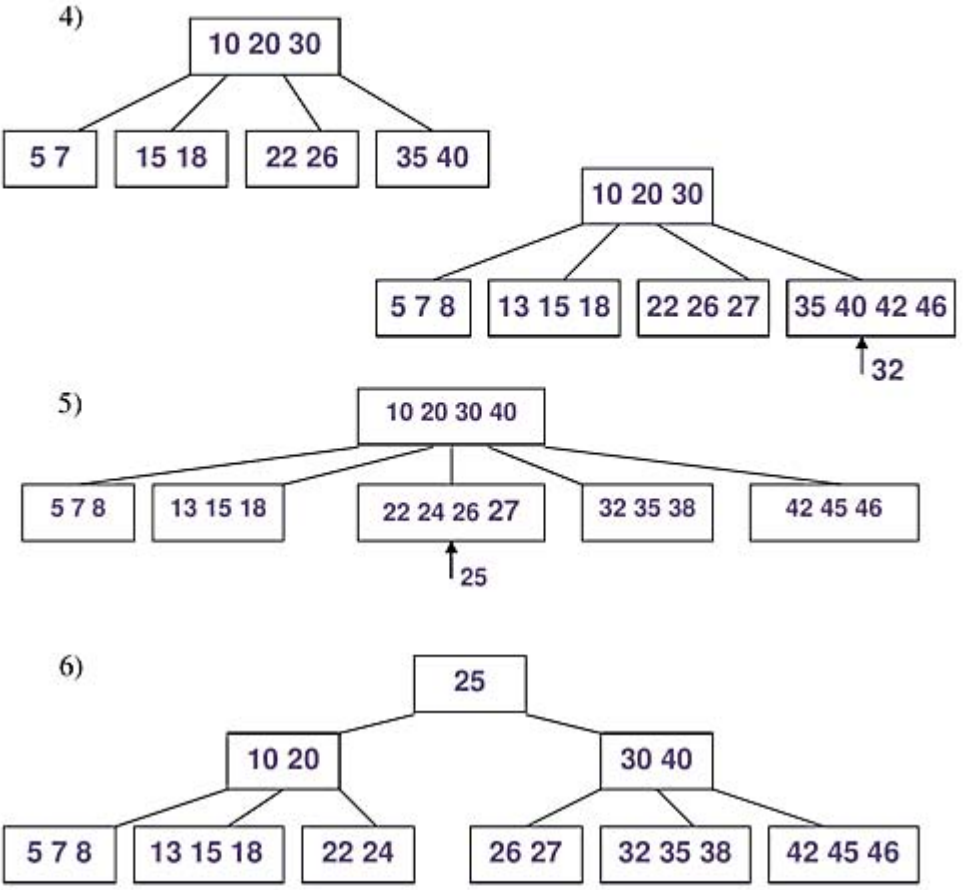


Рис. 4.16. Рост B–дерева порядка 2

4.10. Деревья Фибоначчи

Идеально сбалансированные и просто сбалансированные деревья дают минимальную высоту h для заданного числа узлов n.

Исследуем сбалансированные деревья, находящиеся на другом «полюсе»: для заданного n они имею *максимальную высоту h* (у них для любого узла, кроме концевого, баланс равен ±1).

Последовательности Фибоначчи – это последовательность чисел, в которой каждый член является суммой двух предыдущих членов: 0, 1, 1, 2, 3, 5, 8, 13, 21, …Числа Фибоначчи F_n формально определяются следующим образом: $F_{n+2} = F_{n+1} + F_n$, n>=0. Всякое положительное число m можно представить единственным образом в виде суммы чисел Фибоначчи, причем в этом расположе-

нии наибольшее F_n не будет превосходить m и никакие два F_n не будут соседними членами последовательности Фибоначчи.

Принцип организации таких деревьев напоминает принцип построения *чисел Фибоначчи*, поэтому они называются *деревьями Фибоначчи*. Они определяются следующим образом:

1. Пустое дерево есть дерево Фибоначчи высотой 0: T_0.
2. Один узел есть дерево Фибоначчи высотой 1: T_1.
3. Если T_{h-1} и T_{h-2} – деревья Фибоначчи, то дерево Фибоначчи $T_h = < T_{h-1}, x, T_{h-2} >$ с высотой h.
4. Никакие другие деревья не являются деревьями Фибоначчи.

Число узлов в T_h определяется простым рекуррентным соотношением:

$$N_0 = 0, \quad N_1 = 1$$

$$N_h = N_{h-1} + 1 + N_{h-2}$$

N_i – это количество узлов, для которых можно получить наихудший случай (верхнюю границу h).

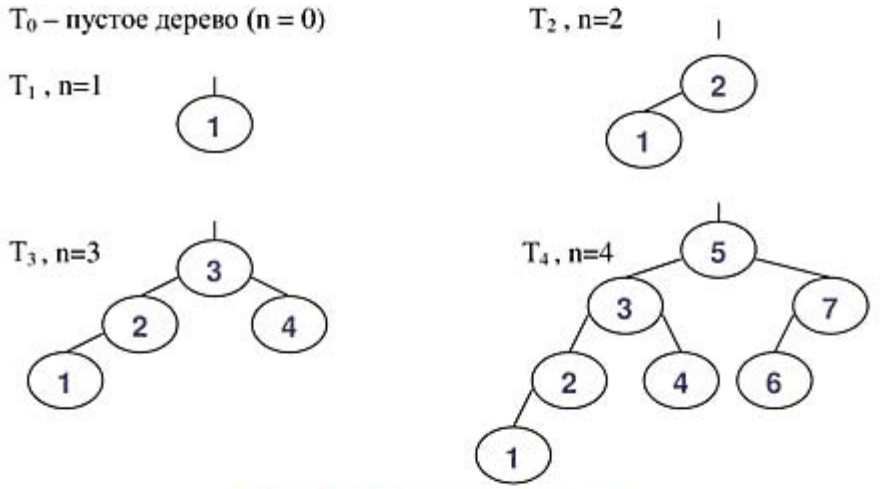


Рис. 4.17. Деревья Фибоначчи

4.11. Кодирование и сжатие информации. Алгоритм Хаффмена

Рассмотрим следующую проблему. Предположим, что у нас есть алфавит из n символов и длинное сообщение, состоящее из символов этого алфавита. Мы хотим закодировать сообщение в виде длинной строки битов следующим образом (бит определяем как единицу, содержащую 0 или 1). Присвоим каждому символу алфавита определенную последовательность битов (код). Затем соединим отдельные коды символов, составляющих сообщение, и получим коди-

ровку сообщения. Например, предположим, что алфавит состоит из четырех символов – A, B, C и D – и что символам назначены следующие коды:

Символ	Код
A	010
B	100
C	000
D	111

Сообщение ABACCDA кодируется как 01010001000000111010. Однако такая кодировка была бы неэффективной, поскольку для каждого символа используются 3 бит, так что для кодировки всего сообщения требуется 21 бит. Предположим, что каждому символу назначен 2-битовый код:

Символ	Код
A	00
B	01
C	10
D	11

Тогда кодировка сообщения была бы 00010010101100; требуется лишь 14 бит. Мы хотим найти код, который минимизирует длину закодированного сообщения [6].

Рассмотрим наш пример еще раз. Каждая из букв B и D появляется в сообщении только один раз, а буква A – три раза. Стало быть, если выбран код, в котором букве A назначена более короткая строка битов, чем буквам B и D, то длина закодированного сообщения будет меньше. Это происходит оттого, что короткий код (представляющий букву A) появляется более часто, чем длинный. В самом деле, коды могут быть назначены следующим образом:

Символ	Частота	Код
A	3	0
B	1	110
C	2	10
D	1	111

При использовании этого кода сообщение ABACCDA кодируется как 0110010101110, что требует лишь 13 бит. В очень длинных сообщениях, которые содержат символы, встречающиеся чрезвычайно редко, экономия существенна. Обычно коды создаются не на основе частоты вхождения символов в отдельно взятом сообщении, а на основе их частоты во всем множестве сообщений. Для каждого сообщения используется один и тот же код. Например, если сообщения состоят из английских слов, могут использоваться известные данные о частоте появления символов алфавита в английском языке.

Отметьте, что если используются коды переменной длины, то код одного символа не должен совпадать с началом кода другого символа. Такое условие должно выполняться, если раскодирование происходит слева направо. Если бы

код символа x–c(x) совпадал с началом кода символа y–c(y), то, когда встречается код c(x), неясно, является ли он кодом символа x или началом кода с (y).

В нашем случае битовая строка сообщения просматривается слева направо. Если первый бит равен 0, то это символ A; в противном случае это B, C или D и проверяется следующий бит. Если второй бит равен 0, то это символ C, иначе это B или D и надо проверить третий бит. Если он равен 0, значит это B, а если 1, то D. Как только распознан первый символ, процесс повторяется для нахождения второго символа, начиная со следующего бита.

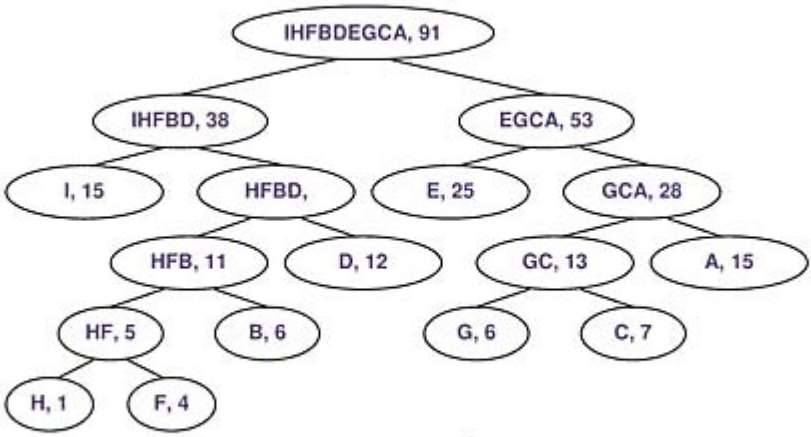
Такая операция подсказывает метод реализации оптимальной схемы кодирования, если известны частоты появления каждого символа в сообщении. Находим два символа, появляющихся наименее часто. В нашем примере это B и D. Будем различать их по последнему биту кодов: 0–B, 1–D. Соединим эти символы в единый символ BD, появление которого означает, что это либо символ B, либо символ D. Частота появления этого нового символа равна сумме частот двух составляющих символов. Стало быть, частота BD – 2. Теперь у нас есть три символа: A (частота 3), C (частота 2) и BD (частота 2). Снова выберем два символа с наименьшей частотой, т.е. C и BD. Будем различать их по последнему биту кодов: 0 – C, 1 – BD. Затем два символа объединяются в один символ CBD с частотой 4. К этому времени осталось только два символа – A и CBD. Они объединяются в один символ ACBD. Будем различать их по последнему биту кодов: 0 – A, 1 – CBD.

Символ ACBD содержит весь алфавит, ему присваивается в качестве кода пустая строка битов нулевой длины. Это означает, что вначале раскодирования до момента проверки какого-либо бита известно, что этот символ содержится в ACBD. Двум символам, составляющим ACBD (A и CBD), присваиваются соответственно коды 0 и 1: если встречается 0, значит, закодирован символ A, а если 1, то это C, B или D. Аналогично двум символам, составляющим CBD (C и BD), назначаются соответственно коды 10 и 11. Первый бит указывает, что символ входит в группу CBD, а второй позволяет отличить C и BD. Символам, составляющим BD (B и D), назначаются соответственно коды 110 и 111. Следуя этому процессу, символам, которые появляются в сообщении часто, присваиваются более короткие коды, чем тем, которые встречаются редко.

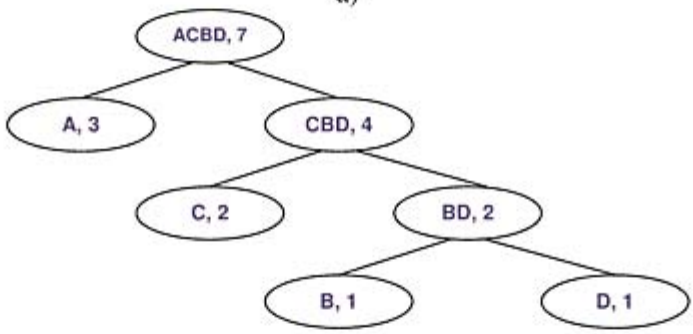
Операция объединения двух символов в один предполагает использование структуры бинарного дерева. Каждый лист представляет символ исходного алфавита. Каждый узел, не являющийся узлом, представляет соединение символов из листов – потомков данного узла. На рис. 4.18а приведено бинарное дерево, построенное с использованием предыдущего примера. Каждый узел содержит символ и его частоту. На рис. 4.18б показано бинарное дерево, построенное по данному методу для приведенной на рис. 4.18в таблицы символов алфавита и частот. Такие деревья называют *деревьями Хаффмена,* по имени изобретателя этого метода кодирования.

Как только дерево Хаффмена построено, код любого символа алфавита может быть определен просмотром дерева снизу вверх, начиная с листа, пред-

ставляющего этот символ. Начальное значение кода – пустая строка. Каждый раз, когда мы поднимаемся по левой ветви, к коду слева приписывается 0; каждый раз при подъеме по правой ветви к коду приписывается 1.



а)



б)

Символ	Частота	Код	Символ	Частота	Код	Символ	Частота	Код
A	15	111	D	12	011	G	6	1100
B	6	0101	E	25	10	H	1	01000
C	7	1101	F	4	01001	I	15	00

в)

Рис. 4.18. Деревья Хаффмена

Отметьте, что дерево Хаффмена строго бинарное. Следовательно, если в алфавите n символов, то дерево Хаффмена (у которого n листьев) может быть представлено массивом узлов размером 2n-1. Поскольку размер памяти, требуемой под дерево, известен, она может быть выделена заранее. Отметим также, что дерево Хаффмена проходится от листьев к корню. Отсюда следует, что

не требуются поля LEFT и RIGHT, и для представления структуры дерева достаточно одного поля FTHER. Знак поля FTHER может использоваться для определения, является ли узел левым или правым сыном, а абсолютное значение поля служит указателем отца узла. Левый сын имеет отрицательное значение поля FTHER, а правый – положительное.

Исходное сообщение (при наличии кодировки сообщения и дерева Хаффмена) может быть восстановлено следующим способом. Начинаем с корня дерева. Каждый раз, когда встречается 0, двигаемся по левой ветви, и каждый раз, когда встречается 1, двигаемся по правой ветви. Повторяем этот процесс до тех пор, пока не дойдем до листа. Новый символ исходного сообщения есть символ, соответствующий этому листу. Проверьте, можете ли вы раскодировать строку 1110100010111011, используя дерево Хаффмена на рис. 4.18 (b).

Вопросы для самоконтроля

1. Что такое степень узла дерева?
2. Чему равна высота сбалансированного дерева?
3. Перечислите способы изображения деревьев.
4. Назовите способы обхода деревьев.
5. Какие способы обхода используются в процедурах Н. Вирта «Построение сбалансированного дерева» и «Печать сбалансированного дерева»?
6. В результате каких операций могут нарушаться условия баланса в дереве?
7. В чем заключается сущность операций балансировки?
8. Чем отличается AVL-дерево от идеально сбалансированного дерева?
9. Какой параметр дерева Фибоначчи является аналогом чисел Фибоначчи?
10. Назовите свойства B-деревьев.
11. В каких условиях используются B-деревья?

ГЛАВА 5
ВНУТРЕННЯЯ СОРТИРОВКА

5.1. Терминология

Сортировка – это упорядочивание набора однотипных данных по возрастанию или убыванию. Сортировка является одной из наиболее приятных для умственного анализа категорией алгоритмов, поскольку процесс сортировки очень хорошо определен. Алгоритмы сортировки были подвергнуты обширному анализу, и способ их работы хорошо понятен. К сожалению, вследствие этой изученности сортировка часто воспринимается как нечто само собой разумеющееся. При необходимости отсортировать данные многие программисты просто вызывают стандартную функцию qsort(), входящую в стандартную библиотеку С (аналогичная функция есть в Pascal и других языках программирования высокого уровня). Однако различные подходы к сортировке обладают разными характеристиками. Несмотря на то, что некоторые способы сортировки могут быть в среднем лучше, чем другие, ни один алгоритм не является идеальным для всех случаев. Поэтому широкий набор алгоритмов сортировки – полезное добавление в инструментарий любого программиста.

Будет полезно кратко остановиться на том, почему вызов qsort() не является универсальным решением всех задач сортировки. Во-первых, функцию общего назначения вроде qsort() невозможно применить во всех ситуациях. Например, qsort() сортирует только массивы в памяти. Она не может сортировать данные, хранящиеся в связанных списках. Во-вторых, qsort() – параметризованная функция, благодаря чему она может обрабатывать широкий набор типов данных, но вместе с тем вследствие этого она работает медленнее, чем эквивалентная функция, рассчитанная на какой-то один тип данных. Наконец, хотя алгоритм быстрой сортировки, примененный в функции qsort (), очень эффективен в общем случае, он может оказаться не самым лучшим алгоритмом в некоторых конкретных ситуациях.

Существует две общие категории алгоритмов сортировки: алгоритмы, сортирующие объекты с произвольным доступом (например, массивы или дисковые файлы произвольного доступа), и алгоритмы, сортирующие последовательные объекты (например, файлы на дисках и лентах или связанные списки1). В данной главе рассматриваются алгоритмы первой категории.

Чаще всего при сортировке данных лишь часть их используется в качестве ключа сортировки. *Ключ* – это часть информации, определяющая порядок эле-

ментов. Таким образом, ключ участвует в сравнениях, но при обмене элементов происходит перемещение всей структуры данных. Например, в списке почтовой рассылки в качестве ключа может использоваться почтовый индекс, но сортируется весь адрес. Для простоты в нижеследующих примерах будет производиться сортировка массивов символов, в которых ключ и данные совпадают.

5.2. Классы алгоритмов сортировки

Существует три общих метода сортировки массивов:

- обмен;
- вставка;
- выбор (выборка).

Чтобы понять, как работают эти методы, можно представить себе колоду игральных карт. Чтобы отсортировать карты методом обмена, их необходимо разложите на столе лицом вверх и менять местами карты, расположенные не по порядку, пока вся колода не будет упорядочена. В методе выбора – разложить карты на столе, выбрать карту наименьшей значимости и положить ее в руку. Затем из оставшихся карт снова выбрать карту наименьшей значимости и положите ее на ту, которая уже находится в руке. Процесс повторяется до тех пор, пока в руке не окажутся все карты; по окончании процесса колода будет отсортирована. Чтобы отсортировать колоду методом вставки, необходимо взять все карты в руку и далее выкладывать их по одной на стол, вставляя каждую следующую карту в соответствующую позицию. Когда все карты окажутся на столе, колода будет отсортирована.

5.3. Оценка алгоритмов сортировки

Существует много различных алгоритмов сортировки. Все они имеют свои положительные стороны, но общие критерии оценки алгоритма сортировки таковы:

Насколько быстро данный алгоритм сортирует информацию в среднем?

Насколько быстро он работает в лучшем и худшем случаях?

Естественно или неестественно он себя ведет?

Переставляет ли он элементы с одинаковыми ключами?

Очевидно, что скорость работы любого алгоритма сортировки имеет большое значение. Скорость сортировки массива непосредственно связана с количеством сравнений и количеством обменов, происходящих во время сортировки, причем обмены занимают больше времени. Сравнение происходит тогда, когда один элемент массива сравнивается с другим; обмен происходит тогда, когда два элемента меняются местами. Время работы одних алгоритмов сортировки растет экспоненциально, а время работы других логарифмически зависит от количества элементов.

Время работы в лучшем и худшем случаях имеет значение, если одна из этих ситуаций будет встречаться довольно часто. Алгоритм сортировки зачастую имеет хорошее среднее время выполнения, но в худшем случае он работает очень медленно.

Поведение алгоритма сортировки называется естественным, если время сортировки минимально для уже упорядоченного списка элементов, увеличивается по мере возрастания степени неупорядоченности списка и максимально, когда элементы списка расположены в обратном порядке. Объем работы алгоритма оценивается количеством производимых сравнений и обменов.

5.4. Обменная сортировка

Название этой группы методов произошло от основного типа операций, используемого в алгоритмах – обмен двух элементов в таблице своими значениями. Эта операция используется и в других группах, поэтому классификацию нельзя признать вполне строгой, но данное разделение тем не менее является традиционным. Таблица, подлежащая сортировке, в общем случае состоит из элементов-записей, включающих информационную часть и ключи, по которым производится упорядочивание по возрастанию. Поскольку информационная часть почти не влияет на процесс сортировки, будем предполагать, что таблицы, используемые в примерах, состоят только из элементов-ключей, а информационная часть записи отсутствует.

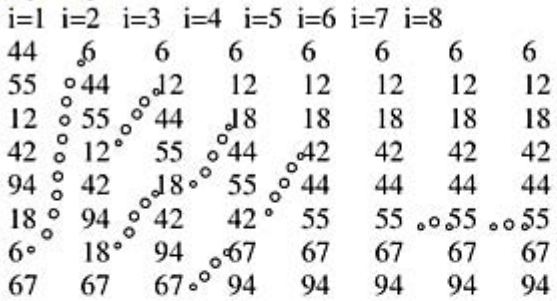
5.4.1. Пузырьковая сортировка

Самый известный (и самый медленный) алгоритм – пузырьковая сортировка (bubble sort, сортировка методом пузырька, или просто сортировка пузырьком). Его популярность объясняется интересным названием и простотой самого алгоритма. Тем не менее, в общем случае это один из самых худших алгоритмов сортировки.

Это метод, где обмен местами двух элементов представляет собой характерную особенность процесса. Алгоритм простого обмена основывается на сравнении и смене мест для пары соседних элементов и продолжении процесса до тех пор, пока не будут упорядочены все элементы. При сортировке этим методом повторяются проходы по массиву. И каждый раз наименьший элемент оставшейся последовательности сдвигается к концу массива.

Если рассматривать массивы как вертикальные, а не горизонтальные, то элементы массива можно сравнить с пузырьками в воде, причем вес каждого соответствует его ключу. В этом случае при каждом проходе один пузырек как бы поднимается до уровня, который соответствует его весу.

Пример



Из примера видно, что последние проходы не влияют на порядок элементов. Очевидно, прием улучшения этого алгоритма – запомнить, были ли перестановки в процессе прохода. Можно чередовать направления просмотра.

Пузырьковая сортировка относится к классу обменных сортировок, т.е. к классу сортировок методом обмена. Ее алгоритм содержит повторяющиеся сравнения (т.е. многократные сравнения одних и тех же элементов) и, при необходимости, обмен соседних элементов. Элементы ведут себя подобно пузырькам воздуха в воде – каждый из них поднимается на свой уровень. Простая форма алгоритма сортировки показана ниже:

```
/* Пузырьковая сортировка */

void bubble (char *items, int count){
register int a, b;
register char t;
for (a=1; a < count; ++a)
for  (b=count-1; b >= a; --b) {
 if (items [b-1] > items [b])
 { /* обмен элементов */
    t = items [b-1] ;
    items [b-1] = items [b] ;
    items [b] = t; }
 }
}
```

Здесь items – указатель на массив символов, подлежащий сортировке, а count – количество элементов в массиве. Работа пузырьковой сортировки выполняется в двух циклах. Если количество элементов массива равно count, внешний цикл приводит к просмотру массива count – 1 раз. Это обеспечивает размещение элементов в правильном порядке к концу выполнения функции даже в самом худшем случае. Все сравнения и обмены выполняются во внутреннем цикле. (Слегка улучшенная версия алгоритма пузырьковой сортировки завершает работу, если при просмотре массива не было сделано ни одного обмена, но это достигается за счет добавления еще одного сравнения при каждом проходе внутреннего цикла.)

С помощью этой версии алгоритма пузырьковой сортировки можно сортировать массивы символов по возрастанию. Например, следующая короткая программа сортирует строку, вводимую пользователем:

```
/*Программа,вызывающая функцию сортировки bubble*/

#include <string.h>
#include <stdio.h>
#include <stdlib.h>
void bubble(char * items, int count);
int main(void) {
 char s[255] ;
 printf(«Введите строку:»);
 gets (s) ;
 bubble(s, strlen(s));
 printf(«Отсортированная строка: %s.\n», s);
 return 0;
}
```

Чтобы наглядно показать, как работает пузырьковая сортировка, допустим, что исходный массив содержит элементы dcab. Ниже показано состояние массива после каждого прохода:

Начало	d c a b
Проход 1	a d c b
Проход 2	a b d c
Проход 3	a b e d

При анализе любого алгоритма сортировки полезно знать, сколько операций сравнения и обмена будет выполнено в лучшем, среднем и худшем случаях. Поскольку характеристики выполняемого кода зависят от таких факторов, как оптимизация, производимая компилятором, различия между процессорами и особенности реализации, мы не будем пытаться получить точные значения этих параметров. Вместо этого сконцентрируем свое внимание на общей эффективности каждого алгоритма.

В пузырьковой сортировке количество сравнений всегда одно и то же, поскольку два цикла **for** повторяются указанное количество раз независимо от того, был список изначально упорядочен или нет. Это значит, что алгоритм пузырьковой сортировки всегда выполняет $(n^2 - n) / 2$ сравнений, где **n** – количество сортируемых элементов. Данная формула выведена на том основании, что внешний цикл выполняется $n-1$ раз, а внутренний выполняется в среднем **n/2** раз. Произведение этих величин и дает предыдущее выражение.

Следует обратить внимание на член n^2 в формуле. Говорят, что пузырько-

вая сортировка является алгоритмом порядка n^2, поскольку время ее выполнения пропорционально квадрату количества сортируемых элементов. Необходимо признать, что алгоритм порядка n^2 не эффективен при большом количестве элементов, поскольку время выполнения растет экспоненциально в зависимости от количества сортируемых элементов.

В алгоритме пузырьковой сортировки количество обменов в лучшем случае равно нулю, если массив уже отсортирован. Однако в среднем и худшем случаях количество обменов также является величиной порядка n^2.

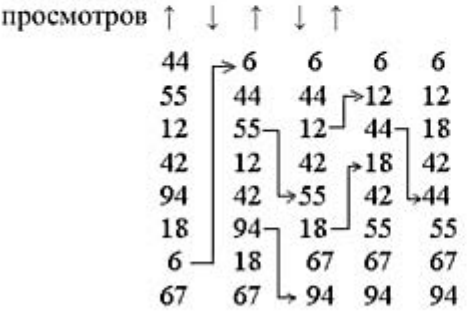
5.4.2. Шейкер-сортировка (ShakerSort)

Алгоритм пузырьковой сортировки можно немного улучшить, если попытаться повысить скорость его работы. Например, пузырьковая сортировка имеет такую особенность: неупорядоченные элементы на «большом» конце массива (например, «a» в примере dcah) занимают правильные положения за один проход, но неупорядоченные элементы в начале массива (например, «d») поднимаются на свои места очень медленно. Этот факт подсказывает способ улучшения алгоритма. Вместо того чтобы постоянно просматривать массив в одном направлении, в последовательных проходах можно чередовать направления. Этим мы добьемся того, что элементы, сильно удаленные от своих положений, быстро станут на свои места. Данная версия пузырьковой сортировки носит название шейкер-сортировки (shaker sort), поскольку действия, производимые ею с массивом, напоминают взбалтывание или встряхивание.

Шейкер – два накрывающих друг друга стакана, в котором встряхиванием вверх-вниз готовят коктейль. Шейкер-сортировка похожа на «пузырек», но здесь следующее улучшение:

1) чередуются направления последовательных просмотров;
2) запоминаются перестановки в процессе некоторого прохода. Если в последнем проходе перестановок не было, то алгоритм можно заканчивать;
3) запоминаются индексы последнего обмена.

Пример.



Ниже показана реализация шейкер-сортировки. Пример программы:

```
/* Шейкер-сортировка */
void shaker(char *items, int count) {
 register int a;
 int exchange;
 char t;
do {
exchange = 0;
for  (a=count-1; a > 0; --a)
{ if (items [a-1]> items [a]) {
 t = items [a-1] ;
 items [a-1] = items [a];
 items [a] = t;
 exchange = 1; }
 }
for (a=1; a < count; ++a) {
 if (items [a-1] > items [a])
 { t = items [a-1] ;
 items [a-1] = items [a];
 items [a] = t; exchange = 1; }
 }
} while (exchange) ; /* сортировать до тех пор, пока не бу-
дет обменов */
 }
```

Хотя шейкер-сортировка и является улучшенным вариантом по сравнению с пузырьковой сортировкой, она по-прежнему имеет время выполнения порядка n^2. Это объясняется тем, что количество сравнений не изменилось, а количество обменов уменьшилось лишь на относительно небольшую константу. Шейкер-сортировка лучше пузырьковой, но есть еще гораздо лучшие алгоритмы сортировки.

5.4.3. Параллельная сортировка Бэтчера (обменная сортировка со слиянием)

В этом методе для сравнений подбираются некоторые пары несоседних ключей (ki,kj).

Здесь программируется последовательность сравнений, предназначенная для отыскивания возможных обменов.

В алгоритме Бэтчера происходит слияние пар отсортированных подпоследовательностей. Поэтому она носит второе название – обменная сортировка со слиянием [5].

Алгоритм

Записи от R1 до Rn переразмещаются на том же месте и после перестановок ключи упорядочены:

$k_1<k_2<\ldots<k_n$, причем n>2.

В этом методе последовательность сравнений предопределена. Каждый раз сравниваются одни и те же пары ключей, независимо от того, что мы могли узнать из предыдущих сравнений.

1. [Начальная установка p.] Установка в $p \leftarrow 2^{t-1}$, где $t=[\log_2 n]$; $2^t \square N$.

2. [Начальная установка q,r,d.] Установить: $q \leftarrow 2^{t-1}$, $r \leftarrow 0, d \leftarrow p$.

3. [Цикл по i.] Для всех i, $0 \le i < n-d$ и $i \& p = r$ выполнить шаг 4. Затем перейти к шагу 5. (Здесь i&p – операция "логическое и" над двоичным представлением чисел i и p).

4. [Сравнение и обмен записей $[R_{i+1}, R_{i+d+1}]$] Если $k_{i+1} > k_{i+d+1}$, то соответствующие записи поменять местами: $R_{i+1} \leftrightarrow R_{i+d+1}$.

5. [Цикл по переменной q.] Если $q \neq p$, то $d \leftarrow q-p$, $q \leftarrow q/2$, $r \leftarrow p$, возврат к шагу 3.

6. [Цикл по p.] К этому моменту перестановки k1, k2, …, kN будут упорядочены. Установить $p \leftarrow [p/2]$. Если $p > 0$, то возвратиться к шагу 2.

5.4.4. Быстрая сортировка

Быстрая сортировка, придуманная Ч. А. Р. Хоаром (Charles Antony Richard Hoare) и названная его именем, является самым лучшим методом внутренней сортировки и обычно считается лучшим из существующих в настоящее время алгоритмом сортировки общего назначения. В ее основе лежит сортировка обменами – удивительный факт, учитывая ужасную производительность пузырьковой сортировки!

Это улучшенный метод, основанный на принципе обмена. Неожиданно оказалось, что усовершенствованный «пузырек» дает лучший результат из всех известных до настоящего времени методов сортировки. Он обладает столь блестящими характеристиками, что его изобретатель Хоар окрестил его «быстрой сортировкой», QuickSort (1962г.).

В методе Бэтчера последовательность сравнений предопределена. Она не зависит от фактического состояния файла (т.е. от того, насколько он уже упорядочен). Разумно предположить, что метод сортировки будет эффективнее, если последовательность сравнений элементов зависит от результатов предыдущих сравнений (т.е. будет учитываться состояние файла).

Q-sort основана на том факте, что для достижения наибольшей эффективности, желательно производить обмены элементов на больших расстояниях.

Алгоритм

Массив

i=1→ ← j=N

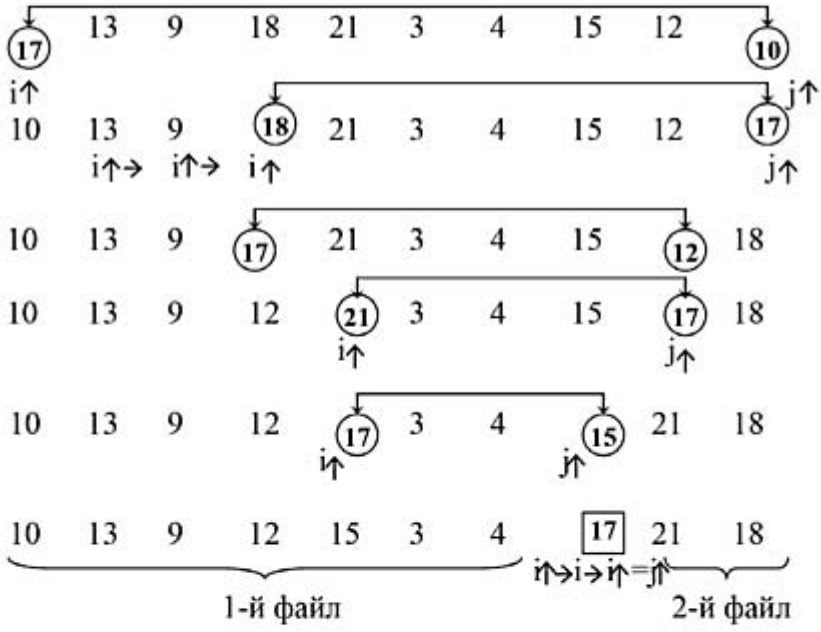
Пусть имеются два указателя i и j. Причем вначале: i=1, j=N.

Сравним k_i и k_j (k_i : k_j). Если обмен не нужен, то уменьшаем j на 1 и повторяем этот процесс. После первого обмена увеличим i на 1 и будем продолжать сравнения, увеличивая i, пока не произойдет еще один обмен, тогда уменьшим j и т.д., пока не станет i=j. К этому моменту запись Ri(Rj) займет свою окончательную позицию, и исходный файл будет разделен на два подфайла: $(R_1,\ldots R_{i-1})R_i(R_{i+1},\ldots,R_n)$. Эти подфайлы надо сортировать независимо (тем же методом).

В процессе сортировки образуется множество подфайлов, которые можно сортировать параллельно, независимо друг от друга.

Если подфайлы будут образовываться последовательно, то информация о неотсортированных подфайлах сбрасывается в стек. При этом целесообразно при образовании двух подфайлов в стек скидывать больший из них. Каждый из подфайлов сортируют методом Q-sort за исключением очень коротких. На рис. 5.1. представлена блок-схема алгоритма «Быстрая сортировка».

Пример.



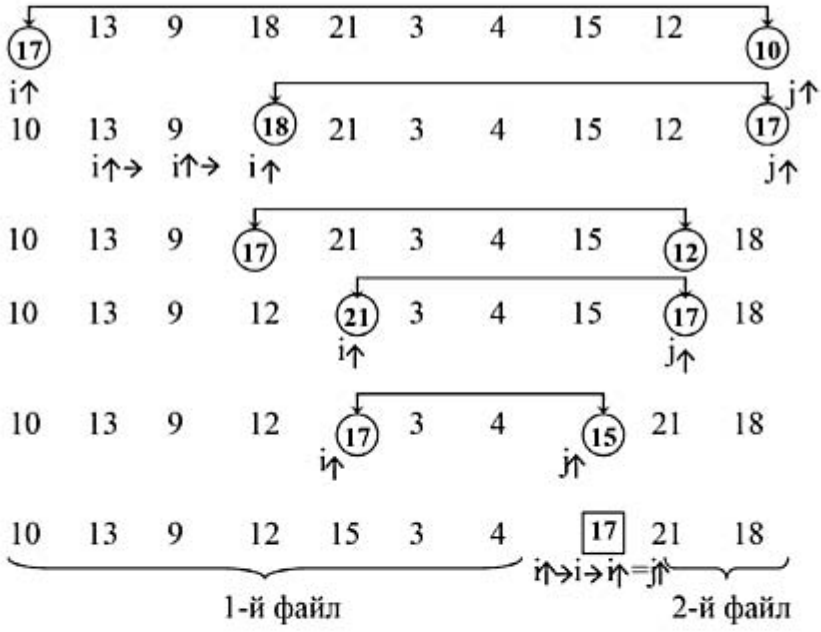
Пусть имеются два указателя i и j. Причем вначале: i=1, j=N.

Сравним k_i и k_j (k_i : k_j). Если обмен не нужен, то уменьшаем j на 1 и повторяем этот процесс. После первого обмена увеличим i на 1 и будем продолжать сравнения, увеличивая i, пока не произойдет еще один обмен, тогда уменьшим j и т.д., пока не станет i=j. К этому моменту запись Ri(Rj) займет свою окончательную позицию, и исходный файл будет разделен на два подфайла: $(R_1,\ldots R_{i-1})R_i(R_{i+1},\ldots,R_n)$. Эти подфайлы надо сортировать независимо (тем же методом).

В процессе сортировки образуется множество подфайлов, которые можно сортировать параллельно, независимо друг от друга.

Если подфайлы будут образовываться последовательно, то информация о неотсортированных подфайлах сбрасывается в стек. При этом целесообразно при образовании двух подфайлов в стек скидывать больший из них. Каждый из подфайлов сортируют методом Q-sort за исключением очень коротких. На рис. 5.1. представлена блок-схема алгоритма «Быстрая сортировка».

Пример.



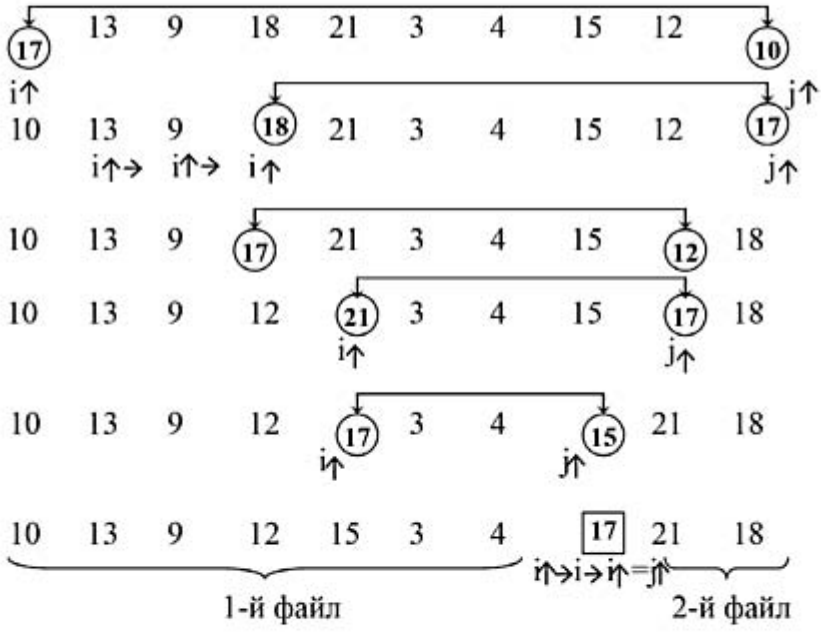
Пусть имеются два указателя i и j. Причем вначале: i=1, j=N.

Сравним k_i и k_j (k_i : k_j). Если обмен не нужен, то уменьшаем j на 1 и повторяем этот процесс. После первого обмена увеличим i на 1 и будем продолжать сравнения, увеличивая i, пока не произойдет еще один обмен, тогда уменьшим j и т.д., пока не станет i=j. К этому моменту запись Ri(Rj) займет свою окончательную позицию, и исходный файл будет разделен на два подфайла: $(R_1,\ldots R_{i-1})R_i(R_{i+1},\ldots,R_n)$. Эти подфайлы надо сортировать независимо (тем же методом).

В процессе сортировки образуется множество подфайлов, которые можно сортировать параллельно, независимо друг от друга.

Если подфайлы будут образовываться последовательно, то информация о неотсортированных подфайлах сбрасывается в стек. При этом целесообразно при образовании двух подфайлов в стек скидывать больший из них. Каждый из подфайлов сортируют методом Q-sort за исключением очень коротких. На рис. 5.1. представлена блок-схема алгоритма «Быстрая сортировка».

Пример.



Быстрая сортировка построена на идее деления. Общая процедура заключается в том, чтобы выбрать некоторое значение, называемое компарандом (comparand), а затем разбить массив на две части. Все элементы, большие или равные компаранду, перемещаются на одну сторону, а меньшие – на другую. Потом этот процесс повторяется для каждой части до тех пор, пока массив не будет отсортирован. Например, если исходный массив состоит из символов fedacb, а в качестве компаранда используется символ d, первый проход быстрой сортировки переупорядочит массив следующим образом:

Начало f e d a c b

Проход 1 b c a d e f

Затем сортировка повторяется для обеих половин массива, то есть bca и def. Очевидно , этот процесс по своей сути рекурсивный, и, действительно, в чистом виде быстрая сортировка реализуется как рекурсивная функция.

Значение компаранда можно выбирать двумя способами – случайным образом либо усреднив небольшое количество значений из массива. Для оптимальной сортировки необходимо выбирать значение, которое расположено точно в середине диапазона всех значений. Однако для большинства наборов данных это сделать непросто. В худшем случае выбранное значение оказывается одним из крайних. Тем не менее, даже в этом случае быстрая сортировка работает правильно. В приведенной ниже версии быстрой сортировки в качестве компаранда выбирается средний элемент массива. Пример.

```
/* Быстрая сортировка. */
void qs (char *items, int left, int right)
{
 register int i, j ;
 char x, y;
i = left; j = right;
 x = items [ (left+right) /2] ; /* выбор компаранда */
do {
while ((items [i] < x) && (i < right)) i++;
while ((x < items [j]) && (j > left)) j-- ;
 if(i <= j) {
 y = items [i] ;
 items [i] = items [j];
 items [ j ] = y;
 i++;j--;
 }
 } while(i <= j) ;
if (left < j) qs (items, left, j);
 if(i < right) qs (items, i, right);
}
void quick (char *items, int count)
{
 qs (items, 0, count-1);
}
```

В этой версии функция quick() готовит вызов главной сортирующей функции qs(), Это обеспечивает общий интерфейс с параметрами items и count, но несущественно, так как можно вызывать непосредственно функцию qs () с тремя аргументами.

Получение количества сравнений и обменов, которые выполняются при быстрой сортировке, требует математических выкладок, которые выходят за рамки данного отчета. Тем не менее, среднее количество сравнений равно n log n, а среднее количество обменов примерно равно n/6 log n. Эти величины намного меньше соответствующих характеристик рассмотренных ранее алгоритмов сортировки.

Необходимо упомянуть об одном особенно проблематичном аспекте быстрой сортировки. Если значение компаранда в каждом делении равно наибольшему значению, быстрая сортировка становится «медленной сортировкой» со временем выполнения порядка пг. Поэтому внимательно выбирайте метод определения компаранда. Этот метод часто определяется природой сортируемых данных. Например, в очень больших списках почтовой рассылки, в которых сортировка происходит по почтовому индексу, выбор прост, потому что почтовые индексы довольно равномерно распределены – компаранд можно определить с помощью простой алгебраической функции. Однако в других базах данных зачастую лучшим выбором является случайное значение. Популярный и довольно эффективный метод – выбрать три элемента из сортируемой части массива и взять в качестве компаранда значение, расположенное между двумя другими.

5.4.5. Обменная поразрядная сортировка

Этот метод открыли П.Хальдебрандт, Г.Исбитц, Х.Райзинг, Ж.Шварц примерно за год до открытия быстрой сортировки.

Данный метод совершенно отличен от всех схем сортировок, которые были описаны раньше. В нем используется двоичное представление ключей, и поэтому предназначен только для двоичных машин. Вместо того, чтобы сравнивать два ключа, в этом методе проверяется равны ли 0 или 1 отдельные биты ключа. В других отношениях он обладает характеристиками обменной сортировки и напоминает быструю сортировку.

В общих чертах этот алгоритм можно описать следующим образом:

Последовательность сортируется по старшему значащему биту так, чтобы все ключи начинающиеся с 0 оказались перед ключами начинающимися с 1. Для этого надо найти самый левый ключ R[i] начинающийся с 1 и самый правый ключ R[j] начинающийся с 0, после чего R[j] и R[i] меняются местами. Процесс повторяется пока не станет i > j.

Пусть F[0] – множество элементов начинающихся с 0, а F[1] – все остальные. Применим поразрядную сортировку к F[0] (начиная со второго бита слева, а не со старшего) до тех пор пока F[0] не будет полностью отсортировано. Теперь проделаем тоже самое с F[1].

Вместо того, чтобы сравнивать между собой два ключа, в этом методе проверяется, равны ли нулю или единице отдельные биты ключа от старшего бита к младшему. Рассмотрим метод, напоминающий метод быстрой сортировки, но использующий двоичное представление ключей. Здесь, вместо арифметических сравнений ключей проверяется, равны ли нулю или единице отдельные биты ключей.

В общих чертах идея метода следующая:

I этап. Файл сортируется по старшему значащему биту ключа так, что все ключи, начинающиеся с нуля, оказываются перед всеми ключами, начинающимися с единицы. Для этого надо найти самый левый ключ k_i, начинающийся с единицы, и самый правый k_j, начинающийся с нуля, после чего записи R_i и R_j меняются местами. Процесс повторяется до тех пор, пока не станет i>j.

II этап. Обозначим через F_0 множество элементов, начинающихся с нуля, а через F_1 – остальные. Применим к F_0 обменную поразрядную сортировку (начинаем теперь со второго бита слева) до тех пор, пока множество F_0 не будет полностью отсортировано. Затем проделываем то же самое с множеством F_1. Как и при быстрой сортировке, для хранения информации о подфайлах, ожидающих сортировки, можно воспользоваться стеком. Стратегию формирования стека здесь применяют другую: в стек помещают не больший из подфайлов, а правый подфайл, при этом элемент стека (r,b) означает, что подфайл с правой границей r ожидает сортировку по биту b. Левую границу можно не запоминать, она всегда задана неявно, так как файл обрабатывается слева направо.

Алгоритм:

Записи от 1-й до N-й $R_1,R_2,\ldots,R_N$ переразмещаются на том же месте. После завершения сортировки их ключи будут упорядочены

$$k_1 \le \ldots \le k_n .$$

Предполагается, что все ключи – это m-разрядные двоичные числа:

$$a_1,a_2,\ldots,a_m .$$

R1 [Начальная установка.] Опустошить стек, установить l←1, r←N, b←1.

R2 [Начать новую стадию.] Сортировать подфайл записей со следующими границами:

$$R_l \le \ldots \le R_r \text{ по биту b.}$$

По смыслу алгоритма l≤r; если l=r, то перейти к R10, в противном случае i←l,j←r.

R3 [Проверить k_i на единицу.] Проверить бит b ключа k_i на единицу. Если $(k_i)_b=1$, то перейти к шагу R6.

R4 [Увеличить i.] Если i≤j, то возвратиться к R3, иначе к шагу R8.

R5 [Проверить k_{j+1} на ноль.] Проверить бит b ключа k_{j+1}.

Если $(k_{j+1})_b=0$, то перейти к шагу R7.

R6 [Уменьшить j.] Уменьшить j на 1. Если $i \le j$, то перейти к шагу R5, иначе к

R7 [Поменять местами записи R_i и R_{j+1}.] Поменять местами R_i и R_{j+1}, затем перейти к шагу R4.

R8 [Проверить особые случаи.] К этому моменту стадия разделения завершена, i=j+1, бит b ключей от k_l до k_j равен нулю, а бит b ключей от k_i до k_r равен единице. Увеличить b на 1. Если b>m, где m – общее число битов в ключах, то перейти к шагу R10. Это значит, что подфайл $R_l \ldots R_r$ отсортирован. Если в файле не может быть равных ключей, то такую проверку можно не делать. Иначе, если j<l или j=r, возвратиться к шагу R2 (все просмотренные биты оказались равными соответственно единице или нулю).

Если j=l, то увеличить l на 1 и перейти к шагу R2 (встретился только один бит, равный нулю).

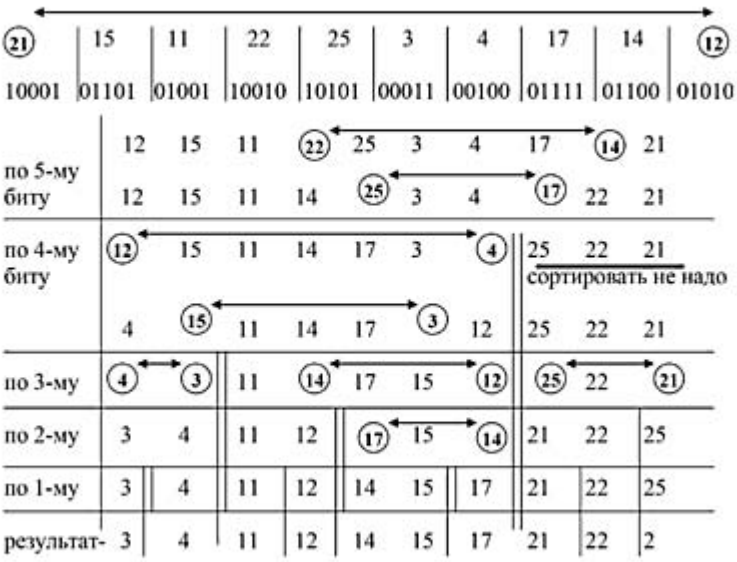
R9 [Поместить в стек.] Поместить в стек элемент (r,b), затем установить r←j и перейти к R2.

R10 [Взять из стека.] Если стек пуст, то сортировка закончена, иначе установить l←r+1, взять из стека элемент (r',b'), установить r←r',b←b' и возвратиться к шагу R2.

Для хранения "информации о границах" подфайлов, ожидающих сортировки, можно воспользоваться стеком.

Вместо того, чтобы сортировать в первую очередь наименьший из подфайлов, удобно просто продвигаться слева направо, так как размер стека в этом случае никогда не превзойдет числа битов в сортируемых ключах.

Пример. Сортировка методом обменная поразрядная. Ключи заданы в восьмеричном виде.



5.5. Методы вставок

5.5.1. Метод простых вставок

В обычной жизни мы сталкиваемся с этим методом при игре в карты. Чтобы отсортировать имеющиеся в Вас карты, Вы вынимаете карту, сдвигаете оставшиеся карты, а затем вставляете карту на нужное место. Процесс повторяется до тех пор, пока хоть одна карта находится не на месте.

Элементы условно подразделяются на готовую последовательность $a_1,\ldots,a_i$ и входную последовательность $a_{i+1},\ldots,a_n$. На каждом шаге, начиная с i=2 и увеличивая i на 1, берут i-й элемент входной последовательности и передают в готовую последовательность, вставляя его на подходящее место.

Пример.

Начальные ключи | 44 | 55 12 42 94 18 06 67
i=2 | 44 55 | 12 42 94 18 06 67
i=3 | 12 44 55 | 42 94 18 06 67
i=4 | 12 42 44 55 | 94 18 06 67
i=5 | 12 42 44 55 94 | 18 06 67
i=6 | 12 18 42 44 55 94 | 06 67
i=7 | 06 12 18 42 44 55 94 | 67
i=8 | 06 12 18 42 44 55 67 94|

Метод основывается на следующем: считается, что перед рассмотрением записи R[j] предыдущие записи R[1], R[2], ..., R[j-1] уже упорядочены, и R[j] вставляется в соответствующее место.

Сортировка таблицы начинается со второй записи. Ее ключ сравнивается с ключом первой записи, и, если упорядоченность нарушена, то записи R[1] и R[2] переставляются. Затем ключ записи R[3] сравнивается с ключами записей R[2] и R[1]. На j-м шаге K[j] сравнивается по очереди с K[j-1], K[j-2], ... (K[1]<=K[2]<=...<=K[j-1]) до тех пор, пока выполняется условие K[j] < K[i] (i = j-1, j-2, ...) или достигнут левый конец упорядоченной подтаблицы (i = 1, K[j] < K[1]). Выполнение условия K[j] >= K[i] означает, что запись R[j] нужно вставить между R[i] и R[i+1]. Тогда записи R[i+1], R[i+2], ..., R[j-1] сдвигаются на одну позицию, и запись R[j] помещается в позицию i+1.

Операции сравнения и перемещения записей удобно совмещать, перемежая их друг с другом (этот способ называется «просеиванием»).

Количество операций сравнения для данного метода вставки равно $N(N-1)/4$. Максимальное количество перестановок при использовании этого метода примерно равно $(N*N)/4$.

Пример.

```
/* array - сортируемая таблица (массив) */
/* size - количество эллементов */
void sort(int *array, int size)
{
register int i, j;
int temp;
for (i = 1; i < size; i++)
{
temp = array[i];
for  (j = i - 1; j >= 0; j--)
{
if(array[j] < temp)
break;
array[j+1] = array[j];
}
array[j+1] = temp;
}
}
```

Метод вставки обычно используется тогда, когда записи вносятся в упорядоченную таблицу. Новая запись должна быть вставлена в таблицу таким образом, чтобы существующая упорядоченность не нарушалась. Этот метод эффективен когда записи частично упорядочены, т.е. записи находятся не далеко от своих конечных положений.

5.5.2. Метод бинарного включения (бинарные вставки)

Этот метод упоминается Джоном Мочли в 1946г. в первой публикации по машинной сортировке.

Данный метод является улучшенной версией предыдущего метода простых вставок. Он основывается на том факте, что мы вначале ищем место куда нужно вставить элемент, а затем уже вставляем. Это дает существенный выигрыш по числу сравнений, но число перестановок по прежнему остается большим.

Поскольку все методы массива перед записью R[j] уже упорядочены, то можно использовать более эффективный метод поиска места куда надо вставить R[j]. В данном случае это алгоритм бинарного поиска. Например, если вставляется 64-я запись, то можно сравнить ее с 32-й, и если 64-я запись окажется меньше, то сравнить ее уже с 16-й, а если окажется больше, то сравнить с 48-й и т.д. Таким образом, место 64-й записи будет определено в худшем случае за шесть сравнений.

Общее число сравнений равно приблизительно N*logN, число перестановок по-прежнему остается квадратичнозависимым от N.

5.5.3. Метод двухпутевых вставок

Существует несколько методов для уменьшения числа перемещений в описанном методе вставок. Один из них – метод двухпутевых вставок, разработанный в начале 50-х годов.

Первый элемент помещается в середину области вывода, и место для последующих выводов освобождается сдвигом влево или вправо, туда куда удобней. Таким образом удается сэкономить примерно половину времени по сравнению с простыми вставками за счет некоторого усложнения алгоритма. Число перестановок сократится примерно в 2 раза до N*N/8.

5.5.4. Вставки одновременно нескольких элементов

Модификация метода простых вставок заключается в том, что вместо одной переменной X можно использовать несколько переменных X1, X2, ... Xm, которые имеют значения элементов, подлежащих вставке в уже упорядоченную часть файла. X1, X2, ... Xm упорядочены по возрастанию, поэтому сравнивая Xm в цикле по переменной i с элементами упорядоченной части, мы можем гарантировать, что, если очередной элемент k[i] больше Xm, то он больше и остальных элементов.

Перенос элементов исходной таблицы вперед в цикле по i выполняется на m элементов, то есть вместо k[i+1] = k[i] в исходном алгоритме в модифицированном алгоритме записывается k[i+m] = k[i]. При нахождении k[i] такого, что он меньше Xm, Xm ставится на место k[i+m] и m уменьшается на 1. Далее цикл по i продолжается с новым m. Экономия числа переносов элементов достигается за счет переносов сразу на m элементов.

5.5.5. Метод Шелла (сортировка вставками с убывающим шагом)

Этот метод был предложен Дональдом Л. Шеллом в 1959 г. Основная идея этого алгоритма заключается в том, чтобы в начале устранить массовый беспорядок в массиве, сравнивая далеко стоящие друг от друга элементы. Как видно, интервал между сравниваемыми элементами постепенно уменьшается до единицы. Это означает, что на поздних стадиях сортировка сводится просто к перестановкам соседних элементов (если, конечно, такие перестановки являются необходимыми)[1,2].

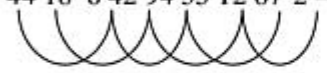
Идея метода: Сортируются группы записей, отстоящих друг от друга на заданном расстоянии (например, 8). После сортировки всех таких групп, сортируются группы записей, отстоящих на меньшее расстояние (4) и т.д., пока не дойдем до расстояния, равного 1.

Пример. Отсортировать вручную при приращениях 4,2,1.

(4),(2),(1) – приращения

44 55 12 42 94 18 6 67 4 – сортировка

44 18 6 42 94 55 12 67 2 – сортировка



6 18 12 42 44 55 94 67 1 – сортировка

6 12 18 42 44 55 67 94 результат

Затраты, которые требуются для сортировки n элементов с помощью алгоритма Шелла, пропорциональны $n^{1,2}$.

Существует формула для расчета приращений:

n – количество записей;

t – количество приращений;

$t=[\log_3 n]$.

$h_t=1$; $h_{k-1}=3*h_k+1$,

где h – приращение.

Пример. Рассчитать ряд приращений.

n=27; t=3;

$h_3=1$; $h_2=4$; $h_1=13$.

Можно использовать самые разные схемы выбора шагов (следует избегать степеней двойки). Как правило, сначала мы сортируем массив с большим шагом, затем уменьшаем шаг и повторяем сортировку. В самом конце сортируем с шагом 1. Хотя этот метод легко объяснить, его формальный анализ довольно труден. В частности, теоретикам не удалось найти оптимальную схему выбора шагов. Кнут провел множество экспериментов и предложил следующую формулу выбора шагов (h) для массива длины N:

h[1] = 1, h[i] = h[i-1]*3 + 1, h[s] > N

Вот несколько первых значений h:

h[1] = 1;

h[2] = 1*3 + 1 = 4;

h[3] = 4*3 + 1 = 13;

h[4] = 13*3 + 1 = 40;

h[5] = 40*3 + 1 = 121;

Получаем ряд приращений 1,4,13,40,121…

Чтобы отсортировать массив длиной 100, прежде всего найдем номер s, для которого h[s] > 100. Согласно приведенным числам, s = 5. Нужное нам значение находится двумя строчками выше. Таким образом, последовательность шагов при сортировке будет такой: 13-4-1. Ну, конечно, нам не нужно хранить эту последовательность: очередное значение h находится из предыдущего по формуле h[i] = h[i+1]/3.

5.6. Методы выбора

5.6.1. Сортировка методом прямоугольного перебора

Алгоритм сортировки методом прямоугольного перебора базируется на конечном количестве значений элементов сортируемого массива. Данный алгоритм может быть использован как для сортировки строк, так и чисел. Он удобен для сортировки небольших чисел, алфавитных списков. Также он может найти применение в небольших программах, где большое значение имеет размер алгоритма. Сам алгоритм имеет такой вид:

1. Организовывается цикл от u до v, где u – минимальное значение массива, v – максимальное, т.е. v-u – количество возможных значений элемента массива.

2. Вложенный цикл перебора. Перебираются значения исходного массива. Если значения массива совпадает с счетчиком предыдущего цикла, то элемент выводится в конечный массив. Пример.

```
/* array - сортируемая таблица (массив) */
/* out - таблица вывода (массив) */
/* size - количество эллементов */
/* u - минимальное значение ключа */
/* v - максимальное значение ключа */
void sort(int *array, int *out, int size, int u, int v)
{
register int i, j, k = 0;
for (i = u; i <= v; i++)
{
for (j = 0; j < size; j++)
{
if(array[j] == i)
{
out[k] = array[j];
k++;}}}
return;}
```

Число сравнений равно примерно M*N, где M – диапозон возможных значений элементов массива (v-u). Число перестановок равно N.

5.6.2. Сортировка простым выбором

На этот раз при просмотре массива мы будем искать наименьший элемент, сравнивая его с первым. Если такой элемент найден, поменяем его местами с первым. В результате такой перестановки элемент с наименьшим значением ключа помещается в первую позицию в таблице. Затем повторим эту операцию, но начнем не с первого элемента, а со второго. И будем продолжать подобным образом, пока не рассортируем весь массив.

Простейшая сортировка, построенная на этой идее, состоит из следующих шагов:

1. Найти наименьший ключ, после чего выбрать соответствующую ему запись и переслать ее в начало некоторой пустой области памяти (область вывода).

Так, как операция ввода соответствует чтению данных и не удаляет физически запись из сортируемого файла, то удаление имитируется заменой ключа на значение «∞» (∞ по предположению больше любого реального ключа).

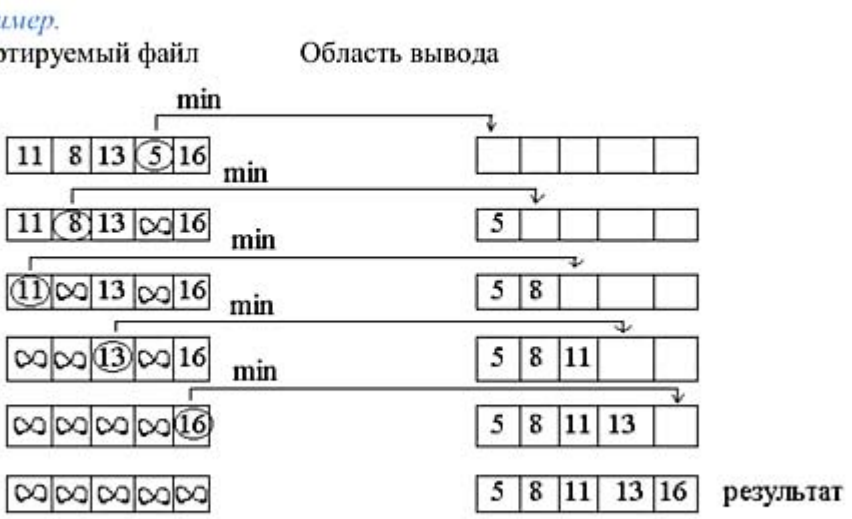
2. Повторить шаг 1. На этот раз будет найден ключ, наименьший из оставшихся.

3. Повторять шаг 1 до тех пор, пока не будут выбраны все N записей.

Этот метод требует наличия всех исходных элементов до начала сортировки, а элементы вывода он порождает последовательно, один за другим. Картина по существу противоположна методу вставок, в котором исходные элементы должны поступать последовательно, но вплоть до завершения сортировки ничего не известно об окончательном выводе.

Описанный метод требует N-1 сравнений каждый раз, когда выбирается очередная запись.

Пример.



Недостатки этого метода очевидны, если использовать последовательное размещение записей файла в памяти:

- большое число сравнений N(N-1);
- необходима дополнительная память.

Этот метод можно модифицировать следующим образом:
выбранную на очередном шаге запись помещать в тот же самый файл на место, соответствующее месту в случае использования области вывода, а находящуюся на этом месте запись перенести на место выбранной. При этом отпадает необходимость в использовании "∞", и количество просматриваемых элементов с каждым шагом сокращается на 1.

Пример программы:

```
/* array - сортируемая таблица (массив) */
/* size - количество эллементов */
void sort(int *array, int size)
{
register int i, j, k;
int temp;
for (i = 0, k = 0; i < size - 1; i++, k++)
{
temp = array[i];
for (j = i + 1; j < size; j++)
{
if(temp > array[j])
{
temp = array[j];
k = j;}}
temp = array[i];
array[i] = array[k];
array[k] = temp;}
return;}
```

Число сравнений в данном методе равно N(N-1)/2. Максимальное количество перестановок при такой сортировке равно (N-1).

5.6.3. Линейный выбор с подсчетом (сравнение и подсчет)

Данный метод впервые упоминается в работе Э.Х. Фрэнда, хотя он и не заявил о ней как о своей собственной разработке.

При упорядочении таблицы этим методом необходима память для хранения исходной таблицы, а также дополнительно должна быть выделена память под счетчик для каждого элемента таблицы.

В этом алгоритме элементы не перемещаются. Он подобен сортировке таблицы адресов, поскольку таблица счетчиков определяет конечное положение элементов.

Просмотр таблицы начинается с первой записи. Ее ключ сравнивается с ключами последующих записей. При этом счетчик большего из сравниваемых ключей увеличивается на 1. При втором просмотре таблицы первый ключ уже не рассматривается, второй ключ сравнивается со всеми последующими. Результаты сравнений фиксируются в счетчиках. Для таблицы, содержащей N элементов, этот процесс повторяется N-1 раз.

После выполнения всех просмотров счетчик каждого элемента указывает, какое количество ключей таблицы меньше ключа этого элемента. Эти счетчики используются затем в качестве индексов элементов результирующей таблицы. Поместив записи в результирующую таблицу в соответствии со значениями их счетчиков, получим упорядоченную таблицу. Другими словами, если известно, что некоторый ключ превышает ровно 27 других, то после сортировки соответствующий элемент должен занять 28-е место.

```
/* array - сортируемая таблица (массив) */
/* count - таблица счетчиков (массив) */
/* size - количество эллементов */
void sort(int *array, int *count, int size)
{
register int i, j;
for (i = 0; i < size; i++)
count[i] = 0;
for (i = 0; i < size - 1; i++)
{
for (j = i + 1; j < size; j++)
{
if(array[i] < array[j])
count[j]++;
else
count[i]++;})
return;}
```

Выполняется N-1 просмотров с N/2 сравнениями в среднем при каждом просмотре. Значения счетчиков изменяются столько раз, сколько выполняется сравнений.

5.6.4. Распределяющий подсчет

Существует другая разновидность сортировки посредством подсчета, которая действительно важна с точки зрения эффективности.

Алгоритм впервые разработан Х. Сьювордом в 1954г. и применялся при поразрядной сортировке.

Этот алгоритм применяется в том случае, когда имеется много равных ключей и все они попадают в диапазон u <= K <= v, где (u – v) невелико (ключи должны быть целыми числами). Эти предположения представляются весьма

ограниченными, но на самом деле существует немало приложений этой идеи, например, если применить этот метод лишь к старшим цифрам ключей, а не ко всем ключам целиком, то массив окажется частично отсортированным и будет уже достаточно просто довести дело до конца.

```
      /* array - сортируемая таблица (массив) */
      /* out - таблица вывода (массив) */
      /* count - таблица счетчиков размером u - v (массив) */
      /* size - количество эллементов */
      /* u - минимальное значение ключа */
      /* v - максимальное значение ключа */
      void sort(int *array, int *out, int *count, int size, int u,
 int v)
      {
      register int i, j;
      for (i = 0; i < size; i++)
      count[i] = 0;
      for (i = 0; i < size; i++)
      count[array[i] - u]++;
      for (i = 1; i <= v - u; i++)
      count[i] += count[i-1];
      for (i = size - 1; i > 0; i--)
      {
      j = count[array[i]];
      out[j] = array[i];
      count[array[i]]--;}
      return;}
```

5.6.5. Пирамидальная сортировка

Последовательность клчей K[1], K[2], ..., K[N] называется пирамидой, если K[j/2] >= K[j] при 1 <= j/2 < j <= N. В этом лучае K[1] >= K[2], K[1] >= K[3], K[2] >= K[4] и т.д. Из этого условия следует, что наименьший ключ находится на «вершине пирамиды».

Если бы мы сумели преобразовать любую произвольную таблицу в пирамиду, то для получения эффективного алгоритма сортировки можно было бы использовать нисходящую процедуру выборки.

```
1

     3 5

     6 9 7

     12 10 12 13

     . . . . . .
```

Эффективный подход к построению дерева предложил Р.У. Флойд, а Дж. У. Уильямс разработал алгоритм сортировки пирамиды.

Алгоритм содержит два последовательных этапа: построение пирамиды и выбор элементов из корня пирамиды с последующим ее восстановлением. Опишем сначала алгоритм восстановления. Если корень меньше хотя бы одного из своих потомков, то меняем местами корень и больший из потомков и применяем эту процедуру последовательно ко всем поддеревьям, в которых мы заменили корень.

Этап построения пирамиды можно вывести из этапа восстановления. Исходный файл можно представить как набор тривиальных пирамид, состоящих из одного элемента. Последовательно объединяя их по алгоритму восстановления, мы в конце концов приходим к полной пирамиде.

```
/* array - сортируемая таблица (массив) */
/* size - количество эллементов */
void sort(int *array, int size)
{
register int i, j, l = size / 2, r = size - 1;
int temp;
while(1)
{
if(l > 0)
{
l--;
temp = array[l];
}
else
{
temp = array[r];
array[r] = array[0];
r--;
if(r == 0)
{
    array[0] = temp;
    return;}}
i = l;
while(1)
{
j = i;
i = i*2 + 1;
if(i < r)
{
    if(array[i] < array[i+1])
    i++;}
else if(i != r)
    break;
if(temp >= array[i])
    break;
array[j] = array[i];}
array[j] = temp;}}
```

При достаточно больших N этот метод превосходит метод Шелла, но никогда не превосходит метод быстрой сортировки. Но, с другой стороны, время работы быстрой сортировки в худшем случае пропорционально квадрату N. Пиромидальная сортировка в худшем сучае не займет больше 16logN + 38N единиц времени. В среднем же случае ее показатель примерно 16logN.

5.6.6. Метод квадратичной выборки

Данный метод по сравнению с сортировкой выборкой уменьшает число сравнений, но требует дополнительного объема памяти. Сортируемая таблица, состоящая из N элементов, разделяется на N групп по sqrt(N) элементов в каждой. Если N не является точным квадратом,то таблица разделяется на N' групп, где N' есть ближайший точный квадрат, больший N. В каждой группе выбирается наименьший элемент, который пересылается во вспомогательный список. Вспомогательный список просматривается, и наименьший его элемент пересылается в зону вывода (в зоне вывода формируется отсортированный список). Далее из группы, содержащей элемент, посылаемый в зону вывода, выбирается новый наименьший элемент, который помещается во вспомогательный список. Затем другой просмотр вспомогательного списка выбырает новый наименьший элемент, который является вторым по величине во всем списке. Он пересылается в зону вывода. Элементы групп, которые уже посланы во вспомогательный список, заменяются большими фиктивными величинами.

Таким образом, при сортировке квадратичной выборкой попеременно просматриваются то вспомогательный список, то группа, до тех пор, пока все группы не будут исчерпаны. Такое состояние наступает, когда все группы посылают во вспомогательный список фиктивные величины. Модификацией данного метода является квадратичная выборка с предварительной сортировкой. В этом методе группы сначала полностью упорядочиваются, а затем уже выполняются сравнения между группами.

Общее число сравнений для случая точного квадрата равно приблизительно 2N*sqrt(N), необходимый резерв памяти – поле длиной N+sqrt(N) элементов.

Рассмотрим следующий метод: пусть файл содержит 16 записей. Разобьем его на четыре группы по четыре записи:

‖ 14 5 18 21 ‖ 2 7 11 3 ‖ 9 26 1 8 ‖ 13 22 6 20 ‖ .

Найдем наибольшие элементы в каждой группе и соберем их вместе в группу «лидеров»':

21 11 26 22

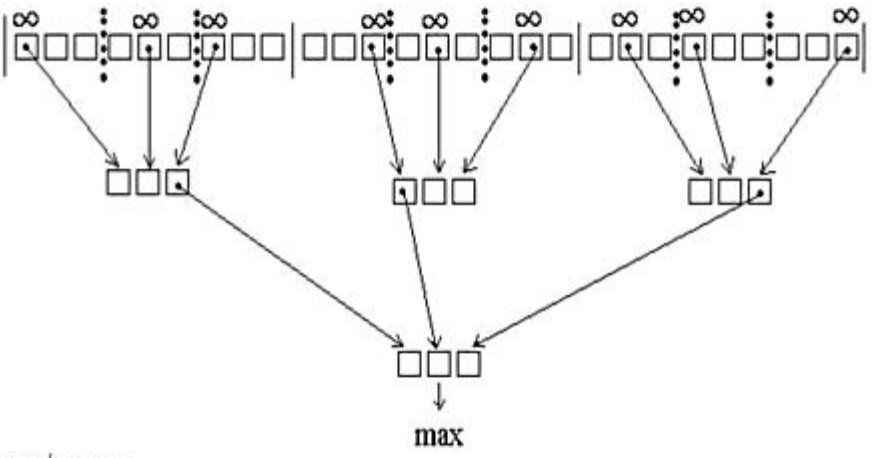
Тогда наибольший среди них и будет наибольшим в файле. Чтобы получить второй по величине элемент, достаточно просмотреть оставшиеся элемен-

ты группы, содержащей 26 (т.е. максимальный), и максимальный из них поместить в «группу лидеров» на место 26 и т.д.

Вообще, если N – точный квадрат, то можно разделить файл на √N групп по √N элементов в каждой. Тогда любой шаг, кроме первого, потребует не более, чем √N – 1 сравнений внутри группы ранее выбранного элемента плюс √N – 1 сравнений в "группе лидеров". Этот метод получил название "квадратичный выбор".

Оказывается такой метод может быть обобщен: существует метод кубического выбора ($\sqrt[3]{N}$ больших групп, каждая из которых содержит $\sqrt[3]{N}$ малых групп по $\sqrt[3]{N}$ записей), выбора четвертой степени и т.д. Если эту идею развить до ее полного логического завершения, то мы придем к выбору n-й степени, основанному на структуре бинарного дерева.

Пример. Схема кубического выбора.



(n=27, $\sqrt[3]{27}$=3);
max –максимальный элемент помещаем в выходную последовательность.

5.6.7. Выбор из дерева

Этот метод является выбором n-й степени, где n выбрано так, что размер самой малой группы минимален и равен 2.

Рассмотрим, например, результаты соревнований по настольному теннису: Сидоров побеждает Володина, а Иванов побеждает Сергеева; затем в следующем туре Иванов выигрывает у Сидорова и т.д. (рис 4.2.) [3]

На этом рисунке показано, что Иванов – чемпион среди восьми спортсменов, а для того, чтобы определить это, потребовалось 8-1=7 матчей (т.е. сравнений).

Рис. 5.2. Дерево выбора

Петров вовсе не обязательно будет вторым по силе игроком: любой из спортсменов, у которых выиграл Иванов (чемпион), включая даже проигравшего в первом туре Сергеева, мог бы оказаться вторым по силе игроком. Второго игрока можно определить, заставив сыграть Сергеева с Сидоровым, а победителя этого матча – с Петровым. Всего два матча требуется для определения второго по силе игрока, исходя из того соотношения сил, которое мы запомнили из предыдущих игр.

Метод сортировки простым выбором основан на повторном выборе наименьшего ключа среди n элементов, затем среди n-1 элементов и т.д. Поиск наименьшего ключа из n элементов потребует n-1 сравнений, из (n-1) элементов – (n-2) сравнений. Как можно улучшить эту сортировку выбором? Это можно сделать только в том случае, если получить от каждого прохода больше информации, чем просто указание на один наименьший элемент.

Например, с помощью n/2 сравнений можно определить наименьший элемент (ключ) из каждой пары, при помощи n/4 сравнений можно выбрать наименьший из каждой пары таких наименьших ключей и т.д. Наконец, при помощи n-1 сравнений можно построить дерево выбора и определить корень как наименьший ключ.

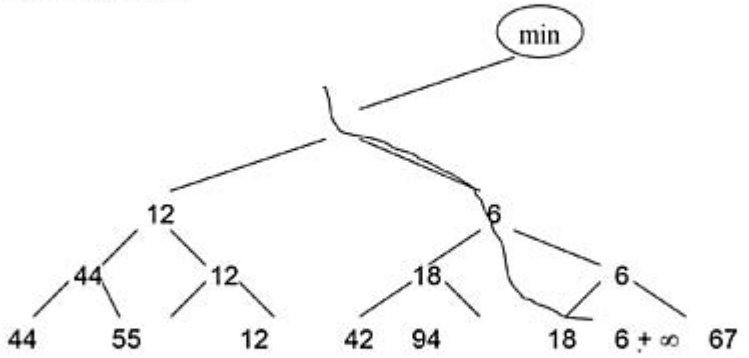


Рис. 5.3. Дерево выбора, выбор 1-го элемента

На втором шаге нужно спуститься по пути, указанному наименьшим ключом, и исключить его, последовательно заменяя либо на «дыру» (или ключ ∞), либо на элемент, находящийся на противоположной ветви промежуточного узла.

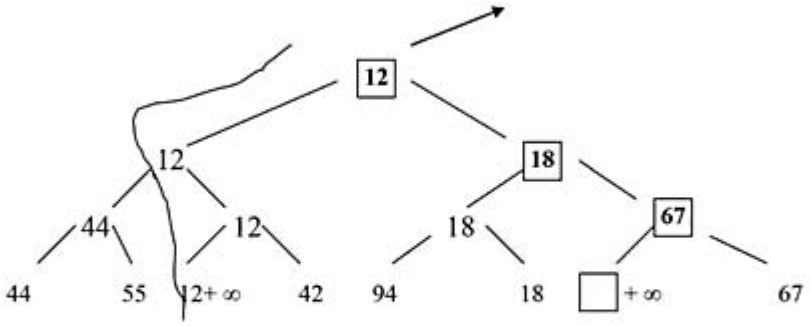


Рис. 5.4. Дерево выбора, выбор 2-го элемента

Элемент, оказавшийся в корне дерева, вновь имеет меньший ключ (среди оставшихся) и может быть исключен.

После n таких шагов дерево становится пустым (т.е. состоит из "дыр"), и процесс сортировки закончен.

Каждый из n шагов требует $\log_2 n$ сравнений. Вся сортировка требует порядка $n*\log_2 n$ элементарных операций, не считая n шагов, которые необходимы для построения дерева.

Это значительное улучшение по сравнению с простым методом, требующим n^2 шагов. При сортировке с помощью дерева задача хранения информации стала сложнее и поэтому увеличилась сложность отдельных шагов. Для хранения возросшего объема информации, получаемой на начальном проходе, нужно строить некоторую древовидную структуру.

5.7. Методы слияния

Сортировка слиянием

Слияние – означает объединение двух или более упорядоченных файлов в один упорядоченный файл.

Можно, например, слить два подфайла,

1-й – 3 6 11 21 30

2-й – 1 4 7 9 28

получив 1 3 4 6 7 9 11 21 28 30.

Простой способ сделать это – сравнить два наименьших элемента, вывести наименьший из них и повторить процедуру.

Начав с

$$\begin{pmatrix} 3 & 6 & 11 & 21 & 30 \\ 1 & 4 & 7 & 9 & 28 \end{pmatrix}$$

получим

$$1\ 3 \begin{pmatrix} 6 & 11 & 21 & 30 \\ 4 & 7 & 9 & 28 \end{pmatrix}$$

затем

$$1 \begin{pmatrix} 3 & 6 & 11 & 21 & 30 \\ 4 & 7 & 9 & 28 \end{pmatrix}$$

и т.д.

Самый простой алгоритм здесь – двухпутевое слияние.

5.7.1. Двухпутевое слияние

Алгоритм М

Этот алгоритм осуществляет слияние двух упорядоченных файлов $x_1 \le x_2 \le \ldots \le x_m$ и $y_1 \le y_2 \le \ldots \le y_n$ в $z_1 \le z_2 \le \ldots \le z_{m+n}$.

М1. [Начальная установка.] Установить $i \leftarrow 1, j \leftarrow 1, k \leftarrow 1$.

М2. [Найти наименьший элемент.] Если $x_i \le y_j$, то перейти к шагу М3, в противном случае перейти к шагу М5.

М3. [Вывести x_i] Установить $z_k \leftarrow x_i$, $k \leftarrow k+1$, $i \leftarrow i+1$. Если $i \le m$, то возвратиться к М2.

М4. [Вывести $y_j, \ldots, y_n$] Установить $(z_k, \ldots, z_{m+n}) \leftarrow (y_j, \ldots, y_n)$ и завершить работу алгоритма.

М5. [Вывести y_j] Установить $z_k \leftarrow y_j$, $k \leftarrow k+1$, $j \leftarrow j+1$. Если $j \le n$, то возвратиться к М2.

М6. [Вывести $x_i, \ldots, x_m$] Установить $(z_k, \ldots, z_{m+n}) \leftarrow (x_i, \ldots, x_m)$ и завершить работу алгоритма.

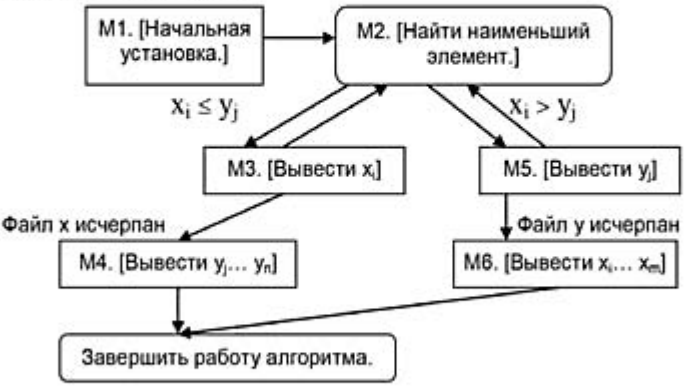


Рис. 5.5. Алгоритм «Двухпутевое слияние»

Общий объем работы, выполняемой алгоритмом M, по существу пропорционален m+n (отсюда слияние – более простая задача, чем сортировка). Задачу сортировки можно свести к слияниям, сливая все более длинные подфайлы до тех пор, пока не будет отсортирован весь файл. Такой подход можно рассматривать как развитие идеи сортировки вставками: вставка нового элемента в упорядоченный файл – частный случай слияния при n=1.

5.7.2. Слияние списков

Алгоритмы слияния имеют два существенных недостатка.

1. Большой расход памяти для вспомогательной рабочей области.
2. Необходимость большого числа перемещений записей.

Эти недостатки можно устранить, создав из сортируемого файла подобие связанного списка, для чего каждая запись R_i должна иметь "поле связи" L_i, в котором будет храниться № записи, следующей за данной записью в порядке возрастания ключей[5]. После сортировки L_0=№ записи с наименьшим ключом.

Алгоритм L

Предполагается, что записи $R_1,\ldots,R_N$ содержат ключи $k_1,\ldots,k_N$ и поля связи $L_1,\ldots,L_N$, в которых могут храниться числа от -(N+1) до (N+1). В начале и в конце файла имеются искусственные записи R_0 и R_{N+1} с полями связи L_0 и L_{N+1}. Этот алгоритм сортировки устанавливает поля связи таким образом, что записи оказываются связанными в возрастающем порядке.

После завершения сортировки L_0 указывает на запись с наименьшим ключом, при $1\le k\le N$ связь L_k указывает на запись, следующую за R_k, а если R_k – запись с наибольшим ключом, то L_k=0. В процессе выполнения этого алгоритма записи R_0 и R_{N+1} служат "головами" двух линейных списков, подсписки которых в данный момент сливаются.

R_0	R_1	R_2	…	R_N	R_{N+1}
	K_1	K_2	…	K_N	
L_0	L_1	L_2	…	L_N	L_{N+1}

Отрицательная связь означает коненц подсписка, о котором известно, что он упорядочен; нулевая связь означает конец списка. Предполагается, что $N\ge 2$.

Через "$|L_s|\leftarrow p$" обозначена операция присвоить L_s значение p или –p, сохранив прежний знак L_s.

L1. [Подготовить два списка]. Установить $L_0\leftarrow 1$, $L_{N+1}\leftarrow 2$, $L_i\leftarrow -(i+2)$ при $1\le i\le N-2$ и $L_{N-1}\leftarrow L_N\leftarrow 0$.

Мы создаем два списка, содержащие соответственно записи $R_1,R_3,R_5,\ldots$ и $R_2,R_4,R_6,\ldots$; отрицательные связи говорят о том, что каждый упорядоченный "подсписок" состоит всего лишь из одного элемента. Другой способ выполнить этот шаг, извлекая пользу из упорядоченности, которая могла присутствовать в исходных данных.

L2. [Начать новый просмотр.] Установить s←0, t←N+1, p←L_s, q←L_t. Если q=0, то работа алгоритма завершена. При каждом просмотре p и q пробегают по спискам, которые подвергаются слиянию; s обычно указывает на последнюю обработанную запись текущего подсписка, а t – на конец только что выведенного подсписка.

L3. [Сравнить $k_p : k_q$]. Если $k_p > k_q$, то перейти к L6.

L4. [Продвинуть p]. Установить $|L_s|$←p, s←p, p←L_p. Если p>0, то возвратиться к шагу L3.

L5. [Закончить подсписок]. Установить L_s←q, s←t. Затем установить t←q и q←L_q один или более раз, пока не станет q≤0, после чего перейти к шагу L8.

L6. [Продвинуть q]. (Шаги L6 и L7 двойственны по отношению к L4 и L5.) Установить $|L_s|$←q, s←q, q←L_q. Если q>0, то возвратиться к шагу L3.

L7. [Закончить подсписок]. Установить L_s←p, s←t. Затем установить t←p и p←L_p один или более раз, пока не станет p≤0.

L8. [Конец просмотра]. К этому моменту p≤0 и q≤0, так как оба указателя подвинулись до конца соответствующих подсписков.

Установить p← -p, q← -q. Если q=0, то установить $|L_s|$←p, $|L_t|$←0 и возвратиться к шагу L2. В противном случае возвратиться к шагу L3.

Выводы

В данном разделе мы рассмотрели множество алгоритмов и поэтому возникает вопрос: какой же алгоритм использовать?

Мы рассмотрели методы внутренней сортировки, которые эффективны при сортировки данных в памяти компьютера. Но часто приходится сортировать данные в файлах на жестком диске, т.к. современные объемы данных превышают возможности оперативной памяти современных компьютеров (кстати так было и будет всегда). Это, во-первых, связано с сегментацией памяти, а во-вторых, с ограниченным размером памяти (например, элемент данных может занимать десятки или сотни байт).

Вот некоторые рекомендации для выбора алгоритма сортировки:

Если сортируется небольшой объем данных (N < 100), то рекомендуется выбирать простой метод сортировки, так как сложные алгоритмы занимают больший размер кода и не эффективны при малом количестве сортируемых элементов.

Если сортируются элементы большого размера, то рекомендуется использовать специальные таблицы с помощью которых осуществляется доступ к самим элементам (например, это могут быть указатели на элементы или их порядковые номера), причем ключ по которому сортируются данные может входить или не входить в данные такой таблицы. После сортировки таблицы можно или переставить исходные данные по таблице, или оставить все как есть и осуществлять дальнейший доступ к данным по уже отсортированной таблице.

L2. [Начать новый просмотр.] Установить s←0, t←N+1, p←L_s, q←L_t. Если q=0, то работа алгоритма завершена. При каждом просмотре p и q пробегают по спискам, которые подвергаются слиянию; s обычно указывает на последнюю обработанную запись текущего подсписка, а t – на конец только что выведенного подсписка.

L3. [Сравнить $k_p : k_q$]. Если $k_p > k_q$, то перейти к L6.

L4. [Продвинуть p]. Установить $|L_s|$←p, s←p, p←L_p. Если p>0, то возвратиться к шагу L3.

L5. [Закончить подсписок]. Установить L_s←q, s←t. Затем установить t←q и q←L_q один или более раз, пока не станет q≤0, после чего перейти к шагу L8.

L6. [Продвинуть q]. (Шаги L6 и L7 двойственны по отношению к L4 и L5.) Установить $|L_s|$←q, s←q, q←L_q. Если q>0, то возвратиться к шагу L3.

L7. [Закончить подсписок]. Установить L_s←p, s←t. Затем установить t←p и p←L_p один или более раз, пока не станет p≤0.

L8. [Конец просмотра]. К этому моменту p≤0 и q≤0, так как оба указателя подвинулись до конца соответствующих подсписков.

Установить p← -p, q← -q. Если q=0, то установить $|L_s|$←p, $|L_t|$←0 и возвратиться к шагу L2. В противном случае возвратиться к шагу L3.

Выводы

В данном разделе мы рассмотрели множество алгоритмов и поэтому возникает вопрос: какой же алгоритм использовать?

Мы рассмотрели методы внутренней сортировки, которые эффективны при сортировки данных в памяти компьютера. Но часто приходится сортировать данные в файлах на жестком диске, т.к. современные объемы данных превышают возможности оперативной памяти современных компьютеров (кстати так было и будет всегда). Это, во-первых, связано с сегментацией памяти, а во-вторых, с ограниченным размером памяти (например, элемент данных может занимать десятки или сотни байт).

Вот некоторые рекомендации для выбора алгоритма сортировки:

Если сортируется небольшой объем данных (N < 100), то рекомендуется выбирать простой метод сортировки, так как сложные алгоритмы занимают больший размер кода и не эффективны при малом количестве сортируемых элементов.

Если сортируются элементы большого размера, то рекомендуется использовать специальные таблицы с помощью которых осуществляется доступ к самим элементам (например, это могут быть указатели на элементы или их порядковые номера), причем ключ по которому сортируются данные может входить или не входить в данные такой таблицы. После сортировки таблицы можно или переставить исходные данные по таблице, или оставить все как есть и осуществлять дальнейший доступ к данным по уже отсортированной таблице.

Если сортируются данные в файле, то необходимо учитывать, что большая часть времени будет тратиться на чтение, запись элемента и перемещение по файлу. В такой ситуации методы вставок являются не эффективными, так как требуют большое число перестановок. (Кстати, при внутренней сортировке время сравнения и перестановки двух элементов практически одинаковое.) Так же возможно считывать небольшие участки данных в память, там их сортировать и записывать обратно в память, после чего файл будет уже частично отсортирован и останется довести дело до конца каким-либо методом, предпочитающим частично отсортированные данные.

Если используются структуры данных, отличные организацией доступа от массива, то необходимо выбирать метод сортировки наиболее подходящий для данной структуры. Например, если сортируется связный список, то для него будет эффективен метод вставок. Надо преобразовать соответствующий алгоритм таким образом, чтобы вначале находилось место для вставки очередного элемента, а затем уже вставлять его (в наших примерах на базе массива приходилось многократно переставлять элементы, что на самом деле не эффективно).

Рекомендуется выбирать алгоритм сортировки с учетом предполагаемого входного состояния данных (является таблица частично отсортированной, отсортированной в обратном порядке и т.д.) и рекомендаций приведенных почти к каждому алгоритму.

Если после сортировки надо получить новую таблицу содержащую отсортированные данные (имеется ввиду, что в памяти хранится две таблица: исходная и отсортированная), то неплохие результаты могут дать алгоритмы основанные на выборках элементов.

Рекомендуется использовать сочетания алгоритмов, например, в методах с разделением использовать простые алгоритмы для малых таблиц.

Вопросы для самоконтроля

1. Назовите группы методов сортировки массивов.
2. Каковы критерии оценки алгоритма сортировки?
3. Назовите методы сортировки из каждой группы.
4. Назовите логарифмические и квадратичные методы внутренней сортировки.
5. Какие существуют меры эффективности алгоритмов внутренней сортировки?
6. Сколько сравнений требуют хорошие (логарифмические) методы сортировки? Простые (квадратичные)?
7. Запишите формулу для расчета приращений в методе Шелла.
8. В чем заключается идея метода: быстрой сортировки; обменной поразрядной сортировки?
9. Что означает «слияние» как метод сортировки?
10. Изобразите структуру записи и объясните назначение полей записи для метода «Слияние списков».

ГЛАВА 6
ВНЕШНЯЯ СОРТИРОВКА

6.1. Специфика задачи внешней сортировки

Задача внешней сортировки возникает в том случае, когда необходимо сортировать огромные объемы данных, когда число сортируемых записей превышает объем быстродействующего запоминающего устройства. Внешняя сортировка в корне отлична от внутренней, хотя в обоих случаях необходимо расположить записи данного файла в неубывающем порядке., Объясняется это тем, что время доступа к файлам на внешних носителях гораздо больше времени доступа к оперативной памяти, где хранятся массивы данных в случае внутренней сортировки. Структура данных здесь должна быть такой, чтобы сравнительно медленные периферийные запоминающие устройства могли справиться с потребностями алгоритма сортировки. Поэтому большинство изученных до сих пор алгоритмов внутренней сортировки (сортировки массивов) – из групп вставок, обмена, выбора фактически бесполезно для внешней сортировки, и в данном случае необходимо рассмотреть всю проблему заново [4].

Рассмотрим проблему «от противного». Допустим, в начале сортировки получено M внутренне упорядоченных частей файла размером 64 К и теперь выполняется этап их попарного слияния с размещением новых частей в отдельном файле, который на диске занимает смежное пространство (благоприятный случай). Новые части формируются в памяти порциями размером 64 К, отправляемыми на диск. Так как обращения к исходному и новому файлу чередуются, на этапе будет 2M ходов размером в файл. Затем новый файл становится исходным для следующего этапа слияния; файлы меняются ролями. Таких этапов {log M}, значит всего будет M {log M} больших ходов. Их окажется свыше 2700, если файл занимает 20 Мбайт (более 300 исходных частей), на них затрачивается время, соизмеримое с временем всех остальных действий сортировки. Воистину, случай «противный».

Если исходный и вспомогательный файл разделены большим дисковым пространством, оно и будет определять величину больших ходов, время которых станет доминировать в общих затратах.

Целесообразно сортировку больших файлов выполнять на отдельном логическом диске, свободном от других файлов, например, заново отформатированном.

Если в качестве внешней памяти для хранения данных используются диски, то выражение сложности внешней сортировки учитывает время:

а) внутренней сортировки частей файла;

б) многократного считывания и записи данных на диск;

в) ходов головки между актами считывания/записи;

г) действий в памяти при слиянии упорядоченных частей.

Компонента «а» превалирует, если число внутренне упорядочиваемых частей невелико. В этом случае влияние компоненты «г» незначительно, и в большинстве случаев ее влияние мало.

Рассмотрим некоторые алгоритмы, которые можно применить в случае внешней сортировки.

6.2. Рекурсивный алгоритм сортировки слиянием

Принципиальную возможность эффективно сортировать файл, работая с его частями и не выходя за его пределы, обеспечивает алгоритм Боуза и Нельсона, который применялся лишь к внутренней сортировке. Он основан на очевидной идее: слить две равные упорядоченные части можно слиянием их начальных половин, слиянием конечных и слиянием 2-й половины 1-го результата с 1-й половиной 2-го результата (рис 6.1).

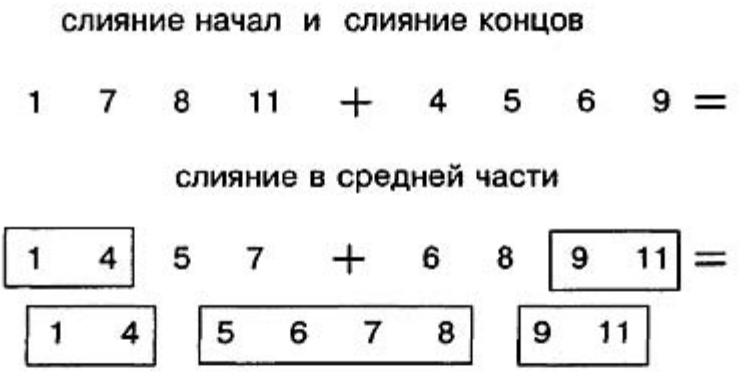


Рис. 6.1. Алгоритм Боуза и Нельсона

В рамки заключены части конечного результата. Если части не равны или не делятся точно пополам, процедуру уточняют надлежащим образом. Аналогично слияние «половинок» можно свести к слиянию «четвертушек»,»восьмушек» и т. д.; имеет место рекурсия.

Эта процедура отвечает нашему стремлению вести основную работу в памяти. Как только в ходе рекурсии получаем части, вписывающиеся в память, выполняем их слияние в памяти, но уже другим, быстрым алгоритмом. Общий эффект слияний всех частей, образующихся в рекурсии, тождественен эффекту слияния исходных частей, сколь бы велики они ни были.

Рассмотрим оценку сложности на примере. Пусть исходные упорядоченные части файла имеют размер RR = 64 К, их число M, а для слияния частей

используются в качестве буферов 3 сегмента памяти размером 64 К*. Удвоение частей (1-й этап слияния) выполняется без рекурсии. На 2-м этапе слияния (получение частей размером в 4 исходных) объем работы и на диске, и в памяти увеличивается в 1,5 раза. На каждом следующем этапе из-за удвоения частей глубина рекурсии возрастает на 1. Если принять сложность 1 этапа за «1», получается прогрессия

$1, 1{,}5, 1{,}5^2, 1{,}5^3, \ldots, 1{,}5^{\{\log M\}-1}$ с суммой $S = 2 \cdot 1{,}5^{\{\log M\}} - 2$.

Время 1 -го этапа пропорционально М и в первом приближении асимптотическая ВС – $0(M^* 1{,}5^{\{\log M\}})$. Испытания при больших значениях Rf (размер файла) и М (Rf = 2 Мб...40 Мб, М = 40...300 частей) подтверждают практически эту зависимость.

В данном анализе не учтены ходы штанги; учесть их весьма непросто, но влияние их можно обнаружить по небольшому расхождению расчетного и практически полученного коэффициента к2 увеличения времени сортировки при удвоении размера Rf файла. Асимптотически к2 стремится к 3.

6.3. Разделительная сортировка

Усовершенствуем метод сортировки разделения. От начала файла или части файла, полученной ранее, будем группировать записи, ключ которых «меньше» выбранного опорного ключа, а начиная от конца – «большие»*. Граница между ними будет новой точкой деления на меньшие 2 части. Части, «вписывающиеся» в оперативную память (терминальные), безотлагательно подвергаются внутренней сортировке, т.е. деление частей имеет предел. Например, дерево, отображающее разделение, может быть таким:

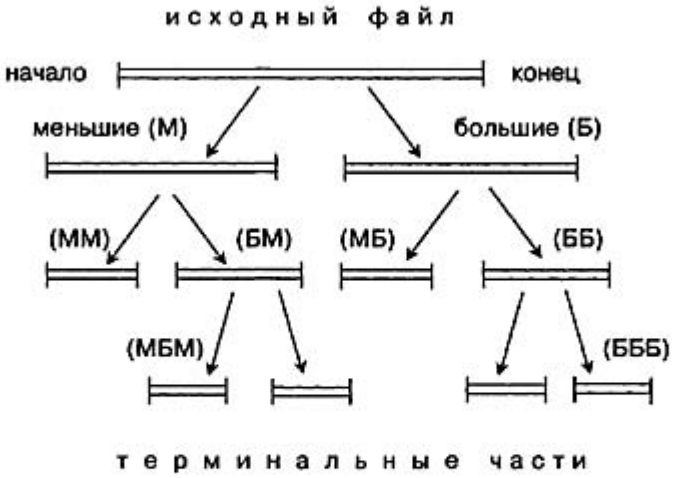


Рис. 6.2. Разделительная сортировка

ММ означает «меньшие из меньших», БМ – «большие из меньших» и т. д. Части «ММ», «МБ» «вписываются» в память (в данном примере), поэтому и не разделяются на меньшие части.

Процесс разделения не требует дополнительного пространства и весьма эффективно использует всю предоставленную память, которая выполняет роль «парома» при перемещении записей от одного конца файла («берега») к другому [2].

В нашем распоряжении к сегментов памяти; к > 1. Заполняем их данными, считывая начало файла или разделяемой части файла и начинаем – в памяти – разделение на «большие» и «меньшие» записи. Для накопления «меньших» используем 1-й сегмент, для «больших» – все остальные. Дело в том, что мы пока у того «берега», где надо группировать «меньшие». Дождавшись, когда в 1-м сегменте останутся одни «меньшие» записи, мы разгружаем его в файл, а из файла передаем в сегмент следующую порцию записей и т. д. Тем временем идет вытеснение «меньших» записей из прочих сегментов, т.е. в них остаются «большие». «Паром» приготовлен к перемещению этих записей на другой «берег»: переходим к концу файла (части файла). Если файл большой, штанга с головками совершает большой ход.

Один сегмент, например 1-й, должен быть свободен, иначе не «разгрузить паром»: сначала освобождаем окончание файла или части файла, передавая его записи в 1-й сегмент. Теперь, сегмент за сегментом, переписываем «большие» записи в файл, параллельно заполняя сегменты новыми записями. Снова начинаем разделение в пространстве к сегментов – на «большие» и «меньшие» записи, но теперь лишь один сегмент нужен для накопления «больших» (потому что сразу передаем их в файл и берем в него из файла следующую порцию записей), а все прочие постепенно заполняются «меньшими», чтобы быть отправленными «на другой берег», где в файле ранее образовалось свободное место для их размещения. Итак, произойдет 2-й большой ход штанги.

Подобные вышерассмотренным акты сортировки и перемещений записей выполняются вновь и вновь, пока все «меньшие» и все «большие» записи не будут сгруппированы и найдена точка раздела между группами.

Оценке метода «Разделительная сортировка». Влияние ходов штанги на время сортировки можно свести, к минимуму. Например, если при разделении на 2 части файла размером 64 Мбайт воспользоваться шестью сегментами памяти по 64 Кбайт каждый, то при случайном наборе значений ключей ходов не будет слишком много: пока готовятся к «отправке» 5 сегментов «больших» записей, примерно столько же выявляется и «меньших», т.е. нерассмотренная часть файла сокращается примерно на 10 сегментов. То же происходит и на другом конце файла, следовательно, ходов окажется около сотни, а средняя их длина определяется размером половины файла (она немного меньше). Нетрудно записать коротенькую программу, выполняющую эти ходы, и убедиться, что они займут считанные секунды. В процессе дальнейшего разделения частей общая длина ходов будет примерно такой же, т.е. время лишь удвоится. Поэто-

му на первый план выступает задержка обращения к данным на диске, которую мы уменьшаем, считывая с диска большие порции данных [2].

Помимо высокого быстродействия достоинством метода является работа в пределах пространства, занятого файлом, т.е. метод предельно компактен.

Комбинированный вариант. Из-за произвольного размера файла и, отчасти, неравномерного деления частей размер терминальных частей произволен – в пределах предоставленной памяти. Неполно используется «быстрый» этап внутренней сортировки. Предлагается «поднять планку» для терминальных частей до уровня 4 размеров предоставленной памяти и завершать сортировку методом поглощения, в котором внутренняя сортировка используется по максимуму.

6.4. Сортировка методом поглощения

Имея M частей размера Z и начав со слияния двух из них, будем сливать все следующие с большей (растущей) упорядоченной частью. Она как бы поглощает часть за частью. Высота дерева слияния – M – 1. Это дает экономию времени и при небольшом M сортировка проходит быстро. Упорядочивание каждой исходной части производят непосредственно перед ее поглощением. В начале сортировки лишь окончание файла считывается и упорядочивается в памяти, возвращается на место. Схема поглощения частей дана ниже; номера соответствуют очередности поглощения частей:

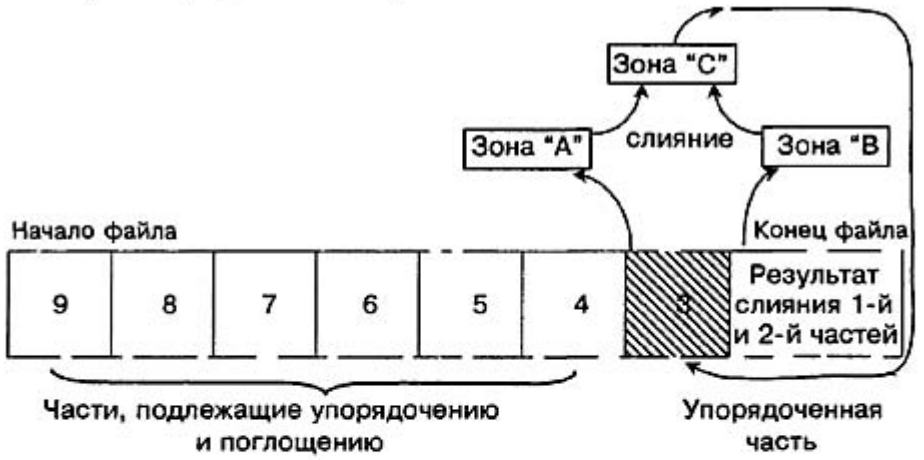


Рис. 6.3. Сортировка методом поглощения

Перед поглощением очередная часть файла считывается в зону «A» памяти, там упорядочивается и остается. Начало ранее упорядоченной части считывается в зону «B» и начинается слияние, прерываемое считыванием в зону «B», когда она опустошается. По мере заполнения зоны «C» записями акта слияния, содержимое

ее переписывается в файл (на место поглощаемой части и далее в сторону конца файла). Если при слиянии взяты все записи поглощаемой части, поглощение завершается передачей в файл из зоны «С» остатка результата. Слияние также завершается, если исчерпана ранее упорядоченная часть. Поглощение ею очередной части произошло.

Чтобы произвести следующее, выполняется ход от конца файла до начала ближайшей неупорядоченной части. Таких ходов M – 1, т.е. относительно немного. Нетрудно понять, что при поглощении части разница адреса считывания и адреса записи вначале равна размеру части Z, а в конце процесса эти адреса почти совпадают. Поэтому при слиянии частей ходы невелики, и мы ими пренебрегаем.

Итак, на каждом из M – 1 этапов поглощения выполняется прогон сегмента файла, начиная от сегмента размером 2Z и кончая размером M • Z всего файла, а в среднем размер сегмента M • Z/2, если учесть отправной сегмент Z. Этот размер и определяет среднюю сложность этапа. Таким образом, сложность сортировки $0(M^2)$. Эксперимент показывает, что оценка несколько занижена.

6.5. Челночное балансное слияние

Методы сортировки слияния и поглощения имеют низкую производительность при больших файлах (когда M велико). Рассмотрим еще один метод.

Челночное слияние. На 1 этапе внутренней сортировки частей в файле создают M упорядоченных частей, по возможности большего размера R. К ним применяют прямое двоичное слияние. Если наложить условие $M = 2^k • 3$ (k – целое), то после некоторого числа слияний в файле останутся лишь 3 примерно равные упорядоченные части.

Отличие от общепринятого метода сортировки состоит в том, что резервное дисковое пространство файла располагается непосредственно до или после сливаемых частей и перемещается к следующим частям по завершении слияния. Его размер не меньше размера R части. Резерв и пространство ближайшей части будут заполнено результатом слияния двух первых частей, при этом пространство 2-й части освободится и станет резервом, необходимым для слияния следующей пары частей и т. д. [2]. По мере слияния частей резерв перемещается от начала файла к концу, потом обратно и т. д., как челнок. Поскольку место результата не удалено от сливаемых частей, ходы невелики, пока сами части небольшие. В процессе сортировки «челнок» увеличивают, ибо растут части:

Рис. 6.4. Слияние частей размером R (промежуточная ситуация)

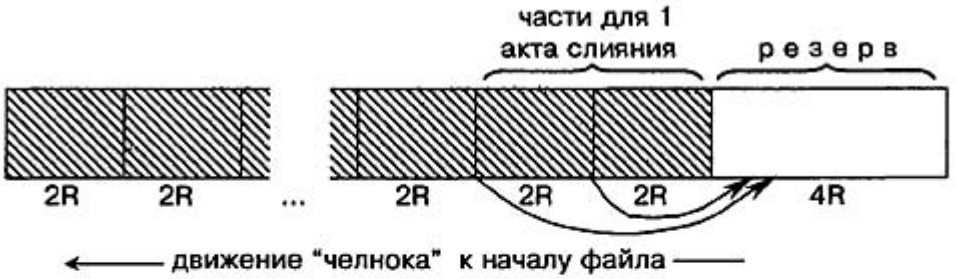


Рис. 6.5. Следующий «прогон» челнока слияния частей размером 2R.

Стрелками показано перемещение данных при слиянии.

Заметим, что резерв можно увеличивать, когда он в конце файла; это происходит один раз за 2 прогона челнока. Хорошо бы ограничить рост размера «челнока», ибо этот рост увеличивает размер ходов. Путем модификации конечных этапов слияния добьемся, чтобы этот рост остановился на размере D = 1/6 размера файла. Для этого, однако, надо, чтобы программа так определила исходные размер и положение резерва (в начале или в конце файла), чтобы в момент, когда останутся всего 3 больших части, резерв, имеющий размер D, стоял в начале файла.

Схемы модифицированных этапов (рис. 6.5, 6.6., 6.7) показаны ниже/

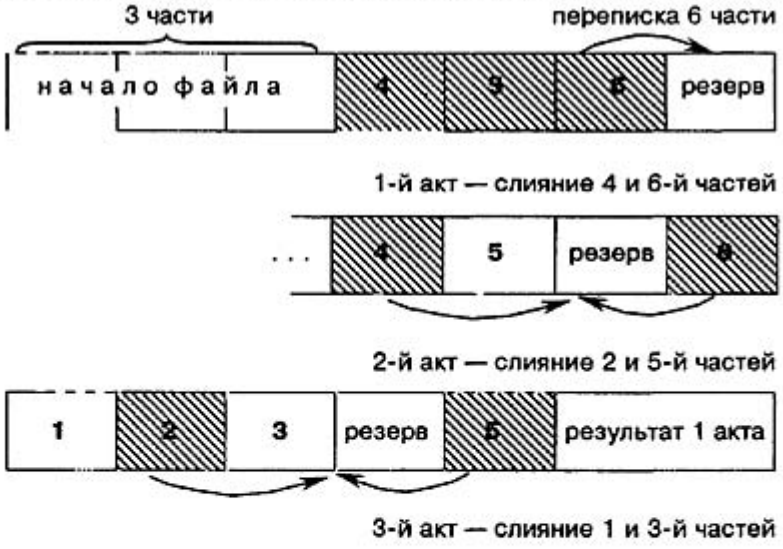


Рис. 6.5. Этап слияния (с подэтапами), когда в файле 6 частей

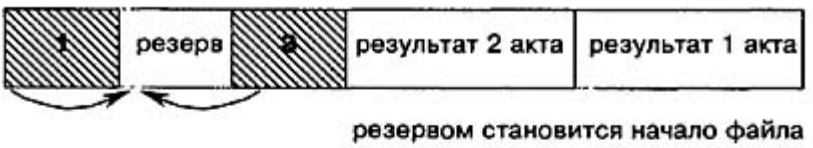


Рис. 6.6. Этап слияния «по половинам», когда частей 3. К
онцевая часть файла на этом этапе не используется

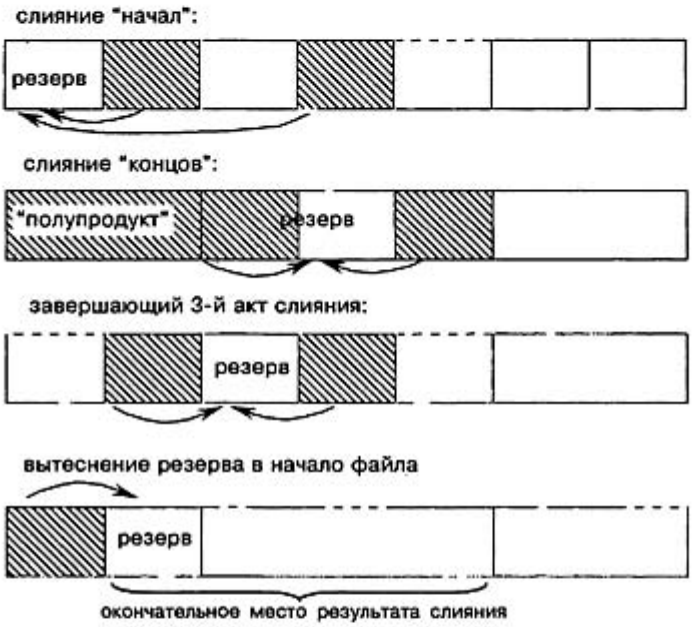


Рис. 6.7. Заключительный этап слияния

Сначала берем для слияния первую половину конечной части и всю начальную часть:

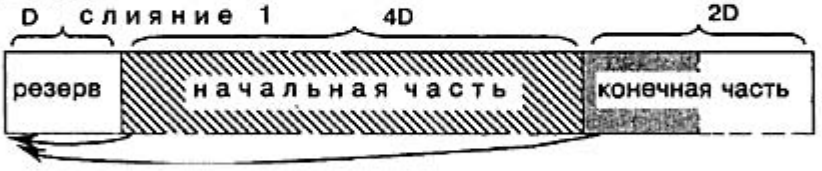


Рис. 6.8. Окончание слияния

По окончании слияния 1 выясняется, надо ли сместить остаток начальной части, чтобы освободить место для конца результата и продолжить слияние

(«подслияние»). Подслияние не нужно, если в ходе слияния начальная часть исчерпана, надо лишь вытеснить резерв в конец файла, переписав окончание.

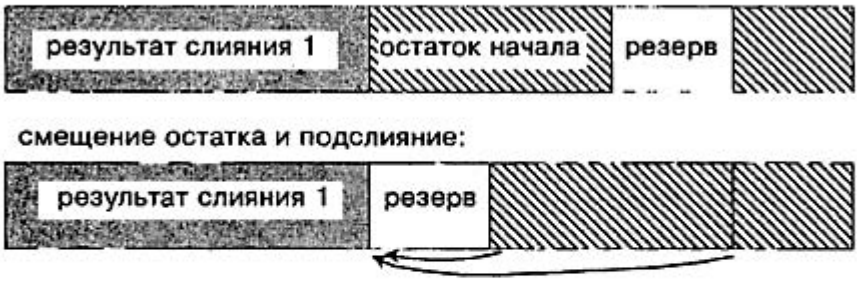


Рис. 6.9. Смещение остатка и подслияние

И при слиянии 1, и при подслиянии размер резерва не меньше минимально требуемого.

Итак, удалось локализовать процесс сортировки в пространстве диска, перекрывающем файл и выходящем за рамки файла примерно на 1/6 размера файла. Сама по себе экономия пространства значима при сортировке очень больших файлов; важно другое: средний размер ходов на последнем этапе слияния сокращается вдвое, по сравнению с известным благоприятным вариантом обычного балансного слияния, когда резерв величиной в половину файла занимает его начало.

Оценки вариантов компактного слияния

Без учета ходов штанги балансное слияние с оптимальным оператором слияния частей имеет оценку сложности 0(M • Z • log M), где M – число исходных упорядоченных частей размером Z; при неизменном Z, если увеличить вдвое размер файла, величина M • Z • log M (и время сортировки) возрастет немногим более, чем в 2 раза. Обозначим этот коэффициент к2. В действительности увеличение времени заметно зависит от размера и числа ходов.

Эксперимент показывает, что к2 ≈ 2.5, тогда как в алгоритме с рекурсивным оператором слияния (п. 6.2) к2 ≈ 3. Для сортировки методом поглощения (п. 6.3) эксперимент подтверждает квадратичную зависимость t от размера файла (к2 ≈ 4). Однако простая программа и хорошая производительность при малом числе M частей не позволяют сдать этот метод в архив, тем более, что он не требует резерва на диске.

Таблица 6.1 значений t (время – в секундах), полученная на ЭВМ с процессором Pentium 166 MMX, выделяет челночную сортировку как объект дальнейшего совершенствования, если задачей является сортировка больших файлов:

Таблица 6.1

Метод \ Число записей	21 500	43 000	86 000	172 000	344 000
Челночный	6.7	19.8	55.3	177.4	462
С рекурсией	12.2	40.6	126.8	375.0	1101
Поглощение	6.2	31.6	128.3	562.8	2399

Размер записи – 116 байт, поэтому файл с 344000 записями занимает почти 40 Мбайт.

Главной проблемой является растущая доля затрат времени на ходы. Убедимся в этом, оценив, хотя бы и приближенно, влияние ходов на длительность последнего этапа слияния в большом файле. Резерв R положим равным Rf/2, где Rf – размер файла.

Размер B буферов чтения и буфера вывода примем одинаковым. Величину хода будем оценивать разностью адресов позиций, между которыми перемещается головка. Будем считать, что и малые, и большие, и средние значения ключа более или менее равномерно попали в каждую сливаемую часть. В этом случае считывание порций данных попеременно происходит из одной и из другой части. Чтения перемежаются с актами вывода в файл данных из буфера результата. Акты чтения в буферы и акты вывода могут чередоваться по одному и по два. Моделированием установлено, что ситуация, когда между 2 актами чтения нет акта вывода, имеет место примерно в 45% случаев. Всего актов чтения – {Rf/B}, следовательно, ходов типа «чтение-чтение» – 45/145 {Rf/B} = 9/29 {Rf/B}, а ходов типа «чтение-запись» (запись-чтение») – 100/145 {Rf/B} = 20/29 {Rf/B}. Всего ходов «чтение-запись», «запись-чтение» 40/29 {Rf/B}; их размер для 1-й части равен вначале R, в конце слияния – 0, а для 2-й части – соответственно Rf/2 + R = 2R и Rf/2 = R.

Усредняя по двум частям и во времени, получаем средний размер хода, равный R. При равномерном считывании данных размер хода «чтение-чтение» остается примерно равным Rf/2, т.е. R, в течение всего акта слияния. Ходы типа «запись-запись» – нулевого размера. Следовательно, среднее для суммы ходов

$$\frac{40+9}{29} * \{Rf / B\} * Rf / 2 \approx \frac{49}{58} Rf^2 / B \qquad (6.1)$$

Этот итог согласуется с результатами моделирования, но характер зависимости очевиден и без выкладок: не вызывает сомнений пропорциональность размера больших ходов и размера файла и то, что число ходов определяется отношением Rf/B. Итак, большими будем называть файлы, для которых Rf на 2 порядка больше B, а гигантскими – те, для которых разница порядков еще больше. Именно в них ходы определяют время сортировки t, ибо квадратичный член доминирует в оценке временной сложности. Положительным является то, что из-за нелинейной зависимости времени движения головок через к цилиндров (когда к мало, движение медленнее) кривая t = f(Rf) не может стать точной

параболой. Поэтому коэффициент к2 приблизится не к 4, а к меньшему значению.

Приходится увеличивать B, т.е. используемую память, переводя гигантские файлы в разряд больших. Например, файл размером 40 Мбайт считаем просто большим. Если пользователь часто обрабатывает большие файлы, нужна конфигурация ЭВМ с большой памятью.

Мы исследовали лишь последний этап слияния. Упрощая, можно считать, что суммы ходов на последнем этапе и на всех этапах, кроме последнего, равны. Столь малый их вклад – достоинство челночного слияния. Уменьшим сумму ходов.

Новый экономный оператор слияния частей. Обычно полагают, что минимум дополнительного пространства для нерекурсивного акта слияния равен размеру меньшей из сливаемых частей (разумеется, части могут быть и равны), но это не так. Вначале выполняем слияние половины меньшей части с большей:

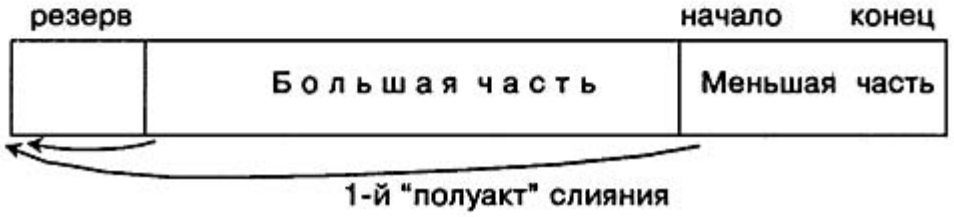


Рис. 6.10. Слияние половины меньшей части с большей

Результат записываем, начиная с резерва и занимая некоторое число позиций большей части. Оно может оказаться любым, даже нулем. Теперь освободилось начало меньшей части – там и резерв для продолжения слияния. 2-й «полутакт» реализует слияние второй половины меньшей части с неучаствовавшими в 1-м «полутакте» записями большей части, но при обратном порядке рассмотрения – от конца. Заполнение переместившегося резерва начинается с конца, ближайшего к остатку меньшей части, поэтому, как и ранее, записям большей части преждевременное «затирание» не грозит. Слова «начало», «конец» здесь условны: резерв может занимать как меньшие, так и большие адреса.

Рассмотрим конкретный пример (рис 6.11):

Анализ последнего этапа слияния дает средний размер хода «чтение запись», равный 5/16 Rf вместо Rf/2, а сумма ходов уменьшается до 34/58 RP/B, т.е. примерно в 1.5 раза.

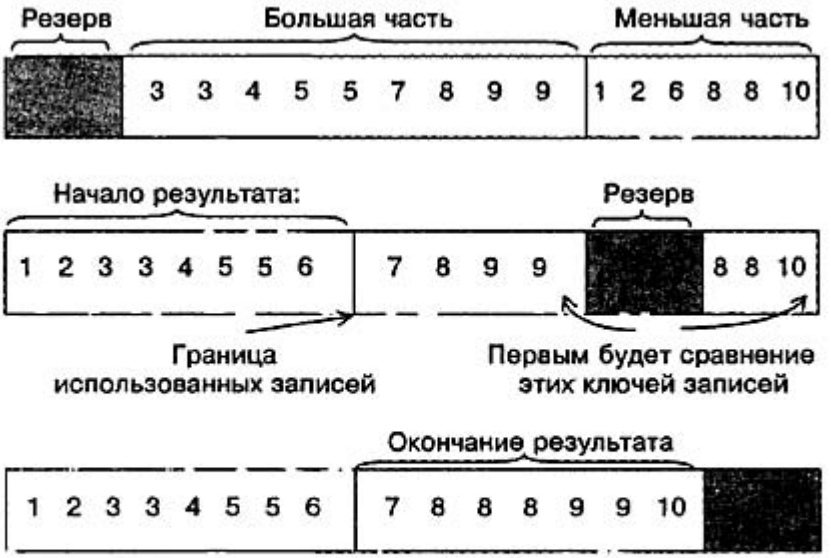


Рис. 6.11. Пример челночно-балансного слияния

6.6. Метод многопутевого челночного слияния

Слияние в одном акте не двух, а нескольких упорядоченных частей позволяет за один прогон файла увеличить размер частей в несколько раз. Соответственно сокращается число прогонов, а следовательно, и время. Число сливаемых в одном акте частей называют числом путей слияния kp.

Например, один прогон файла при 8-путевом слиянии заменяет три прогона в 2-путевом слиянии. Почему бы не слить все M исходных частей в одном акте M-путевого слияния, при одном прогоне файла? Алгоритмические проблемы, как увидим далее, разрешимы. Однако потребуется разместить в памяти M буферов ввода; при большом M они будут невелики, а считывание и запись на диск малыми порциями – неэффективны, к тому же число ходов штанги будет значительным.

Ситуация такова, что с увеличением kp факторы, замедляющие сортировку, в некоторый момент становятся «сильнее» ускоряющего фактора. Ряд исследователей полагают значение kp = 4 наиболее подходящим в большинстве случаев. В новых реализациях сортировки на диске рекомендуемое значение kp должно оказаться большим.

Универсальный оператор многопутевого слияния. Типичной является жесткая, настроенная на конкретное значение kp реализация. Поскольку в акте слияния одни части исчерпаются ранее других, требуются стадии kp- , (kp-1)-, (kp – 2)-,..., 2-путевого слияния. Чтобы избежать запутанного алгоритма и применить жесткую схему, делают представителями исчерпанных частей иллюзор-

ные записи с максимальным (нереальным) значением ключа. Они участвуют в сравнениях, но никогда не будут выбраны. Имеется лишь одна стадия – стадия kp-путевого слияния.

Откажемся от жесткой схемы. Применим дерево сортировки – оглавление, ссылающееся на текущие (сравниваемые) записи сливаемых частей. Первый элемент оглавления будет всегда указывать очередную выигрывающую запись (минимальную из текущих), направляемую в буфер вывода. Ее заменой в оглавлении станет ссылка на следующую запись части; затем выполняется пересылка этой ссылки, чтобы 1-е место заняла ссылка на новую минимальную запись. Всякий раз по исчерпании одной из сливаемых частей оглавление-дерево укорачивается на один элемент. Оглавление из одного элемента – это ссылка на окончание единственной неисчерпанной части, которое в завершение акта слияния направляется в буфер вывода.

Челночный вариант. Предлагаем здесь реализацию – обобщение двухпутевой челночной сортировки. Резерв, если он расположен в файле от его начала, имеет размер (kp ≈ 1) z, где z – размер сливаемых частей:

Рис. 6.12. Сортировка двухпутевого челночного слияния

Результат слияния занимает резерв и перекрывает ближайшую к резерву часть. Освобождаясь, прочие (kp – 1) частей образуют резерв, необходимый для слияния следующих kp частей, если они еще есть. Резерв продвигается к концу файла. На следующем прогоне – движется в обратную сторону.

Показанное выше расположение резерва обязательно для заключительного этапа слияния, когда резерв должен быть вытеснен в конец файла и отсечен с помощью процедуры Truncate. Как видно из рисунка, размер Rf файла практически удваивается к концу сортировки. Итак, kp-путевое слияние недостаточно компактно. На предпоследнем прогоне файла размер резерва такой же, как и на последнем, и два последних прогона дают наибольшую сумму ходов.

Вопросы для самоконтроля

1. Почему алгоритмы внутренней сортировки неприменимы в случае внешней сортировки?
2. Назовите составляющие времени сортировки данных на диске.
3. Назовите алгоритмы внешней сортировки.
4. Как часто увеличивается «резерв» в методе челнока?
5. В чем заключается идея сортировки метода Боуза-Нельсона?

ГЛАВА 7
ПОИСК

7.1. Постановка задачи поиска

Эта глава посвящена изучению задачи: как находить данные, хранящиеся с определенной идентификацией. Нас будет интересовать процесс накопления информации в памяти вычислительной машины с последующим возможно более быстрым извлечением этой информации. Будем предполагать, что хранится множество из N записей и необходимо определить положение соответствующей записи. Найти – означает определить номер (позицию) искомой записи. Как и в случае сортировки, предположим, что каждая запись содержит специальное поле, которое называется *ключом*. Обязательное требование здесь, чтобы эти N ключей были различны. Тогда каждый ключ будет однозначно определять свою запись. Совокупность всех записей называется *таблицей* или *файлом*. Большой файл или группа файлов представляют *базу данных*.

В алгоритмах поиска присутствует алгоритм поиска K. Задача состоит в отыскании записи, имеющей K своим ключом. Существует две возможности окончания поиска: поиск оказался *удачным*, т.е. позволил определить положение соответствующей записи, содержащей K, или он оказался *неудачным* т.е. показал, что аргумент K не может быть найден ни в одной из записей.

Методы поиска можно условно классифицировать следующим образом:

1. Последовательные методы поиска
2. Поиск в упорядоченных таблицах
3. Поиск на деревьях в основной памяти
4. Поиск на деревьях во внешней памяти
5. Хеширование

7.2. Последовательный поиск

Это наиболее простой и очевидный способ отыскания записи по заданному ключу K. Он состоит в последовательном просмотре всех записей и сравнении их ключей с заданным значением ключа. Последовательность записей R_1, R_2, . . . , R_n снабжены ключами K_1, K_2, . . . , K_n. Необходимо найти запись с заданным ключом K. На рис.7.1 приведена схема алгоритма «Последовательный поиск», на рис.7.2 – схема алгоритма «Быстрый последовательный поиск». Ускоряющий принцип во втором алгоритме – только одно сравнение во внутреннем цикле.

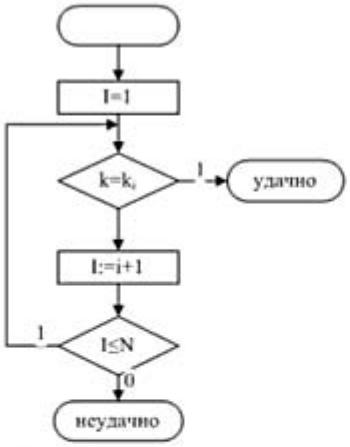


Рис. 7.1. Схема алгоритма «Последовательный поиск»

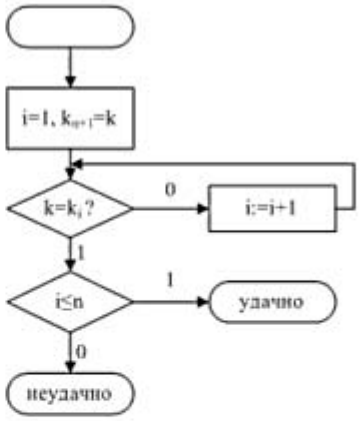


Рис. 7.2. Схема алгоритма «Быстрый последовательный поиск»

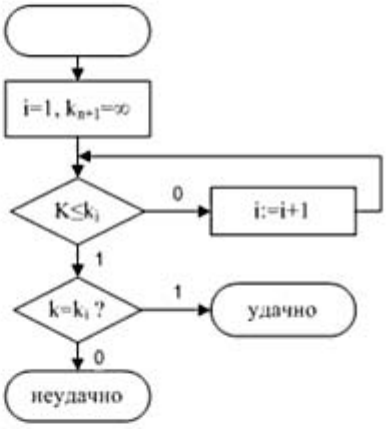


Рис. 7.3. Схема алгоритма «Последовательный поиск в упорядоченной таблице»

Существует способ сделать этот алгоритм поиска еще быстрее. Если известно, что исходная последовательность расположена по возрастанию ключей, то алгоритм поиска можно сделать еще более эффективным. Для этого нужно отсортировать записи, например, по возрастанию ключей. На рис.7.3 представлена схема алгоритма "Последовательный поиск в упорядоченной таблице". Здесь отсутствие нужной записи обнаруживается примерно в два раза быстрее.

7.3. Поиск в упорядоченной таблице

7.3.1. Блочный поиск

Этот метод называется также «Последовательный поиск с пропусками». Он работает на упорядоченных таблицах (файлах).

Идея этого метода заключается в следующем. Вместо того чтобы просматривать весь файл (или таблицу), можно разбить его на блоки и вести поиск по блокам. Поиск начинается с блока, содержащего элементы с наименьшими значениями. Проверяется наибольшее значение элемента в этом блоке. Если это значение меньше требуемого, переходим к следующему блоку.

Размер блока, при котором достигается минимальное число проверок, равен $\sqrt{Np}$, где N_p – число элементов во всем файле.

Пример. Ищем запись с ключом 27: K = 27

Рис. 7.4. Блочный поиск, поиск записи с K = 27

7.3.2. Бинарный поиск

Совершенно очевидно, что других способов убыстрения поиска не существует, если, конечно, нет еще какой-либо информации о данных, среди которых идет поиск. Хорошо известно, что поиск можно сделать значительно более эффективным, если данные будут упорядочены. Поэтому приведем алгоритм, основанный на знании того, что массив A упорядочен, т.е. удовлетворяет условию: $a_{k-1} \le a_k$, где $1 \le k < N$.

Основная идея – выбрать случайно некоторый элемент, предположим a_m, и сравнить его с аргументом поиска x. Если он равен x, то поиск заканчивается, если он меньше x, то делается вывод, что все элементы с индексами, меньшими или равными m, можно исключить из дальнейшего поиска; если же он больше x, то исключаются индексы больше и равные m. Выбор m совершенно не влияет на корректность алгоритма, но влияет на его эффективность. Очевидно, что чем большее количество элементов исключается на каждом шаге алгоритма, тем этот алгоритм эффективнее. Оптимальным решением будет выбор среднего элемента, так как при этом в любом случае будет исключаться половина массива.

С помощью этого алгоритма разыскивается аргумент K в таблице записей $R_1, R_2, \ldots, R_n$, ключи которых расположены в возрастающем порядке ($K_1, K_2, \ldots, K_n$). Идея бинарного поиска заключается в следующем. Сначала нужно срав-

нить К со средним ключом в таблице. Результат сравнения позволит определить, в какой половине файла продолжать поиск, применяя к ней ту же процедуру, и т.д.

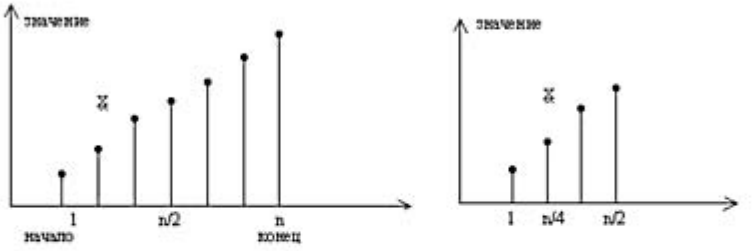


Рис. 7.5. Поиск делением пополам

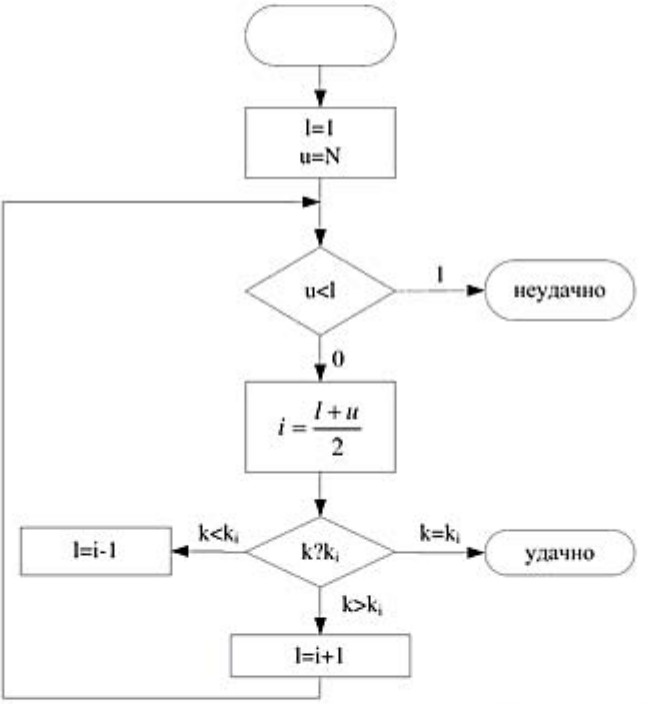


Рис. 7.6. Бинарный поиск в упорядоченной таблице

После не более чем $\log_{2N}$ сравнений ключ либо будет найден, либо будет установлено его отсутствие. Этот метод поиска также называют «Двоичный поиск», «Логарифмическим поиск» или «Метод деления пополам», но наиболее употребительный термин – «Бинарный поиск».

Одна из наиболее популярных реализаций метода использует два указателя: l и u, соответствующие верхней и нижней границам поиска. На рис. 7.6 приведена схема алгоритма бинарного поиска.

Среднее число проверок при достаточно большом N_1 равно примерно $\log_2 N_1 - 1$. Это значительно меньше, чем при последовательном просмотре или при блочном поиске.

Бинарный поиск и пример использования:

```
#include <iostream>
using namespace std;

template <class T>
int search (T *array, int size, T key)
{
    int first = 0, last = size, mid;
    if (array[0] > key)
            return -1;
    else
       {
               if (array[size - 1] < key)
                 return -1;
       }
    while (first < last)
    {
      mid = first + (last - first) / 2;
      if (key <= array[mid])
                last = mid;
      else
                first = mid + 1;
    }

    if (array[last] == key)
            return last;
    else
            return -1;
}

int main()
{
  int array[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
  int x = search(array, 10, 7);

     if (x != -1)
          cout<<«Индекс элемента «<< x <<endl;
  else
          cout<<«Данный элемент не найден»<<endl;

     system(«PAUSE»);
  return 0;
}
```

7.3.3. Поиск Фибоначчи

Этот метод поиска основан на использовании чисел Фибоначчи, которые используются для построения бинарного дерева.

Алгоритм построения дерева поиска Фибоначчи:

1. Если k=0 или k=1, дерево сводится к 0.
2. Если k>=2, корнем является F_k ; левое поддерево есть дерево Фибоначчи порядка k-1; правое поддерево есть дерево Фибоначчи порядка k-2 с числами в узлах, увеличенными на F_k.

Замечание:

Желательно, чтобы число ключей в таблице N удовлетворяло условию: $N<F_{k+1}-1$.

На рис.7.7 изображено дерево поиска Фибоначчи для N=12 и K=6. Здесь K – порядковый номер числа Фибоначчи (порядок дерева Фибоначчи).

k123456 7 8 9 1011

F_k 01123581321345569

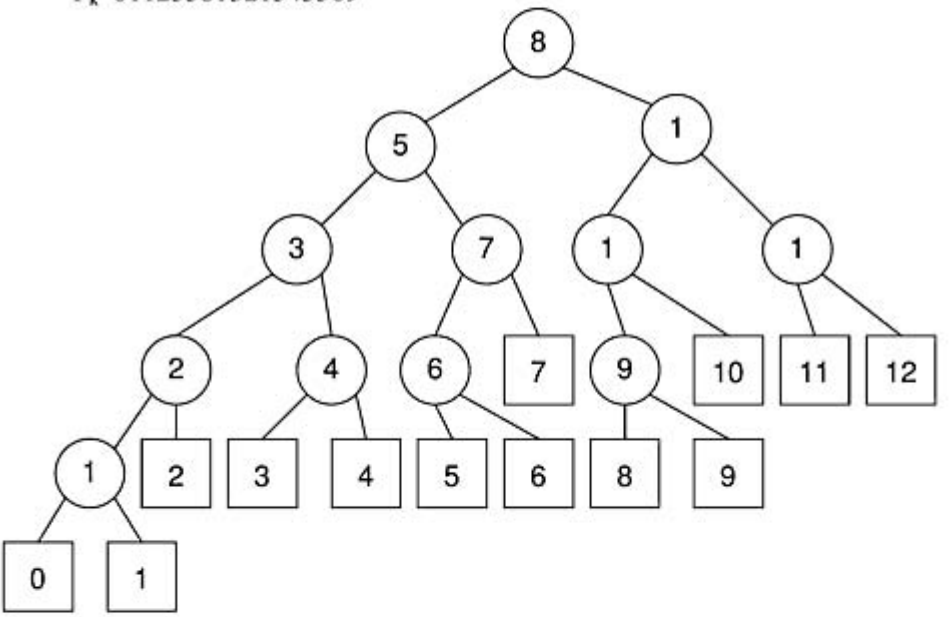


Рис. 7.7. Дерево поиска Фибоначчи порядка 6

Алгоритм поиска Фибоначчи

Алгоритм представляется для поиска аргумента K в таблице записей $R_1, R_2, \ldots, R_n$, расположенных в порядке возрастания ключей $K_1, K_2, \ldots, K_n$.

Предлагается, что N+1 есть число Фибоначчи F_{k+1}. Подходящей начальной установкой данный метод можно сделать пригодным для любого N. В алгоритме переменные p и q – последовательные числа Фибоначчи.

Схема алгоритма поиска Фибоначчи приведена на рис. 7.8.

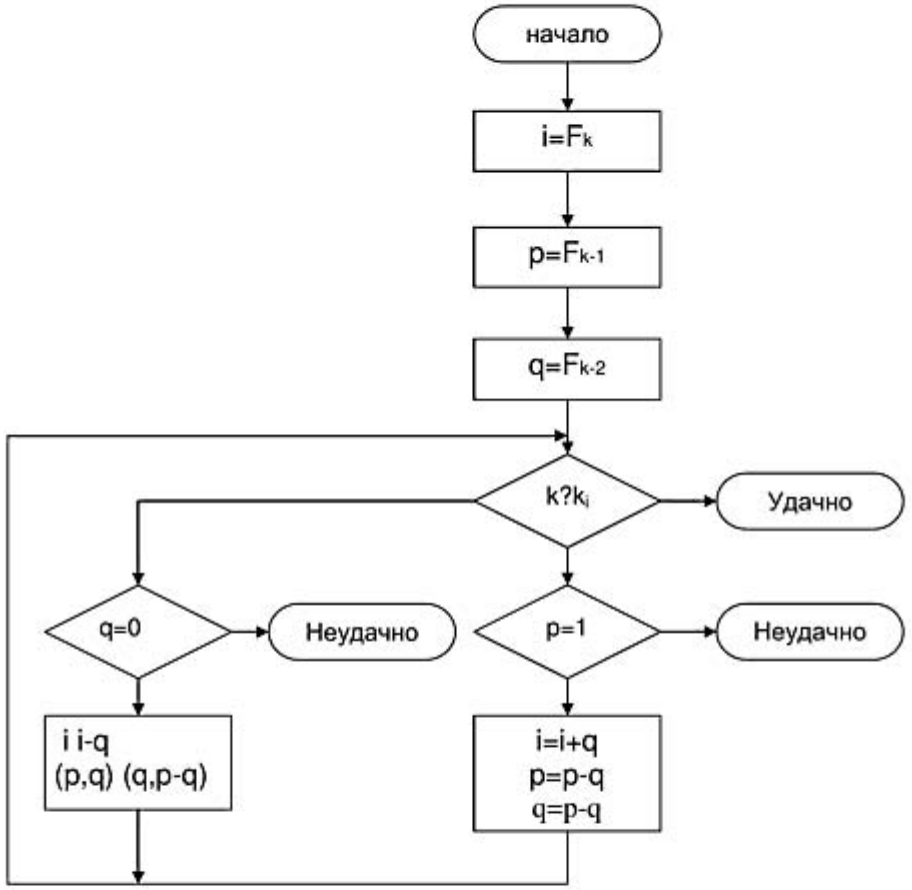


Рис. 7.8. Поиск Фибоначчи

7.3.4. Интерполяционный поиск элемента в массиве

Забудем на минуту о вычислительных машинах и проанализируем, как производит поиск человек. Иногда повседневная жизнь подсказывает путь к созданию хороших алгоритмов. Представьте, что вы ищете слово в словаре. Маловероятно, что вы сначала заглянете в середину словаря, затем отступите от начала на 1/4 или 3/4 и т. д., как в бинарном поиске, и уж совсем невероятно, что вы воспользуетесь фибоначчиевым поиском!

Если нужное слово начинается с буквы А, вы, по-видимому, начнете поиск где-то в начале словаря. Во многих словарях имеются "побуквенные высечки" для большого пальца, которые показывают страницу, где начинаются слова на данную букву. Такую пальцевую технику легко приспособить к ЭВМ, что ускорит поиск.

Когда найдена отправная точка для поиска, ваши дальнейшие действия мало похожи на рассмотренные методы. Если вы заметите, что нужное слово должно находиться гораздо дальше открытой страницы, вы пропустите порядочное их количество, прежде чем сделать следующую попытку. Это в корне отличается от предыдущих алгоритмов, которые не делают различия между «много больше» и «чуть больше». Мы приходим к алгоритму, называемому интерполяционным поиском: Если известно, что К лежит между K_l и K_u, то следующую пробу делаем на расстоянии

$$(u-l)(K-K_l)/(K_u-K_l)$$

от l, предполагая, что ключи являются числами, возрастающими приблизительно в арифметической прогрессии.

Интерполяционный поиск и пример использования:

```
#include <iostream>
using namespace std;

template <class T>
int interpolatingSearch (T *array, int arraySize, T keyOfSearch)
{
     int low = 0;
     int high = arraySize - 1;
     int mid;

     while (array[low] < keyOfSearch && array[high] >= keyOfSearch)
     {
          mid = low + ((keyOfSearch - array[low]) * (high - low)) /
(array[high] - array[low]);
          if (array[mid] < keyOfSearch)
               low = mid + 1;
          else if
               (array[mid] > keyOfSearch)
          high = mid - 1;
          else
               return mid;
     }

     if (array[low] == keyOfSearch)
          return low;
     else
          return -1;
}
```

```
int main()
{
  int array[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
  int x = interpolatingSearch(array, 10, 4);

     if (x != -1)
          cout<<«Индекс элемента «<< x <<endl;
  else
          cout<<«Данный элемент не найден»<<endl;

     system(«PAUSE»);
  return 0;
}
```

Интерполяционный поиск работает за $\log(\log_2 N)$ операций, если данные распределены равномерно. Как правило, он используется лишь на очень больших таблицах, причем делается несколько шагов интерполяционного поиска, а затем на малом подмассиве используется бинарный или последовательный варианты.

7.4. Поиск по деревьям в основной памяти

7.4.1. Поиск по бинарному дереву

Использование неявной структуры бинарного дерева облегчает понимание бинарного и фибоначчиева поисков. Рассмотрение соответствующих деревьев позволило заключить, что при данном N среди всех методов поиска путем сравнения ключей бинарный поиск совершает минимальное, число сравнений. Но методы предыдущего пункта предназначены главным образом для таблиц фиксированного размера, так как последовательное расположение записей делает вставки и удаления, довольно трудоемкими. Если таблица динамически изменяется, то экономия от использования бинарного поиска не покроет затрат на поддержание упорядоченного расположения ключей.

Явное использование структуры бинарного дерева позволяет быстро вставлять и удалять записи и производить эффективный поиск по таблице. В результате мы получаем метод, полезный как для поиска, так и для сортировки. Такая гибкость достигается путем добавления в каждую запись двух полей для хранения ссылок [5].

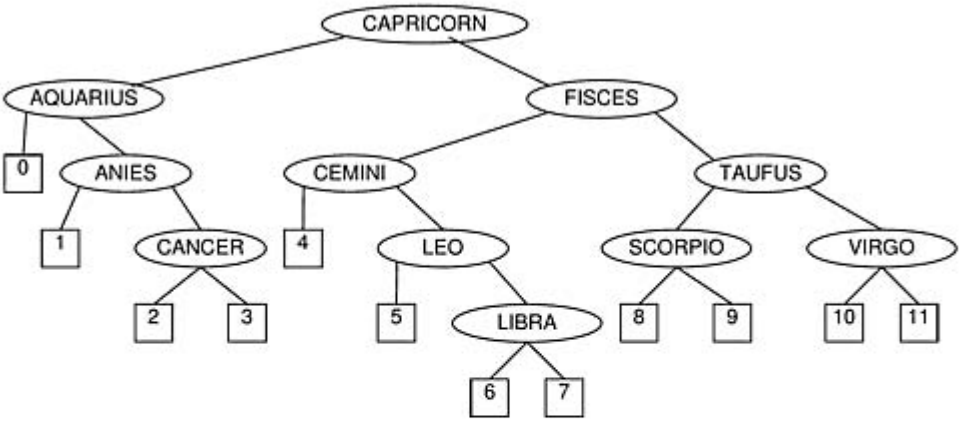


Рис. 7.9. Бинарное дерево поиска

Методы поиска по растущим таблицам часто называют алгоритмами таблиц символов, так как ассемблеры, компиляторы и другие системные программы обычно используют их для хранения определяемых пользователем символов. Например, ключом записи в компиляторе может служить символический идентификатор, обозначающий переменную в некоторой программе на Фортране или Алголе, а остальные поля записи могут содержать информацию о типе переменной и ее расположении в памяти. Или же ключом может быть символ программы для MIX, а оставшаяся часть записи может содержать эквивалент этого символа. Программы поиска с вставкой по дереву, которые будут описаны ниже, отлично подходят для использования в качестве алгоритмов таблиц символов, особенно если желательно распечатывать символы в алфавитном порядке. На рис. 7.9 изображено бинарное дерево поиска, содержащее названия одиннадцати знаков зодиака. Если теперь, отправляясь от корня дерева, мы будем искать двенадцатое название, SAGITTARIUS,то увидим. что оно больше, чем CAPRICORN, поэтому нужно идти вправо; оно больше, чем PISCES,–снова идем вправо; оно меньше, чем TAURUS – идем влево; оно меньше, чем SCORPIO – мы попадаем во внешний узел – поиск был неудачным; теперь по окончании поиска мы можем вставить SAGITTARIUS, «подвязывая» его к дереву вместо внешнего узла. Таким образом, таблица может расти без перемещения существующих записей. Рисунок 7.9 получен последовательной вставкой, начиная с пустого дерева, ключей CAPRICORN, AQUARIUS, PISCES, ARIES, TAURUS, GEMINI, CANCER, LEO, VIRGO, LIBRA, SCORPIO в указанном порядке.

На рис. 7.9 все ключи левого поддерева корня предшествуют по алфавиту слову CAPRICORN, а в правом поддереве стоят после него. Аналогичное утверждение справедливо для левого и правого поддеревьев любого узла. Отсюда

следует, что при обходе дерева в симметричном порядке ключи располагаются строга в алфавитном порядке:

AQUARIUS, ARIES, CANCER, CAPRICORN, GEMINI, LEO, ... ,VIRGO,

так как симметричный порядок основан на прохождении сначала левого поддерева каждого узла, затем самого узла, а затем его правого поддерева.

Ниже дается подробное описание процесса поиска с вставкой.

7.4.2. Поиск со вставкой по дереву

Дана таблица записей, образующих бинарное дерево. Производится поиск заданного аргумента K. Если его нет в таблице, то в подходящем месте в дерево вставляется новый узел, содержащий K.

Предполагается, что, узлы дерева содержат, по крайней мере, следующие поля [4]:

KEY(P)=ключ, хранящийся в узле NODE(P);

LLINK(P)= указатель на левое поддерево узла NODE(P);

RLINK(P)= указатель на правое поддерево узла NODE(P).

Пустые поддеревья представляются устыми указателями Λ. Переменная ROOT указывает на корень дерева. Для удобства предполагается, что дерево не пусто (ROOT ≠ Λ), так как при ROOT = Λ операции становятся тривиальными[4].

T1. [Начальная установка.] Установить P ← ROOT. (Указатель P будет продвигаться вниз по дереву.)

T2. [Сравнение.] если K < KEY(P); то перейти на T3; если K> KEY(P), то перейти на T4; если K = KEY(P), поиск завершен удачно.

T3. [Шаг влево.] Если LLINK(P) ≠ Λ, установить P← LLINK(P) и вернуться на T2. В противном случае перейти на T5.

T4.. [Шаг вправо.] Если RLINK(P) ≠ Λ установить. P ← RLINK(P) и вернуться на T2.

T5. [Вставка в дерево.] (Поиск неудачен; теперь мы поместим K в дерево.) Выполнить Q ⇐ AVAIL; Q ,теперь указывает на новый узел. Установить KEY(Q) ← K, LLINK(Q) ←RLINK(Q) ← Λ. (На самом деле нужно произвести начальную установку и других полей нового узла.) Если K было меньше KEY(P), Установить LLINK(P) ← Q; в противном случае установить RLINK(P)←Q. (В этот момент мы могли бы присвоить P ← Q и удачно завершить работу алгоритма).

Поиск в бинарном дереве является простым и эффективным методом, который считается одним из наиболее фундаментальных в компьютерной науке. Он чрезвычайно легок в использовании, но на самом деле он довольно широко используется во многих ситуациях.

Рассмотрим следующую структуру – дерево поиска:

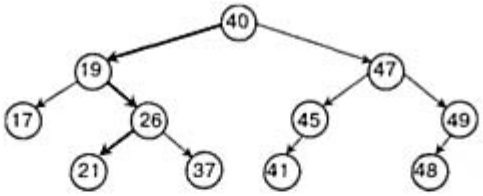


Рис. 7.10. Дерево поиска

Поиск по совпадению является самым легким: идем влево, если искомая запись меньше ключа в узле и вправо, если больше либо равна.

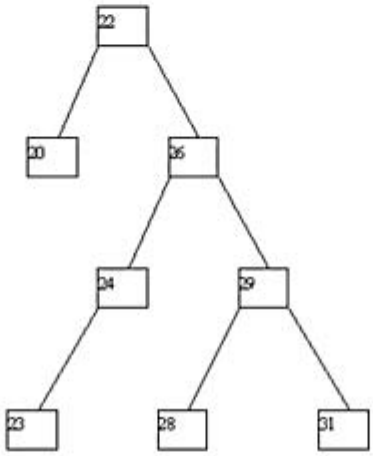


Рис. 7.11. Дерево поиска (другой пример)

Двоичные деревья обычно представляются как динамические структуры с базовым типом записи T, в число полей которого входят два указателя на переменные типа T.

При использовании в целях поиска элементов данных по значению уникального ключа применяются двоичные деревья поиска, обладающие тем свойством, что для любой вершины дерева значение ее ключа больше значения ключа любой вершины ее левого поддерева и больше значения ключа любой вершины правого поддерева (рис. 7.10 и рис. 7.11). Для поиска заданного ключа в дереве поиска достаточно пройти по одному пути от корня до (возможно, листовой) вершины (рис. 7.12). Высота идеально сбалансированного двоичного дерева с n вершинами составляет не более, чем $\log_n$ (логарифм двоичный), по-

этому при применении таких деревьев в качестве деревьев поиска (рис. 7.13) потребуется не более $\log_n$ сравнений.

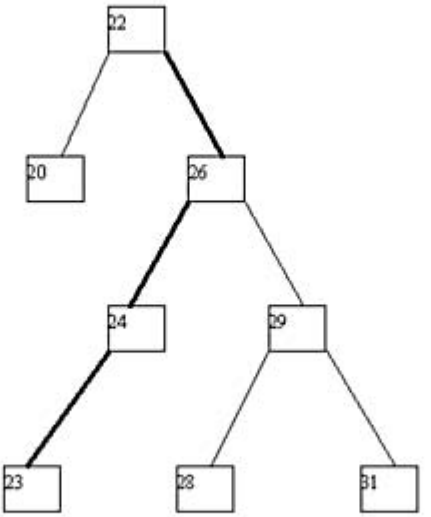


Рис. 7.12. Путь поиска ключа по значению «23»

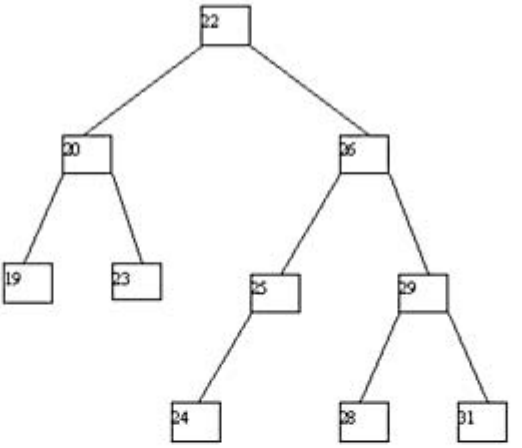


Рис. 7.13. Идеально сбалансированное двоичное дерево

Применение деревьев как объектов с динамической структурой особенно полезно, если допускать выполнение не только операции поиска по значению ключа, но и операций включения новых и исключения существующих ключей. Если не принимать во внимание потенциальное желание поддерживать идеальную балансировку дерева, то процедуры включения и исключения ключей

очень просты. Для включения в дерево вершины с новым ключом x по общим правилам поиска ищется листовая вершина, в которой находился бы этот ключ, если бы он входил в дерево. Возможны две ситуации: (a) такая вершина не существует; (b) вершина существует и уже занята, т.е. содержит некоторый ключ y. В первой ситуации создается недостающая вершина, и в нее заносится значение ключа x. Во второй ситуации после включения ключа x эта вершина в любом случае становится внутренней, причем если x > y, то ключ x заносится в новую листовую вершину – правого сына y, а если x < y – то в левую. Четыре потенциально возможных случая проиллюстрированы на рис. 7.14.

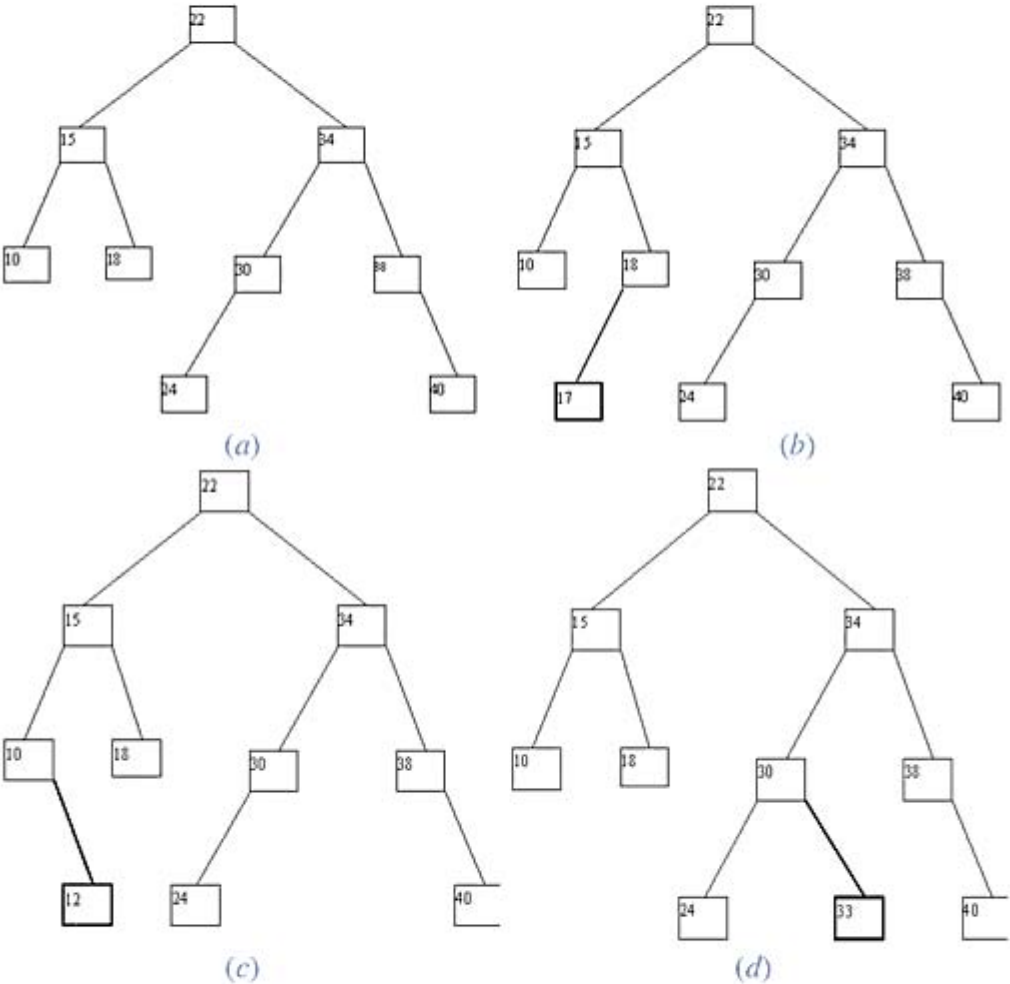
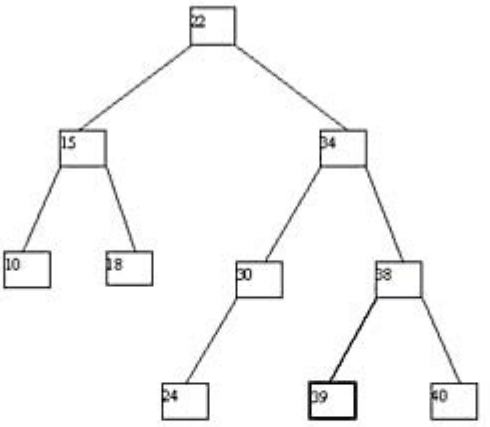


Рис. 7.14. Варианты включения ключа в дерево поиска



(*e*)

Рис. 7.14 (окончание). Варианты включения ключа в дерево поиска

7.4.3. Деревья оптимального поиска

В приводившихся выше рассуждениях по поводу организации деревьев поиска предполагалось, что вероятность поиска любого возможного ключа одна и та же, т.е. распределена равномерно. Но встречаются ситуации, в которых можно получить информацию о вероятности обращений к отдельным ключам. Обычно в таких случаях дерево поиска строится один раз, имеет неизменяемую структуру, в него не включаются новые ключи, и из него не исключаются существующие ключи. Примером соответствующего приложения является сканер компилятора, одной из задач которого является определение принадлежности очередного идентификатора к набору ключевых слов языка программирования. На основе сбора статистики при многочисленной компиляции программ можно получить достаточно точную информацию о частотах поиска по отдельным ключам.

Пусть дерево поиска содержит n вершин, и обозначим через p_i вероятность обращения к i-той вершине, содержащей ключ k_i. Сумма всех p_i, естественно, равна 1. Постараемся теперь организовать дерево поиска таким образом, чтобы обеспечить минимальность общего числа шагов поиска, подсчитанного для достаточно большого количества обращений. Будем считать, что корень дерева имеет высоту 1 (а не 0, как раньше), и определим взвешенную длину пути дерева как сумму p_i*h_i ($1<=i<=n$), где h_i – длина пути от корня до i-той вершины. Требуется построить дерево поиска с минимальной взвешенной длиной пути.

В качестве примера рассмотрим возможности построения дерева поиска для трех ключей 1, 2, 3 с вероятностями обращения к ним 1/7, 2/7 и 4/7 соответственно (рисунок 5.15).

Посчитаем взвешенную длину пути для каждого случая. В случае (a) взвешенная длина пути P(a) = 1*4/7 + 2*2/7 + 3*1/7 = 11/5. Аналогичные подсчеты дают результаты P(b)=12/7; P(c)=12/7; P(d)=15/7; P(e)=17/5. Следовательно, оптимальным в интересующем нас смысле оказалось не идеально сбалансированное дерево (c), а вырожденное дерево (a).

На практике приходится решать несколько более общую задачу, а именно, при построении дерева учитывать вероятности неудачного поиска, т.е. поиска ключа, не включенного в дерево. В частности, при реализации сканера желательно уметь эффективно распознавать идентификаторы, которые не являются ключевыми словами. Можно считать, что поиск по ключу, отсутствующему в дереве, приводит к обращению к «специальной» вершине, включенной между реальными вершинами с меньшим и большим значениями ключа соответственно. Если известна вероятность qj обращения к специальной j-той вершине, то к общей средней взвешенной длине пути дерева необходимо добавить сумму $q_j * e_j$ для всех специальных вершин, где e_i – высота j-той специальной вершины.

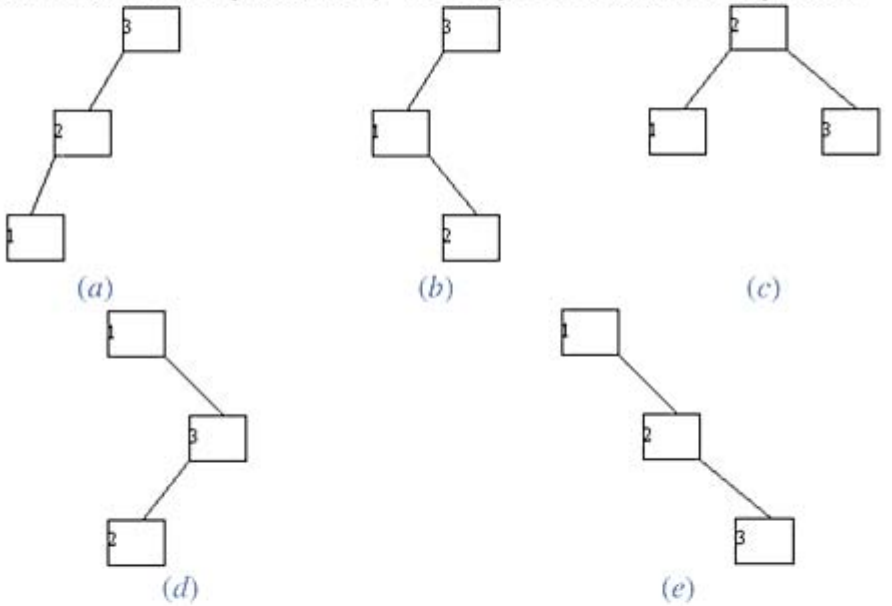


Рис. 7.15. Варианты построения дерева для трех ключей

При построении дерева оптимального поиска вместо значений p_i и q_j обычно используют полученные статистически значения числа обращений к соответствующим вершинам. Процедура построения дерева оптимального поиска достаточно сложна и опирается на тот факт, что любое поддерево дерева оптимального поиска также обладает свойством оптимальности. Поэтому известный алгоритм строит дерево «снизу-вверх», т.е. от листьев к корню. Сложность этого алгоритма и расходы по памяти составляют O(n2). Имеется эври-

стический алгоритм, дающий дерево, близкое к оптимальному, со сложностью $O(n*\log_n)$ и расходами памяти – O(n).

7.4.4. Деревья цифрового поиска

Методы цифрового поиска достаточно громоздки и плохо иллюстрируются. Поэтому мы кратко остановимся на наиболее простом механизме – бинарном дереве цифрового поиска. Как и в деревьях, обсуждавшихся в предыдущих разделах, в каждой вершине такого дерева хранится полный ключ, но переход по левой или правой ветви происходит не путем сравнения ключа-аргумента со значением ключа, хранящегося в вершине, а на основе значения очередного бита аргумента.

Поиск начинается от корня дерева. Если содержащийся в корневой вершине ключ не совпадает с аргументом поиска, то анализируется самый левый бит аргумента. Если он равен 0, происходит переход по левой ветви, если 1 – по правой. Если не обнаруживается совпадение ключа с аргументом поиска, то анализируется следующий бит аргумента и т.д., пока либо не будут проверены все биты аргумента, или мы не наткнемся на вершину с отсутствующей левой или правой ссылкой.

На рисунке 7.16 показан пример дерева цифрового поиска для некоторых заглавных букв латинского алфавита. Считается, что буквы кодируются в соответствии с кодовым набором ASCII, а для их представления и поиска используются 5 младших бит кода. Например, код буквы A равен 41(16), а представляться A будет как последовательность бит 00001.

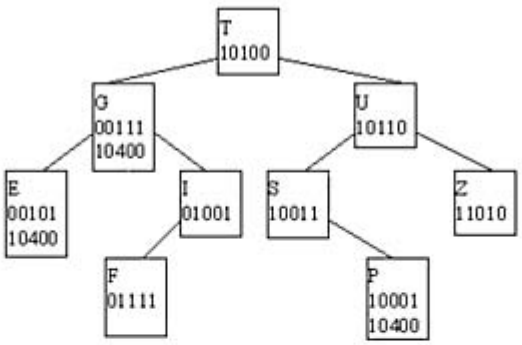


Рис. 7.16. Дерево цифрового поиска

7.5. Методы поиска во внешней памяти на основе деревьев

Как и в случае необходимости сортировки данных, не размещаемых целиком в основной памяти, для поиска данных во внешней памяти существует ряд методов, несколько напоминающих те, которые упоминались в предыдущей части, но сильно отличающихся по своей сути. Базовым «древовидным» аппаратом для поиска данных во внешней памяти являются B–деревья. В основе этого механизма лежат следующие идеи. Во-первых, поскольку речь идет о структурах данных во внешней памяти, общее время доступа к которой определяется в основном не объемом последовательно расположенных данных, а временем подвода магнитных головок, то выгодно получать за одно обращение к внешней памяти как можно больше информации, учитывая при этом необходимость экономного использования основной памяти. При сложившемся подходе к организации основной памяти в виде набора страниц равного размера естественно считать именно страницу единицей обмена с внешней памятью. Во-вторых, желательно обеспечить такую поисковую структуру во внешней памяти, при использовании которой поиск информации по любому ключу требует заранее известного числа обменов с внешней памятью.

7.5.1. Классические B–деревья

Механизм классических B–деревьев был предложен в 1970 г. Бэйером и Маккрейтом. B–дерево порядка n представляет собой совокупность иерархически связанных страниц внешней памяти (каждая вершина дерева – страница), обладающая следующими свойствами:

1. Каждая страница содержит не более 2*n элементов (записей с ключом).
2. Каждая страница, кроме корневой, содержит не менее n элементов.
3. Если внутренняя (не листовая) вершина B–дерева содержит m ключей, то у нее имеется m+1 страниц-потомков.
4. Все листовые страницы находятся на одном уровне.

Пример B–дерева степени 2 глубины 3 приведен на рисунке 7.15.

Поиск в B–дереве производится очевидным образом. Предположим, что происходит поиск ключа K. В основную память считывается корневая страница B–дерева. Предположим, что она содержит ключи k_1, k_2, ..., k_m и ссылки на страницы p_0, p_1, ..., p_m. В ней последовательно (или с помощью какого-либо другого метода поиска в основной памяти) ищется ключ K.

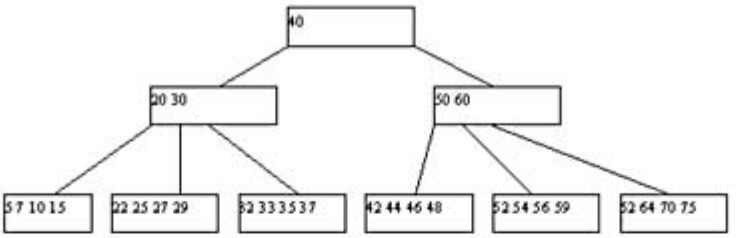


Рис. 7.15. Классическое B–дерево порядка 2

Если он обнаруживается, поиск завершен. Иначе возможны три ситуации:

1. Если в считанной странице обнаруживается пара ключей k_i и k_{i+1} такая, что $k_i < K < k_{i+1}$, то поиск продолжается на странице p_i.
2. Если обнаруживается, что $K > k_m$, то поиск продолжается на странице p_m.
3. Если обнаруживается, что $K < k_1$, то поиск продолжается на странице p_0.

Для внутренних страниц поиск продолжается аналогичным образом, пока либо не будет найден ключ K, либо мы не дойдем до листовой страницы. Если ключ не находится и в листовой странице, значит ключ K в B–дереве отсутствует.

Включение нового ключа K в B–дерево выполняется следующим образом. По описанным раньше правилам производится поиск ключа K. Поскольку этот ключ в дереве отсутствует, найти его не удастся, и поиск закончится в некоторой листовой странице A. Далее возможны два случая. Если A содержит менее 2*n ключей, то ключ K просто помещается на свое место, определяемое порядком сортировки ключей в странице A. Если же страница A уже заполнена, то работает процедура расщепления. Заводится новая страница C. Ключи из страницы A (берутся 2*n-1 ключей) + ключ K поровну распределяются между A и C, а средний ключ вместе со ссылкой на страницу C переносится в непосредственно родительскую страницу B. Конечно, страница B может оказаться переполненной, рекурсивно сработает процедура расщепления и т.д., вообще говоря, до корня дерева. Если расщепляется корень, то образуется новая корневая вершина, и высота дерева увеличивается на единицу. Одношаговое включение ключа с расщеплением страницы показано на рисунке 7.18.

Процедура исключения ключа из классического B–дерева более сложна. Приходится различать два случая – удаление ключа из листовой страницы и удаления ключа из внутренней страницы B–дерева. В первом случае удаление производится просто: ключ просто исключается из списка ключей. При удалении ключа во втором случае для сохранения корректной структуры B–дерева его необходимо заменить на минимальный ключ листовой страницы, к которой ведет последовательность ссылок, начиная от правой ссылки от ключа K (это минимальный содержащийся в дереве ключ, значение которого больше значе-

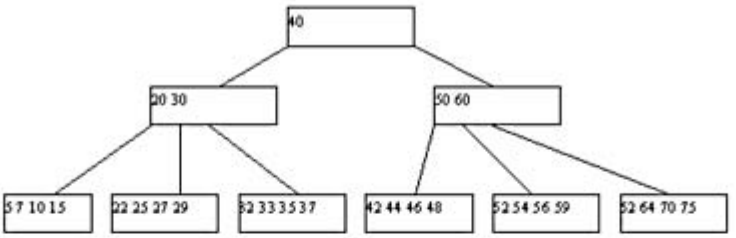


Рис. 7.15. Классическое B–дерево порядка 2

Если он обнаруживается, поиск завершен. Иначе возможны три ситуации:

1. Если в считанной странице обнаруживается пара ключей k_i и k_{i+1} такая, что $k_i < K < k_{i+1}$, то поиск продолжается на странице p_i.
2. Если обнаруживается, что $K > k_m$, то поиск продолжается на странице p_m.
3. Если обнаруживается, что $K < k_1$, то поиск продолжается на странице p_0.

Для внутренних страниц поиск продолжается аналогичным образом, пока либо не будет найден ключ K, либо мы не дойдем до листовой страницы. Если ключ не находится и в листовой странице, значит ключ K в B–дереве отсутствует.

Включение нового ключа K в B–дерево выполняется следующим образом. По описанным раньше правилам производится поиск ключа K. Поскольку этот ключ в дереве отсутствует, найти его не удастся, и поиск закончится в некоторой листовой странице A. Далее возможны два случая. Если A содержит менее 2*n ключей, то ключ K просто помещается на свое место, определяемое порядком сортировки ключей в странице A. Если же страница A уже заполнена, то работает процедура расщепления. Заводится новая страница C. Ключи из страницы A (берутся 2*n-1 ключей) + ключ K поровну распределяются между A и C, а средний ключ вместе со ссылкой на страницу C переносится в непосредственно родительскую страницу B. Конечно, страница B может оказаться переполненной, рекурсивно сработает процедура расщепления и т.д., вообще говоря, до корня дерева. Если расщепляется корень, то образуется новая корневая вершина, и высота дерева увеличивается на единицу. Одношаговое включение ключа с расщеплением страницы показано на рисунке 7.18.

Процедура исключения ключа из классического B–дерева более сложна. Приходится различать два случая – удаление ключа из листовой страницы и удаления ключа из внутренней страницы B–дерева. В первом случае удаление производится просто: ключ просто исключается из списка ключей. При удалении ключа во втором случае для сохранения корректной структуры B–дерева его необходимо заменить на минимальный ключ листовой страницы, к которой ведет последовательность ссылок, начиная от правой ссылки от ключа K (это минимальный содержащийся в дереве ключ, значение которого больше значе-

Поскольку в любом случае в одной из листовых страниц число ключей уменьшается на единицу, может нарушиться то требование, что любая, кроме корневой, страница B–дерева должна содержать не меньше n ключей. Если это действительно случается, начинает работать процедура переливания ключей. Берется одна из соседних листовых страниц (с общей страницей-предком); ключи, содержащиеся в этих страницах, а также средний ключ страницы-предка поровну распределяются между листовыми страницами, и новый средний ключ заменяет тот, который был заимствован у страницы-предка (рисунок 7.20).

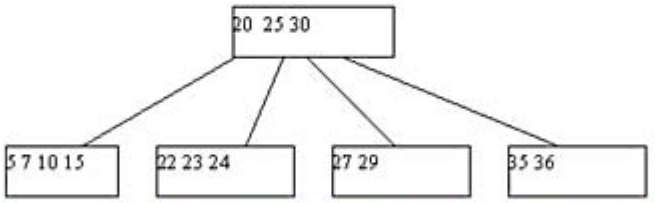
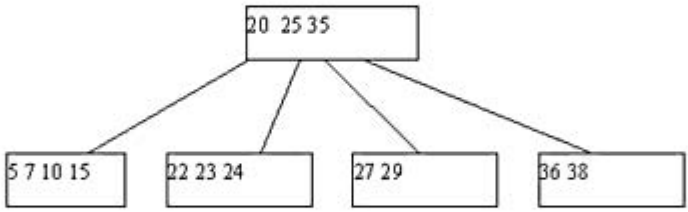


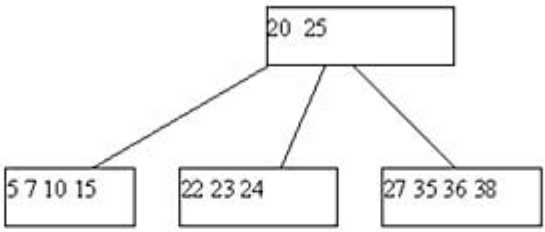
Рис. 7.20. Результат удаления ключа 38 из B–дерева с рис. 7.19

Может оказаться, что ни одна из соседних страниц непригодна для переливания, поскольку содержат по n ключей. Тогда выполняется процедура слияния соседних листовых страниц. К 2*n-1 ключам соседних листовых страниц добавляется средний ключ из страницы-предка (из страницы-предка он изымается), и все эти ключи формируют новое содержимое исходной листовой страницы. Поскольку в странице-предке число ключей уменьшилось на единицу, может оказаться, что число элементов в ней стало меньше n, и тогда на этом уровне выполняется процедура переливания, а возможно, и слияния. Так может продолжаться до внутренних страниц, находящихся непосредственно под корнем B–дерева. Если таких страниц всего две, и они сливаются, то единственная общая страница образует новый корень. Высота дерева уменьшается на единицу, но по-прежнему длина пути до любого листа одна и та же. Пример удаления ключа со слиянием листовых страниц показан на рисунке 7.21.



а) Начальный вид B–дерева

Рис. 7.21,а. Пример удаления ключа из B–дерева со слиянием листовых страниц



b) B–дерево после удаления ключа 29

Рис. 7.21,b. Пример удаления ключа из B–дерева со слиянием листовых страниц

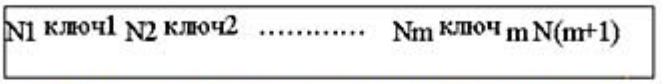
7.5.2. B+–деревья

Схема организации классических B–деревьев проста и элегантна, но не очень хороша для практического использования. Прежде всего это связано с тем, что в большинстве практических применений необходимо хранить во внешней памяти не только ключи, но и записи. Поскольку в B–дереве элементы располагаются и во внутренних, и в листовых страницах, а размер записи может быть достаточно большим, внутренние страницы не могут содержать слишком много элементов, по причине дерево может быть довольно глубоким. Поэтому для доступа к ключам и записям, находящимся на нижних уровнях дерева, может потребоваться много обменов с внешней памятью. Во-вторых, на практике часто встречается потребность хранения и поиска ключей и записей переменного размера. Поэтому тот критерий, что в каждой странице B–дерева содержится не меньше n и не больше 2*n ключей, становится неприменимым.

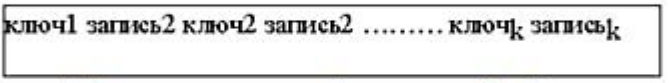
Широкое практическое применение получила модификация механизма B–деревьев, которую принято называть B+-деревьями. Эти деревья похожи на обычные B–деревья. Они тоже сильно ветвистые, и длина пути от корня к любой листовой странице одна и та же. Но структура внутренних и листовых страниц различна. Внутренние страницы устроены так же, как у B–дерева, но в них хранятся только ключи (без записей) и ссылки на страницы-потомки. В листовых страницах хранятся все ключи, содержащиеся в дереве, вместе с записями, причем этот список упорядочен по возрастанию значения ключа (рисунок 7.22).

Поиск ключа всегда доходит до листовой страницы. Аналогично операции включения и исключения тоже начинаются с листовой страницы. Для применения переливания, расщепления и слияния используются критерии, основанные на уровне заполненности соответствующей страницы. Для более экономного и сбалансированного использования внешней памяти при реализации B+-деревьев иногда используют технику слияния трех соседних страниц в две и расщепления двух соседних страниц в три. Хотя B+-деревья хранят избыточ-

ную информацию (один ключ может храниться в двух страницах), они, очевидно, обладают меньшей глубиной, чем классические B–деревья, а для поиска любого ключа требуется одно и то же число обменов с внешней памятью.



a) Структура внутренней страницы B+–дерева



b) Структура листовой страницы B+–дерева

Рис. 7.22. Структуры страниц B+–дерева

Дополнительной полезной оптимизацией B+–деревьев является связывание листовых страниц в одно- или двунаправленный список. Это позволяет просматривать списки записей для заданного диапазона значений ключей с лишь одним прохождением дерева от корня к листу.

7.5.3. Разновидности B+–деревьев для организации индексов в базах данных

B+–деревья наиболее интенсивно используются для организации индексов в базах данных. В основном это определяется двумя свойствами этих деревьев: предсказуемостью числа обменов с внешней памятью для поиска любого ключа и тем, что это число обменов по причине сильной ветвистости деревьев не слишком велико при индексировании даже очень больших таблиц.

При использовании B+–деревьев для организации индексов каждая запись содержит упорядоченный список идентификаторов строк таблицы, включающих соответствующее значение ключа. Дополнительную сложность вызывает возможность организации индексов по нескольким столбцам таблицы (так называемых «составных» индексов). В этом случае в B+–дереве может появиться очень много избыточной информации по причине наличия в разных составных ключах общих подключей. Имеется ряд технических приемов сжатия индексов с составными ключами, улучшающих использование внешней памяти, но, естественно, замедляющих выполнение операций включения и исключения.

7.5.4. R–деревья и их использование для организации индексов в пространственных базах данных

Коротко рассмотрим еще одно расширение механизма B–деревьев, используемое главным образом для организации индексов в пространственных базах данных, R–деревья. Подобно B+–деревьям, R–дерево представляет собой

ветвистую сбалансированную древовидную структуру с разной организацией внутренних и листовых страниц.

Информация, хранящаяся в R–дереве несколько отличается от той, которая содержится в B–деревьях. В дополнение к находящимся в листовых страницах идентификаторам пространственных объектов, в R–деревьях хранится информация о границах индексируемого объекта. В случае двумерного пространства сохраняются горизонтальные и вертикальные координаты нижнего левого и верхнего правого углов наименьшего прямоугольника, содержащего индексируемый объект. Пример простого R–дерева, содержащего информацию о шести пространственных объектах, приведен на рисунке 7.23.

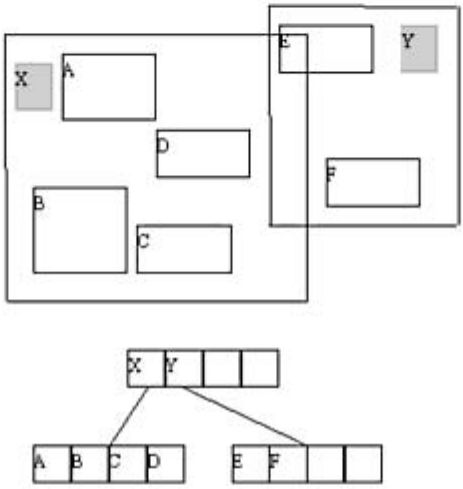


Рис. 7.23. Простое R–дерево для представления шести пространственных объектов

7.5.5. Индексно-последовательный поиск

Для увеличения эффективности поиска в отсортированном файле существует другой метод, но он приводит к увеличению требуемого пространства. Этот метод называется индексно – последовательным методом поиска.

В дополнение к отсортированному файлу заводится некоторая вспомогательная таблица, называемая индексом. Каждый элемент индекса состоит из ключа k_{index} и указателя на запись в файле, соответствующую этому ключу k_{index}. Элементы в индексе, так же как и элементы в файле, должны быть отсортированы по этому ключу.

Если индекс имеет размер, составляющий одну восьмую от размера файла, то каждая восьмая запись в файле представлена первоначально в индексе. Это показано на рисунке 7.24 .

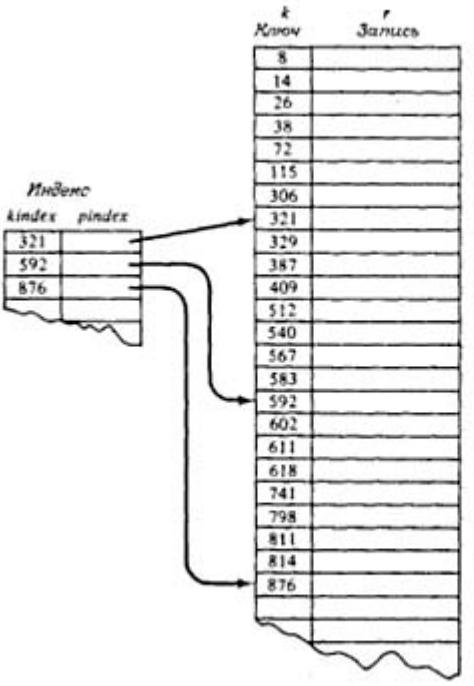


Рис. 7.24. Индексно-последовательный поиск

После перехода по индексу поиск осуществляется либо линейным способом, либо двоичным.

Вопросы для самоконтроля

1. Назовите алгоритмы поиска данных. Приведите классификацию методов поиска.
2. Запишите формулу вычисления пробы в методе интерполяционного поиска.
3. В чем заключается идея бинарного поиска?
4. Изобразите схему блочного поиска на примере.
5. Как происходит поиск в дереве Фибоначчи?
6. Какое практическое применение находят сильно-ветвящиеся деревья?

ГЛАВА 8
ХЕШИРОВАНИЕ

8.1. Терминология

Рассмотренные алгоритмы поиска основываются на абстрактной операции сравнения. Однако, метод поиска с использованием индексирования по ключу является существенным исключением из этого утверждения. При поиске с использованием индексирования по ключу значения ключей используются в качестве индексов массива, а не участвуют в сравнениях; при этом метод основывается на том, что ключи являются различными целыми числами из того же диапазона, что и индексы таблицы. В этой главе мы рассмотрим хеширование (hashing) – расширенный вариант поиска с использованием индексирования по ключу, применяемый в более типовых приложениях поиска, в которых не приходится рассчитывать на наличие ключей со столь удобными свойствами. Конечный результат применения данного метода коренным образом отличается от результата применения основанных на сравнении методов – вместо перемещения по структурам данных словаря со сравнением ключей поиска с ключами в элементах, предпринимается попытка обращения к элементам в таблице непосредственно, за счет выполнения арифметических операций для преобразования ключей в адреса таблицы.

Хеширование есть разбиение множества ключей (однозначно характеризующих элементы хранения и представленных, как правило, в виде текстовых строк или чисел) на непересекающиеся подмножества (наборы элементов), обладающие определенным свойством. Это свойство описывается *функцией хеширования*, или *хеш-функцией*, и называется *хеш-адресом*. Решение обратной задачи возложено на *хеш-структуры* (*хеш-таблицы*): по хеш-адресу они обеспечивают быстрый доступ к нужному элементу. В идеале для задач поиска хеш-адрес должен быть уникальным, чтобы за одно обращение получить доступ к элементу, характеризуемому заданным ключом (идеальная хеш-функция). Однако, на практике идеал приходится заменять компромиссом и исходить из того, что получающиеся наборы с одинаковым хеш-адресом содержат более одного элемента.

Термин «**хеширование**» (**hashing**) в печатных работах по программированию появился 1967 г., хотя сам механизм был известен и ранее. Глагол «hash» в английском языке означает «рубить, крошить». Для русского языка академиком А.П. Ершовым был предложен достаточно удачный эквивалент – «**расстановка**», созвучный с родственными понятиями комбинаторики, такими как «подстановка» и «перестановка». Однако он не прижился.

С хешированием мы сталкиваемся едва ли не на каждом шагу: при работе с браузером (список Web-ссылок), текстовым редактором и переводчиком (словарь), языками скриптов (Perl, Python, PHP и др.), компилятором (таблица символов). По словам Брайана Кернигана, это «одно из величайших изобретений информатики». Заглядывая в адресную книгу, энциклопедию, алфавитный указатель, мы даже не задумываемся, что упорядочение по алфавиту является не чем иным, как хешированием.

Алгоритмы поиска, которые используют хеширование, состоят из двух отдельных частей. Первый шаг – вычисление хеш-функции, которая преобразует ключ поиска в адрес в таблице. В идеале различные ключи должны были бы отображаться на различные адреса, но часто два и более различных ключа могут преобразовываться в один и тот же адрес в таблице. Поэтому вторая часть поиска методом хеширования – процесс разрешения конфликтов, который обрабатывает такие ключи. В одном из рассматриваемых методов разрешения конфликтов используются связные списки, поэтому он находит непосредственное применение в динамических ситуациях, когда заранее трудно предвидеть количество ключей поиска. В других двух методах разрешения конфликтов высокая производительность поиска обеспечивается для элементов, хранящихся в фиксированном массиве. Исследуем также способ усовершенствования этих методов с целью их расширения для случаев, когда нельзя заранее предсказать размеры таблицы.

Как отмечает Дональд Кнут, идея хеширования впервые была высказана Г.П. Ланом при создании внутреннего меморандума IBM в январе 1953 г. с предложением использовать для разрешения коллизий хеш-адресов метод цепочек. Примерно в это же время другой сотрудник IBM – Жини Амдал – высказала идею использования открытую линейную адресацию. В открытой печати хеширование впервые было описано Арнольдом Думи (1956), указавшим, что в качестве хеш-адреса удобно использовать остаток от деления на простое число. А. Думи описывал метод цепочек для разрешения коллизий, но не говорил об открытой адресации. Подход к хешированию, отличный от метода цепочек, был предложен А.П. Ершовым (1957), который разработал и описал метод линейной открытой адресации. Среди других исследований можно отметить работу Петерсона (1957). В ней реализовывался класс методов с открытой адресацией при работе с большими файлами. Петерсон определил открытую адресацию в общем случае, проанализировал характеристики равномерного хеширования, глубоко изучил статистику использования линейной адресации на различных задачах. В 1963 г. Вернер Буккольц опубликовал наиболее основательное исследование хеш-функций.

К концу шестидесятых годов прошлого века линейная адресация была единственным типом схемы открытой адресации, описанной в литературе, хотя несколькими исследователями независимо была разработана другая схема, основанная на неоднократном случайном применении независимых хеш-функции. В течение нескольких последующих лет хеширование стало широко

использоваться, хотя не было опубликовано никаких новых работ. Затем Роберт Моррис дал обширный обзор по хешированию и ввел термин «рассеянная память» (scatter storage). Эта работа привела к созданию открытой адресации с двойным хешированием.

Хеширование – хороший пример компромисса между временем и объемом памяти. Если бы на объем используемой памяти ограничения не накладывались, любой поиск можно было бы выполнить за счет всего лишь одного обращения к памяти, просто используя ключ в качестве адреса памяти, как это делается при поиске с использованием индексирования по ключу. Однако, часто этот идеальный случай оказывается недостижимым, поскольку требуемый объем памяти неприемлем, когда ключи являются длинными. С другой стороны, если бы не существовало ограничений на время выполнения, можно было бы обойтись минимальным объемом памяти, используя метод последовательного поиска. Хеширование предоставляет способ использования как приемлемого объема памяти, так и приемлемого времени с целью достижения компромисса между этими двумя крайними случаями. В частности, можно поддерживать любое выбранное соотношение, просто настраивая размер таблицы, а не переписывая код или выбирая другие алгоритмы.

Хеширование – одна из классических задач компьютерных наук: различные алгоритмы подробно исследованы и находят широкое применение. Мы увидим, что при ряде общих допущений можно надеяться на обеспечение поддержки операций search и insert в таблицах символов при постоянном времени выполнения независимо от размера таблицы.

Это ожидаемое постоянное время выполнения – теоретический оптимум производительности для любой реализации таблицы символов, но хеширование не является панацеей по двум основным причинам. Во-первых, время выполнения зависит от длины ключа, которая может быть значительной в реальных приложениях, использующих длинные ключи. Во-вторых, хеширование не обеспечивает эффективные реализации для других операций, таких как select или sort, с таблицами символов. Эти и другие вопросы подробно рассматриваются в этой главе.

Далее будут рассмотрены основные виды хеш-функций и некоторые их модификации, методы разрешения коллизий, проблемы удаления элементов из хеш-таблицы, а также некоторые варианты применения хеширования.

8.2. Хеш-функции

Хеш-функция – это некоторая функция h(K), которая берет некоторый ключ K и возвращает адрес, по которому производится поиск в хеш-таблице, чтобы получить информацию, связанную с K. Например, K – это номер телефона абонента, а искомая информация – его имя. Функция в данном случае нам точно скажет, по какому адресу найти искомое.

Коллизия (конфликт)– это ситуация, когда h(K1) = h(K2), в то время как K1 ≠ K2. В этом случае, очевидно, необходимо найти новое место для хранения данных. Очевидно, что количество коллизий необходимо минимизировать. Методикам разрешения коллизий будет посвящен отдельный раздел ниже.

Хорошая хеш-функция должна удовлетворять двум требованиям:

- ее вычисление должно выполняться очень быстро;
- она должна минимизировать число коллизий.

Итак, первое свойство хорошей хеш-функции зависит от компьютера, а второе – от данных. Если бы все данные были случайными, то хеш-функции были бы очень простые (несколько битов ключа, например). Однако на практике случайные данные встречаются крайне редко, и приходится создавать функцию, которая зависела бы от всего ключа.

Теоретически невозможно определить хеш-функцию так, чтобы она создавала случайные данные из реальных неслучайных файлов. Однако на практике реально создать достаточно хорошую имитацию с помощью простых арифметических действий. Более того, зачастую можно использовать особенности данных для создания хеш-функций с минимальным числом коллизий (меньшим, чем при истинно случайных данных).

Хеш-таблица – это обычный массив с необычной адресацией, задаваемой хеш-функцией. Например, на **hashTable** (рис. 8.1) – это массив из 8 элементов. Каждый элемент представляет собой указатель на линейный список, хранящий числа. Хеш-функция в этом примере просто делит ключ на 8 и использует остаток как индекс в таблице. Это дает нам числа от 0 до 7 Поскольку для адресации в **hashTable** нам и нужны числа от 0 до 7, алгоритм гарантирует допустимые значения индексов.

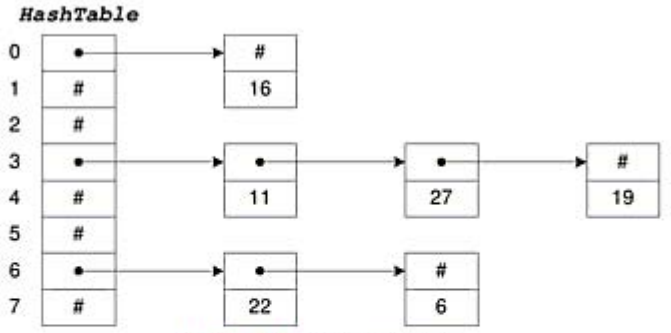


Рис. 8.1. Хеш-таблица

Чтобы вставить в таблицу новый элемент, мы хешируем ключ, чтобы определить список, в который его нужно добавить, затем вставляем элемент в начало этого списка. Например, чтобы добавить 11, мы делим 11 на 8 и получаем остаток 3. Таким образом, 11 следует разместить в списке, на начало которого указывает **hashTable[3]**. Чтобы найти число, мы его хешируем и проходим по

соответствующему списку. Чтобы удалить число, мы находим его и удаляем элемент списка, его содержащий.

Если хеш-функция распределяет совокупность возможных ключей равномерно по множеству индексов, то хеширование эффективно разбивает множество ключей. Наихудший случай – когда все ключи хешируются в один индекс. При этом мы работаем с одним линейным списком, который и вынуждены последовательно сканировать каждый раз, когда что-нибудь делаем. При иллюстрации методов предполагается, что *unsigned char* располагается в 8, *unsigned short int* – в 16, *unsigned long int* – в 32.

Возможно, одним из самых очевидных и простых способов хеширования является метод середины квадрата, когда ключ возводится в квадрат и берется несколько цифр в середине. Здесь и далее предполагается, что ключ сначала приводится к целому числу, для совершения с ним арифметических операций. Однако такой способ хорошо работает до момента, когда нет большого количества нолей слева или справа. Многочисленные тесты показали хорошую работу двух основных типов хеширования, один из которых основан на делении, а другой на умножении. Впрочем, это не единственные методы, которые существуют, более того, они не всегда являются оптимальными.

Метод деления

Наиболее широко распространенная функция хеширования основывается на методе деления и определяется в виде

$$H(x) = x \bmod m + 1,$$

где m – делитель (m – может быть равным размеру хеш-таблицы). Эта функция хеширования – одна из первых и наиболее широко используемых.

При отображении ключей в адреса методом деления до некоторой степени сохраняется существующая на множестве ключей равномерность распределения. Ключи с близкими значениями отображаются при этом в уникальные адреса. Например, при делителе равном 101 такая функция отобразила бы ключи 2000, 2001, ..., 2017 в адреса 82,83, ..., 99. К сожалению, если два скопления ключей или более отображаются в одни и те же адреса, то сохранение равномерности будет недостатком. Например, если имеются также ключи 3310, 3311,3313,3314, ...,3323,3324, то при делителе 101 они будут отображены в адреса 79, 80, 82, 83, ..., 92, 93, и с группой ключей, значения которых начинаются с 2000, произойдет много коллизий. Причина этого в том, что ключи из этих двух групп совпадают по модулю 101.

Вообще, если по модулю d совпадает много ключей, а m и d не являются взаимно простыми числами, то использование значения m в качестве делителя может привести к низкой эффективности хеширования, основанного на методе

деления. Это показано в предыдущем примере, в котором m = d = 101. Возьмем другой пример: если все ключи в совокупности записей совпадают по модулю 5 и делителем является число 65, то значения ключей отображаются лишь в 13 различных позициях. Если m является большим простым числом, то обычно ключи не совпадают по модулю m, и поэтому в качестве делителя следует выбирать простое число. Исследования, однако, показывают, что удовлетворительные результаты получаются и при нечетном делителе, не имеющем множителей менее 20. В особенности следует избегать четных делителей, так как при этом четные и нечетные ключи отображаются соответственно в нечетные и четные адреса (в предположении, что адресное пространство имеет вид {1,2, ..., m}). При этом возникали бы трудности в организации таблиц, содержащих в основном четные или в основном нечетные ключи.

Пример. Деление (размер таблицы **hashTableSize** – простое число). Хеширующее значение **hashValue**, изменяющееся от 0 до (**hashTableSize – 1**), равно остатку от деления ключа на размер хеш-таблицы. Вот как это может выглядеть:

```
typedef int hashIndexType;
hashIndexType hash(int Key) {
return Key % hashTableSize;
}
```

Для успеха этого метода очень важен выбор подходящего значения **hashTableSize**. Если, например, **hashTableSize** равняется двум, то для четных ключей хеш-значения будут четными, для нечетных – нечетными. Ясно, что это нежелательно – ведь если все ключи окажутся четными, они попадут в один элемент таблицы. Аналогично, если все ключи окажутся четными, то **hashTableSize**, равное степени двух, попросту возьмет часть битов **Key** в качестве индекса. Чтобы получить более случайное распределение ключей, в качестве **hashTableSize** нужно брать простое число, не слишком близкое к степени двух.

Метод умножения (мультипликативный)

Для мультипликативного хеширования используется следующая формула:

$$h(K) = [M * ((C * K) \bmod 1)]$$

Здесь производится умножение ключа на некую константу C, лежащую в интервале [0..1]. После этого берется дробная часть этого выражения и умножается на некоторую константу M, выбранную таким образом, чтобы результат не вышел за границы хеш-таблицы. Оператор [ ] возвращает наибольшее целое, которое меньше аргумента.

Если константа C выбрана верно, то можно добиться очень хороших результатов, однако, этот выбор сложно сделать. Дональд Кнут отмечает, что умножение может иногда выполняться быстрее деления.

Мультипликативный метод хорошо использует то, что реальные файлы неслучайны. Например, часто множества ключей представляют собой арифметические прогрессии, когда в файле содержатся ключи {K, K + d, K + 2d, …, K + td}. Например, рассмотрим имена типа {PART1, PART2, …, PARTN}. Мультипликативный метод преобразует арифметическую прогрессию в приближенно арифметическую прогрессию h(K), h(K + d), h(K + 2d),… различных хеш-значений, уменьшая число коллизий по сравнению со случайной ситуацией. Впрочем, справедливости ради надо заметить, что метод деления обладает тем же свойством.

Частным случаем выбора константы является значение золотого сечения $\varphi = (\sqrt{5} - 1)/2 \approx 0{,}6180339887$. Если взять последовательность $\{\varphi\}, \{2\varphi\}, \{3\varphi\},...$ где оператор { } возвращает дробную часть аргумента, то на отрезке [0..1] она будет распределена очень равномерно. Другими словами, каждое новое значение будет попадать в наибольший интервал. Это явление было впервые замечено Я. Одерфельдом (J. Oderfeld) и доказано С. Сверчковски (S. Świerczkowski) . В доказательстве играют важную роль числа Фибоначчи. Применительно к хешированию это значит, что если в качестве константы C выбрать золотое сечение, то функция будет достаточно хорошо рассеивать ключи вида {PART1, PART2, …, PARTN}. Такое хеширование называется хешированием Фибоначчи. Впрочем, существует ряд ключей (когда изменение происходит не в последней позиции), когда хеширование Фибоначчи оказывается не самым оптимальным.

Пример 1. Мультипликативный метод (размер таблицы **hashTableSize** есть степень 2^n).

Значение **Key** умножается на константу, затем от результата берется необходимое число битов. В качестве такой константы Кнут[11] рекомендует золотое сечение $(\sqrt{5}-1)/2 = 0.6180339887499$. Пусть, например, мы работаем с байтами. Умножив золотое сечение на 2^8, получаем 158. Перемножим 8-битовый ключ и 158, получаем 16-битовое целое. Для таблицы длиной 2^5 в качестве хеширующего значения берем 5 младших битов младшего слова, содержащего такое произведение. Вот как можно реализовать этот метод:

```
/* 8-bit index */
typedef unsigned char hashIndexType;
static const hashIndexType K = 158;

/* 16-bit index */
typedef unsigned short int hashIndexType;
static const hashIndexType K = 40503;
```

```
/* 32-bit index */
typedef unsigned long int hashIndexType;
static const hashIndexType K = 2654435769;

/* w=bitwidth(hashIndexType), size of table=2**m */
static const int S = w - m;
hashIndexType hashValue = (hashIndexType)(K * Key) >> S;
```

Пример 2. Мультипликативный метод (размер таблицы hashTableSize равен 1024 (2^{10}). Тогда нам достаточен 16-битный индекс и S будет присвоено значение 16 – 10 = 6. В итоге получаем:

```
typedef unsigned short int hashIndexType;

hashIndexType hash(int Key) {
static const hashIndexType K = 40503;
static const int S = 6;
return (hashIndexType)(K * Key) >> S;
```

Аддитивный метод для строк переменной длины (размер таблицы равен 256)

Для строк переменной длины вполне разумные результаты дает сложение по модулю 256. В этом случае результат `hashValue` заключен между 0 и 244.

```
typedef unsigned char hashIndexType;

hashIndexType hash(char *str) {
hashIndexType h = 0;
while (*str) h += *str++;
return h;
}
```

Исключающее ИЛИ для строк переменной длины (размер таблицы равен 256)

Этот метод аналогичен аддитивному, но успешно различает схожие слова и анаграммы (аддитивный метод даст одно значение для XY и YX). Метод, как легко догадаться, заключается в том, что к элементам строки последовательно применяется операция “исключающее или”. В нижеследующем алгоритме добавляется случайная компонента, чтобы еще улучшить результат.

```
typedef unsigned char hashIndexType;
unsigned char Rand8[256];
hashIndexType hash(char *str) {
unsigned char h = 0;
while (*str) h = Rand8[h ^ *str++];
return h;
}
```

Здесь *Rand8* – таблица из 256 восьмибитовых случайных чисел. Их точный порядок не критичен. Корни этого метода лежат в криптографии; он оказался вполне эффективным.

Исключающее ИЛИ для строк переменной длины (размер таблицы ≤ 65536). Если мы хешируем строку дважды, мы получим хеш-значение для таблицы любой длины до 65536. Когда строка хешируется во второй раз, к первому символу прибавляется 1. Получаемые два 8-битовых числа объединяются в одно 16-битовое.

```
typedef unsigned short int hashIndexType;
unsigned char Rand8[256];
hashIndexType hash(char *str) {
hashIndexType h;
unsigned char h1, h2;
if (*str == 0) return 0;
h1 = *str; h2 = *str + 1;
str++;
while (*str) {
h1 = Rand8[h1 ^ *str];
h2 = Rand8[h2 ^ *str];
str++;
}
/* h is in range 0..65535 */
h = ((hashIndexType)h1 << 8)|(hashIndexType)h2;
/* use division method to scale */
return h % hashTableSize
}
```

Размер хеш-таблицы должен быть достаточно велик, чтобы в ней оставалось разумное число пустых мест. Как видно из таблицы 8.1, чем меньше таблица, тем больше среднее время поиска ключа в ней. Хеш-таблицу можно рассматривать как совокупность связанных списков. По мере того, как таблица растет, увеличивается количество списков и, соответственно, среднее число узлов в каждом списке уменьшается. Пусть количество элементов равно n. Если размер таблицы равен 1, то таблица вырождается в один список длины n. Если размер таблицы равен 2 и хеширование идеально, то нам придется иметь дело с двумя списками по $n/100$ элементов в каждом. Как мы видим в таблице 6.1, имеется значительная свобода в выборе длины таблицы.

Метод середины квадрата

При хешировании по методу середины квадрата ключ умножается сам на себя, а адрес получается отсечением битов или цифр от обоих концов произведения, которое выполняется до тех пор, пока число оставшихся битов или цифр не станет равным требуемой длине адреса. Во всех получаемых произведениях при этом должны использоваться одни и те же позиции.

Таблица 6.1

size	*time*	*size*	*time*
1	869	128	9
2	432	256	6
4	214	512	4
8	106	1024	4
16	54	2048	3
32	28	4096	3
64	15	8192	3

В качестве примера рассмотрим шестизначный ключ 113586. При возведении его в квадрат получается 12901779396. Если требуется четырехзначный адрес, могут быть выбраны позиции 5-8, дающие адрес 1779. Метод середины квадрата подвергался критике, но его применение к некоторым наборам ключей дает хорошие результаты[18]..

Метод свертывания

В методе свертывания ключ разбивается на части, каждая из которых имеет длину, равную длине требуемого адреса (кроме, возможно, последней части). Чтобы сформировать адрес, части затем складываются, при этом игнорируется перенос в старшем разряде. Если ключи представлены в двоичном виде, то вместо сложения может быть использована операция исключающего ИЛИ. Существуют различные вариации этого метода, которые лучше всего проиллюстрировать на конкретном примере ключа 187249653. В методе свертывания со сдвигом складываются 187, 249 и 653, при этом получается адрес 89.

В методе граничного свертывания инвертируются цифры в крайних частях ключа. Таким образом, в нашем примере складываются числа 781. 249 и 356, что дает адрес 386. Свертывание является функцией хеширования, удобной для сжатия многословных ключей и последующего перехода к другим функциям хеширования.

Метод «Преобразование системы счисления»

По этому методу хеширования делается попытка получить случайное распределение ключей по адресам в адресном пространстве. Ключ, представленный в системе счисления q (q обычно равно 2 или 10), рассматривается как число в системе счисления p, где p больше q, причем p и q – взаимно простые. Это число из системы счисления с основанием p переводится в систему счисления с основанием q, и адрес формируется путем выбора правых цифр (или битов) нового числа или применением метода деления.

Например, ключ 530476_{10}, рассматриваемый как 530476_{11}, переводится в десятичную систему счисления с помощью следующих вычислений:

$530476_{11} = 5 * 11^5 + 3 * 11^4 + 4 * 11^2 + 7 * 11 + 6 = 849745_{10}$.

8.3. Динамическое хеширование

Описанные выше методы хеширования являются статическими, т.е. сначала выделяется некая хеш-таблица, под ее размер подбираются константы для хеш-функции. К сожалению, это не подходит для задач, в которых размер базы данных меняется часто и значительно. По мере роста базы данных можно:

- пользоваться изначальной хеш-функцией, теряя производительность из-за роста коллизий;
- выбрать хеш-функцию «с запасом», что повлечет неоправданные потери дискового пространства;
- периодически менять функцию, пересчитывать все адреса.

Это отнимает очень много ресурсов и выводит из строя базу на некоторое время. Существует техника, позволяющая динамически менять размер хеш-структуры. Это – динамическое хеширование. Хеш-функция генерирует так называемый псевдоключ (pseudokey), который используется лишь частично для доступа к элементу. Другими словами, генерируется достаточно длинная битовая последовательность, которая должна быть достаточна для адресации всех потенциально возможных элементов. В то время, как при статическом хешировании потребовалась бы очень большая таблица (которая обычно хранится в оперативной памяти для ускорения доступа), здесь размер занятой памяти прямо пропорционален количеству элементов в базе данных. Каждая запись в таблице хранится не отдельно, а в каком-то блоке (bucket). Эти блоки совпадают с физическими блоками на устройстве хранения данных. Если в блоке нет больше места, чтобы вместить запись, то блок делится на два, а на его место ставится указатель на два новых блока.

Задача состоит в том, чтобы построить бинарное дерево, на концах ветвей которого были бы указатели на блоки, а навигация осуществлялась бы на основе псевдоключа. Узлы дерева могут быть двух видов: узлы, которые показывают на другие узлы или узлы, которые показывают на блоки. Например, пусть узел имеет такой вид, если он показывает на блок:

Zero	Null
Bucket	Указатель
One	Null

Если же он будет показывать на два других узла, то он будет иметь такой вид:

Zero	Адрес a
Bucket	Null
One	Адрес b

Вначале имеется только указатель на динамически выделенный пустой блок. При добавлении элемента вычисляется псевдоключ, и его биты поочередно используются для определения местоположения блока. Например, элементы с псевдоключами 00… будут помещены в блок A, а 01… – в блок B. Когда A будет переполнен, он будет разбит таким образом, что элементы 000… и 001… будут размещены в разных блоках (рис. 6.2).

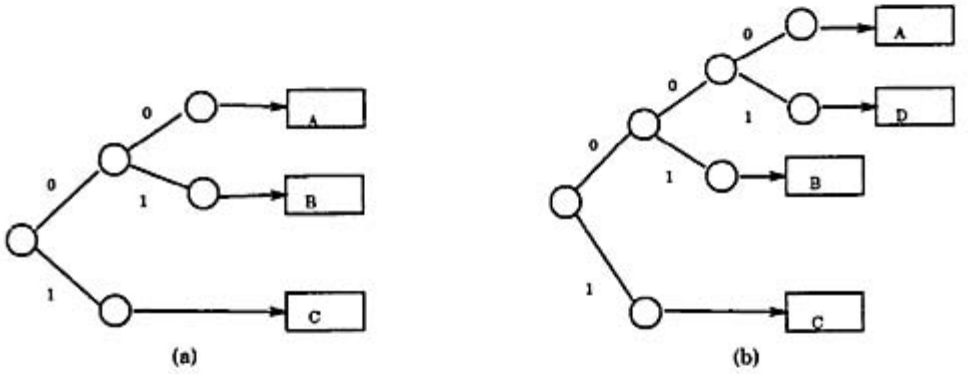


Рис. 8.2. Динамическое хеширование

8.4. Расширяемое хеширование (extendible hashing)

Расширяемое хеширование близко к динамическому. Этот метод также предусматривает изменение размеров блоков по мере роста базы данных, но это компенсируется оптимальным использованием места. Т.к. за один раз разбивается не более одного блока, накладные расходы достаточно малы .

Вместо бинарного дерева расширяемое хеширование предусматривает список, элементы которого ссылаются на блоки. Сами же элементы адресуются по некоторому количеству i битов псевдоключа (рис. 8.3.). При поиске берется i битов псевдоключа и через список (directory) находится адрес искомого блока. Добавление элементов производится сложнее. Сначала выполняется процедура,

аналогичная поиску. Если блок неполон, добавляется запись в него и в базу данных. Если блок заполнен, он разбивается на два, записи перераспределяются по описанному выше алгоритму. В этом случае возможно увеличение числа бит, необходимых для адресации. В этом случае размер списка удваивается и каждому вновь созданному элементу присваивается указатель, который содержит его родитель. Таким образом, возможна ситуация, когда несколько элементов показывают на один и тот же блок. Следует заметить, что за одну операцию вставки пересчитываются значения не более, чем одного блока. Удаление производится по такому же алгоритму, только наоборот. Блоки, соответственно, могут быть склеены, а список – уменьшен в два раза.

Итак, основным достоинством расширяемого хеширования является высокая эффективность, которая не падает при увеличении размера базы данных. Кроме этого, разумно расходуется место на устройстве хранения данных, т.к. блоки выделяются только под реально существующие данные, а список указателей на блоки имеет размеры, минимально необходимые для адресации данного количества блоков. За эти преимущества разработчик расплачивается дополнительным усложнением программного кода.

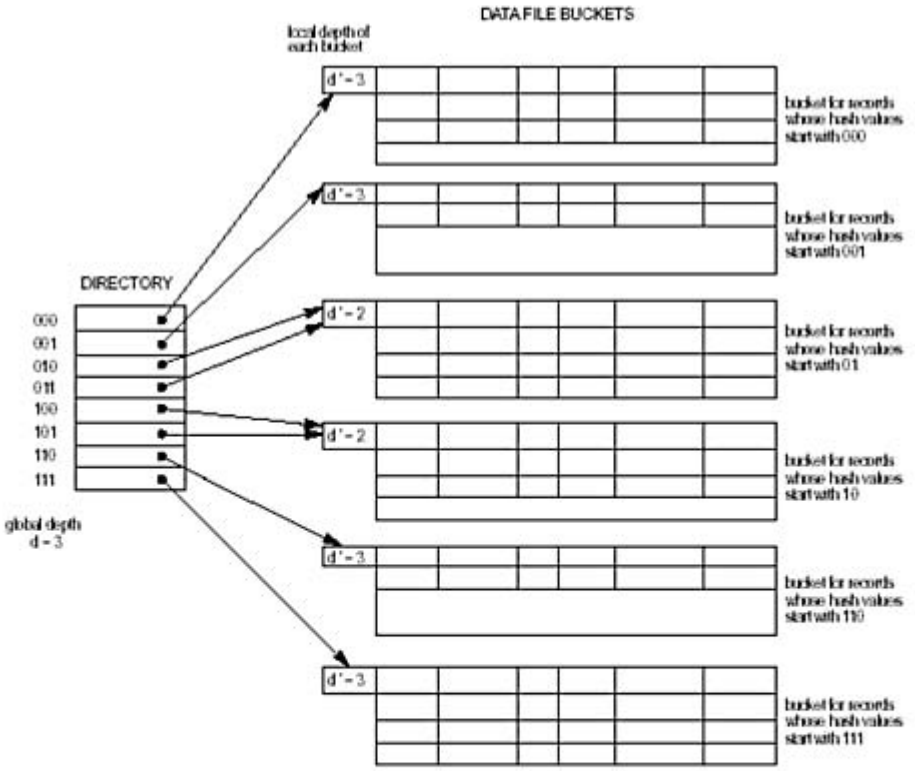


Рис. 8.3. Расширяемое хеширование

8.5. Функции, сохраняющие порядок ключей (Order preserving hash functions)

Существует класс хеш-функций, которые сохраняют порядок ключей. Другими словами, выполняется:

$$K1 < K2 \rightarrow h(K1) < h(K2)$$

Эти функции полезны для сортировки, которая не потребует никакой дополнительной работы. Другими словами, мы избежим множества сравнений, т.к. для того, чтобы отсортировать объекты по возрастанию достаточно просто линейно просканировать хеш-таблицу.

В принципе, всегда можно создать такую функцию, при условии, что хеш-таблица больше, чем пространство ключей. Однако, задача поиска правильной хеш-функции нетривиальна. Разумеется, она очень сильно зависит от конкретной задачи. Кроме того, подобное ограничение на хеш-функцию может пагубно сказаться на ее производительности. Поэтому часто прибегают к индексированию вместо поиска подобной хеш-функции, если только заранее неизвестно, что операция последовательной выборки будет частой.

8.6. Минимальное идеальное хеширование

Как уже упоминалось выше, идеальная хеш-функция должна быстро работать и минимизировать число коллизий. Назовем такую функцию идеальной (perfect hash function) . С такой функцией можно было бы не пользоваться механизмом разрешения коллизий, т.к. каждый запрос был бы удачным. Но можно наложить еще одно условие: хеш-функция должна заполнять хеш-таблицу без пробелов. Такая функция будет называться минимальной идеальной хеш-функцией. Это идеальный случай с точки зрения потребления памяти и скорости работы. Очевидно, что поиск таких функций – очень нетривиальная задача. Один из алгоритмов для поиска идеальных хеш-функций был предложен Р. Чичелли . Рассмотрим набор некоторых слов, для которых надо составить минимальную идеальную хеш-функцию. Пусть это будут некоторые ключевые слова языка C++. Пусть это будет какая-то функция, которая зависит от некоего численного кода каждого символа, его позиции и длины. Тогда задача создания функции сведется к созданию таблицы кодов символов, таких, чтобы функция была минимальной и идеальной. Алгоритм очень прост, но занимает очень много времени для работы. Производится полный перебор всех значений в таблице с откатом назад в случае необходимости, с целью подобрать все значения так, чтобы не было коллизий. Если взять для простоты функцию, которая складывает коды первого и последнего символа с длиной слова (да, среди слов умышленно нет таких, которые дают коллизию), то алгоритм дает следующий результат.

Таблица 6.2

Буква	Код	Буква	Код	Буква	Код	Буква	Код	Буква	Код
a	-5	e	4	i	2	n	-1	t	10
b	-8	f	12	k	31	o	22	u	26
c	-7	g	-7	l	29	r	5	v	19
d	-10	h	4	m	-4	s	7	w	2

Таблица 6.3

Слово	Хеш	Слово	Хеш	Слово	Хеш	Слово	Хеш
Auto	21	Double	0	Int	15	Struct	23
Break	28	Else	12	Long	26	Switch	17
Case	1	Enum	4	Register	18	Typedef	29
Char	2	Extern	9	Return	10	Union	30
Const	8	Float	27	Short	22	Unsigned	24
Continue	5	For	20	Signed	3	Void	13
Default	7	Goto	19	Sizeof	25	Volatile	31
Do	14	If	16	Static	6	While	11

8.7. Разрешение коллизий

Составление хеш-функции – это не вся работа, которую предстоит выполнить программисту, реализующему поиск на основе хеширования. Кроме этого, необходимо реализовать механизм разрешения коллизий. Как и с хеш-функциями существует несколько возможных вариантов, которые имеют свои достоинства и недостатки.

8.7.1. Метод цепочек

Метод цепочек – самый очевидный путь решения проблемы. В случае, когда элемент таблицы с индексом, который вернула хеш-функция, уже занят, к нему присоединяется связный список. Таким образом, если для нескольких различных значений ключа возвращается одинаковое значение хеш-функции, то по этому адресу находится указатель на связанный список, который содержит все значения. Поиск в этом списке осуществляется простым перебором, т.к. при грамотном выборе хеш-функции любой из списков оказывается достаточно коротким. Но даже здесь возможна дополнительная оптимизация. Дональд Кнут отмечает, что возможна сортировка списков по времени обращения к элементам. С другой стороны, для повышения быстродействия желательны большие размеры таблицы, но это приведет к ненужной трате памяти на заведомо пустые элементы. На рис.8.4. показана структура хеш-таблицы и образование цепочек при возникновении коллизий.

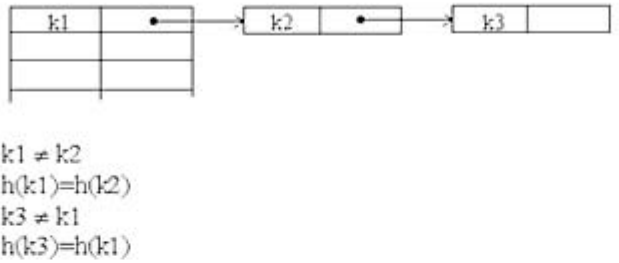


Рис. 8.4. Разрешение коллизий при добавлении элементов методом цепочек

8.7.2. Открытая адресация

Другой путь решения проблемы, связанной с коллизиями, состоит в том, чтобы полностью отказаться от ссылок, просто просматривая различные записи таблицы по порядку до тех пор, пока не будет найден ключ K или пустая позиция. Идея заключается в формулировании правила, согласно которому по данному ключу определяется «пробная последовательность», т.е. последовательность позиций таблицы, которые должны быть просмотрены при вставке или поиске ключа K. Если при поиске встречается пустая ячейка, то можно сделать вывод, что K в таблице отсутствует, т.к. эта ячейка была бы занята при вставке, т.к. алгоритм проходил ту же самую цепочку. Этот общий класс методов назван открытой адресацией.

Линейная адресация

Простейшая схема открытой адресации, известная как линейная адресация (линейное исследование, linear probing) использует циклическую последовательность проверок

h(K), h(K – 1), …, 0, M – 1, M – 2, …, h(K) + 1

и описывается следующим алгоритмом. Он выполняет поиск ключа K в таблице из M элементов. Если таблица не полна, а ключ отсутствует, он добавляется.

Ячейки таблицы обозначаются как TABLE[i], где $0 \le i < M$ и могут быть или пустыми, или занятыми. Вспомогательная переменная N используется для отслеживания количества занятых узлов. Она увеличивается на 1 при каждой вставке.

1. Установить i = h(K)
2. Если TABLE[i] пуст, то перейти к шагу 4, иначе, если по этому адресу искомый, алгоритм завершается.
3. Установить i = i – 1, если i < 0, то i = i +M. Вернуться к шагу 2.

4. Вставка, т.к. поиск оказался неудачным. Если N = M – 1, то алгоритм завершается по переполнению. Иначе увеличить N, пометить ячейку TABLE[i] как занятую и установить в нее значение ключа K.

Опыты показывают, что алгоритм хорошо работает в начале заполнения таблицы, однако по мере заполнения процесс замедляется, а длинные серии проб становятся все более частыми.

Квадратичная и произвольная адресация

Вместо постоянного изменения на единицу, как в случае с линейной адресацией, можно воспользоваться следующей формулой:

$h = h + a^2$,

где a – это номер попытки. Этот вид адресации достаточно быстр и предсказуем (он проходит всегда один и тот же путь по смещениям 1, 4, 9, 16, 25, 36 и т.д.). Чем больше коллизий в таблице, тем дольше этот путь. С одной стороны, этот метод дает хорошее распределение по таблице, а с другой занимает больше времени для просчета.

Произвольная адресация использует заранее сгенерированный список случайных чисел для получения последовательности. Это дает выигрыш в скорости, но несколько усложняет задачу программиста.

8.7.3. Адресация с двойным хешированием

Этот алгоритм выбора цепочки очень похож на алгоритм для линейной адресации, но он проверяет таблицу несколько иначе, используя две хеш-функции h1(K) и h2(K). Последняя должна порождать значения в интервале от 1 до M – 1, взаимно простые с M.

1. Установить i = h1(K).
2. Если TABLE[i] пуст, то перейти к шагу 6, иначе, если по этому адресу искомый, алгоритм завершается.
3. Установить c = h2(K).
4. Установить i = i – c, если i < 0, то i = i +M.
5. Если TABLE[i] пуст, то переход на шаг 6. Если искомое расположено по этому адресу, то алгоритм завершается, иначе возвращается на шаг 4.
6. Вставка. Если N = M – 1, то алгоритм завершается по переполнению. Иначе увеличить N, пометить ячейку TABLE[i] как занятую и установить в нее значение ключа K.

Очевидно, что этот вариант будет давать значительно более хорошее распределение и независимые друг от друга цепочки. Однако, он несколько медленнее из-за введения дополнительной функции.

Дональд Кнут предлагает несколько различных вариантов выбора дополнительной функции. Если M – простое число и h1(K) = K mod M, можно положить h2(K) = 1 + (K mod (M – 1)); однако, если M – 1 четно (другими словами,

M нечетно, что всегда выполняется для простых чисел), было бы лучше положить h2(K) = 1 + (K mod (M – 2)).

Здесь обе функции достаточно независимы. Гари Кнотт (Gary Knott) в 1968 предложил при простом M использовать следующую функцию:

h2(K) = 1, если h1(K) = 0 и h2(K) = M – h1(K) в противном случае (т.е. h1(K) > 0).

Этот метод выполняется быстрее повторного деления, но приводит к увеличению числа проб из-за повышения вероятности того, что два или несколько ключей пойдут по одному и тому же пути.

8.7.4. Удаление элементов хеш-таблицы

Многие программисты настолько слепо верят в алгоритмы, что даже не пытаются задумываться над тем, как они работают. Для них неприятным сюрпризом становится то, что очевидный способ удаления записей из хеш-таблицы не работает. Например, если удалить ключ, который находится в цепочке, по которой идет алгоритм поиска, использующий открытую адресацию, то все последующие элементы будут потеряны, так как алгоритм производит поиск до первого незанятого элемента.

Вообще говоря, обрабатывать удаление можно, помечая элемент как удаленный, а не как пустой. Таким образом, каждая ячейка в таблице будет содержать уже одно из трех значений: пустая, занятая, удаленная. При поиске удаленные элементы будут трактоваться как занятые, а при вставке – как пустые, соответственно.

Однако, очевидно, что такой метод работает только при редких удалениях, поскольку однажды занятая позиция больше никогда не сможет стать свободной, а, значит, после длинной последовательности вставок и удалений все свободные позиции исчезнут, а при неудачном поиске будет требоваться M проверок (где M, напомним, размер хеш-таблицы). Это будет достаточно долгий процесс, так как на каждом шаге №4 алгоритма поиска (см. раздел Адресация с двойным хешированием) будет проверяться значение переменной i.

Рассмотрим алгоритм удаления элемента i при линейной адресации:

1. Пометить TABLE[i] как пустую ячейку и установить j = i.
2. i = i – 1 или i = i + M, если i стало отрицательным.
3. Если TABLE[i] пуст, алгоритм завершается, т.к. нет больше элементов, о доступе к которым следует заботиться. В противном случае мы устанавливаем r = h(KEY[i]), где KEY[i] – ключ, который хранится в TABLE[i]. Это нам даст его первоначальный хеш-адрес. Если $i \le r < j$ или если $r < j < i$ либо $j < i \le r$ (другими словами, если r циклически лежит между этими двумя переменным, что говорит о том, что этот элемент находится в цепочке, звено которой мы удалили выше), вернуться на шаг 1.
4. Надо переместить запись TABLE[j] = TABLE[i] и вернуться на первый шаг.

Можно показать, что этот алгоритм не вызывает снижения производительности. Однако, корректность алгоритма сильно зависит от того факта, что используется линейное исследование хеш-таблицы, поэтому аналогичный алгоритм для двойного хеширования отсутствует.

Данный алгоритм позволяет перемещать некоторые элементы таблицы, что может оказаться нежелательно (например, если имеются ссылки извне на элементы хеш-таблицы). Другой подход к проблеме удаления основывается на адаптировании некоторых идей, использующихся при сборке мусора: можно хранить количество ссылок с каждым ключом, говорящим о том, как много других ключей сталкивается с ним. Тогда при обнулении счетчика можно преобразовывать такие ячейки в пустые.

8.7.5. Применение хеширования

Одно из побочных применений хеширования состоит в том, что оно создает своего рода слепок, «отпечаток пальца» для сообщения, текстовой строки, области памяти и т. п. Такой «отпечаток пальца» может стремиться как к «уникальности», так и к «похожести» (яркий пример слепка – контрольная сумма CRC). В этом качестве одной из важнейших областей применения является криптография. Здесь требования к хеш-функциям имеют свои особенности. Помимо скорости вычисления хеш-функции требуется значительно осложнить восстановление сообщения (ключа) по хеш-адресу. Соответственно необходимо затруднить нахождение другого сообщения с тем же хеш-адресом. При построении хеш-функции однонаправленного характера обычно используют функцию сжатия (выдает значение длины n при входных данных больше длины m и работает с несколькими входными блоками). При хешировании учитывается длина сообщения, чтобы исключить проблему появления одинаковых хеш-адресов для сообщений разной длины.

8.7.6. Хеширование паролей

Ниже предполагается, что для шифрования используется 128-битный ключ. Разумеется, это не более, чем конкретный пример. Хеширование паролей – метод, позволяющей пользователям запоминать не 128 байт, то есть 256 шестнадцатеричных цифр ключа, а некоторое осмысленное выражение, слово или последовательность символов, называющуюся паролем. Действительно, при разработке любого криптоалгоритма следует учитывать, что в половине случаев конечным пользователем системы является человек, а не автоматическая система. Это ставит вопрос о том, удобно, и вообще реально ли человеку запомнить 128-битный ключ (32 шестнадцатиричные цифры). На самом деле предел запоминаемости лежит на границе 8-12 подобных символов, а, следовательно, если мы будем заставлять пользователя оперировать именно ключом, тем самым мы практически вынудим его к записи ключа на каком-либо листке

бумаги или электронном носителе, например, в текстовом файле. Это, естественно, резко снижает защищенность системы.

Для решения этой проблемы были разработаны методы, преобразующие произносимую, осмысленную строку произвольной длины – пароль, в указанный ключ заранее заданной длины. В подавляющем большинстве случаев для этой операции используются так называемые хеш-функции. Хеш-функцией в данном случае называется такое математическое или алгоритмическое преобразование заданного блока данных, которое обладает следующими свойствами:

1. Хеш-функция имеет бесконечную область определения.
2. Хеш-функция имеет конечную область значений.
3. Она необратима.
4. Изменение входного потока информации на один бит меняет около половины всех бит выходного потока, то есть результата хеш-функции.

Эти свойства позволяют подавать на вход хеш-функции пароли, то есть текстовые строки произвольной длины на любом национальном языке и, ограничив область значений функции диапазоном 0..2N-1, где N – длина ключа в битах, получать на выходе достаточно равномерно распределенные по области значения блоки информации – ключи.

Вопросы для самоконтроля

1. Что означает хеширование ключа?
2. Что представляет собой хеш-таблица?
3. Назовите классические хеш-функции.
4. Что такое коллизия?
5. Назовите способы разрешения коллизий.
6. Зачем применяются динамические хеш-таблицы?
7. Какими свойствами обладает качественная хеш-функция?

ЗАКЛЮЧЕНИЕ

Содержание данного учебного пособия соответствует основным разделам программы по дисциплине «Структуры и алгоритмы компьютерной обработки данных», преподаваемой на кафедре «Программное обеспечение вычислительной техники» в Южно-Российском государственном политехническом университете (Новочеркасский политехнический институт – НПИ) им. М.И. Платова.

Целью изучения дисциплины «Структуры и алгоритмы компьютерной обработки данных» является освоение студентами основных положений теории структур данных, применяемых в программировании, операций над ними, способов их описания; умение применять набор типовых проектных решений по их конструированию, умение применять классические алгоритмы обработки данных (сортировка, поиск информации) для решения прикладных задач.

В пособии рассмотрены структуры данных, их представление и алгоритмы их обработки, без знания которых невозможно современное компьютерное программирование. Представлены наиболее известные и используемые из алгоритмов сортировки и поиска данных, методы проектирования линейных и древовидных структур данных. Подробно описаны алгоритмы сортировки и поиска данных, хеширования. Дано детальное описание некоторых алгоритмов в виде пошагового описания и в виде схем алгоритмов. Приведено множество примеров программ на языке C++.

Пособие позволяет освоить основные принципы организации данных в языках высокого уровня, типовые проектные решения по конструированию списковых (стеки, очереди) и древовидных структур данных, а также получить практические навыки программирования классических алгоритмов сортировки и поиска.

Материал данного пособия используется в курсах «Теория языков программирования», «Базы данных», «Системы искусственного интеллекта».

Предназначено для студентов вузов, обучающихся по специальности 23100062 «Программная инженерия» (квалификация «бакалавр»). Также может быть рекомендовано для студентов специальности 010503 «Математическое обеспечение и администрирование информационных систем» и специальностей направления 23010062 «Информатика и вычислительная техника» (квалификация «бакалавр») дневной и заочной форм обучения.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. *Вирт Н.* Алгоритмы и структуры данных. Новая версия для Оберона. М. : ДМК Пресс, 2010.
2. *Зубов В.С.* Справочник программиста. Базовые методы решения графовых задач и сортировки. М. : Филинъ, 1999.
3. *Иванченко А.Н., Мясникова Н.А.* Структуры и алгоритмы обработки данных : учеб. пособие для дистанционного обучения / ЮРГТУ (НПИ). Новочеркасск, 2002.
4. *Кнут Д.Э.* Искусство программирования для ЭВМ. В 3 т. Т. 1. Основные алгоритмы. М. : Вильямс, 2004.
5. *Кнут Д.Э.* Искусство программирования для ЭВМ. В 3 т. Т. 3. Сортировка и поиск. М. : Вильямс, 2003.
6. *Лэнгсам Й., Огенстайн М., Тэненбаум А.* Структуры данных для персональных ЭВМ. М. : Мир, 1989.
7. *Макконнелл Д.* Анализ алгоритмов: Активный обучающий подход : учеб. пособие ; пер. с англ. 3-е изд., доп. М. : Техносфера, 2009.
8. *Макконнелл С.* Совершенный код: Практическое руководство по разработке программного обеспечения ; пер. с англ. СПб. : Питер, 2007.
9. *Мэйерс С.* Эффективное использование C++. 55 верных советов улучшить структуру и код ваших программ. ДМК Пресс, 2006.
10. *Мясникова Н. А.* Структуры и алгоритмы обработки данных / ЮРГТУ (НПИ). Новочеркасск, 2009.
11. *Нагибаев В.Н.* Основы алгоритмизации и программирования на языке C++ : учеб. пособие. М. : МИИТ, 2006.
12. *Окулов С.М.* Основы программирования. М. : БИНОМ. Лаборатория знаний, 2010.
13. *Павловская Т.А.* C/C++ Программирование на языке высокого уровня. СПб. : Питер, 2008.
14. *Подбельский В.В.* Стандартный C++ : учеб. пособие для вузов. М. : Финансы и статистика, 2008.
15. *Потопахин В.В.* Искусство алгоритмизации. М. : ДМК Пресс, 2011.
16. *Седжвик Р.* Фундаментальные алгоритмы на C. В 5 ч. ; пер.с англ. М. ; СПб. ; Киев : Dia-Soft, 2003.
17. *Страуструп Б.* Язык программирования C++ : специальное издание ; пер с англ. под ред. Ф. Андреева и А. Ушакова. М. : Бином-Пресс, 2005.
18. *Трамбле Ж., Соренсон П.* Введение в структуры данных ; пер. с англ. (В.И. Бринккер и др.) под ред. А.Е. Костина, В.Ф. Шаньгина. М. : Машиностроение, 1982.

19. *Франка П.* C++ : учеб. пособие ; пер. с англ. СПб. : Питер, 2006.

20. *Фридман А.Л.* Язык программирования C++: Курс лекций : учеб. пособие. 2-е изд. испр. М. : Интернет-Университет Информационных технологий, 2004.

21. *Хусаинов Б.С.* Структуры и алгоритмы обработки данных: примеры на языке Си : учеб. пособие. М. : Финансы и статистика, 2004.

22. *Шилдт Г.* Искусство программирования на C++. СПб. : БХВ-Петербург, 2005.

23. *Шилдт Г.* Полный справочник по C++ ; пер. с англ. 4-е изд. М. ; СПб. ; Киев : Вильямс, 2006.

24. *Шилдт Г.* Самоучитель C++ ; пер. с англ. 3-е изд. перераб. и доп. СПб. : БХВ-Петербург, 2009.

25. *Liskov B., Zilles S.* Programming with abstract data types. SIGPlan Notices. 1974. Vol. 9, no. 4.