

ФЕДЕРАЛЬНОЕ АГЕНТСТВО СВЯЗИ
Федеральное государственное образовательное бюджетное учреждение
высшего профессионального образования
«ПОВОЛЖСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ТЕЛЕКОММУНИКАЦИЙ И ИНФОРМАТИКИ»

Кафедра информационных систем и технологий

П. А. Назаренко

АЛГОРИТМЫ И СТРУКТУРЫ ДАНЫХ

Учебное пособие

Самара
2015

УДК 004.65+ 004.43
ББК 32.973
Н 19

Рекомендовано к изданию методическим советом ПГУТИ,
протокол № 26, от 05.05.2015 г.

Рецензент:

профессор кафедры ИВТ ПГУТИ,
к. т. н., доцент Алексеев А. П.

Назаренко, П. А.

Н **Алгоритмы и структуры данных:** учебное пособие / П. А. Назаренко – Самара : ПГУТИ, 2015. – 196 с.

Учебное пособие «Алгоритмы и структуры данных» содержит теоретический материал по основным структурам данных и их практической реализации в языках программирования Си/Си++ и Паскаль. Приведена классификация структур данных. Рассмотрены основные алгоритмы обработки структур данных, включая создание и удаление элементов, прохождение, сортировку и поиск, с их реализациями на языках программирования Си/Си++ и Паскаль.

Учебное пособие разработано в соответствии с Федеральным государственным образовательным стандартом высшего профессионального образования по направлению подготовки 02.03.03 – «Математическое обеспечение и администрирование информационных систем» и предназначено для студентов третьего курса факультета информационных систем и технологий, а также для студентов других специальностей, изучающих и использующих структуры данных и алгоритмы их обработки, преподавателей, магистрантов и аспирантов.

© Назаренко П. А., 2015

Содержание

Введение	5
1. Понятие о структурах данных	8
1.1. Основные определения	8
1.2. Уровни структур данных	9
1.3. Классификация структур данных	10
1.4. Информация и ее представление в памяти ЭВМ	17
2. Простые структуры и типы данных	20
2.1. Понятие о типах данных	20
2.2. Перечисляемый тип данных	22
2.3. Стандартные типы данных	22
2.4. Указатели	30
2.5. Алгоритмы обработки простых структур данных ..	38
3. Линейные статические структуры данных	41
3.1. Массивы	41
3.2. Динамические массивы	42
3.3. Многомерные массивы	43
3.4. Связь массивов с указателями	46
3.5. Строки	49
3.6. Массивы указателей	50
3.7. Интерпретация составных описателей	51
3.8. Алгоритмы обработки статических линейных структур	53
4. Ссылки. Временные структуры данных	56
5. Составные типы данных	59
5.1. Структуры	59
5.2. Битовые поля	62
5.3. Объединения	63
6. Файлы	67
7. Очереди	72
7.1. Кольцевая очередь	76
7.2. Приоритетная очередь	80
7.3. Дек	81
8. Стеки	83

9. Связные списки	90
9.1. Линейный односвязный список	92
9.2. Линейный двусвязный список	100
9.3. Операции с двусвязным списком	101
9.4. Кольцевые списки	103
9.5. Процедуры работы с двусвязным кольцевым списком на языке Си++	106
9.6. Многосвязные списки	108
10. Древовидные структуры данных	111
10.1. Классификация	112
10.2. Двоичные деревья поиска	113
10.3. Операции с деревьями	117
10.4. Сбалансированные деревья	135
10.5. Многоключевые деревья	143
11. Элементы теории графов	145
11.1. Способы представления графов	145
11.2. Алгоритмы на графах	154
12. Поиск	158
12.1. Последовательный поиск	158
12.2. Двоичный поиск	160
12.3. Специальные виды поиска	163
13. Сортировка	166
13.1. Классификация алгоритмов сортировки	166
13.2. Пузырьковая сортировка	169
13.3. Сортировка отбором	174
13.4. Сортировка вставками	177
13.5. Алгоритм Шелла	180
13.6. Алгоритм быстрой сортировки	184
13.7. Параллельная сортировка Бэтчера	188
Заключение	192
Библиографический список	193

Введение

Структуры данных – необходимые компоненты любой программы или программного комплекса. Поэтому знание теории структур данных и, в частности, методов представления данных на логическом и машинном уровнях, а также допустимых операций над различными структурами, необходимо для глубокого изучения и уяснения таких разделов, как автоматизированные системы управления, компиляторы языков программирования, операционные системы, а также системы программного имитационного моделирования, управления базами данных, искусственного интеллекта и т.д.

Выбор структур данных является одним из важных этапов разработки программ и от правильности этого выбора зависит эффективность программы, трудоёмкость её написания и время решения программой тех задач, ради которых она создавалась. Это же справедливо и для алгоритмов обработки данных и их структур. Появление в составе современных языков программирования библиотек и классов структур данных, например, векторов, списков, различных видов деревьев, карт и т.п. не отменяет необходимости знания высококвалифицированными специалистами тонкостей использования этих структур данных и алгоритмов их обработки. Учебное пособие по дисциплине «Алгоритмы и структуры данных» предназначено для того, чтобы помочь студентам полнее усвоить соответствующий лекционный курс, подготовиться к выполнению лабораторных работ по этой дисциплине и заложить основы дальнейшего более глубокого изучения конкретных алгоритмов обработки данных и вопросов практического применения специфических структур данных.

Настоящее учебное пособие предназначено для студентов специальностей «Технология программирования», «Информационные системы и технологии», «Разработка программно-информационных систем», «Управление и информатика в технических системах» и других. Предполагается, что студенты, в зависимости от специальности, изучили дисциплины «Программирование», «Дискретная математика», «Вычислительная математика», «Технология программирования». Отдельные разделы пособия имеют связь с курсом «Базы данных».

В процессе изучения курса студенты познакомятся с основными типами структур данных – *списковыми, древовидными, сетевыми, файловыми*; основными алгоритмами обработки структур данных – *пополнением, удалением, поиском, прохождением, упорядочением*; будут получены навыки разработки алгоритмов обработки структур данных. Текст учебного пособия разбит на 13 разделов, каждый из которых посвящён отдельной группе структур данных с операциями их обработки, или группе алгоритмов, например, поиску или сортировке. К сожалению, ограниченный объём издания не позволяет подробно рассмотреть все алгоритмы обработки данных и их структур в их огромном многообразии, но базовые алгоритмы, составляющие основу этой дисциплины, рассматриваются в обязательном порядке.

Структуры данных и алгоритмы обработки данных и их структур рассматриваются в большом количестве книг и других источников, перечисленных в библиографическом указателе, все из которых в той или иной степени были использованы при составлении учебного пособия. Несмотря на то, что отдельные книги были изданы около 40 лет назад, изложенная в них теория структур данных или алгоритмов их обработки до сих пор не потеряла своего значения. Некоторые из авторов этих книг являются лауреатами премии Алана Тьюринга, так же как и создатели отдельных рассматриваемых в пособии алгоритмов.

Библиографический список не претендует на абсолютную полноту, да это и невозможно, учитывая, что книги по алгоритмам и структурам данных публикуются или переиздаются практически каждый год. Для углублённого изучения структур данных и алгоритмов предлагается использовать какие-либо из этих источников. Можно, например, порекомендовать книги Томаса Кормена с соавторами [1], Роберта Седжвика [14 – 18] и подобные [12, 20]. Из многочисленных Интернет-источников выбраны наиболее заслуживающие доверия.

При составлении учебного пособия его автор опирался на требования федерального государственного образовательного стандарта по дисциплине «Структуры и алгоритмы компьютерной обработки данных» для специальности «Технология программирования».

Изучение курса «Алгоритмы и структуры данных» не ориентировано на применение какого-либо конкретного языка программирования, но в тексте учебного пособия приводятся фрагменты программ и примеры процедур на языках Си++ и Паскаль. Язык Си++ выбран как компактный и одновременно достаточно выразительный, при этом не используются объектно-ориентированные механизмы этого языка, т.к. этим вопросам посвящён отдельный курс. В то же время это пособие не является учебником или справочником по языку Си++ (возможно, за исключением отдельных вопросов, например, стандартных типов данных, указателей, массивов и ссылок). Заинтересованный читатель сможет найти все необходимые материалы в имеющихся многочисленных источниках.

1. Понятие о структурах данных

1.1. Основные определения

Под программным обеспечением ЭВМ понимают совокупность программ, предназначенных как для поддержания должного функционирования ЭВМ, так и для выполнения ею полезных функциональных задач некоторой прикладной области. Когда употребляют термин «программа», подразумевают не только последовательность операторов некоторого языка программирования, но и набор различных информационных объектов, над которыми выполняют те или иные действия операторы программы. Такие информационные объекты программы называют *данными*.

Существует по крайней мере несколько не противоречащих друг другу определений того, что такое данные. В соответствии с международным стандартом ISO, в наиболее общем виде *данные* – это представлений фактов, понятий, инструкций, идей или какой-либо другой информации в формализованном виде, приемлемом для обработки, интерпретации, общения или передачи как человеком, так и техническими средствами, при помощи некоторых процессов или алгоритмов.

Данные – неперенный атрибут любой программы. Ими могут быть отдельные биты, последовательности независимых битов, числа в разных формах представления (с фиксированной или плавающей точкой, обычной или удвоенной точностью и т.д.), байты и группы независимых байтов, представляющие символы в различных системах кодирования, массивы чисел, информация хранимая в памяти вычислительной машины в форме связанных списков, а также информация на устройствах внешней памяти, организованная в виде отдельных файлов и систем взаимосвязанных файлов. Перечисленные примеры иллюстрируют разный уровень сложности, или организованности данных. Характер этой организованности и является одним из воплощений понятия «*структура данных*».

Термин «*структуры данных*» может употребляться по крайней мере в двух разных значениях. Во-первых, *структура данных* – это логическая или математическая модель организации данных. Фактически, структура данных может рассматри-

ваться как представление именно этих данных в памяти ЭВМ (*физическая структура*), и является общим свойством любого информационного объекта, с которым имеет дело какая-либо программа.

Во-вторых, структура данных – это собственно реализация *логического* понятия данных, объект (большой или меньший) в программе и в памяти ЭВМ. Это может быть отдельная переменная, массив или более сложный программный объект, например, список, дерево и т.п. Важно помнить, что любая структура данных размещается в памяти ЭВМ (в первую очередь, в оперативной памяти), занимает некоторое, возможно весьма большое, количество ячеек этой памяти и характеризуется начальным адресом своего размещения. Для описания этой ситуации используется понятие «*место в памяти*».

Очень часто структуры данных рассматриваются во взаимосвязи с алгоритмами обработки данных. Одно из распространённых определений *алгоритма* – однозначно определённая конечная последовательность команд (или инструкций, предписаний), задающая порядок выполнения операций для решения задачи. Алгоритмы обработки данных и их структур можно разделить на две группы – алгоритмы обработки собственно данных, например, подавление шумов, преобразование Фурье, цифровая фильтрация и т.п. Это именно те алгоритмы, для реализации которых и создаются программы, поэтому такие алгоритмы можно назвать *основными*. Вторая группа – это алгоритмы обработки именно структур данных, те, которые уже были перечислены во введении – пополнение, поиск, упорядочение (сортировка), просмотр (прохождение) и т.п. Они занимают подчинённое положение по отношению к основным алгоритмам, поэтому могут быть названы *вспомогательными*, но для собственно структур данных такие алгоритмы являются первостепенными.

1.2. Уровни структур данных

Структуры данных могут рассматриваться на разных уровнях. Применяются три уровня структур данных:

1. содержательный,
2. логический,

3. физический.

На *содержательном уровне* структур данных исследуются конкретные объекты обработки, их свойства и отношения между объектами. На этом уровне важны не только значения, но и смысл данных.

На *логическом* или *абстрактном* (логические структуры) *уровне* структура данных считается машинно-независимым логическим понятием, и выделяются следующие задачи: определение массивов данных как объектов исследования, выделение состава массива, определение структуры данных по заданным требованиям, разработка количественных методов оценки эффективности различных видов структур данных.

Физический уровень (физическая структура). На этом уровне рассматриваются среда хранения данных (память ЭВМ) и представление в ней значений (ячейки, разряды ячеек, их адреса и взаимное расположение значений.), т.е. отображение данных в памяти ЭВМ. В общем случае между логической и соответствующей ей физической структурами существует различие, степень которого зависит от самой структуры и особенностей той физической среды, в которой она должна быть отражена.

Различия между уровнями структур данных могут быть пояснены следующим примером. Пусть требуется выполнить ввод данных в память ЭВМ, используя некоторую буферную область (содержательный уровень). С этой целью удобно использовать кольцевую очередь (логический уровень), которая в работающей программе может быть реализована при помощи одномерного массива как непрерывного блока в памяти или при помощи связанного списка, допускающего разнесённое размещение в памяти своих элементов (физический уровень).

1.3. Классификация структур данных

Структуры данных можно классифицировать по нескольким различным признакам. Рассмотрим по крайней мере два варианта классификации. По одному из них структуры данных разделяются на достаточно большое количество категорий. Рассмотрим эти категории с примерами, соответствующими каждой из них.

Наиболее простым и понятным критерием классификации является сложность структур данных.

По уровню сложности структуры данных разделяются на:

1. *простые* структуры – обычные переменные или константы стандартных для языков программирования типов, а также динамические переменные этих же типов;

2. *наборы однотипных данных* – массивы одномерные (или векторы), двумерные (матрицы) и многомерные;

3. *составные* структуры, отличные от массивов – записи и объекты классов и им подобные структуры;

4. *динамические структуры с внутренними связями* – связные списки, деревья, графы.

С точки зрения *архитектуры* можно выделить:

1. *линейные* структуры – одномерные массивы (или векторы), линейные списки, линейные очереди, стеки;

2. *прямоугольные* структуры – двумерные (матрицы) и многомерные массивы;

3. *кольцевые* структуры – кольцевые списки, кольцевые очереди, некоторые реализации графов;

4. *ветвящиеся* структуры – деревья различных видов, некоторые реализации графов;

5. *сетевые* структуры – графы.

По способу создания структуры данных можно разделить на

1. *обычные* – переменные стандартных типов, обычные (т.е. не динамические) массивы, записи и т.п.;

2. *динамические* (создаваемые и разрушаемые с помощью специальных операций или процедур динамического выделения и освобождения памяти) – динамические массивы, динамические переменные, связные списки, деревья.

В зависимости от *наличия* или отсутствия *связей* между элементами структуры данных различают:

1. *несвязные* структуры – векторы, массивы, строки, стеки, очереди;

2. *связные* – списки, деревья, графы.

В зависимости от *постоянства* во время работы программы различают:

1. *статические* (неизменяющиеся) структуры – переменные различных типов, записи, массивы, в том числе динамиче-

ские, а также списки, деревья и графы в тех случаях, когда они являются фиксированными и могут быть построены на основе, например, массива.

2. *динамические* (изменяющиеся) – списки, деревья, очереди, стеки, в общем случае графы.

Здесь следует обратить внимание на то, что термин «*динамические*» употребляется в разных смыслах, и структуры, динамические по способу создания, могут быть статическими по своему поведению. Структуры, динамические по поведению, как правило, оказываются динамическими и по способу создания. Также следует обратить внимание на то, что термин «статические» используется и в некоторых языках программирования, например, в Си и Си++, и там его смысл в значительной степени отличается от подразумеваемого в приведённой классификации.

По месту размещения в памяти ЭВМ:

1. существующие в оперативной памяти (или *внутренние*) – ими могут быть все ранее рассмотренные примеры структур данных. Кроме того, такие структуры могут размещаться:

- в регистрах процессора – *регистровые*, обычно переменные стандартных типов, размер которых не превышает разрядности процессора;
- в стеке – *локальные* (в некоторых языках принято обозначение «*автоматические*») – любые не статические и не динамические структуры;
- в свободной (динамически выделяемой) памяти или «куче» – глобальные, локальные статические (для языков Си и Си++) и все динамические структуры.

2. хранящиеся на *внешних* носителях (внешних запоминающих устройствах) – файлы и системы файлов.

Следующий критерий классификации справедлив только для некоторых языков программирования, а именно тех, которые допускают несовпадение типов данных в операциях и неявное преобразование типов с созданием временных объектов. Для этого критерия достаточно сложно подобрать название в одном или нескольких словах, поэтому сразу приведём его варианты:

1. данные, создаваемые самим программистом (независимо от способа и времени создания), в англоязычной литературе по программированию часто обозначаемые словом *persistent* – «*ус-*

тойчивый» («*постоянный*», «*постоянно существующий*»). Таким являются структуры всех данных, явно созданных программистом при написании программы;

2. создаваемые и разрушаемые непосредственно во время работы программы без участия программиста, обычно во вспомогательных целях, например, при несовпадении типов данных в каких-либо операциях. Для них используется термин *transient* – «временный», «преходящий». Их действительно можно коротко назвать временными, поскольку они обычно уничтожаются сразу после их использования. Как правило, такие структуры данных являются безымянными. В некоторых языках программирования, например, Си++, такими данными могут быть данные любых типов. О временных структурах часто говорят, что они создаются не программистом, а самим компилятором (ещё один вариант – компилятор строит код программы так, чтобы эти данные были созданы и разрушены в нужные моменты работы программы).

Приведём ещё один вариант классификации структур данных [9], совпадающий до некоторой степени с рассмотренным и представленный на рисунках 1.1 – 1.3. При этом необходимо помнить, что любая классификация является достаточно условной и может не во всех случаях отражать реальность. Например, на рисунке 1.3 такие структуры данных как очередь, стек и дек отнесены к односвязным линейным структурам. В реальности они могут быть и двусвязными, и кольцевыми, и даже вообще несвязными.

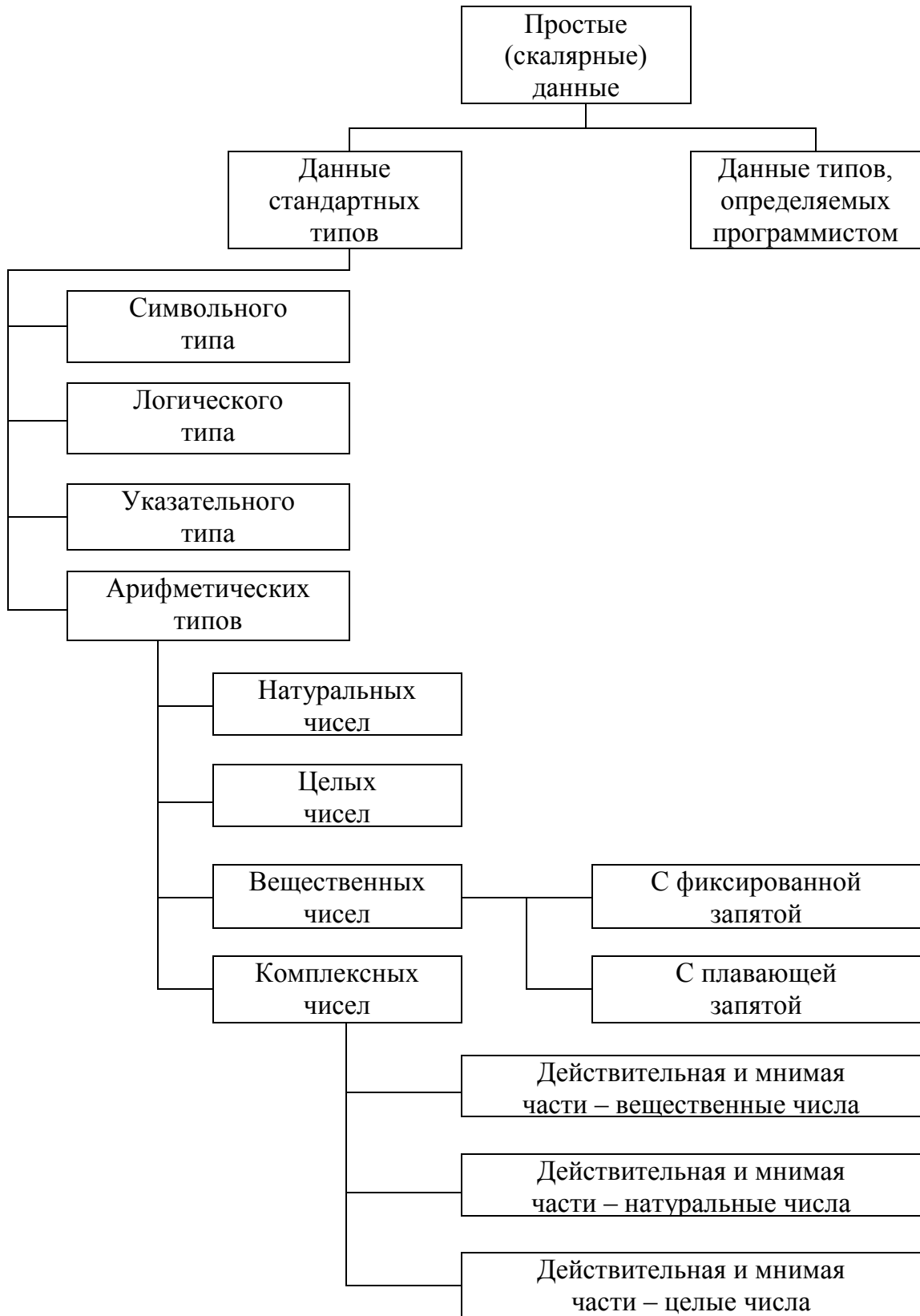


Рис. 1.1 – Простые статические структуры данных

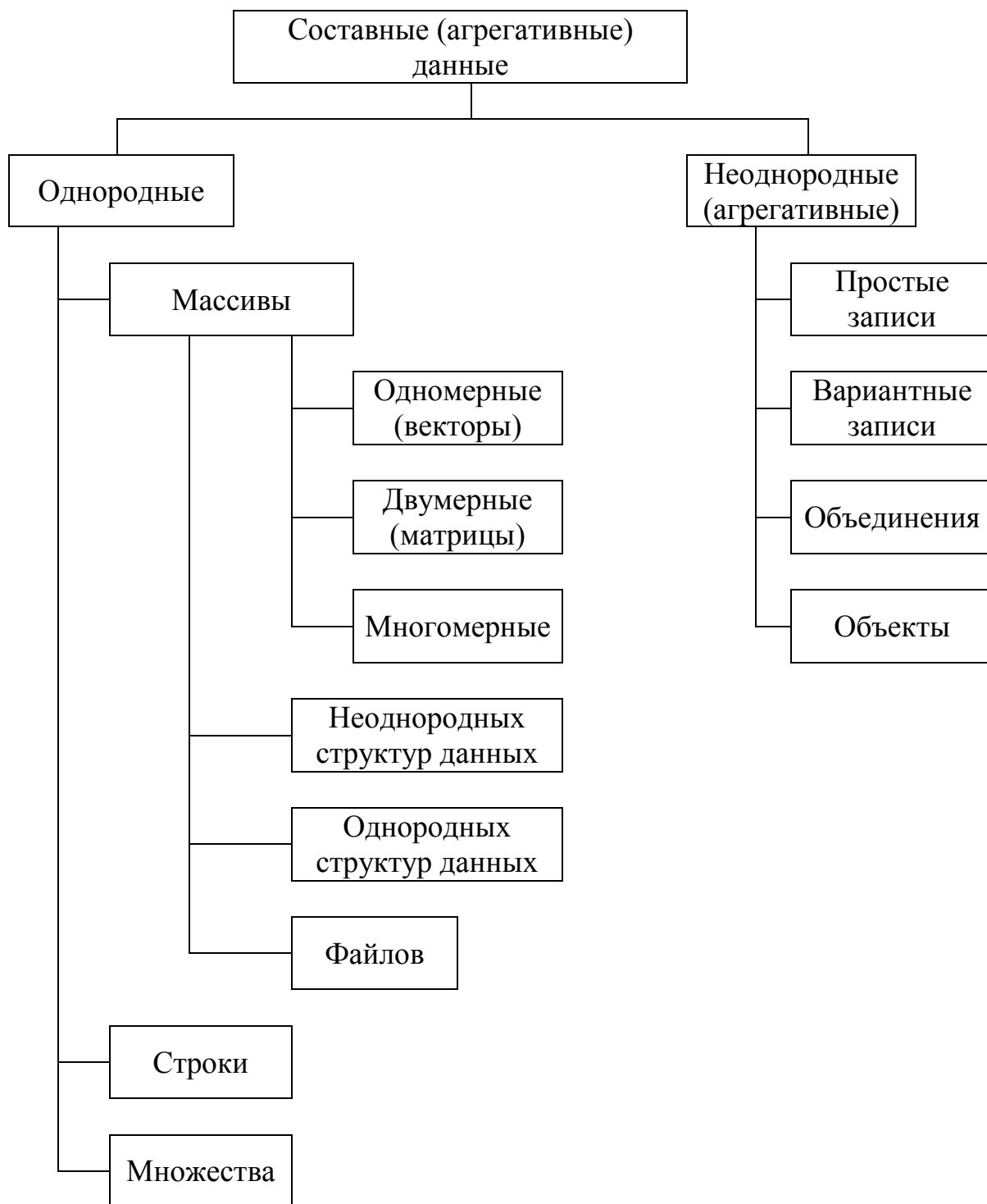


Рис. 1.2 – Составные статические структуры данных

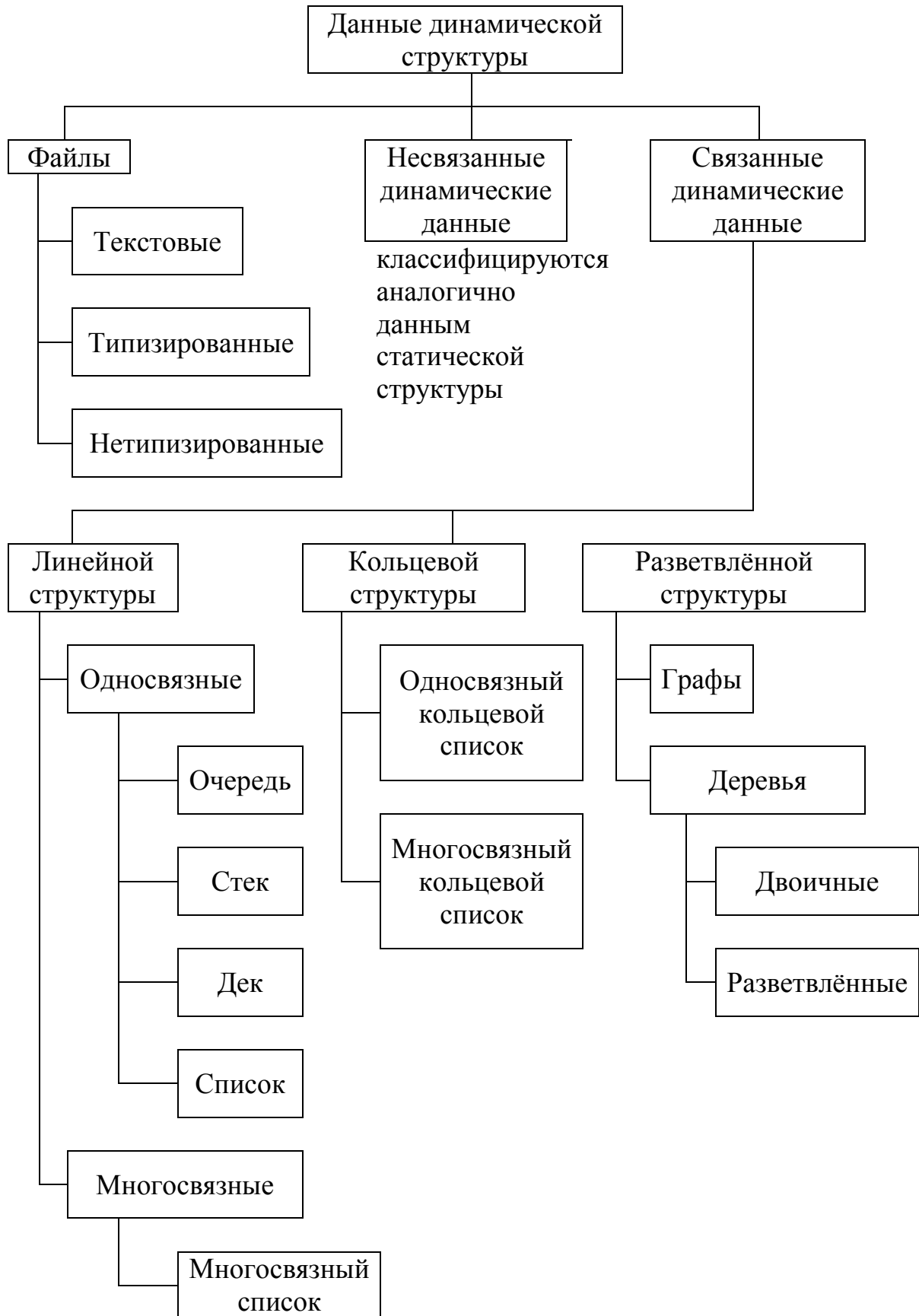


Рис. 1.3 – Динамические структуры данных

1.4. Информация и ее представление в памяти ЭВМ

Базовой единицей информации является бит, который может принимать одно из двух взаимоисключающих значений. Для представления двух возможных состояний некоторого бита используются двоичные цифры – нуль и единица [слово «бит» (английское bit) есть сокращение от английских слов «двоичная цифра» (binary digit)]. Более крупной единицей информации является байт. Группы смежных битов объединяются в поле. Поле, состоящее из 8 битов называется байтом, причем левый двоичный разряд имеет наибольший вес и считается старшим разрядом, а правый разряд – наименьший вес (младшим разрядом). Нумерация разрядов байтов осуществляется слева направо и начинается с нуля. Кроме 8 информационных битов байт может содержать дополнительный контрольный бит четности.

Группа смежных байтов образуют поле байтов, характеризующееся длиной поля – числом входящих в него байтов и адресом поля – адресом старшего, самого левого, байта в поле, т.е. байта с наименьшим адресом. В общем случае поле байтов может иметь произвольную длину и адрес. Для некоторых частных видов полей имеются специальные названия:

- полуслово (для поля, имеющего длину 2 байта),
- слово (4 байта) и
- двойное слово (8 байт).

Графическое соотношение между битом, байтом, полусловом, словом, двойным словом для 16-разрядных вычислительных систем (в настоящее время практически устаревших) показано на рисунке 1.4.



Рис. 1.4 – Байты и машинные слова

Возможно также четверное слово, состоящее из двух двойных слов. Размер слова может определяться разрядностью системы. В 32-разрядной системе слово будет состоять из 4-х байт, а полуслово – из 2-х и, таким образом, не будет совпадать с байтом.

Поле байтов длиной 1024 байт имеет специальное обозначение – 1Кбайт, кроме того поле длиной 1024x1024 байт обозначается через 1Мбайт, а поле длиной 1024x1024x1024 через 1Гбайт. Следует обратить внимание на то, что хотя здесь используются традиционные десятичные приставки «кило», «мега» и «гига» (а также, возможно, «тера»), они имеют значения не степеней числа 10 (10^3 , 10^6 и 10^9 соответственно), а значения степеней числа 2 (2^{10} (кибибайт), $2^{20}=2^{10} \times 2^{10}$ (мебибайт) и $2^{30} = 2^{10} \times 2^{10} \times 2^{10}$ (гибибайт)). Такое использование десятичных приставок в двоичной системе счисления иногда приводит к неоднозначному толкованию значений данных. Например, емкость накопителей на жестких дисках может быть обозначена именно в десятичной системе счисления, и тогда объём в 10 Гбайт будет означать 10.000.000.000 байт, а не 10.737.418.240 байт. Как видно, разница составляет более 700 миллионов байт!

Вопросы и задания для самоконтроля

- 1.1. Что означает понятие «структуры данных»?
- 1.2. По каким признакам классифицируются структуры данных?
- 1.3. На каких уровнях рассматриваются структуры данных?
- 1.4. Поясните различие между уровнями структур данных.
- 1.5. Как структуры данных различаются по сложности? Приведите примеры структур данных различной степени сложности.
- 1.6. Какие структуры данных обладают линейной архитектурой?
- 1.7. Как структуры данных различаются по архитектуре? Приведите примеры структур данных различных архитектур.
- 1.8. Какие структуры данных обладают прямоугольной архитектурой?

1.9. Что может означать термин «динамическая структура данных»?

1.10. Что означает термин «связная структура данных»?

1.11. Какие структуры данных являются связными?

1.12. Как структуры данных различаются по месту размещения в памяти?

1.13. К каким структурам данных относятся файлы?

1.14. Пусть имеется 32-разрядная вычислительная система, в которой действительное число занимает двойное машинное слово. Сколько байт занимает в памяти массив, содержащий 2К действительных чисел ($1К = 1024$)?

1.15. Сколько киббайт занимает блок памяти из 65536 од-нобайтовых ячеек?

2. Простые структуры и типы данных

2.1. Понятие о типах данных

Вычислительный процесс на ЭВМ реализуется с помощью программы и данных, которые программа использует или формирует. Но и сама программа представляет собой совокупность данных, и поэтому можно считать, что данные описывают любую информацию, с которой может работать ЭВМ. Все данные в общем случае характеризуются рядом признаков, или атрибутов, в том числе своим значением, смысл которого различен для разных данных. Независимо от содержания и сложности любые данные в памяти ЭВМ выражаются последовательностями двоичных рядов, или битов, а их значениями могут служить соответствующие двоичные числа. Данные, рассматриваемые в виде последовательности битов, имеют очень простую организацию. Однако 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 можно получить на основе понятия *типов данных*.

Тип некоторых данных определяется множеством значений этих данных и набором операций, которые можно выполнять над этими значениями в соответствии с их известными свойствами. Следовательно, тип данных специфицирует те операции, которые допустимы над соответствующими данными. На машинном уровне тип данных используется для выбора машинных команд, в выполнении которых участвуют такие данные. Тип данных, кроме того, используется при выборе представления данных в памяти, а также описывает *размер* программного объекта (т.е. сколько ячеек памяти он занимает), допустимый *диапазон* значений, которые может принимать этот объект, и *набор операций*, которые могут с ним выполняться. Фактически тип и определяет структуру данных.

Во многих случаях новые типы данных определяются с помощью ранее определенных типов данных. Значения такого нового типа обычно представляют собой совокупности значений компонент, относящихся к определенным ранее составляющим

типам, такие значения называются составными. Если имеется только один составляющий тип, т.е. все компоненты относятся к одному типу, то он называется базовым.

Поскольку составляющие типы также могут быть составными, то можно построить целую иерархию структур, но конечные компоненты любой структуры, разумеется, должны быть *атомарными*, не предполагающими дальнейшего дробления. Следовательно, система понятий должна допускать введение и простых, элементарных типов. Самый прямолинейный метод описания простого типа – *перечисление* всех значений, относящихся к этому типу. Ещё один аспект понятия «*атомарный тип*» – это такой тип данных, значения которого целиком обрабатываются за одну операцию или один машинный цикл. Например, число длинного целого типа (long int в языках Си/Си++) для персональных компьютеров общего назначения имеет размер 4 байта (32 бита) и не может быть обработано за одну операцию на 16-разрядных ЭВМ. Это может оказаться критически важным, например, в промышленных контроллерах при обработке вводимых данных от датчиков, когда разные байты многобайтового числа могут содержать значения, принятые в разные моменты времени, и итоговое значение не будет соответствовать истинной ситуации.

Если для значений некоторого типа существует отношение порядка, то такой тип называется упорядоченным или *скалярным*. Кроме того, значения скалярного типа должны быть отдельными числами или символами, не связанными (в смысле единого типа данных) с другими программными объектами. Скалярными являются все арифметические типы, символьные типы и, кроме того, типы, описывающие адреса программных объектов (т.к. адреса также являются числами). Остальные типы можно назвать *составными*. Скалярные типы обычно реализованы в языках программирования и получили название *стандартных*. Количество стандартных типов обычно невелико – от 10 до 15-20 в разных языках программирования. На основе стандартных типов программист создает те типы, которые ему нужны. Такие типы (созданные программистом) называются *производными*. К производным типам относятся в том числе и массивы, т.к. массив це-

лых чисел из 5 элементов и такой же массив из 10 элементов – это совершенно различные типы данных.

2.2. Перечисляемый тип данных

Любой новый простейший тип определяется простым перечислением входящих в него различных значений. Такой тип называется перечисляемым. Его определение, например, на языках Си/Си++, имеет следующий вид:

```
enum Seasons {winter, spring, summer, autumn}
s1, s2;
```

Имена в перечислении Seasons – целые константы: winter = 0, spring = 1 и т.д.

```
enum days {Su=5, Mo, Tu, We=11, Th, Fr, Sa};
```

2.3. Стандартные типы данных

К простейшим стандартным типам относятся целые числа, логические значения, символы, вещественные числа. Иногда все перечисленные типы, кроме вещественных, называют *интегральными*.

2.3.1. Целочисленные типы

ЭВМ оперирует с числами, содержащими конечное число разрядов. Возможны следующие типы чисел:

Целые двоичные числа длиной 8, 16 и 32 и двоично-кодированные десятичные числа длиной 8 бит. Двоичные числа допускают интерпретацию как целых без знака и целых со знаком, а десятичные числа знака не имеют. В двоичных целых числах без знака все разряды считаются значащими и диапазон представляемых чисел составляет 0...255 для чисел длиной 8 бит и 0...65535 для чисел длиной 16 бит.

Все разряды числа являются значащими, а двоичная точка находится справа или, как говорят, фиксирована после младшего значащего разряда. Отсюда появляется еще одно название этого формата и других аналогичных форматов – формат с фиксированной точкой. Следовательно, в этом формате представимы

только целые числа $0, 1, 2, \dots, 2^n-1$ и любая комбинация битов (двоичный набор) является допустимой.

Среди стандартных типов языков программирования может быть целый набор целочисленных типов, например, в Си/Си++:

```
char k1; // знаковый символьный тип (целочисленный)
int a; // знаковый целый тип
int b, c; // знаковый целый тип
```

Переменные беззнаковых типов должны быть явно указаны как беззнаковые:

```
unsigned char k2; // беззнаковый символьный тип (1 байт)
```

```
unsigned int d; // беззнаковый целый тип (2 – 4 байта, в зависимости от разрядности ЭВМ)
```

```
unsigned e; // беззнаковый целый тип
```

```
unsigned long int a1; // беззнаковый длинный целый тип (не короче 4-х байт)
```

```
unsigned long a2; // беззнаковый длинный целый тип
```

```
unsigned short int s1; // беззнаковый короткий целый тип (не длиннее 2-х байт)
```

```
unsigned short s1; // беззнаковый короткий целый тип
```

Символьные типы языков Си/Си++ также являются целочисленными и имеют размер в 1 байт.

Целые знаковые двоичные числа. Чтобы компьютеры могли оперировать положительными и отрицательными числами, один из разрядов необходимо отвести для изображения знака чисел S . Обычно им является старший (левый) бит, а стандартное кодирование имеет такой вид: $S=0$ – число положительное; $S=1$ – число отрицательное.

В языках Си/Си++ знаковые типы могут быть явно указаны:

```
signed char k1; // знаковый символьный тип
```

```
signed int a; // знаковый целый тип
```

но в большинстве случаев в этом нет необходимости:

```
char k1; // знаковый символьный тип
```

```
int b, c; // знаковый целый тип (слово signed можно не указывать)
```

```
long int alpha; // знаковый длинный целый тип
```

```
long beta; // знаковый длинный целый тип (слова int
можно не указывать)
```

```
short int gamma; // знаковый короткий целый тип
```

```
short delta; // знаковый короткий целый тип
```

Для 64-разрядных процессоров семейства x86 в более новых версиях языка Си++ используется 64-разрядный целочисленный тип `long long int`, в ранних версиях языка обозначавшийся как `int64`.

2.3.2. Вещественные числа

Система вещественных чисел, применяемая в обычных математических (ручных) вычислениях, предполагается бесконечной и непрерывной. Это означает, что не существует никаких ограничений на диапазон используемых чисел и на точность (количество значащих цифр) их представления. Для любого вещественного числа имеется бесконечно много чисел, которые больше и меньше его, а между любыми двумя вещественными числами также находится бесконечно много чисел.

Реализовать такую систему чисел в технических устройствах нельзя. Во всех компьютерах размеры регистров и ячеек памяти фиксированы, что ограничивает систему представимых чисел. Ограничения касаются как диапазона, так и точности представления чисел, т.е. система машинных чисел оказывается конечной и дискретной, образуя подмножество системы вещественных чисел.

Обычно вещественные числа хранятся и используются в ЭВМ в формате с плавающей точкой.

Для вещественных чисел применяется формат с плавающей точкой в вариантах обычной (или одинарной), двойной и расширенной точности (ОТ, ДТ, РТ). Значащие цифры числа находятся в поле мантииссы (или дроби), поле порядка (или экспоненты) показывает фактическое положение точки в разрядах мантииссы, а бит знака S определяет знак числа. Мантиисса представлена в прямом коде.

Порядок E задается в смещенной форме; он равен истинному порядку, увеличенному на значение смещения. Смещенный

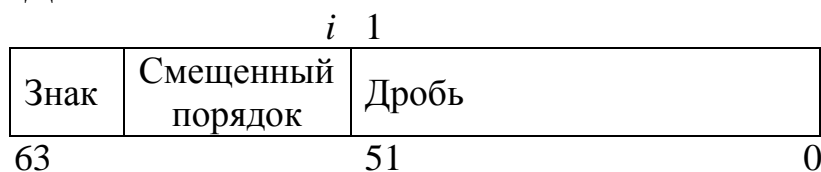
порядок называется также характеристикой или *экспонентой*; ее можно считать целым положительным или беззнаковым числом. Задание порядка в форме со смещением упрощает операцию сравнения чисел с плавающей точкой, превращая ее в операцию сравнения целых чисел, а выполнение остальных операций практически не усложняется. Так как операции с целыми числами выполняются значительно быстрее операций над числами с плавающей точкой, сравнение чисел с плавающей точкой производится быстрее, что важно в алгоритмах с большим числом сравнений, например в алгоритмах сортировки. Значения смещений для основных форматов следующие: OT – 127, ДТ – 1023, РТ – 16383. Смещенные порядки 000...00 и 111...11 зарезервированы для специальных значений.

Числа с плавающей точкой длиной 32 и 64 бита применяются во многих компьютерах и обычно называются числами с одинарной и двойной точностью. Как правило, порядок имеет фиксированную длину, определяя один и тот же диапазон представимых чисел, а для повышения точности вводятся дополнительные биты мантииссы. При этом схемы арифметико-логических устройств в процессоре усложняются незначительно. Однако при удвоении длины числа предпочтительнее часть бит отвести для расширения порядка. Такой порядок обеспечивает, в частности, получение точного произведения без переполнения и антипереполнения, когда сомножители представлены в формате OT. Параметры для трех форматов вещественных чисел приведены на рисунке 2.1.

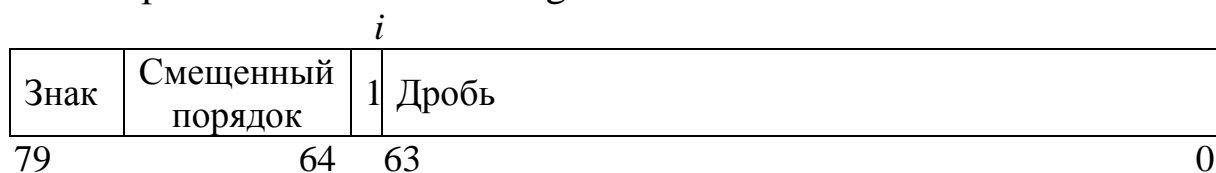
Одинарной точности – float



Двойной точности – double



Расширенной точности – long double



i – позиция скрытой десятичной точки

Рис. 2.1 – Форматы вещественных чисел

Отметим наличие в мантиссе бита F0 единиц. В форматах чисел с плавающей точкой большинства компьютеров бит F0 отсутствует, и мантисса оказывается правильной дробью. Мантисса обычно представляется в нормализованной форме, т.е. старший бит равен 1. Следовательно за исключением числа нуль мантисса состоит из целой части и дроби в следующем виде :

$$1.F_1F_2F_3\dots F_N,$$

где F_i равно 0 или 1. Благодаря нормализации достигается однозначное представление чисел и устраняются старшие нули в числах, меньше единицы, что максимизирует количество значащих цифр мантиссы при ее фиксированной длине.

В форматах ОТ и ДТ бит F0 при передачах чисел и хранении их в памяти не используется. Это так называемый скрытый или неявный бит, который в нормализованных числах содержит 1. Следовательно, в этих форматах невозможно представить числа, которые не нормализованы (за исключением нулевого порядка). Кроме того, скрытый бит не позволяет представить в этих форма-

тах нуль и он должен кодироваться как специальное значение. Скрытый бит можно реализовать только при основании, на степень которого умножается мантисса, равным 2.

Числа в формате РТ имеют явный бит F0. Такой формат позволяет несколько повысить скорость выполнения операций и обеспечить некоторые преимущества, благодаря простоте представления чисел, не являющихся нормализованными. Отличия в форматах представления вещественного числа показаны на рисунке 2.2 для значения $-247,375$

1	10000110	11101110110...0
1	10000000110	11101110110...0
1	100000000000110	111101110110...0

Рис. 2.2 – Пример двоичного представления вещественного числа

Числа в форматах ОТ и ДТ существуют только в памяти. При загрузке числа в одном из этих форматов в регистр арифметического устройства оно автоматически преобразуется в 80-битный формат РТ. Аналогично данные из регистров можно преобразовать в форматы ОТ и ДТ для сохранения их в памяти. Формат РТ также допускает сохранение в памяти и обычно применяется для промежуточных результатов.

В процессорах Intel принято хранение многобайтных элементов по принципу «младшее – по меньшему адресу». Логически во всех форматах левый байт является старшим, а правый младшим (*big-endian*). В физической памяти архитектуры x86 первым, т.е. по меньшему адресу, хранится младший байт (этот адрес считается и адресом всего числа), а последним, т.е. по большему адресу, – старший байт (*little-endian*). Передача данных всегда начинается с младшего адреса.

Специальные числовые значения

К специальным значениям относятся денормализованные вещественные числа, нули, отрицательные и положительные бесконечности, нечисла, неопределенность и неподдерживаемые форматы. Для представления специальных значений зарезервированы минимальный 00000...00 и максимальный 111...11 смещенные порядки.

1). **Денормализованные вещественные числа.** Это числа, которые меньше минимального нормализованного числа для каждого вещественного формата. Такие числа имеют минимальный смещенный порядок и ненулевую мантиссу.

2). **Истинный нуль.** Значение нуль в вещественных и десятичном форматах является знаковым, а двоичный целый нуль всегда положительный. В вычислениях знак нуля не учитывается, и обычно программист может не заботиться о наличии знакового нуля. Нуль в вещественных форматах кодируется с нулевым смещенным порядком 0000...00 и такой же мантиссой (0.000..00).

3). **Бесконечность.** Вещественные форматы поддерживают знаковые представления бесконечностей. Эти значения кодируются со смещенным порядком из всех единиц 1111...11 и мантиссой 1.000...00. Знаки бесконечностей учитываются и сравнения возможны.

4). **Нечисла.** Нечисло является представителем класса специальных значений, существующих только в вещественных форматах. Оно имеет смещенный порядок из всех единиц (1111...11), любой знак и любую мантиссу за исключением 1.000..00.

Арифметическое устройство формирует специальное нечисло, называемое **неопределенностью**, реагируя на замаскированный особый случай недействительной операции. Оно имеет отрицательный знак, смещенный порядок 111...111, а мантисса равна 1.1000...00. Неопределенность предусмотрена для вычислительных ситуаций, в которых человек говорит «не знаю», например деление 0 на 0.

В некоторых операционных системах, например, UNIX, нечисла обозначаются как NaN (Not a Number – не число) и используются как возвращаемые значения функций или в операциях сравнения.

5). *Неподдерживаемые форматы*. Формат РТ имеет много двоичных наборов, которые не попадают ни в один из ранее рассмотренных классов.

Переменные вещественных (или действительных) типов определяются в языках Си/Си++ следующим образом:

```
float f1; // плавающий (вещественный) тип обычной
точности (4 байта)
double d2; // плавающий тип двойной точности (8 байт)
long float LF; // то же, что double (в некоторых реали-
зациях компиляторов языка Си++, например, GCC, тип long
float может не поддерживаться, что может привести к ошибке)
long double Ld; // длинный плавающий двойной точно-
сти (10 байт)
```

2.3.3. Представление и структуры хранения логической информации

Логический тип данных (в языке Паскаль – BOOLEAN) представляет одно из двух истинных логических значений (истина/ложь). Эти значения обозначаются посредством стандартных идентификаторов

true (истина);
false (ложь).

Над значениями булевского типа допускаются операции сравнения, причем считается, что *false* < *true*. Кроме того, имеются четыре стандартные логические операции, обозначаемые служебными словами:

and – логическое умножение;
or – логическое сложение;
xor – сложение по модулю 2 (исключающее или);
not – логическое отрицание.

Для запоминания логического значения достаточно одного бита. Однако с целью ускорения доступа к логическому значению его представление в памяти может быть избыточным.

В более поздних реализациях компиляторов языка Паскаль, например Турбо Паскаль 7.0, добавлено еще три логических типа: ByteBool (размер 1 байт), WordBool (размер – 2 байта) и

LongBool (размер 4 байта). Они введены для унификации с другими языками программирования и с операционной системой Windows. Отличие их от стандартного типа Boolean заключается в фактической величине параметра этого типа. Значению FALSE соответствует число 0, записанное в соответствующее количество байтов. Значению же TRUE для типа BOOLEAN соответствует число 1, записанное в его байт, а для других типов значению TRUE соответствует любое число, отличное от нуля.

В ранних версиях языков Си и Си++ не было специального типа для представления логических значений. Все целые типы, включая символьные, могут использоваться для представления логических значений. Нулевое значение означает «ложь», любое ненулевое, включая отрицательное – «истина». В более поздних версиях языка Си++ введён логический тип `bool`, аналогичный типу `boolean` в Паскале.

2.4. Указатели

2.4.1. Назначение и смысл указателей

Все программные объекты и структуры данных по их размещению в памяти ЭВМ характеризуются своими адресами. Эти адреса могут (а часто и *должны*) каким-либо образом использоваться в программах, а для этого они должны где-то храниться. Для хранения адресов и используются указатели.

Указатели – это *переменные*, содержащие адреса других переменных или функций, в том числе членов классов (каких-либо программных объектов, например, переменных, массивов, записей, функций). Указатель на функцию содержит адрес точки входа в функцию. Размер памяти, требуемый для хранения адреса и формат этого адреса, зависят от вычислительной платформы и реализации компилятора. В современных системах общего назначения размер указателя на данные любого типа составляет 4 байта:

```
sizeof(void*...long double*) ~ 4
```

Указатели являются *необычными* переменными по своему *назначению* – хранение адресов других переменных, но *обычными* по некоторым своим *свойствам*, например, выполняемым операциям. Поскольку адреса являются целыми беззнаковыми числами (длинными), то указатели также относятся к *простым* структурам данных (целочисленным).

Так как указатель – это адрес некоторого объекта, то через него можно обращаться к этому объекту. Унарная операция $\&$ позволяет получить адрес программного объекта, поэтому оператор

$y = \&x;$

присваивает адрес объекта x переменной y . Операцию $\&$ можно применять только к объектам, действительно размещенным в памяти, конструкции вида

$\&(x+7)$

$\&28$

недопустимы.

Унарная операция $*$ воспринимает свой операнд как адрес некоторого объекта и использует этот адрес для выборки содержимого, поэтому оператор

$z = *y;$

присваивает переменной z значение, записанное по *адресу* y .

Рассмотрим ситуацию размещения в памяти переменной x :

	A000
значение x	A001
	A002

← Адрес x ($\&x$) → $y = \&x$

	A000
$*y$	y
	A002

$z = *x$

$z = *(\&x)$

$z = x$

Итак, если $y = \&x$ и $z = *y$, то $z = x$

x – переменная

y – её адрес (указатель)
z – содержимое адреса y
*y – содержимое адреса y

Объекты, состоящие из знака * и адреса, необходимо объявлять.

```
int *a, *b, *c;
```

Указатели объявляются при помощи символа *. Объявление вида

```
char *d;
```

говорит о том, что значение, записанное по адресу d, имеет тип char. Описатель, записываемый после символа *, может представлять собой более сложную конструкцию. При этом тип объекта, на который «указывает» указатель, определяется совокупностью оставшейся части описателя и спецификатора типа:

```
char *(*(*var)())[10];
```

Указатель может указывать на значение, имеющее стандартный тип, в том числе перечислимый, а также на структуры, объединения, массивы, функции, объекты классов, члены классов и другие указатели.

При определении указателя часто бывает целесообразно выполнить его инициализацию. В качестве инициализирующего выражения должно использоваться константное выражение, в частности:

- явно заданный адрес участка памяти;
- указатель, уже имеющий значение;
- выражение, позволяющее получить адрес объекта с помощью операции &:

```
int i, *ip = &i;
```

После определения с инициализацией указателя ip доступ к переменной i возможен как с помощью её **имени** i, так и с помощью её **адреса**, являющегося значением указателя ip.

Присвоив указателю адрес объекта, можно не только получать, но и изменять содержимое этого объекта.

Выражение вида *ip обладает некоторыми правами имени

объекта, т.е. `*ip` служит синонимом или псевдонимом имени `i`. Выражение `*ip` может использоваться везде, где допустимо использование имён объектов того типа, к которому относится указатель, но только в том случае, если указатель инициализирован при определении явным образом. Например, указатель `a` не инициализирован, поэтому попытки использовать выражение `*a` в *левой* части операции присваивания, или в функции ввода неправомерны (именно поэтому мы говорим лишь о *некоторых* правах имени объекта). Значение указателя (т.е. адрес как место в памяти) неизвестно, а результат занесения значения в неопределённый участок памяти непредсказуем и может привести к аварийному событию.

```
*a = 1; // Ошибочное применение
```

L-выражения или **леводопустимые выражения** – выражения, ссылающиеся на некоторую *именованную* область или ячейку памяти и поэтому имеющие смысл в *левой* части операции присваивания (откуда и произошло это название). Простейшим примером L-выражения является имя переменной – оно ссылается на ячейку памяти, которая хранит значение этой переменной.

Адрес ячейки, на которую ссылается L-выражение, может быть получен с помощью операции получения адреса `&`, за некоторыми исключениями: не могут быть получены адреса битовых полей и переменных, имеющих класс памяти `register`.

К **модифицируемым L-выражениям** (т.е. ссылающимся на ячейку памяти, значение которой доступно изменениям) относятся:

- идентификаторы переменных целых, плавающих, перечислимых типов, указателей, объектов классов, структур и объединений;
- индексные выражения, кроме тех, которые имеют тип массива;
- выражения выбора элемента класса, структуры или объединения, если выбранный элемент сам является одним из допустимых L-выражений;

- выражения косвенной адресации, т.е. получение значения по указателю, если только их значения не имеют типы «массив» или «функция»;
- L-выражения в скобках;
- ссылки на объекты;
- выражения преобразования типа переменной, если размер результирующего типа не превышает размера первоначального типа (см. пример).

```
char *p;
int i;
long n;
(long*)p = &n; // допустимо
(long)i = n;   // ошибка
```

Немодифицируемые L-выражения (т.е. праводопустимые) – такие, адрес которых может быть получен, но использоваться в левой части бинарной операции присваивания они не могут: идентификаторы массивов, функций, констант, переменных, объявленных с модификатором `const`, операции вызова функций (для этих операций существуют исключения, которые будут рассмотрены ниже).

Специальное применение имеет указатель на тип `void` («пустой»), который означает, что данный указатель адресует любой объект, не имеющий тип `const` или `volatile`. Указатель на тип `void` в языках Си/Си++ является аналогом нетипизированного указателя `pointer` в Паскале. Любой указатель (кроме указателей на члены классов) может быть преобразован к указателю на `void`. Такое преобразование может быть неявным. Никакие другие преобразования типов указателей по умолчанию не выполняются, т.е. для обратного преобразования, – указателя на `void` в указатель на реальный объект, – требуется операция приведения типа. Другими словами, все указатели как бы «знают», что они «произошли» от указателя на `void`, но указатель на `void` «не знает», что от него «произошли» какие-то другие указатели.

```

char c, *p = &c;
void* v = p;           // Неявное преобразование типа
char* q = (char*)v;   // Явное преобразование типа

```

Такое явное приведение типа указателя на `void` к типу, отличному от `void*`, требуется при выполнении операций с указателями на `void`, либо с адресуемым им объектом. Например, если объявлена переменная `i` типа `int` и указатель `p` на тип `void`, то можно присвоить `p` указателю адрес переменной `i`, но нельзя изменить значение указателя на `void`.

```

int i;
void* p;
p = &i;
p++; // Ошибка
(int*)p++;

```

Можно создать функцию с типом возвращаемого значения – указатель на `void`, возвращенное ею значение может быть присвоено указателю на тот тип, который требуется.

Константы при использовании указателей распространяются как на сам указатель, так и на объект, адресуемый этим указателем. Либо то, либо другое, либо оба вместе в этом случае являются константой. В объявлении указателя позиция модификатора `const` определяет, что указатель или им адресуемый объект не должен изменяться. Рассмотрим примеры:

```

char a,b;
const char *psc = &a; // Указатель на const char – неизменяемый объект
psc = &b; // Правильно
*psc = 'z'; // Неправильно
char *const psc = &a; // Константный указатель на char – неизменяемый указатель
psc = &b; // Неправильно
*psc = 'z'; // Правильно

```

Адрес константного объекта не может быть присвоен неконстантному указателю, т.к. такое присваивание может изменить константу через указатель на неё.

2.4.2. Операции с адресами

Указатели могут использоваться в выражениях. Если y – указатель на целое значение, т.е. имело место объявление

```
int *y;
```

то $*y$ может, как уже было рассмотрено в §2.4.1, появляться там же, где и любая другая переменная, не являющаяся указателем.

Вполне допустимы выражения:

```
*y = 7; // Запись числа 7 по адресу y
```

```
*x*= 5; // Увеличение значения по адресу x в 5 раз
```

```
(*z)++; // Увеличение на 1 значения по адресу z
```

Скобки в последнем выражении необходимы, т.к. операции $*$ и $++$ имеют одинаковый приоритет и выполняются справа налево. Если $*z = 5$, то после операции $(*z)++$ $*z \sim 6$, а $*z++$ только увеличит сам адрес, хранящийся в указателе z .

```
int *p; // Указатель на тип int
```

```
int i = 33;
```

```
p = &i;
```

```
*p = *p + 1; // i ~ 34;
```

Указатели можно использовать как операнды в арифметических операциях:

```
y++;
```

Унарная операция $++$ увеличивает его значение, теперь оно является адресом следующего элемента. Указатели и целые числа можно складывать. Конструкция вида

```
y + n
```

задает адрес n -го объекта, адресуемого указателем y . Это справедливо для объектов любых стандартных типов (`char`, `int` и т.д.), транслятор будет масштабировать приращение адреса в соответствии с типом, определённым из соответствующего объявления. Арифметические операции, выполняемые над указателем и целым значением, имеют осмысленный результат, если указатель адресуется непрерывный блок памяти, хранящий множество элементов одного типа.

Один указатель может быть вычтен из другого, если они указывают на один и тот же тип данных. Разность между двумя указателями преобразуется к знаковому целому значению путем деления разности на длину типа, данные которого адресуются указателями. Результат представляет число ячеек памяти данного типа между двумя адресами.

Поскольку операции ++, --, * и & имеют одинаковый приоритет и выполняются справа налево, то нужно внимательно относиться к последовательности их выполнения при их сочетаниях, например, выражение

*p++

вычисляется как сначала *, затем ++ и значение указателя увеличивается на 1. Рассмотрим пример. Пусть имеются некоторые переменные и указатель, над которым последовательно выполняются операции.

```
int i1=10, i2=20, i3=30;
int *p=&i2;
```

i3	i2	i1
30	20	10

*&i2 -> 20

*&+i2 -> 21

30	21	10
----	----	----

*p -> 21

*p++ -> 21

*p -> 10

30	21	10
----	----	----

++*p -> 11

30	21	11
----	----	----

*--p -> 21

30	21	10
----	----	----

++*--p -> 31

31	21	10
----	----	----

```

++&i2;    // Ошибка - не L-выражение
--&i2++;  // Ошибка - не L-выражение

```

Любой адрес можно проверить на равенство операцией == или на неравенство операцией != со специальной константой NULL, которая записывается вместо нуля (значение, эквивалентное нулю). Гарантируется, что никакой программный объект никогда не будет иметь адрес NULL. Указатель, имеющий значение NULL, не адресует никакую область памяти. Он называется нулевым указателем.

2.4.3. Указатели на указатели

Поскольку указатель – это объект в памяти, то можно определить указатель на указатель, указатель на указатель на указатель, и так сколько нужно раз.

```

int i = 88;
int *pi = &i;
int **ppi = &pi;
int ***pppi = &ppi;

***pppi ~ 88

```

Порядок выполнения операции * – справа налево, поэтому обеспечивается последовательный доступ к участку памяти с адресом pppi, затем – с адресом (*pppi) ~ pi, затем – с адресом (*pi) ~ i.

```

pppi -> (*pppi) ~ pi -> (*pi) ~ i
***pppi ~ *( (*pppi) )

```

2.5. Алгоритмы обработки простых структур данных

Уровень сложности простых структур данных таков, что понятие «*алгоритм обработки*» к ним мало применимо. Лучше использовать термин «*операция*». Из основных алгоритмов обработки, перечисленных во введении, к простым структурам применимы операции просмотра и модификации в разных вариантах, например на языке Си++:

```

#include <iostream.h> // #include – аналог дирек-
тивы uses в Паскале
int main() // Точка входа в программу (начало про-
граммы)
{
    int a; // Создание (определение) локальной пе-
ременной без инициализации
    cin >> a; // Ввод с клавиатуры (из внешнего потока
ввода)
    a++; // Модификация (инкремент)
    a--; // Декремент
    if(a == 0) // Сравнение
        a = 1; // Модификация (присваивание значения)
    cout << a; // Считывание – вывод на дисплей (во
внешний поток вывода)
    return 0; // Завершение программы
}

```

Вопросы и задания для самоконтроля

- 2.1. Какие структуры данных относятся к простым?
- 2.2. Что означает понятие «тип данных»?
- 2.3. Какую информацию можно извлечь из типа данных?
- 2.4. На какие группы разделяются типы данных в основных языках программирования?
- 2.5. На какие группы могут разделяться типы, предназна-
ченные для описания целочисленных значений?
- 2.6. Пусть имеется некоторое отрицательное целое число.
Какое значение имеет старший разряд этого числа?
- 2.7. Считываемые из файла текстовые данные выводятся на
дисплей неверно. В чём может быть причина этого?
- 2.8. Работая с программой, пользователь вводит число
100000. При контроле введённых данных на дисплей выводится
число -31072. В чём может быть причина этого?
- 2.9. Можно ли в двоичной системе счисления точно пред-
ставить значение 0,0625?

- 2.10. Можно ли в двоичной системе счисления с ограниченной разрядностью точно представить значение 0,95?
- 2.11. Что такое указатели?
- 2.12. Для чего используются указатели?
- 2.13. Какие операции можно выполнять над указателями?
- 2.14. Что представляют собой указатели на указатели? Для чего они могут применяться?
- 2.15. Можно ли получить адрес указателя?
- 2.16. Что означает понятие «модифицируемое L-выражение»?
- 2.17. Что означает понятие «немодифицируемое L-выражение»?
- 2.18. Может ли указатель содержать нулевое значение?
- 2.19. Может ли явно созданная в памяти структура данных иметь адрес, равный нулю?

3. Линейные статические структуры данных

3.1 Массивы

Как было отмечено выше, арифметические операции, выполняемые над указателем и целым значением, имеют осмысленный результат, если указатель адресует непрерывный блок памяти, хранящий множество элементов одного типа. Именно этот блок и называется *массивом*. Одномерные массивы часто называются *векторами*.

Массив *объявляется* указанием числа элементов массива, которое должно быть *положительным целым константным* выражением, заключённым в квадратные скобки.

```
float echo[10];
```

Индексация элементов массива в языке C++ начинается с нуля.

Массивы в принципе являются *статическими* структурами данных, т.е. не могут изменять изначально заданный размер. В то же время в языках Си/Си++ могут существовать переменные любых типов и массивы, отдельно специфицированные как *статические* (т.е. имеющие статический *класс памяти*).

```
static int mas1[20];
```

Это означает, во-первых, что такие переменные и массивы существуют в программе от точки создания до конца работы программы и, во-вторых, гарантируется, что такие переменные и массивы инициализируются нулями или эквивалентными значениями.

В разных языках программирования информация о размере массива может храниться в самом массиве (*дескрипторный* способ в языке Паскаль), либо размещаться в системных переменных и быть недоступной для программиста (обычные массивы с Си/Си++). В последнем случае при работе с массивом требуется явно указывать его размер.

Массивы могут быть инициализированы набором значений, заключённых в фигурные скобки. Если при инициализации число значений меньше заданного числа элементов, то оставшиеся элементы инициализируются нулями или эквивалентными значе-

ниями.

```
int a[5] = {7, 6, 9}; // a[3]=a[4]=0
```

Если указаны все инициализирующие значения, то размер одномерного массива (или число элементов по первой размерности для многомерного массива) может отсутствовать. Для одномерного массива оно равно числу инициализирующих значений.

```
int a[] = {7, 6, 9, 3, 4};
```

Массив символов может быть инициализирован строковой константой.

```
char vls2[] = "2nd very long string.\n";
```

3.2. Динамические массивы

Как выше уже отмечалось, массивы по своей организации в принципе являются статическими структурами данных, их размер сохраняется неизменным в течение всей работы программы или от момента создания до момента уничтожения. Термин «*динамические*» в данном случае относится к способу создания массивов. Такие массивы определяются при помощи указателей, а для их создания и удаления используются специальные процедуры, функции или операции динамического выделения и освобождения памяти. Такие операции могут использоваться для работы не только с динамическими массивами, но и с другими динамическими программными объектами, например, отдельными переменными.

```
int *p; // Указатель
p = new int; // Динамическая переменная знакового
целого типа
float *b2 = new float (20.5); // Инициализация
динамической переменной
int *p2;
p2 = new int [5]; // Динамический массив целых
чисел из 5 элементов
```

Динамический массив отличается от обычного (автоматического, статического) еще и тем, что размер его может задаваться не только константой, но и переменной. При этом во время рабо-

ты программы массив может уничтожаться, память, им занимаемая, будет освобождена, а затем указатель, при необходимости, может быть ещё раз использован для организации массива. Имя такого массива останется тем же самым, но, фактически, это будет совершенно другой массив, возможно, другого размера и расположенный в другой области памяти.

```
delete p;          // Удаление динамической переменной
delete b2;
delete [] p2;     // Удаление динамического массива
(освобождение памяти)
```

При удалении динамического массива указание **квадратных скобок** обязательно (они могут быть пустыми в большинстве случаев). Иначе будет освобождена память, занимаемая только начальным элементом этого массива, а остальная память останется занятой и к ней не будет доступа («**утечка памяти**»). Если указатель, используемый в операции `delete`, адресует не начало блока, выделенного операцией `new`, то последствия операции `delete` могут быть непредсказуемы. Если указатель содержит значение `NULL`, то `delete` не выполняет никаких действий.

```
int size = 40;
p2 = new int [size]; // Пересоздание нового массива
с тем же именем
```

Динамический массив в языке Си++ имеет дескриптор, недоступный для программиста, но необходимый для правильной работы операции освобождения памяти. Именно поэтому квадратные скобки при удалении массива могут указываться **пустыми**.

Динамические массивы, в отличие от обычных, не могут быть инициализированы.

3.3. Многомерные массивы

В языках Си/Си++ считается, что массивы имеют только одну размерность. Многомерные массивы представляются массивами указателей на массивы (или массивами массивов).

Двумерные массивы как структуры данных на *логическом уровне* представляются матрицей значений. Массивы большей размерности представляются параллелепипедом (или набором матриц), «гиперпараллелепипедом» (набором наборов матриц) и т.д. В этой связи подобные структуры данных называют *прямоугольными*. Поскольку память ЭВМ представляет собой линейную (т.е., фактически, одномерную) последовательность ячеек с уникальными адресами, то на *физическом уровне* существует сложность представления массивов с *размерностью* большей единицы. Такие массивы представляются массивами одномерных массивов, фактически, массивами указателей на одномерные массивы.

В случае двумерного массива, т.е. матрицы, в каждом одномерном массиве такого массива указателей на массивы хранятся элементы столбцов или строк матрицы. Выбор столбцов или строк определяется языком программирования. Для языков Си/Си++ это будут строки. Для трехмерного массива параллелепипед раскладывается на слои, каждый из которых является матрицей. Реально, слой – это указатель на матрицу, все слои – массив указателей на матрицы. В свою очередь, матрица – массив указателей на векторы. Таким образом, для доступа к элементам многомерных (и одномерных) массивов могут использоваться указатели. Такой способ доступа будет рассмотрен ниже.

В языках Си/Си++ многомерные массивы определяются следующим образом.

```
int foxtrot[5][20];
```

Элементы двумерного массива хранятся по строкам, т.е. если проходить по ним в порядке их расположения в памяти, то быстрее всего изменяется крайний правый индекс. То же справедливо и для массивов большей размерности.

При инициализации многомерных массивов соблюдаются те же соглашения, что и при инициализации одномерных массивов, за исключением того, что можно не указывать размер только по первой (самой левой) размерности:

```
int b[][2]={{1,2},{3,4},{5,6}}; // b[3][2]
```

Если заданы все значения, то внутренние скобки могут отсутствовать.

```
int d[][2]={{1},{2},{3}};  
int c[][2]={1,2,3,4,5,6};
```

Имя любого массива, как одно-, так и многомерного, само по себе является адресом начального (т.е. нулевого) элемента этого массива и, одновременно, *константой*, т.е. константным указателем.

```
foxtrot ~ &foxtrot[0][0]
```

В таком качестве имя массива может использоваться везде, где использовался бы константный указатель. Единственным исключением является использование имени массива в операции `sizeof`. Результатом операции в этом случае будет число байт, занимаемых всем массивом, а не указателем.

Многомерный динамический массив может быть создан в языке Си++ по крайней мере двумя способами. **Первый способ:**

```
char (*ram1)[512];    // Указатель на символьный  
массив из 512 элементов  
int SizeY = 40;  
ram1 = new char [SizeY][512];    // Двумерный ди-  
намический символьный массив
```

В этом способе обязательно должен использоваться указатель на массив (1-, 2- или многомерный, в зависимости от конечной размерности) и только первый (самой левый) размер создаваемого массива может быть задан при помощи переменной.

```
long (*lp) [3][4];  
lp = new long[SizeY][3][4];
```

Удаляются такие динамические массивы также, как одномерные:

```
delete [] ram1;  
delete [] lp;
```

Второй способ является более гибким и допускает использование переменных по всем размерностям.

```

    int height = 256, width = 512; // Высота и ширина
матрицы
    unsigned char **ram; // Указатель на указатель (для 2-
мерного массива)
    ram=new unsigned char *[height]; // Создание од-
номерного массива указателей
    for(int i = 0; i < height; ++i)
        ram[i] = new unsigned char[width]; // Создание
строк – одномерных массивов

```

Каждый одномерный массив-строка может быть неодинакового размера.

Удаление двумерного динамического массива, созданного таким способом, выполняется следующим образом:

```

for(int i = 0; i < height; i++)
    delete[] ram[i];
delete [] ram;

```

Операции создания и удаления двумерного динамического массива относятся к линейным алгоритмам.

3.4. Связь массивов с указателями

Любое действие, которое достигается индексированием массива, может быть выполнено также с помощью указателей, причём последний вариант будет быстрее.

Объявление

```
int a[5];
```

определяет массив из 5 элементов. Если объект *у объявлен как

```
int *y;
```

то оператор

```
y = &a[0];
```

присваивает переменной у адрес элемента а[0]. Если переменная у указывает на очередной элемент массива а, то у+1 указывает на следующий элемент, причём здесь выполняется масшта-

бирование для приращения адреса с учётом длины объекта. Поскольку *имя массива* есть *адрес его начального элемента* (и указатель на этот элемент), то инструкцию $y = \&a[0]$ можно записать в виде

$$y = a;$$

Тогда элемент $a[i]$ можно представить как $*(a+i)$. Если y – указатель, следующие записи эквивалентны:

$$a[i] \sim *(a+i)$$
$$y[i] \sim *(y+i)$$

Таким образом, любой массив (и индексное выражение) можно представить при помощи указателей.

$a[5]$ – массив

a – имя массива

$y = \&a[0]$ – адрес начального элемента массива

a – адрес начального элемента массива

$x0$ – содержимое начального элемента массива

$$a[0] = x0;$$
$$a[1] = x1;$$
$$x0 = *y = *a = a[0]$$
$$x1 = *(y+1) = *(a+1) = a[1]$$

В то же время между именем массива и соответствующим указателем есть одно существенное различие. Указатель – это *переменная* и $y = a$; или $y++$; – допустимые операции. Имя массива – это *константа*. Поэтому конструкции вида

$$a = y; \quad a++;$$

использовать нельзя, т.к. значение константы постоянно и не может быть изменено.

Если указатели адресуют элементы одного массива, то их можно сравнивать (в операциях отношения). В то же время нельзя сравнивать, либо применять в арифметических операциях указатели на разные массивы. Указатели на элементы одного массива можно вычитать, результатом будет число элементов массива, расположенных между уменьшаемым и вычитаемым указателями.

Доступ к элементам многомерных массивов также возможен с помощью указателей:

```
b[i][j][k] ~ *(*(* (b+i)+j)+k)
```

Допускается комбинировать обе формы доступа к элементам многомерного массива:

```
b[i][j][k] ~ *(b[i][j]+k)
```

Обращение к элементу массива в языке C++ относится к постфиксному выражению вида

```
PE[IE]
```

Постфиксное выражение PE должно быть указателем на нужный тип, выражение IE должно быть одного из целых типов. Таким образом, если PE – указатель, адресующий массив, то PE[IE] – индексированный элемент этого массива. Выражение

```
*(PE + IE)
```

– другой способ доступа к этому же элементу массива. Возможна эквивалентная запись

```
*(IE + PE)
```

т.к. операция сложения коммутативна, и следовательно

```
IE[PE]
```

адресует тот же элемент, что и PE[IE].

```
int m[] = {1, 2, 3, 4,};
```

```
m[0] ~ 1
```

```
0[m] ~ 1
```

Иногда индексы могут иметь отрицательные значения. В этом случае указатель PE должен указывать не на начало массива.

Можно определить указатель, инициализированный именем массива, т.е. адресом начального элемента этого массива.

```
int m[] = {1, 2, 3, 4,};
```

```
int *mp = m;
```

```
mp[0] ~ 1, *(mp+3) ~ 4
```

Иногда для такого указателя используется термин *«указатель на массив»*. Следует внимательно относиться к употреблению этого термина. *Его применение для указателя, используе-*

мого в рассмотренном примере, является, в сущности, программистским жаргоном, не вполне оправданно и совсем не правильно с точки зрения типов данных. В дальнейшем будем для обозначения указателя, адресующего массив, применять термин «указатель на массив» в кавычках, для того, чтобы не путать его с типами данных указателей на массив.

«Указатель на массив», в отличие от имени массива, является обычным указателем, неконстантным, со всеми вытекающими последствиями. Операция `sizeof`, примененная к такому указателю, даёт количество байт, занятых именно *указателем*, а не адресуемым им массивом. Операция получения адреса `&` позволяет получить *адрес указателя*, а не адрес начала массива. Наконец, значение такого указателя может быть изменено, и он уже не будет адресовать массив.

Однако такой указатель, также как и имя массива, может использоваться для доступа к элементам массива с помощью операций `[]` и `*`, как в рассмотренном примере.

3.5. Строки

Строки как структуры данным могут быть организованы несколькими различными способами. Первый способ – *дескрипторный*. В этом случае строка представляет собой массив символов, первый элемент массива является *дескриптором*, т.е. содержит информацию о длине строки. Непосредственно символы строки хранятся, начиная только со второго элемента массива. *Недостатком* такого способа организации является ограниченная длина строки. Так, если каждый элемент является байтом, то длина строки ограничена 255 символами. Именно такой способ используется в Паскале для строк типа `String`.

Второй способ организации строк – *маркерный*. И в этом случае строка также представляется массивом символов, но символы хранятся, начиная с первого (начального) элемента массива, а заканчивается строка специальным символом, который называется *маркером*. Такой способ используется в Паскале для строк типа `ANSIIZ` и в Си/Си++. Длина строки при маркерном способе организации, теоретически, неограниченна. Пример маркерного

способа организации строк в Си/Си++ уже был приведён:

```
char vls2[] = "2nd very long string.\n";
```

Третий способ организации – в виде *линейного связного списка*. Такой способ будет рассмотрен ниже.

В языках Си/Си++ указатели на тип `char` могут инициализироваться с помощью строковых констант:

```
char *vls1 = "1st very long string.";
```

Следует отметить, что присваивание для строковых констант (и вообще строк) возможно только при инициализации. После определения символьных массивов или указателей на тип `char` в программе для назначения им значений следует пользоваться библиотечной функцией `strcpy`:

```
char* str;  
strcpy(str, "строка");
```

Строки при этом обладают всеми свойствами массивов, а указатели – это обычные «указатели на массивы», значением такого указателя является адрес начального символа строки.

3.6. Массивы указателей

Существует возможность создать массив указателей. Определение вида

```
int* arp[20];
```

создает массив указателей на объекты типа `int`. Элементами этого массива являются указатели на объекты типа `int`.

Массивы указателей можно инициализировать, например массив строк:

```
char* month1[] = {"январь", "февраль", "март"};
```

Элементами массива `month1` являются адреса начальных символов строк. Следует отметить, что хотя показанный способ и не приводит к ошибке при компиляции программы, он не является корректным, т.к. позволяет выделить память только для начальных символов каждой строки. При этом сохранность содер-

жимого остальных байт в течение всего времени работы программы не гарантируется. Выходом из этой ситуации является выделение памяти требуемого размера (в том числе в виде массива) под каждую строку отдельно.

Возможен второй вариант размещения данных (в том числе строк) в памяти, при котором расход памяти больше, но память выделяется гарантированно:

```
char month2[][10]={"январь","февраль","март"};
```

Одной из областей применения массивов указателей является сортировка сложных объектов неодинакового размера.

3.7. Интерпретация составных описателей

Правило интерпретации составных описателей может быть названо чтением «изнутри – наружу». Начать интерпретацию нужно с *идентификатора* и проверить, есть ли *справа* от него *открывающие* скобки, квадратные или круглые. Если *да* – то рассматривается *правая* часть описателя, иначе – левая. Затем следует проверить, есть ли *слева* от идентификатора символ *, если *да* – то рассматривается левая часть. Если на каком либо шаге интерпретации *справа* встретится закрывающая круглая скобка, которая используется для изменения порядка интерпретации, то необходимо сначала закончить интерпретацию *внутри* данной пары круглых скобок, а затем продолжит интерпретацию *справа* от закрывающей круглой скобки. На последнем шаге интерпретируется спецификация типа. После этого тип объявленного объекта полностью известен.

```
char * (* (*var) ()) [10];  
7 6 4 2 1 3 5
```

1. Идентификатор `var` – это
2. Указатель на
3. Функцию без аргументов, возвращающую
4. Указатель на
5. Массив из 10 элементов, являющихся
6. Указателями на
7. Значения типа `char`.

Рассмотрим примеры:

1. `int* var[5];` // `var` – массив указателей на значения типа `int`;
2. `int (*var)[5];` // `var` – указатель на массив значений типа `int`;
3. `long* var();` // `var` – функция, возвращающая указатель на значения типа `long`;
4. `long (*var)();` // `var` – указатель на функцию, возвращающую значение типа `long`;
5. `struct both {
 int a;
 char b;
}` `(*var[5])();` // `var` – массив указателей на функции, возвращающие структуры типа `both`.

Чаще определения структурных типов и их использование в программах оказывается разнесенным по исходному тексту. В таком случае этот пример будет выглядеть следующим образом (на языке C++):

```
struct both {  
    int a;  
    char b; };  
both (*var[5])();
```

6. `double (*var())[3];` // `var` – функция, возвращающая указатель на массив из 3-х значений типа `double`;
7. `union sign {
 int x;
 char y; }`
`**var[5][5];` // `var` – массив из 5-ти элементов, являющихся массивами указателей на указатели на объединения типа `sign` (двумерный массив указателей на указатели).

Ситуация, аналогичная примеру 5. С учетом отдельного определения объединения и переменной `var`, последнее будет выглядеть следующим образом:

```
sign **var[5][5];
```

8. `sign * (*var[5])[5];` // `var` – массив из 5-ти элементов, являющихся указателями на массив указателей на объединения типа `sign`.

3.8 Алгоритмы обработки статических линейных структур

Алгоритмы обработки статических линейных структур чаще всего также носят линейный характер и представляют собой циклы, подобные циклам создания и удаления двумерного динамического массива. Наряду с операциями, используемыми для простых структур данных – просмотр, модификация, ввод и вывод, – к массивам и другим линейным структурам (не только статическим) применяются более сложные операции, например, поиск, реверсирование (обращение).

Алгоритмы поиска, в том числе, в линейных структурах, будут рассмотрены ниже.

Операция **реверсирования** представляет собой перестановку элементов структуры данных, например, массива, в обратном порядке. Реверсирование может оказаться полезным в случае, когда массив упорядочен, например, по убыванию, и требуется переупорядочить его по возрастанию.

```
const int N = 10; // Размер массива
float mas[N] = {9., 8., 7., 6., 5., 4., 3.,
2., 1., 0. };
float buf;
void main()
{
//...
for(int a = 0; a < N/2; a++)
{
    buf = mas[a];
    mas[a] = mas[N-a];
    mas[N-a] = buf;
}
//...
}
```

Часто требуется скопировать содержимое одного массива в другой. Наиболее просто такая операция выполняется, если массив-приёмник по размерам не превосходит массив-источник. Удобно оформить эту операцию в виде отдельной процедуры.

```
float mas1[N];
void MasCopy(float* a, float* b, int n)
{
    for(int x = 0; x<n; x++)
        b[x] = a[x];
}
```

Алгоритмы характеризуются своей трудоёмкостью, т.е. количеством операций, необходимых для завершения алгоритма, а также требованиями к объёму памяти, необходимой для хранения исходных данных и промежуточных результатов. Кроме того, количество операций фактически означает время выполнения алгоритма. Для обозначения обеих характеристик – времени выполнения и требуемого объёма памяти, – используется одна и та же система обозначений, получившая название «О большое» – верхняя асимптотическая оценка трудоёмкости алгоритма. В этой системе обозначений время обработки структуры данных размером N с помощью рассмотренных выше линейных алгоритмов (кроме поиска) характеризуется как $O(N)$.

Вопросы и задания для самоконтроля

3.1. К какой группе структур данных относятся автоматические массивы?

3.2. К какой группе структур данных относятся статические массивы?

3.3. К какой группе структур данных относятся динамические массивы?

3.4. По каким признакам можно классифицировать двумерный динамический массив?

3.5. Можно ли изменить размер динамического массива в процессе его использования?

3.6. Можно ли использовать один и тот же указатель для создания в разные моменты времени динамических массивов разного размера?

3.7. В чем заключается связь между указателями и массивами?

3.8. Какие операции обязательны при работе с динамическими массивами?

3.9. Перечислите основные свойства динамических массивов.

3.10. В чем заключается отличие между автоматическими и статическими массивами?

3.11. Какое требование нужно соблюдать при присваивании адреса массива указателю?

3.12. Какие ограничения накладываются на определение многомерных динамических массивов?

3.13. В чем заключается отличие между именем массива и указателем?

3.14. Что представляют собой строки?

3.15. Какие существуют способы организации строк?

4. Ссылки. Временные структуры данных

В языках Си и Си++ возможны ситуации появления временных структур данных – как переменных стандартных типов, так и более сложных: экземпляров записей (структурных типов) и объектов классов. Такими ситуациями являются, в частности, несовпадение типов операндов в арифметических операциях и несовпадение типов в формальных и фактических аргументах функций.

Ссылки в языке Си++ представляют собой *второе имя* (или псевдоним) объекта; объявляются при помощи символа *&*, *при объявлении обязательно должны быть инициализированы* именами тех объектов, псевдонимами которых они будут являться.

```
double agent = .028;
double &bond = agent;
// ...
bond /= 4.0 // agent ~ .007
```

В общем случае в качестве инициализирующего выражения должно выступать имеющее значение L-выражение (имя объекта, реально существующего в памяти). Значением *ссылки* после определения с инициализацией становится *адрес* этого объекта. Повторное присвоение значения ссылке не допускается, т.е. невозможно сделать так, чтобы некоторая конкретная ссылка ссылалась на другой объект.

В определении ссылки символ *&* не является частью типа, т.е. имя `bond` имеет тип `double`.

Могут существовать ссылки на любые используемые в программе типы, как стандартные, так и созданные программистом (записи, классы и т.п.), кроме перечисленных ниже.

Функционально ссылка ведет себя подобно обычной переменной того же типа, а фактически является другой формой указателя.

Ссылки повышают эффективность программ при передаче больших объектов в функции, поскольку не требуют копирования объекта в стек;

– предоставляют функциям механизм изменения значения

передаваемого им аргумента (за счёт передачи адреса объекта, подобно указателю, а не копии объекта);

– используются, главным образом, при определении компонентных функций классов;

– не являются самостоятельными объектами и существуют только после инициализации; какие-либо операции выполняются не над ссылками, а над объектами, на которые они ссылаются;

```
int i;  
int &ir = i;  
ir = 3; // i = 3  
int j;  
j = i * ir; // j = 9  
ip = &ir; // ip получает адрес i
```

– *не могут* ссылаться на *другую ссылку* или на *битовое поле*. *Не может* быть ни *массивов ссылок*, ни *указателей на ссылку* (т.к. ссылка – это не самостоятельный объект), но *может* быть объявлена *ссылка на указатель* (поскольку указатель – самостоятельный объект).

```
int *&rpi; // Ссылка на указатель
```

Ссылка не может иметь тип `void`, для ссылки *нельзя* выделить участок памяти операцией `new`. Значение ссылки не может изменяться. Но любой объект может быть адресован любым количеством ссылок (также как и указателей).

Можно определить ссылку на константу, используя модификатор *const*. Ссылка на константу не позволяет изменить значение того объекта, с которым она связана.

```
double agent = .028;  
const double &Bond = agent;  
agent = 0.035; // правильно  
Bond = .008; // ошибка
```

Если тип инициализированной ссылки не совпадает с типом объекта, то создается *временный анонимный объект*, для которого ссылка является псевдонимом, инициализирующее значение преобразуется к типу ссылки и используется для установки значения анонимного объекта.

```
double d;  
int &ir = d; // несовпадение типов => анонимный объект  
ir = 3.0;    // значение d не изменилось, анонимный  
объект получил значение 3
```

Такая ситуация справедлива как для локальных или глобальных переменных, так и для аргументов функций, являющихся ссылками. При передаче в функцию некоторой структуры данных (любого типа) по ссылке, в случае несовпадения типа и возможности его преобразования (например, целый тип в действительный тип или производный класс в базовый класс) внутри функции создаётся временная безымянная структура данных требуемого типа, с которой и выполняются все предписанные программой действия. Затем временная структура автоматически разрушается при выходе из функции и её содержимое теряется. Состояние объекта, передававшегося в функции (фактического аргумента) остаётся неизменным.

5. Составные типы данных

5.1. Структуры

Структура является набором логически связанной информации, возможно, различных типов, объединённым в единый программный объект. Термин «*структура*» в Си/Си++ является синонимом понятия «*запись*», используемого в Паскале. Оба термина являются взаимозаменяемыми. **Структура** может означать как тип данных, так и объект этого типа. Определение структурного типа в Си/Си++ напоминает определение записи в Паскале, но в Си/Си++ отсутствуют некоторые ограничения Паскаля на размещение указателей.

```
struct Student
{
    char FirstName[10];
    char SecondName[15], Surname[20];
    int Age;
    char Dept[10];
    int Year;
};
```

Отдельные компоненты структур часто называют полями. Объекты структурных типов, также часто называемые структурами, определяются следующим образом:

```
Student Petrov; // Объявление структуры типа Student
Student i31[21]; // массив структур
```

Объекты структурного типа могут быть инициализированы, если имеют статический класс памяти, набором значений соответствующих типов, заключённых в фигурные скобки и расположенных в порядке объявления полей структуры, подобно тому, как инициализируются массивы. Поля, оставшиеся неинициализированными, получают нулевые значения или эквивалентные нулю.

Статические массивы структур также могут быть инициализированы. Инициализирующие значения для каждого элемента

массива помещаются внутри отдельной пары фигурных скобок, аналогично инициализации двумерного массива. Если число инициализирующих значений равно числу полей структуры во всех элементах массива, то внутренние фигурные скобки могут быть опущены.

Доступ к компонентам структуры происходит при помощи операции «точка»:

```
Petrov.Year = 3;
```

Имя структурного типа при его определении может отсутствовать, если в программе имеется одна или несколько структур **только одного** типа.

```
struct { /* поля структуры */ } st1, st2, st3;
```

Операция & позволяет получить **адрес** структуры, который может храниться в указателе.

```
Student *r32; // указатель на структуру  
r32 - адрес структуры типа Student  
*r32 - сама структура
```

Если имеется указатель на структурный тип, содержащий адрес реально существующей структуры, то доступ к полям с помощью указателя выполняется следующим образом:

```
(*r32).Age = 30;
```

Приоритет операции «точка» выше, чем операции *, поэтому круглые скобки в данном случае необходимы. Возможна другая, чаще используемая, форма доступа к полям структуры через указатель при помощи операции -> :

```
r32->Age = 30;
```

Могут существовать **ссылки на структурные типы**. Обычно такие ссылки используются как возвращаемые значения и аргументы функций для предотвращения копирования структур в стек при их передаче в функции:

```
Student& PrintInfo(Student&);
```

Поля структуры могут быть любого типа, стандартного или созданного программистом и известного к моменту его использования. Существует только одно исключение. Поле структуры *не может* иметь такой же тип, как и структура, компонентом которого это поле является. Но поле может быть указателем на структуру определяемого типа.

```
struct Err
{ Err d; ... };      // Ошибка
struct Corr
{ Corr *pc; ... };  // Правильно
```

Поле определяемого структурного типа *может* быть структура, тип которой *уже* определён.

```
struct Beg{...} golf;
struct next{ Beg st; ...};
```

Если в определении структурного типа нужно в качестве элемента использовать указатель на структуру другого типа, то разрешена такая последовательность определений.

```
struct A; // Неполное определение
struct B {struct A *pa};
struct A {struct B *pb};
```

Неполное определение структурного типа А можно использовать в определении структурного типа В, так как определение указателя ра на структуру типа А не требует информации о размере структуры типа А. Последующее определение структурного типа А **обязательно**. Использование в структурах типа А указателей на структуры уже введенного типа В не обладает какими-либо дополнительными особенностями и не требует пояснений.

Структурный тип может быть определен с помощью спецификатора *typedef*.

```
typedef struct { double Re,Im; } complex;
complex z1,z2;
```

```
struct a1
{ ...
  int oscar;
```

```

};
struct a2
{ ...
  a1 lima; // поле ранее определенного структурного типа
} tango; // совместное определение структурного типа и его
объекта
//...
tango.lima.oscar=256; // доступ к вложенному полю объ-
екта tango

```

5.2. Битовые поля

Структуры могут содержать битовые поля – последовательности двоичных разрядов внутри одного целого значения. Язык C++ позволяет использовать любой целый тип. Будет он знаковым или беззнаковым, зависит от реализации компилятора, так же как и порядок размещения битовых полей в машинном слове. Именно поэтому битовые поля имеют имена, для обеспечения независимости от машинной реализации. Приведём пример структурного типа, содержащего битовые поля.

```

struct kilo
{
  int a:2;
  unsigned b:3;
  int :5 // не используется
  int c:1;
  unsigned d:5;
};

```

Состав этих полей показан на рисунке 5.1.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
d					c	не используется			b			a			

Рис. 5.1 – Состав битовых полей

Для битового поля можно явно определить тип `signed` или `unsigned`. В битовом поле типа `signed` крайний левый бит является знаковым. Такое поле шириной в 1 бит может хранить значения только 0 или -1.

Битовое поле без имени просто резервирует указанное число разрядов. Битовое поле без имени с нулевым размером указывает, что следующее битовое поле начинается на границе машинного слова.

Доступ к битовому полю производится также как к компоненту структуры.

```
kilo ssc;  
ssc.a = 1;
```

Битовые поля используются для упаковки значений нескольких переменных в одно машинное слово с целью экономии памяти, отводимой под данные, но это увеличивает размер исполняемого кода для доступа к этим данным и снижает производительность программы.

Битовые поля позволяют также выполнять преобразование данных в другие форматы.

Битовые поля не могут быть массивами и не имеют адресов, поэтому к ним нельзя применить операцию получения адреса `&`, на них не может быть указателей и ссылок. Можно получить адрес структуры, содержащей битовые поля.

5.3. Объединения

Объединение – это некоторая переменная, которая может хранить *в разное время* объекты различных типа и размера. Такая переменная имеет размер, достаточный для размещения наибольшего из своих полей. Объединения объявляются при помощи служебного слова `union`.

```
union romeo  
{  
    int ir;  
    double fr;  
    char ch;  
};
```

Объединения могут иметь (и обычно имеют) имя, которое становится идентификатором типа, в противном случае объединения являются анонимными. К полям анонимных объединений можно обращаться так же, как если бы они были самостоятельными переменными. Глобальные анонимные объединения должны быть объявлены как статические.

```
romeo zulu;  
//...  
zulu.ir = 15; // обращение к полю (не инициализация)
```

Статические объединения могут быть инициализированы значениями в фигурных скобках, причём инициализировано может быть лишь первое поле.

```
romeo zulu = {15}; // инициализация  
romeo zulu = {'F'}; // неправильно, т.к. первое поле  
имеет тип int, а не char
```

Статические массивы объединений также могут быть инициализированы

```
romeo x_ray[ ] = {3, 5, 7};
```

Могут существовать указатели

```
romeo *mike;
```

и ссылки на объединения:

```
romeo& foxtrot = zulu;
```

Доступ к объединениям через ссылку

```
foxtrot.fr = ...
```

через указатель

```
(*mike).ir = ... или mike->ch = ...
```

Если поля объединения имеют одинаковый тип и длину, отличаясь только именами, то использование объединения подобно применению ссылки.

Основное достоинство объединений – возможность разных трактовок одного и того же содержимого некоторого участка па-

мяти, т.е. доступ к одному и тому же участку памяти с помощью объектов разных типов:

```
union {          // Безымянное объединение
    float F;
    unsigned long K; } FK;
FK.F = 3.141593;
FK.K ~ 1078530012
```

Битовые поля могут входить в состав объединений как структуры:

```
union a1 { struct b2{ int a0:1; int a1:2;
...}; };
```

Вопросы и задания для самоконтроля

- 5.1. Что представляет собой структурный тип данных?
- 5.2. Данные каких типов могут входить в состав структур?
- 5.3. Данные каких типов не могут входить в состав структур?
- 5.4. Могут ли все поля структуры быть одного типа?
- 5.5. Могут ли структурные типы быть безымянными? Если да, то как создаются их объекты?
- 5.6. Могут ли быть полями структурного типа объекты другого структурного типа? Если да, то накладываются ли при этом на структурные типы какие-либо ограничения?
- 5.7. Может ли один структурный тип определяться внутри другого структурного типа?
- 5.8. Может ли структурный тип быть локальным внутри функции?
- 5.9. Могут ли существовать указатели на структурный тип?
- 5.10. Могут ли существовать ссылки на структурный тип?
- 5.11. Могут ли существовать массивы объектов структурного типа?
- 5.12. Что такое объединения?
- 5.13. Что общего у объединений и структур и в чём их различие?

5.14. Что такое битовые поля?

5.15. Что нужно использовать, если требуется попеременно рассматривать один и тот же участок памяти как вещественное число или как набор байт?

5.16. Как проверить по битам содержимое вещественного числа в формате расширенной точности, рассмотренном в параграфе 2.3.2? Предложите программу или процедуру, которая позволяет это сделать.

5.17. С помощью какой операции выполняется доступ к полям структуры?

5.18. С помощью какой операции выполняется доступ к полям структуры, адресуемой указателем?

5.19. Как обеспечить связь между структурами и функциями?

6. Файлы

Файл представляет собой последовательность элементов не обязательно одного типа, хранящихся на внешнем запоминающем устройстве, переменной длины, имеющую уникальное имя среди всех остальных файлов на этом же устройстве.

В разных языках программирования высокого уровня файлы могут разделяться по типам, как например, в Паскале, где могут быть файлы *текстовые*, *типизированные* или *нетипизированные* (см. рис. 1.3). В других языках, например, Си/Си++ нет явного деления файлов по типам. Один и тот же файл может быть открыт для записи или для считывания как *двоичный* или как *текстовый*, в зависимости от текущих требований программиста.

Существует несколько способов доступа к файлам в программах при использовании языка Си++. Один из них – использование указателя на тип FILE, который может считаться аналогом файловой переменной в Паскале, только такая «файловая переменная» никак не связана с типом файлов. Этот указатель требуется для большинства библиотечных функций работы с файлами, описанных в заголовочном файле `stdio.h`. Этот указатель, так же как и файловую переменную в языке Паскаль, можно представить как логическое имя файла, используемое библиотечными функциями.

Для доступа к элементам файла он должен быть открыт. Функция `fopen` приводит физическое имя файла в соответствии с логическим и открывает файл.

```
#include <stdio.h>
FILE *file;
int main()
{
    //...
    file = fopen("имя_файла", "w");
    //...
```

Файл может быть открыт для: записи – "w", чтения – "r", дополнения – "a". Если файл открывается для записи или дополнения, но ещё не существует, то он создаётся (если это можно

сделать). Открытие существующего файла для записи приводит к уничтожению его имеющегося содержимого.

Примеры других режимов открытия файлов: "r+" – открытие файла для добавления, "r+t" – открытие текстового файла для добавления, "wt" – открытие текстового файла для записи, "rb" – открытие двоичного файла для чтения. Именно здесь и выбирается, как будет трактоваться файл – как текстовый, или как двоичный.

После окончания работы с файлом он должен быть закрыт:

```
//...
fclose(file);
}
```

Другой способ доступа к файлам – использование в качестве «файловой переменной» целого числа (конкретно знакового длинного целого числа) – *handle*, *handler*. В этом случае используются библиотечные функции такого же характера, что и при использовании указателя на тип FILE, но немного отличающиеся по названию.

Ещё один способ – использование объектов из библиотеки классов потокового ввода-вывода, аналогичных объектам *cin* и *cout* языка Си++, в частности, объектов классов *ifstream* и *ofstream*. В этом случае также используются соответствующие библиотечные функции.

При создании программ для операционной системы Windows можно также применять функции доступа к файлам из библиотеки *Windows API*, а при использовании какой-либо инструментальной системы, например, *C++ Builder* – функции записи и считывания в/из файл из состава классов этой системы.

Поскольку файл представляет собой последовательность элементов, с ним можно работать, как с линейной структурой данных, т.е. использовать последовательные алгоритмы. В то же время при помощи соответствующих библиотечных функций можно обращаться к произвольным элементам файла, аналогично возможности обращения к произвольным элементам массива.

Рассмотрим библиотечные функции работы с файлами, ис-

пользующие указатель на тип FILE. Эти функции можно разделить на две группы: *вспомогательные* и *записи/чтения*.

Вспомогательные функции:

FILE* fopen(char* name, char* code) – открытие файла (рассмотрена выше).

int fclose(FILE* file) – закрытие файла.

int access(char* name, int n) – проверка файла; режим проверки определяется аргументом **n**, **n** = 0 – проверка на существование, 1 – на выполняемость, 2 – на запись, 4 – на чтение, 6 – на запись и чтение, 7 – на запись, чтение и выполняемость и т.д. При успешном завершении функция возвращает 0.

int fileno(FILE* file) – возвращает целочисленный идентификатор (*handler*) файла. Эта функция обеспечивает связь этого идентификатора и указателя на тип FILE. Целочисленный идентификатор необходим для следующей функции.

long filelength(int handle) – возвращает длину файла в байтах. Знаковый тип *long* позволяет создавать файлы длиной не более 2 Гбайт.

int fseek(FILE* file, long pos, int sign) – перемещение текущей позиции записи/считывания в файле в точку, указываемую аргументом **pos**, отсчёт выполняется от начала файла, если аргумент **sign** = 0, от конца файла при **sign** = 2, от текущей позиции при **sign** = 1.

long ftell(FILE* file) – возвращает текущую позицию в файле. Эта функция полезна для предыдущей.

int rewind(FILE* file) – перемещение текущей позиции в начало файла.

int rename(char* old_name, char* new_name) – переименование файла.

int unlink(char* name) – удаление файла.

Функции *записи/считывания*.

int getc(FILE* file) – считывание символа (байта), несмотря на то, что возвращает значение типа `int`.

int getw(FILE* file) – считывание целого числа

int putc(int c, FILE* file) – запись символа

int fputs(const char* s, FILE* file) – запись строки

char* fgets(char* s, int n, FILE* file) – считывание строки

длиной $n-1$ символ

`int fread(void* block, size_t size, size_t items, FILE* file)` – считывание из файла в блок памяти, адресуемый указателем `block`, данных в количестве `items` элементов, каждый элемент размером `size` байт. Указатель на тип `void` используется потому, что неизвестен заранее точный тип считываемых данных. `size_t` – один из переименованных целых типов (для 32-разрядных систем – часто беззнаковый целый), определён в заголовочном файле `stddef.h`. Аргументы `size` и `items` – взаимозаменяемые, например, можно прочитать 1 блок данных размером 1000 байт или 1000 блоков размером 1 байт.

`size_t fwrite(const void* block, size_t size, size_t items, FILE* file)` – запись блока в файл. Блок помечен как константный, чтобы внутри функции его нельзя было модифицировать.

Кроме этих функций запись в текстовый файл может выполняться функцией `fprintf`, а считывание – функцией `fscanf`. Обе эти функции относятся к функциям с переменным числом аргументов, рассмотрение которых выходит за рамки настоящего курса.

Вопросы и задания для самоконтроля

- 6.1. Что такое файл?
- 6.2. К какой группе структур данных относятся файлы?
- 6.3. Какие действия необходимо выполнить для работы с файлом?
- 6.4. Различаются ли файлы по типам?
- 6.5. Как в программах устанавливается связь с файлами?
- 6.6. Какие способы организации связи с файлами вам известны?
- 6.7. Какие операции можно выполнять над файлами?
- 6.8. Как открыть файл для записи?
- 6.9. Как открыть файл для считывания?
- 6.10. Какая функция позволяет узнать длину файла?
- 6.11. Как проверить, можно ли произвести запись в выбранный файл?

6.12. Можно ли считать данные из произвольного места в файле? Если да, то как это сделать?

6.13. Можно ли перемещаться по файлу? Если да, то с помощью какой функции?

6.14. Чем отличается запись действительных чисел в текстовый и двоичный файлы?

6.15. Требуется создать файл размером 8 Гбайт. Можно ли для работы с таким файлом использовать тип данных `long`?

6.16. Как получить размер файла?

6.17. Можно ли в файл записывать разнотипную информацию?

6.18. Как удалить содержимое существующего файла?

6.19. Как обеспечить связь между файлами и функциями?

7. Очереди

Все линейные *динамические структуры данных* могут быть отнесены к *списковым структурам*. Именно такими являются разные виды очередей и стеки. *Динамическими* эти структуры считаются по той причине, что их размер в процессе работы может изменяться – список может удлиняться или укорачиваться, – а элементы структур (включая самые первые) создаются и разрушаются при помощи операций или процедур динамического выделения и освобождения памяти.

Начнём рассмотрение этих структур данных с наиболее простых, т.е. с очередей.

В общем случае, *очередь* – это линейный список, доступ к элементам которого происходит по принципу *FIFO* (First In and First Out – первым пришел и первым ушел).

Для очереди характерны две *операции* – занесение элемента в очередь и извлечение (считывание) элемента из очереди. В простой очереди для работы с данными доступны две позиции – *начало* (из этой позиции происходит извлечение) и *конец* (в эту позицию заносится входящий элемент) или «голова» и «хвост». Произвольный доступ к элементам, в отличие от массивов, формально *не допускается*. Операция извлечения (считывания) формально является *разрушающей*. Это означает, что считанные данные становятся недоступными. Возможно, явного разрушения (уничтожения) данных и не происходит, но к ним нет доступа, используя стандартные операции работы с очередью.

Области применения очередей могут быть разделены на две группы – системное применение и прикладное. К применению очередей в *системных* целях относятся:

- диспетчеризация задач операционной системой;
- буферизация ввода/вывода;

Прикладное применение:

- моделирование процессов (например, систем массового обслуживания);
- использование очередей как вспомогательных структур данных в каких-либо алгоритмах (например, при поиске в графах).

Одним из примеров очереди является *машина Тьюринга*. Более простым механическим примером является труба с текущей только в одном направлении жидкостью.

Классификация очередей. По *архитектуре* очереди делятся на линейные и кольцевые (циклические). По количеству позиций записи и считывания – на простые и *приоритетные*. Кроме того существует специальный вид очереди – *двухходовая очередь* или *дек* (*DEQue* – Double Ended Queue; *queue* – очередь).

На практике очереди могут реализовываться при помощи *одномерных массивов* или *связных списков*, что хорошо иллюстрирует различие между *логическим* и *физическим* уровнями структур данных (массив на физическом уровне является очередью на логическом).

Ниже приведены примеры процедур занесения и извлечения для линейной очереди на языках Паскаль и Си++.

Программа работы с линейной очередью на языке Паскаль.

```
Programm Queue_Test;
var
  q:array[0..30] of Integer; (* Глобальный массив, на котором и строится очередь *)
  qnext, (* Индекс занесения *)
  qindex, (* Индекс извлечения *)
  qlength:Integer; (* Длина очереди *)

procedure qstore(i:Integer); (* Процедура записи в очередь *)
begin
  if (qnext + 1) < qlength then
    begin
      qnext := qnext + 1;
      q[qnext] := i
    end
  else
    writeln('Мест нет')
  end;

function qretrieve():Integer; (* Функция считывания из очереди *)
```

```

begin
  if qindex = qnext then
    begin
      writeln('Очередь пуста');
      qretrieve := 0;
    end
  else
    begin
      qindex := qindex + 1;
      qretrieve := q[qindex]
    end;
  end;
end;

```

Функция `qretrieve` извлекает из очереди первый элемент; хранившиеся в этом элементе данные «разрушаются». Если из очереди удалены все элементы, она становится пустой. На самом деле `qretrieve` в этом примере не разрушает информацию, но информацию можно считать удаленной, так как дальнейший доступ к ней средствами самой очереди невозможен.

Собственно программа с результатами её выполнения.

<code>begin</code>		Состояние очереди
<code> qlength := 30;</code>		
<code> qnext := 0;</code>		
<code> qindex := 0;</code>		
<code> qstore(1);</code>		1
<code> qstore(2);</code>		2 1
<code> qstore(3);</code>		3 2 1
<code> qretrieve();</code>	{возвращает 1}	3 2
<code> qstore(4);</code>		4 3 2
<code> qretrieve();</code>	{возвращает 2}	4 3
<code> qretrieve();</code>	{возвращает 3}	4
<code> qretrieve();</code>	{возвращает 4}	Очередь пуста
<code>end.</code>		

Следует отметить, что при работе с очередью данные не перемещаются по массиву (хотя это и возможно, но крайне неэффективно, рисунок 7.1а). Вместо этого изменяются значения соответствующих переменных-индексов (рисунок 7.1б).

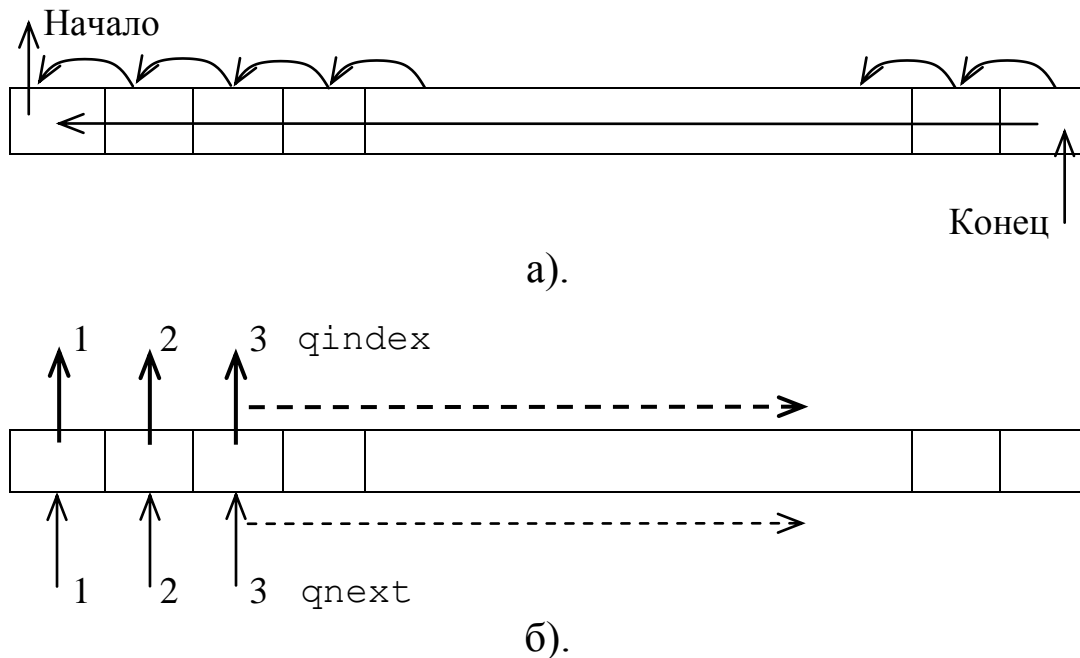


Рис. 7.1 – Принципы работы очереди

Аналогичная программа на языке Си++.

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

int q[30];
int qnext = 0, qindex = 0, qlength = 30;

// Занесение элемента в очередь.
void Insert(int i)
{
    if(qnext + 1 == qlength)
    {
        cout << "Мест нет!\n";
        return;
    }
    qnext++; // Смещение позиции записи
```

```

    q[qnext] = i; // Ввод данных
}

// Извлечение элемента из очереди.
int Retrieve()
{
    if(qindex == qnext)
    {
        cout<<"Очередь пуста.\n";
        return 0;
    }
    qindex++; // Смещение позиции считывания
    return q[qindex]; // Считывание
}

int main()
{
    Insert(1);
    Insert(2);
    Insert(3);
    cout<<Retrieve()<<endl; // Возвр. 1
    Insert(4);
    cout<<Retrieve()<<' \n'; // Возвр. 2
    cout<<Retrieve()<<endl; // Возвр. 3
    return 0;
}

```

	Результат	Состояние
Insert(1);		1
Insert(2);		2 1
Insert(3);		3 2 1
cout<<Retrieve()<<endl; // Возвр. 1		3 2
Insert(4);		4 3 2
cout<<Retrieve()<<' \n'; // Возвр. 2		4 3
cout<<Retrieve()<<endl; // Возвр. 3		4

Работа с линейной очередью, построенной на основе массива, становится невозможной (как запись, так и считывание), когда будет достигнут предельный размер массива, используемого для хранения очереди.

7.1. Кольцевая очередь

От указанного выше недостатка свободна **кольцевая** (или циклическая) очередь. В такой очереди массив, в котором хранятся элементы очереди, используется как кольцевой список, а не как линейный.

Программа работы с кольцевой очередью на языке Паскаль.

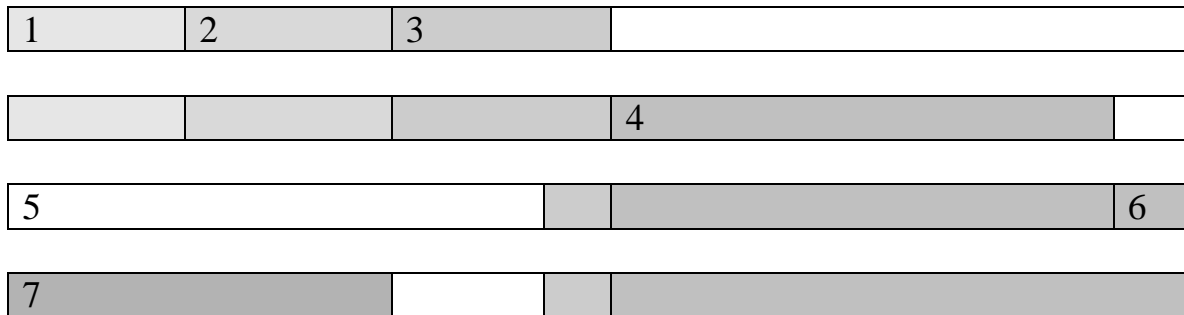
```
Programm Queue_Test2;
var
  q:array[0..30] of Integer;
  qnext, qindex, qlength:Integer;

procedure qstore(i:Integer);
begin
  if (qnext+1) <> qlength then
  begin
    q[qnext] := i;
    qnext := qnext + 1;
    if qnext = qlength then
      qnext := 0      (* Циклический переход *)
    end
  else
    writeln('Мест нет')
  end;
end;

function qretrieve():Integer;
begin
  if qindex = qlength then
    qindex := 0;      (* Циклический переход *)
  if qindex = qnext then
  begin
    writeln('Очередь пуста');
    qretrieve := 0;
  end
  else
  begin
    qretrieve := q[qindex];
    qindex := qindex + 1
  end;
end;
end;
```

Индексы сохранения `qnext` и извлечения `qindex` циклически возвращаются к началу массива (т.е. к нулю). В такую очередь можно поместить любое количество элементов, если они не только помещаются, но и извлекаются из очереди (рисунок 7.2).

Если очередь заполнена не полностью, то при последовательных записи и считывании свободная область циклически перемещается по очереди.









	свободная область	1...4, 6, 7 – этапы ввода данных
	данные, введённые на 1-м этапе	5 – считывание данных
	данные, введённые на 2-м этапе	
	данные, введённые на 3-м этапе	
	данные, введённые на 4-м и 6-м этапах	
	данные, введённые на 7-м этапе	

Рис. 7.2 – Принцип работы кольцевой очереди

Если $qindex$ на 1 больше $qnext$ – очередь заполнена и запись новых элементов невозможна, пока не будут прочитаны элементы, хранящиеся в очереди в данный момент. Если $qindex$ равен $qnext$ – очередь пуста. В остальных случаях существует место для помещения по крайней мере еще одного элемента.

begin	Состояние очереди
<code>qlength := 30;</code>	
<code>qnext := 0;</code>	
<code>qindex := 0;</code>	
<code>qstore(1);</code>	1
<code>qstore(2);</code>	2 1
<code>qstore(3);</code>	3 2 1
<code>qretrieve();</code> {возвращает 1}	3 2
<code>qstore(4);</code>	4 3 2
<code>qretrieve();</code> {возвращает 2}	4 3

```

    qretrieve();    {возвращает 3} 4
    qretrieve();    {возвращает 4} Очередь пуста
end.

```

Аналогичная программа на языке Си++.

```

// Занесение элемента в очередь.
void Insert_c(int i)
{
    if(qnext+1 == qindex || (qnext+1 == qlength
&& !qindex))
    {
        cout << "Мест нет!\n";
        return;
    }
    q[qnext] = i; // Запись
    qnext++;     // Смещение позиции записи
    if(qnext == qlength)
        qnext = 0; // Циклический переход.
}

// Извлечение элемента из очереди.
int Retrieve_c()
{
    if(qindex == qlength)
        qindex = 0; // Циклический переход.
    if(qindex == qnext)
    {
        cout << "Очередь пуста.\n";
        return 0;
    }
    qindex++; // Смещение позиции считывания
    return q[qindex-1]; // Считывание
}

```

Более короткие варианты этих же процедур (без проверки на переполнение и пустоту очереди).

```

const int maxN=15;
int N=maxN+1,head=N,tail=0; // размер, «голо-
ва» и «хвост»
int q[maxN+1]; // Очередь

void Ins(int i)
{
    q[tail++] = i; // Смещение и запись
    tail = tail % N; // Циклический переход
}

int Ret()
{
    head = head % N; // Циклический переход
    return q[head++]; // Смещение и чтение
}

```

Именно кольцевая очередь используется на практике, в том числе в качестве буферной памяти клавиатуры. Часто при заиклиивании программ в операционных системах MS-DOS или Windows вводимые с клавиатуры и поступающие в её буфер данные не считываются из этого буфера – очередь переполняется. Индикацией этого события является звуковой сигнал.

Другим вариантом построения очереди (в некоторых случаях, возможно, более гибким, хотя и требующим большего расхода памяти) является использование *связного списка*. В этом случае, во-первых, снимается ограничение на размер очереди, и, во-вторых, по другому используются переменные, указывающие на начало и конец очереди. При этом фактически устраняются различия между линейной и кольцевой очередями, независимо от типа связного списка – линейного или кольцевого. Такая реализация очереди будет рассмотрена позднее.

7.2. Приоритетная очередь

Приоритетная очередь отличается от обычной тем, что имеет дополнительные позиции считывания (одну или несколько), расположенные ближе к хвосту очереди, или дополнительные позиции записи (одну или несколько), расположенные ближе

к её голове. Чем ближе позиция считывания находится к хвосту очереди, тем выше приоритет этой позиции. Чем ближе позиция записи к голове очереди, тем выше приоритет. Предельным (но обычно не используемым) случаем такой очереди является одномерный массив, в котором приоритет позиции определяется её номером.

Процедуры, обслуживающие приоритетную очередь, должны изменять несколько индексов записи и/или считывания – основной (как в выше приведённых примерах), имеющий самый низкий приоритет, и один или несколько индексов с более высоким приоритетом (возрастающим, если таких индексов несколько). Данные могут быть считаны из дополнительной позиции (или записаны в неё), только если соответствуют уровню приоритета.

Для организации приоритетных очередей могут использоваться более сложные структуры данных, например, **Фибоначчиева пирамида** или **куча**.

7.3. Дек

Дек или **двухходовая** очередь отличается от обычной очереди тем, что данные в такой очереди могут «перемещаться» (физически или виртуально) в обоих направлениях; «голова» дека одновременно является его «концом» и наоборот. Одним из вариантов реализации дека является **двусвязный кольцевой список**.

Вопросы и задания для самоконтроля

- 7.1. Что представляет собой очередь?
- 7.2. Какие известны виды очередей?
- 7.3. На основе каких структур данных могут организовываться очереди?
- 7.4. Какой характер имеет операция считывания для очередей?
- 7.5. Какими свойствами обладают очереди?
- 7.6. Каким недостатком обладает простая очередь? Каков способ борьбы с этим недостатком?
- 7.7. Чем отличается приоритетная очередь от простой?

- 7.8. К каким структурам данных относятся очереди?
- 7.9. Какой метод доступа к элементам используется в очередях? Опишите особенности этого метода.
- 7.10. Какие операции над элементами характерны для очередей?
- 7.11. Перечислите основные отличия очереди от массива.
- 7.12. Для решения каких задач применяются очереди?
- 7.13. В чем преимущества циклической очереди? Чем она отличается от простой очереди?
- 7.14. К каким позициям очереди возможен доступ при записи и чтении информации?
- 7.15. Предложите процедуры работы с очередью на основе односвязного линейного списка. Обеспечьте надёжность работы такой очереди.
- 7.16. Что представляет собой дек?
- 7.17. Предложите методы работы с деком при помощи двусвязного кольцевого списка.

8. Стеки

Стек – разновидность очереди (а значит и разновидность списка), но с другим правилом доступа – **LIFO** (Last In and First Out – последним пришел и первым ушел). Ещё один термин (редко используемый) для обозначения этой структуры данных – **магазин**.

В стеке доступна единственная позиция – та, в которой находится последний введённый элемент – **вершина**. Иногда позицию, от которой стек начал заполняться данными, называют **дном** стека (хотя особого применения этот термин не имеет, поскольку дно стека, в котором содержится хотя бы один элемент данных, формально недоступно. Одно из немногих рациональных применений термина «дно» – признак пустого стека, когда дно и вершина совпадают).

Для стека, как и для очереди, характерны две операции – сохранение и извлечение, но применяются они к одной и той же позиции. Операция извлечения удаляет элемент из списка и уничтожает его содержимое. Произвольный доступ к элементам стека не допускается.

Области применения стеков такие же, как и у очередей:

Системные –

- передача процедурам и функциям аргументов по значению;
- сохранение и восстановление содержимого регистров общего назначения процессора при вызове процедур (прологи и эпилоги функций).

Прикладные –

- стековая (магазинная) память, например в языке программирования **Forth**;
- вспомогательные структуры данных (например, при поиске в графе).

Примеры стеков. Второе название стека – магазин, – приводит к механическому примеру стека – магазин стрелкового оружия. Ещё один простой пример – стакан.

Стеки, также как и очереди, могут реализовываться при помощи **одномерных массивов** или **связных списков**.

Ниже приведены примеры процедур занесения и извлечения данных для стека на основе массива на языках Паскаль и Си++.

Программа работы со стеком на языке Паскаль.

```
Programm Stack_Test;
const N=30;
var
  stack: array[0..N] of char;
  slen, pos: integer;

procedure push(i:char);
begin
  if pos < slen then
  begin
    stack[pos] := i;
    pos := pos + 1
  end
  else
    writeln('Стек полон.')
  end;

function pop():char;
begin
  if pos=0 then
  begin
    writeln('Стек пуст.');
```

```
    pop := 0
  end
  else
  begin
    pos := pos - 1;
    pop := stack[pos]
  end;
end;
```

Pos – индекс следующей открытой позиции – вершины стека. Если Pos больше индекса последнего сохранения, значит стек заполнен; если Pos = 0 – стек пуст.

begin	Содержимое стека
slen := N;	
pos := 0;	
push('A');	A
push('B');	B A
push('C');	C B A
pop(); {Возвращает C}	B A
push('D');	D B A
pop(); {Возвращает D}	B A
pop(); {Возвращает B}	A
pop(); {Возвращает A}	Стек пуст
end.	

Аналогичная программа на языке Си++.

```

#include <iostream>
using namespace std;
const int N = 30;
char stack[N];
int pos = 0, slen = N;

// Занесение элемента в стек.
void Push(char i)
{
    if (pos == slen)
    {
        cout<<"Стек полон!\n";
        return;
    }
    stack[pos] = i;
    pos++;
}

// Извлечение элемента из стека.
char Pop()
{
    if (pos == 0)
    {

```

```

        cout<<"Стек пуст.\n";
        return 0;
    }
    pos--;
    return stack[pos];
}

int main()
{
    Push('A');
    Push('B');
    Push('C');
    cout>>Pop()>>endl; // Возвр. C
    Push('F');
    cout>>Pop()>>'\\n'; // Возвр. F
    cout>>Pop()>>endl; // Возвр. B
    return 0;
}

```

	Содержимое
Push('A');	A
Push('B');	B A
Push('C');	C B A
cout>>Pop()>>endl; // Возвр. C	B A
Push('F');	F B A
cout>>Pop()>>'\\n'; // Возвр. F	B A
cout>>Pop()>>endl; // Возвр. B	A

Представим стек, как динамическую цепочку звеньев, т.е. **связный список**. Первое звено – вершина. Так как доступна только вершина, то в таком списке главное звено становится излишним. Стек как единый объект представляет указатель, значение его – адрес вершины стека. Каждое звено цепочки содержит указатель на следующее звено, «дно» стека – элемент, занесенный первым, имеет этот указатель равным `nil`.

Программа работы с динамическим стеком на языке Паскаль.

```

Programm Stack_Test;
type      (* Секция описания типов *)
    stype = real;
    next = ^elem;
    elem=record      (* Запись *)
        dr: next;
        data: stype
    end;
    stack = next;
var
    p: stack;

```

```

procedure push(var st: stack; newl: stype);
var
    q:next;
begin
    new(q);
    q^.data := newl;      (* новое звено *)
    q^.dr := st;
    st := q;             (* новое звено - вершина *)
end;

```

Наличие у процедуры аргумента `push(var st: stack, ...)` позволяет работать с несколькими стеками.

```

function pop(var st: stack):stype;
var
    q:next;
begin
    if st = nil then
        writeln('Стек пуст.')
    else
        begin
            pop := st^.data;
            q := st;
            st := st^.dr;
            dispose(q)
        end;
end;

var P: stack;

```

Если стек пуст, то значение указателя `P` равно `nil`. К началу использования стека его нужно сделать пустым при помощи оператора `p:=nil`.

```

begin
    p := nil;
    push(p, 43);
    writeln(pop(p):3:9);
    push(p, 54);
    writeln(pop(p):3:9);

```

```
    readln;
end.
```

Аналогичные процедуры на языке Си++.

```
typedef float stype; /* Необязательное пере-
именование типа float (может использоваться любой
тип) в имя stype */
```

```
struct elem
{
    elem* dr; // указатель на другой такой же элемент
    stype data; // хранимые в стеке данные
};
```

```
void Push(elem*& st, stype newel)
{
    elem* q = new elem;
    q->data = newel; // Новое звено
    q->dr = st;
    st = q; // Новое звено становится вершиной стека.
}
```

```
stype Pop(elem*& st)
{
    stype a;
    elem *q;
    if (st == NULL)
    {
        cout<<"Стек пуст.\n";
        return;
    }
    a = st->data;
    q = st;
    st = st->dr;
    delete q;
    return a;
}
```


Вопросы и задания для самоконтроля

- 8.1. Что представляет собой стек?
- 8.2. На основе каких структур данных могут организовываться стеки?
- 8.3. Чем отличается стек на основе массива от стека на основе связного списка?
- 8.4. Чем отличается стек на основе связного списка от собственно связного списка?
- 8.5. К каким структурам данных относятся очереди и стеки?
- 8.6. Перечислите основные отличия стека от очереди.
- 8.7. Какой метод доступа используется при доступе к элементам стека? В чем его особенности?
- 8.8. Перечислите основные операции, применяемые при работе со стеками. К каким позициям в стеке они могут применяться?
- 8.9. Какой характер имеет операция считывания для стеков?
- 8.10. Перечислите задачи, для решения которых применяются стеки.
- 8.11. Изобразите структуру звена динамического стека.
- 8.12. Как определить наличие или отсутствие элементов в стеке?
- 8.13. Как определить количество элементов в стеке?

9. Связные списки

Понятие «*списки*» включает в себя самые разные структуры данных. Это могут быть и массивы, в том числе массивы записей, и специальные динамические структуры данных, и даже деревья. Общим для них является то, что они содержат набор записей одного вида, ограниченный по размеру или неограниченный, упорядоченный или неупорядоченный. Данные, хранящиеся в этих записях, обычно *логически* связаны между собой, например, фамилии студентов одной группы и т.п. В тексте программы такая связь может выражаться в том, что все такие элементы хранятся в одном и том же массиве как непрерывном блоке памяти. Но кроме общего имени массива (и адреса его начала) между этими элементами никакой другой *физической* связи нет, и на *физическом* уровне подобные списки могут быть названы «*несвязными*» (или, что то же самое, «*несвязанными*»). Т.е. внутри них нет связей (их элементы не связаны друг с другом *физически*).

Типичным примером физически несвязного списка является массив. В этой главе мы рассмотрим те самые «специальные динамические структуры данных», которые и получили название *связных списков*.

Вспомним общие черты очередей и стеков:

- строгие правила доступа к данным;
- операции извлечения (считывания) данных являются разрушающими.

Связные списки свободны от этих ограничений. Они допускают гибкие методы доступа; извлечение (чтение) элемента из списка не приводит к удалению его из списка и потере данных. Для фактического удаления элемента из связного списка требуется специальная процедура.

Связные списки представляют собой динамические (фактически, линейные!) структуры данных (*динамические цепочки звеньев*), в которых однотипные элементы (звенья) каким-либо образом связаны между собой, обычно на *физическом* уровне. Связь между элементами можно осуществить за счёт хранения в одном элементе адреса другого *такого же* элемента (того же типа). Т.е. каждый информационный элемент содержит внутри себя указатель на собственный тип. Учитывая, что кроме этого указа-

теля должны присутствовать полезные данные, тип информационного элемента оказывается *записью*. Например, для простейшего вида списка этот тип может быть следующим (на языке Си++):

```
struct Link1
{
    int data;
    Link1* next;
};
```

Классификация связанных списков. По числу связей (и одновременно, *направлению*) списки бывают *односвязными* (однонаправленными), *двусвязными* (двунаправленными) и *многосвязными*.

По способу организации связей (или по *архитектуре*) списки могут быть *линейными* и *кольцевыми* (циклическими). (Если список не является ни линейным, ни кольцевым, то остаётся единственный вариант – ветвящийся список, фактически являющийся одной из *древовидных* структур данных.)

По степени упорядоченности хранимых данных списки могут быть *упорядоченными* и *неупорядоченными*. Такое разделение иногда бывает удобно на практике, но для поддержания упорядоченности списков приходится прибегать к специальным мерам.

Самое первое звено списка может содержать полезные данные и обрабатываться как все остальные звенья списка (подобно стеку, рассмотренному в главе 8). В этом случае получается список *без выделенного ведущего звена*. Либо самое первое звено списка используется только для унификации операций добавления и удаления звеньев и, обычно, не содержит полезных данных и никогда не удаляется, пока список существует. В таком случае речь идёт о списке *с выделенным ведущим звеном*.

Для списков, по сравнению с очередями и стеками, имеется значительно больше *операций*, которые включают в себя:

- добавление нового звена списка (вставка звена);
- удаление звена;
- просмотр (или прохождение) списка;

- поиск данных в списке;
- создание ведущего (заглавного) звена (при необходимости);
- сортировка списка;
- обращение (реверсирование) списка, т.е. перестановка всех его звеньев в обратном порядке.

Рассмотрим основные из этих операций для каждого вида списка отдельно, вместе с особенностями этих видов списков.

9.1. Линейный односвязный список

Линейный односвязный список является самым простым видом связанных списков. Такой список можно определить, в том числе, с помощью приведённого выше описания типа.

Процедуры работы с линейным односвязным списком на языке Паскаль.

```

Type
  rell = ^elem; (* Указатель на запись *)
  elem = record

    next: rell;
    data: <Тип хранимых данных> (* Любой допус-
тимый тип данных *)
  end;
var
  L1: rell;

```

Структура такого списка показана на рисунке .

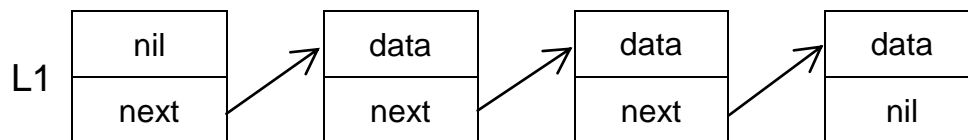


Рис. 9.1 – Односвязный линейный список

Для того, чтобы такие операции, как добавление или удаление звена, выполнялись одинаково, независимо от места их выполнения в списке, удобно использовать *ведущее* или *заглавное*

звено – самое первое звено списка, в котором не обязаны храниться полезные данные. Его создание для односвязного списка можно осуществить следующим образом:

```
var a, L1: list1;
begin
  ...
  new(L1);
  L1^.next := nil;
  a := L1;
  ...
```

Указатель на начало списка L1, значением которого является адрес ведущего звена, представляет список как единый программный объект.

Указатель на следующий элемент в последнем звене списка имеет значение nil (NULL или просто 0 в Си/Си++), что является признаком *линейного* списка.

```
procedure insert1(link: rell; info: <Тип>);
var q: rell;
begin
  new(q);
  q^.data := info;
  q^.next := link^.next;
  link^.next := q
end;
```

```
procedure delete1(link:rell);
var q: rell;
begin
  if link^.next <> nil then
    begin
      q := link^.next;
      link^.next := link^.next^.next;
      dispose(q);
    end
end;
```

```

function search1(l: rell; info: <Тип>; var r:
rell):boolean;
var q: rell;
begin
  search1 := false;
  r := nil;
  q := l^.next;
  while (q <> nil) and (r <> nil) do
  begin
    if q^.data = info then
    begin
      search := true;
      r := q
    end;
    q := q^.next
  end
end;

```

Процедуры добавления и удаления звеньев являются **критическими** с точки зрения сохранения целостности списка. При неправильном выполнении этих процедур (т.е. при неправильной очередности выполнения операций присваивания) возможны 2 ошибочные ситуации:

1. Список «рвётся» по месту вставки или удаления звена, и звено, оказавшее последним, замыкается либо само на себя (чаще всего) (т.е. указатель `next` или аналогичный ему в этом звене получает значение адреса этого же звена), либо на одно из предшествующих звеньев (в зависимости от неправильной реализации операций вставки или удаления звена). При попытке просмотра списка процедура просмотра зацикливается и бесконечно выводит содержимое одного и того же звена (или нескольких звеньев).

2. Список так же «рвётся» по месту вставки или удаления звена, но указатель в звене, ставшем последним, получает какое-то произвольное значение, которое трактуется как адрес следующего звена (реально не существующего), у которого также есть указатель `next`, содержащий какой-то адрес, и так далее, до тех пор, пока случайно не попадётся блок данных, для которого указатель `next` не будет равен **нулю**. При попытке просмотра списка на дисплей сначала выводятся правильные данные, а затем слу-

чайный набор символов.

В обоих случаях к звеньям в «оторвавшейся» части («хвосте») списка больше нет доступа, и хранящиеся в них данные можно считать потерянными.

Для предотвращения возникновения таких ошибок следует соблюдать **правильный порядок проведения связей** (т.е. присваивания указателей) при вставке нового звена и удалении существующего (очередность операций указана в комментариях к листингу процедуры).

Добавление звена в произвольную позицию за ведущим звеном.

```
void Insert1(Link1* link, int data) // link -  
звено, за которым вставляется новое  
{  
    Link1* q = new Link1; // 1 Выделение памяти  
под новое звено  
    q->data = data; // 2 Ввод данных  
    q->next = link->next; // 3 Проведение связи  
от нового звена к следующему  
    link->next = q; // 4 Проведение связи  
от "старого" звена у новому  
}
```

Возможность перемещаться по односвязному списку только в одном направлении приводит к тому, что при **удалении** звена приходится задавать не реально удаляемое звено, а **предшествующее** ему. Это делается для того, чтобы можно было скорректировать связь для предшествующего звена, добраться до которого от удаляемого иначе невозможно.

Удаление звена из любого места списка за ведущим звеном.

```
void Delete1(Link1* link) // link - звено,  
предшествующее удаляемому  
{  
    Link1* q;  
    if (link->next) // Проверка на нали-  
чие звена, следующего за link, то же, что и  
if(link->next!=NULL)
```

```

    {
        q = link->next          // 1 Запоминание
удаляемого звена для операции delete
        link->next = q->next; // 2 Проведение но-
вой связи в обход удаляемого звена
        delete q;             // 3 Освобождение памяти
    }
}

```

Одной из наиболее простых операций со всеми типами спи-сков является их **прохождение**, т.е. поочерёдное получение дос-тупа ко всем элементам. Приведём процедуру, реализующую эту операцию для **просмотра** списка (другие варианты использова-ния прохождения – поиск данных и сохранение списка в файл). В случае перемещения по двусвязному списку в «прямом» направ-лении эта процедура является **одинаковой** для одно- и двусвязно-го **линейных** списков.

Просмотр односвязного линейного списка.

```

void Show(Link1* link)
{
    Link1 *q = link->next; // Учитывается наличие
"пустого" ведущего звена
    while (q)              // или while (q!=NULL)
    {
        cout << q->data << ' '; // или другая операция
        q = q->next;           // Переход по списку
    }
    cout << endl;
}

```

Поиск в списке является вариантом операции просмотра и отличается тем, что:

1. вместо операции вывода на экран (`cout << q->data`) используется операция сравнения искомым данным с хранящими-ся в звеньях списка;

2. если искомые данные найдены, нет необходимости пере-мещаться по списку дальше.

Поиск в *односвязных* списках имеет следующую особенность. Если он выполняется в сочетании с удалением, то результатом поиска может (или должно) быть не то звено, в котором содержатся искомые данные, а *предшествующее* ему. В итоге для поиска в списке могут потребоваться либо две различные процедуры – «обычная» и находящая предыдущее звено, либо одна универсальная, позволяющая найти оба звена – с искомыми данными и предшествующее ему. Рассмотрим в качестве примера именно этот вариант.

Универсальная процедура поиска (находит звено с ключом поиска и предшествующее ему).

```

int Search(Link1* Start, // Точка начала поиска
          Link1*& Find, // Указатель для звена с искомыми данными
          Link1*& Pred, // Указатель для предыдущего звена
          int Key ) // Ключ поиска
{
    Link1* Cur = Start->next; // Текущее звено
    Pred = Start; // Предыдущее звено ("отстаёт" на 1 шаг от текущего)
    int Success = 0; // Признак успеха поиска (установлен в 0)
    while (Cur && !Success) // Операция логическое "И"
    {
        if (Cur->data == Key) // Нашли
        {
            Find = Cur; // Запоминание найденного звена
            Success = 1; // Установка в 1 признака успеха
            break; // Выход из цикла при удачном поиске
        }
        Pred = Cur; // Перемещение предыдущего звена
        Cur = Cur->next; // Переход по списку вперёд
    }
    return Success;
}

```

Следует отметить, что возможны разные (в том числе более короткие) варианты реализации такого алгоритма, например, без переменной `Success` (вместо неё используется указатель `Find`, который до начала поиска должен получить значение `NULL` и сохранить его при неудачном поиске).

Процедура, которая находит только искомое звено, является более простой, – в ней не нужен указатель `Pred` и все операторы, в которых он используется.

Похожая процедура применяется для односвязного *кольцевого* списка. Отличие её от рассмотренного примера заключается в условии продолжения цикла в операторе `while`:

```
while(Cur != Start && !Success) // Для кольце-
вого списка
```

Ведущее (или заглавное) звено. Все приведённые выше примеры на языке Си++ подразумевают наличие в списке ведущего звена. Создаваться это звено может либо отдельной процедурой, либо следующим набором операторов (эти же операторы и будут находиться в процедуре):

```
Link1 *L1 = new Link1; // Выделение памяти под звено
L1->next = NULL; // Ведущее звено одновременно явля-
ется последним
```

Возможна работа со связным списком и без выделенного ведущего звена, т.е. первое звено является обычным, в нём содержатся полезные данные. Именно такой вид списка использовался для организации стека. Ещё один пример использования подобных списков – *очередь*.

Занесение в очередь и извлечение из очереди, построенной на основе двусвязного списка.

```
void queue_in(Link2*& q, // «Голова» очереди
Link2*& e, // «Хвост» очереди
int k) // Вводимые данные
{
Link2* n = new node; // Новый узел
n->data = k;
n->next = q;
```

```

    if (q)
        q->prev = n;
    else
        e = n;
    n->prev = 0;
    q = n;
}

int queue_out(Link2*& q, Link2*& e)
{
    int k = e->data;
    Link2* d = e;
    e = d->prev;
    if (e)
        e->next = 0;
    else
        q = e;
    delete d;
    return k;
}

```

Для упрощения обработки «головы» и «хвоста» очереди использован двусвязный список.

В остальных случаях (т.е. для обычных списков) использование ведущего звена является предпочтительным, т.к. позволяет избежать проверок (операторов `if`) при добавлении и удалении звеньев, т.е. унифицирует операции со списком независимо от места их выполнения. В противном случае либо требуются отдельные процедуры, например, вставки звена в начало списка, либо более универсальные процедуры с проверкой места вставки или удаления, как в показанных выше процедурах работы с очередью. В обоих случаях общий размер машинного кода этих процедур вряд ли окажется меньше размера ведущего звена и говорить о каком-либо выигрыше в объёме памяти при отказе от ведущего звена не приходится.

Недостатки односвязного списка заключаются в следующем:

1. По такому списку можно перемещаться только от начального звена к конечному. Начинать можно с любого звена в спи-

ске, но если вдруг возникнет необходимость обратиться к предшествующим элементам, придется начинать с начального звена, что неудобно, нерационально и усложняет алгоритмы обработки данных.

2. Наличие только одной связи снижает надёжность хранения данных в односвязном списке.

3. Следствием первого недостатка является усложнение взаимодействия операций *поиска* и *удаления*.

Достоинствами этого списка являются меньший расход памяти по сравнению с другими связными динамическими структурами данных (всего один указатель) и простота операций.

9.2. Линейный двусвязный список

Этот список свободен от недостатков, присущих односвязному списку. Для этого в каждое звено добавлен *еще один указатель на тип звена*, значением которого является адрес *предыдущего звена* списка. Тип звена на языке Си++:

```
struct Link2
{
    int data;
    Link2 *next, *prev;
};
```

Структура списка будет выглядеть следующим образом (значения нулевых указателей показаны для языка Паскаль):

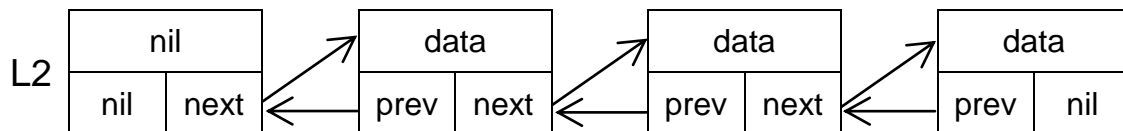


Рис. 9.2 – Линейный двусвязный список

Ведущее звено этого списка создаётся следующим набором операторов.

```
Link2 *L2 = new Link2;
L2->next = NULL;
L2->prev = NULL;
```

По операторам создания ведущего звена можно судить о том, является список одно- или двусвязным, линейным или кольцевым. Нулевые значения указателей `next` и `prev` являются признаком линейного списка.

Графически состояние списка после создания ведущего звена может быть отображено следующим образом:



Рис. 9.3 – Ведущее звено пустого двусвязного линейного списка

Преимущества двусвязного списка:

- есть возможность перестроить поврежденный список;
- проще выполняются некоторые операции (например, удаление).

9.3. Операции с двусвязным списком

Всё, что касалось операций добавления звена и его удаления для односвязного списка, справедливо и для двусвязного. Так же должен соблюдаться правильный порядок проведения (или удаления) связей между звеньями, но таких операций стало больше, т.к. должны обрабатываться *два* указателя. Кроме того, каждая из операций обладает дополнительными особенностями:

1. При добавлении нового звена в пустой список, т.е. содержащий только ведущее звено (см. предыдущий рисунок), или при добавлении звена в конец списка (для пустого списка обе эти ситуации совпадают) необходимо проверять наличие звена, которое будет следующим за добавляемым (для пустого списка или конца списка такого звена *нет*, а значит *нет* и указателя, обозначающего связь). Такая же проверка должна выполняться при удалении звена.

2. Возможность перемещаться по списку в обоих направлениях позволяет *напрямую* задавать удаляемое звено (а не ему предшествующее, как в односвязном списке). Следствием этого является упрощение как операции удаления, так и операции по-

иска. При поиске достаточно получить только искомое звено.

Добавление звена в произвольную позицию за ведущим звеном.

```
void Insert2(Link2* St, int data)
{
    Link2* q = new Link2; // 1 Выделение памяти
    под звено
    q->data = data; // 2 Ввод данных
    q->next = St->next; // 3 Проведение связи от
    нового звена вперёд
    q->prev = St; // 4 Проведение связи от ново-
    го звена назад
    St->next = q; // 5 Проведение связи от пре-
    дыдущего звена к новому
    if (q->next) // Проверка наличия следующего
    звена
        q->next->prev = q; // 6 Проведение связи
    от следующего звена к новому
}
```

Удаление звена из любого места списка за ведущим звеном.

```
void Delete2(Link2* del)
{
    del->prev->next = del->next; // 1 Обработка
    связи вперёд
    if (del->next)
        del->next->prev = del->prev; // 2 Обработ-
    ка связи назад
    delete del; // 3 Освобождение памяти
}
```

Поиск в двусвязном списке.

```
int Search2(Link2* Start, Link2*& Find, int
Key)
{
    Link2* Cur=Start->next;
    int Success = 0;
```

```

while (Cur && !Success)
{
    if (Cur->data == Key)
    {
        Find = Cur;
        Success = 1;
        break;
    }
    Cur = Cur->next;
}
return Success;
}

```

Эта процедура поиска фактически является (за исключением типа данных Link2 вместо Link1) процедурой «обычного» поиска (т.е. не для удаления) в односвязном списке.

Операция *просмотра списка в прямом направлении* ничем не отличается от просмотра односвязного списка.

9.4. Кольцевые списки

Если значение указателя последнего звена линейного односвязного списка заменить с nil (или NULL) на адрес ведущего звена, то линейный список превратится в односвязный кольцевой список.

Для двусвязного списка, кроме того, нужно заменить с nil на адрес последнего звена значение второго указателя в ведущем звене. Получится двусвязный кольцевой (циклический) список.

В односвязном кольцевом списке можно переходить от последнего звена к заглавному, а в двусвязном – еще и от заглавного к последнему.

Односвязный кольцевой список выглядит так:

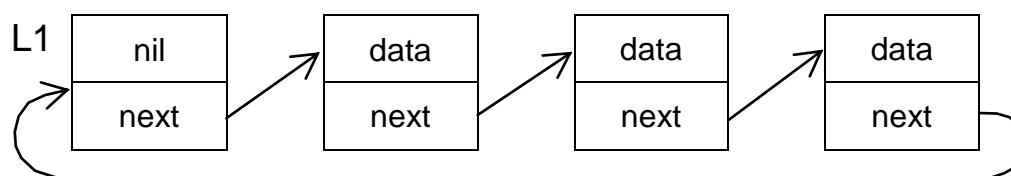


Рис. 9.4 – Кольцевой односвязный список

Кольцевой список, как и линейный, идентифицируется как единый программный объект указателем, например L1, значением которого является адрес заглавного звена.

Возможен другой вариант организации кольцевого списка:

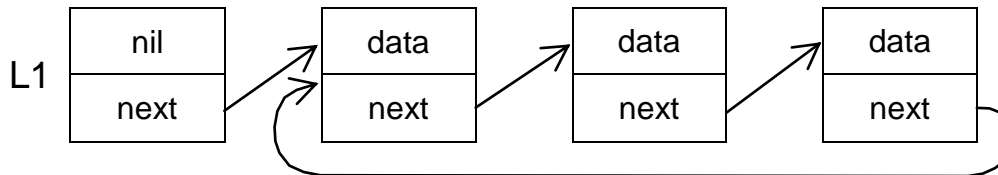


Рис. 9.5 – Кольцевой односвязный список. Второй вариант

Оба варианта сопоставимы по сложности. Для *первого* варианта проще выполняется вставка нового элемента как в начало списка (после заглавного звена), так и в конец – так как вставка звена в конец кольцевого списка эквивалентна вставке перед заглавным звеном, но каждый раз при циклической обработке списка нужно проверять, не является ли текущее звено заглавным (или не совпадает ли текущее звено с точкой начала обработки).

Рассмотрим *операции с кольцевыми списками*.

Отсутствие «последнего» звена приводит к ещё большему упрощению операций добавления и удаления, по сравнению с одно- и двусвязным линейным списком. Например, для односвязного кольцевого списка в процедуре *удаления* отсутствует оператор `if` – проверка на существование звена, следующего за заданным (в кольцевом списке такое звено всегда есть). Такие же операторы *отсутствуют* в процедурах добавления и удаления звеньев для двусвязного кольцевого списка.

При циклической обработке кольцевого списка нужно учесть, что формально последнего звена нет.

Процедуры и программа работы с двусвязным кольцевым списком на языке Паскаль.

```

Type
  rel2 = ^elem2;
  elem2 = record
    next: rel1;
    prev: rel2;
  
```



```

    data: <Тип хранимых данных>
end;
list2 = rel2;

procedure insert2(pred: rel2; info: <Тип>);
var
    q: rel2;
begin
    new(q); (* Создание нового звена *)
    q^.data := info;
    q^.next := pred^.next;
    q^.prev := pred^.next^.prev;
    pred^.next.prev := q;
    pred^.next := q
end;

```

При вставке в начало списка (после заглавного звена) нужно указать в качестве аргумента `pred` адрес заглавного звена, то есть указатель на список `L2`.

```

procedure delete2(del: rel2);
begin
    del^.next^.prev := del^.prev;
    del^.prev^.next := del^.next;
    dispose(del);
end;

function search2(list: rel2; info: <Тип>; var
point: rel2): boolean;
var
    p,q: rel2;
b: boolean;
begin
    b := false;
    point := nil;
    p := list;
    q := p^.next;
    while (p <> q) and (not b) do
    begin
        if q^.data = info then

```

```

begin
    b := true;
    point := q
end;
q := q^.next
end;
search2 := b
end;

...
...

var
    l2: list2;
    r: rel2;
begin (* Создание заглавного звена *)
    new(r);
    r^.next := r;
    r^.pred := r;
    l2 := r;

    ...
    ...

end.

```

9.5. Процедуры работы с двусвязным кольцевым списком на языке Си++

Тип данных для кольцевого двусвязного списка такой же, как и для двусвязного линейного. То же самое справедливо для односвязных списков. ***По типу звена списка нельзя судить о его архитектуре, можно только о числе связей.***

Добавление звена в произвольную позицию за ведущим звеном.

```

void Insert2(Link2* Pred, int data)
{
    Link2* Loc = new Link2;    // 1
    Loc->data = data;          // 2
}

```

```

    Loc->next = Pred->next;    // 3
    Loc->prev = Pred;         // 4
    Pred->next = Loc;        // 5
    Loc->next->prev = Loc;    // 6
}

```

Удаление звена из любого места списка за ведущим звеном.

```

void Delete2(Link2* Del)
{
    Del->prev->next = Del->next;    // 1
    Del->next->prev = Del->prev;    // 2
    delete Del;                  // 3
}

```

Поиск.

```

int Search2(Link2* Start, Link2*& Find, int
Key)
{
    Link2* Cur = Start->next;
    int Success = 0;
    while (Cur != Start && !Success)
    {
        if (Cur->data == Key)
        {
            Find = Cur;
            Success = 1;
            break;
        }
        Cur = Cur->next;
    }
    return Success;
}

```

Прекращение поиска в случае неудачи происходит при достижении «текущим» указателем Cur точки начала поиска Start, т.е. при невыполнении условия в операторе while:

```

while(Cur != Start ... )

```

Ведущее звено кольцевого 2-связного списка создаётся набором операторов:

```
Link2 *L2 = new Link2;  
L2->next = L2;  
L2->prev = L2;
```

Графически состояние списка после создания этого звена может быть представлено таким образом:

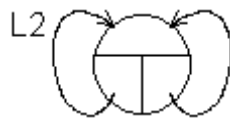


Рис. 9.6 – Ведущее звено пустого двусвязного кольцевого списка

9.6. Многосвязные списки

Многосвязные списки представляют собой динамические структуры данных, в основу которых положены одно- или двусвязные списки, в которых имеются дополнительные связи между звеньями. Чаще всего, такие связи проводятся между далеко отстоящими звеньями, например, обозначающими категории данных. Пример многосвязного списка показан на следующем рисунке.



Рис. 9.7 – Многосвязный список

Переход между звеньями AA и BA может быть выполнен по дополнительной связи, в обход звеньев AB и AV. Из-за такого характера перемещения эти списки иногда называют **скип-списками** (skip – перепрыгивать). А при характере размещения данных, подобном показанному на этом рисунке, такие списки называют **словарными** (иногда просто **словарями**, но термин «словарь» может использоваться в теории структур данных в разных значениях). Возможны и другие варианты многосвязных

списков. Фактически, в некоторых случаях такие списки удобно представлять в виде графов.

Вопросы и задания для самоконтроля

- 9.1. Что представляют собой связные списки?
- 9.2. К каким классификационным группам структур данных относятся списки?
- 9.3. Какие существуют разновидности связных списков?
- 9.4. В чем состоит отличие несвязного списка от массива?
- 9.5. В чем состоит отличие связного списка от массива?
- 9.6. В чем состоит отличие линейного списка от кольцевого?
- 9.7. В чем заключаются недостатки односвязного списка?
- 9.8. В чем состоит отличие односвязного списка от двусвязного?
- 9.9. Какие операции применяются для связных списков?
- 9.10. В чем отличие считывания информации из списка от считывания из очереди или стека?
- 9.11. Особенности операций вставки и удаления для связных списков.
- 9.12. В чем отличие операции вставки в двусвязный список от вставки в односвязный список?
- 9.13. В чем отличие операции удаления из двусвязного списка от удаления из односвязного списка?
- 9.14. В чем заключаются особенности работы с кольцевыми списками?
- 9.15. Какой тип должно иметь звено связного списка? Почему?
- 9.16. Что обязательно должно содержать звено связного списка?
- 9.17. В чем состоит отличие звена двусвязного списка от звена односвязного списка?
- 9.18. В чем состоит отличие связного списка от стека, организованного в виде связного списка?
- 9.19. Перечислите сходства и отличия списков и очередей.
- 9.20. Перечислите достоинства и недостатки линейных односвязных списков.
- 9.21. В чем заключается поиск в списке?

9.22. Измените функцию поиска в односвязном списке так, чтобы не использовалась локальная переменная `Success`.

9.23. Упростите функцию поиска в односвязном списке так, чтобы она находила только звено с искомыми данными, но не предыдущее звено.

9.24. Упростите функцию поиска в односвязном списке так, чтобы она находила только предыдущее по отношению к искомому звено.

9.25. Изобразите структуру линейного односвязного списка.

9.26. Изобразите структуру линейного двусвязного списка.

9.27. Перечислите достоинства и недостатки линейных двусвязных списков.

9.28. Изобразите возможные структуры двусвязных кольцевых списков.

9.29. Объясните назначение выделенного заглавного звена в списках.

9.30. На каких структурах данных могут строиться списки?

9.31. Предложите алгоритм удаления из связного списка каждого чётного звена.

9.32. Предложите методы использования двусвязного кольцевого списка для работы с деком.

10. Древовидные структуры данных

Существует несколько определений того, что такое *дерево* как структура данных. Например, дерево – это *лес*, состоящий из одного связного компонента [16] (т.е. из одного дерева) (рекурсивное определение). Или (с точки зрения *теории графов*), дерево – это граф без циклов (*Directed Acyclic Graph, DAG*) [6]. Рассмотрим деревья с точки зрения структур данных, которые должны использоваться в программах и храниться в памяти.

Деревья или, в более широком смысле, *древовидные структуры данных*, представляют собой динамические связные структуры, отличающиеся от списков тем, что система связей не носит линейного характера, а образует *ветви*, подобно природному дереву (откуда и произошло название этой структуры данных).

Эти структуры данных в общем случае можно разделить на две группы, которые отличаются друг от друга способом построения и (как следствие) реализацией процедур обработки – собственно *деревья* и *пирамиды* (или «*кучи*»). Но у обеих этих групп сохраняется общий древовидный характер и часто их объединяют под общим коротким названием «*деревья*». Отличие *пирамид* от *деревьев* (и значения термина «*куча*») будет рассмотрено позже.

Часто предполагается, что дерево – это *ориентированная* (направленная) структура данных, и при реализации связей, аналогичной спискам, направление (т.е. ориентация) задаётся автоматически. Кроме того, некоторые алгоритмы обработки деревьев предполагают, что дерево является ориентированным (что не всегда удобно), в то время как в других алгоритмах считается, что дерево – *неориентированная* структура (с точки зрения физического уровня – двунаправленная, т.к. для обеспечения возможности перемещения по ветвям дерева во всех направлениях – и вверх, и вниз, – физические связи должны быть направлены в обе стороны).

10.1. Классификация

Существует очень большое количество видов *древовидных структур данных*, которые можно классифицировать по нескольким различным признакам. Одно такое разделение уже было приведено – *деревья и пирамиды*.

Из других признаков классификации следует указать (в первую очередь) *число ветвей*, отходящих от каждого узла дерева. Деревья могут быть:

- *двоичными* (или *бинарными*) – имеющими не более двух ветвей (примеров таких деревьев очень много, они будут приведены позже);

- с числом ветвей больше 2 – часто такие деревья называют *мультивариантными* (*многопутевыми, сильноветвящимися*) или *К-деревьями* (т.е. *К-мерными*). Примерами могут быть *Б-деревья* (различные деревья Байера-МакКрейта с вариантами, например, *2-3-деревья, 2-3-4-деревья*) и т.п. Видов *К-деревьев* очень много (свыше 40 различных древовидных структур для представления многомерной информации: *R-Tree, R+-Tree, Hilbert R-Tree, hB-Tree, hBII-Tree, BV-Tree, BD-Tree, GBD-Tree, G-Tree, SKD-Tree, kd-Tree, kd2B-Tree, BSP-Tree, LSD-Tree, P-Tree, TR-Tree, Cell-Tree, Quadtree, Grid File, Z-Hashing, SS-Tree*).

Важным признаком является состояние *сбалансированности* дерева и способы его достижения. Деревья могут быть:

- сбалансированными;
- идеально сбалансированными (отличие этих двух первых состояний будет рассмотрено ниже);
- вырожденными;
- отличающимися более или менее сильно от сбалансированных и от вырожденных.

Автоматическая балансировка дерева выполняется, например, для *AVL-деревьев, красно-чёрных* деревьев (все эти деревья являются *двоичными*), *Б-деревьев* и некоторых других видов деревьев.

Ещё один возможный признак классификации – *применение* деревьев (хотя не всегда из названия дерева следует, что оно применяется именно с этой целью). В качестве примера можно указать деревья *поиска* или *сортировки*, но из названия «*дерево*

поиска» не следует, что оно применяется только для поиска.

Отдельные типы деревьев применяются для решения специальных задач, например *остовные* деревья на графах (*каркасы* графов, *Spanning Tree*) или *PATRICIA*-деревья (*Practical Algorithm To Retrieve Information Coded In Alphanumeric*), используемые для поиска данных на основе их поразрядного двоичного представления.

Несколько несвязанных между собой деревьев образуют структуру данных, которая называется «лес» (иногда её называют «бор»).

10.2 Двоичные деревья поиска

Ещё раз дадим определение дерева как *структуры данных*.

Дерево – динамическая нелинейная структура данных, каждый элемент которой содержит собственно информацию (или ссылку на то место в памяти ЭВМ, где хранится информация) и ссылки на несколько (не менее двух) других таких же элементов. Для двоичного (бинарного) дерева таких ссылок две: на правый соседний и левый соседний элементы.

В общем случае данные, хранящиеся в дереве, не обязаны быть упорядочены каким-либо образом. Но часто требуется соблюдение упорядоченности по некоторому принципу. Именно по такому принципу и отличаются «*пирамиды*» и «*деревья*»:

– *пирамида* («*куча*») – древовидная структура данных, в которой значения всех узлов, размещённых на одном уровне, больше (или меньше) значений узлов, размещённых на выше лежащем уровне (см. рисунок 10.1);

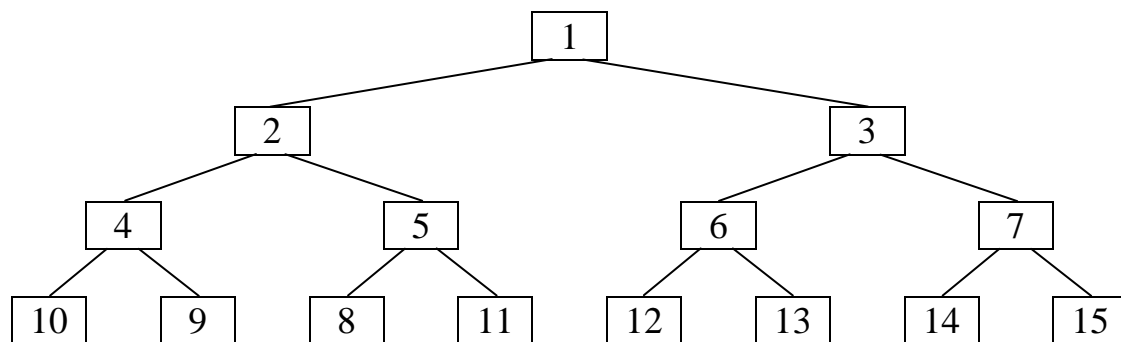


Рис. 10.1 – Двоичная пирамида

– *дерево* – древовидная структура данных, в которой значения всех узлов, размещённых правее некоторого узла, больше значений узлов, размещённых левее, причём это справедливо как для всего дерева, так и для любой его части (см. рисунок). Здесь необходимо сделать одно существенное уточнение: по указанному принципу строится дерево, получившее название «*двоичного дерева поиска*» (*Binary Search Tree, BST*), т.е. дерево, в котором очень удобно искать данные, причём с высокой эффективностью этого поиска. В дальнейшем под двоичными деревьями будут подразумеваться именно двоичные деревья поиска.

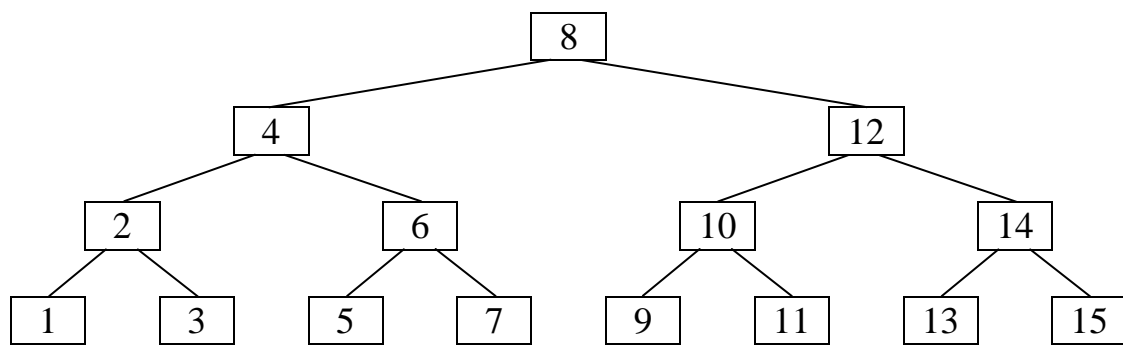


Рис. 10.2 – Двоичное дерево поиска

Как видно из сравнения рисунков 10.1 и 10.2, структура у пирамиды и дерева практически одинакова, отличаются они только характером размещения данных.

Также на этих рисунках видны и связи между элементами деревьев (*узлами*), похожие на связи между звеньями списка. Чем же отличается двоичное дерево от двусвязного списка? Иногда ничем, не только от двусвязного, но и от односвязного списка. Основное отличие – у списка соседними являются предшествующий и последующий элементы, и структура линейна; а у двоичного дерева поиска соседними являются элементы с меньшим и большим ключом, и структура, в общем случае ветвящаяся – в виде дерева, откуда она и получила свое название. Кроме того, любая часть дерева по своей структуре повторяет всё дерево в целом, т.е. дерево (любое, не только двоичное) – это *рекурсивная* структура данных (фактически дерево можно считать фрактальной структурой).

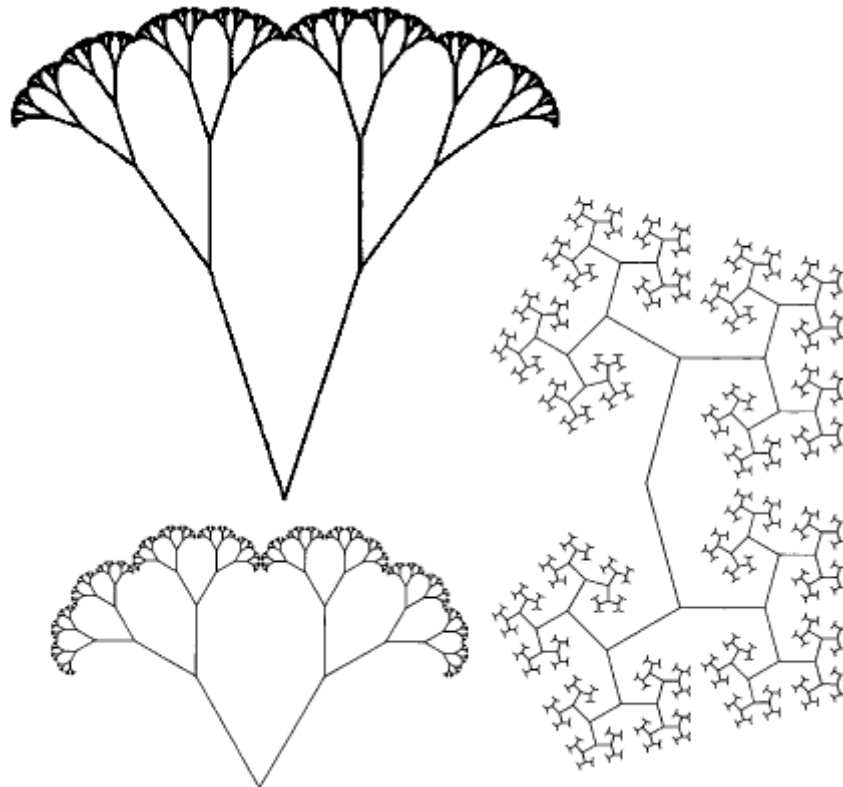


Рис. 10.3 – Фрактальные деревья

Нужно помнить, что дерево – это только метод логической организации информации в памяти, а память – линейна.

В случае, если информация хранится в произвольном (и неизвестном программисту) месте памяти, а ключ и информация – разные понятия (иногда они могут и совпадать), структура узла дерева может быть представлена следующим образом:

key	
^data	
left	right

Рис. 10.4 – Обобщённая структура узла дерева

Этой структуре соответствует набор типов на языке Паскаль.

```

type
  info = (* тип хранимых данных *);
  ptr = ^node;

```

```

node = record
  key: Integer;
  left, right: ptr;
  data: ^info
end;

```

```

var
  tree: ptr;

```

И на языке Си++.

```

struct node
{
  int key;           // Ключ, по которому строится дерево
  float data;       // Полезные данные (любого типа)
  node *left, *right; // Указатели на соседние узлы
};

```

Двоичное дерево, образованное такими элементами, показано ниже.

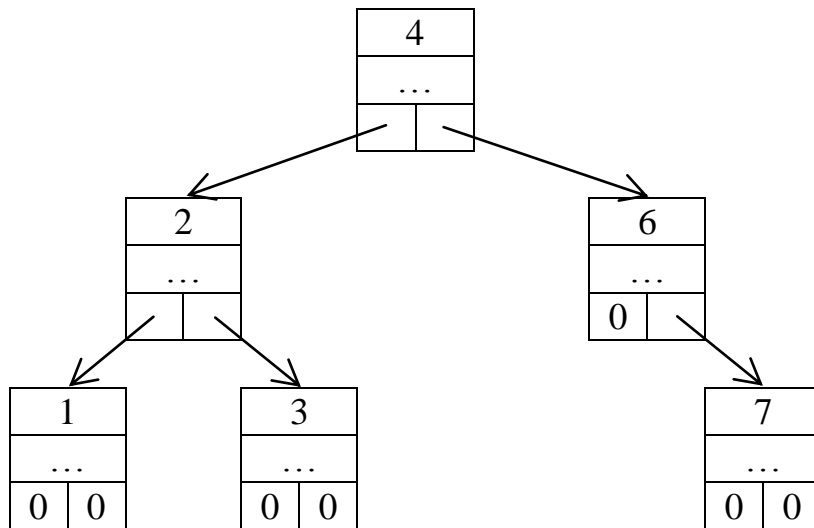


Рис. 10.5 – Двоичное дерево

Элемент дерева получил название «узел» (*node*). Единственный начальный элемент (откуда дерево развивается) – *корень* (*root*). Фрагмент (часть) дерева – поддереву или *ветвь*. Узел, от которого не отходят ветви – конечный или *терминальный* узел,

иногда его называют *листом*. Как видно на рисунках, узлы располагаются по уровням. Количество уровней – *высота* дерева (иногда вместо высоты используют термин «*глубина*»). Поскольку дерево – рекурсивная структура данных, то любой его узел может считаться корнем какой-либо ветви, в том числе пустой.

Области применения деревьев: для хранения информации, как таковой; в процедурах поиска и сортировки; для описания хранения файлов на дисковых носителях; при построении эффективных кодов для сжатия информации (коды Шеннона–Фано и Хаффмена).

Преимущество двоичных деревьев поиска: в случае, если дерево отсортировано (а это дерево именно такое), то операция поиска выполняется быстро, с эффективностью, близкой к эффективности двоичного поиска.

10.3 Операции с деревьями

Для деревьев имеются те же основные операции, что и для списков – добавление элемента, удаление, прохождение (просмотр) и поиск элементов. Кроме них можно использовать операции принудительной балансировки, подсчёта числа узлов в дереве, измерения высоты дерева, поиска ближайшего общего корня (*ближайшего общего предка – Nearest Common Ancestor (nca)*) для двух элементов (в многопутевых деревьях – для более чем двух элементов). При балансировке деревьев используются операции *поворотов (вращений)*, простых и двойных, левых и правых.

Поскольку дерево – рекурсивная структура данных, то наиболее эффективными при работе с деревьями оказываются рекурсивные процедуры. Хотя возможно применение и нерекурсивных (итеративных) процедур. Рассмотрим разные варианты построения этих процедур.

10.3.1. Добавление узла в дерево

Рассмотрим принцип построения двоичного дерева поиска при занесении в него информации. Пусть последовательно вводятся записи с ключами 70, 60, 85, 87, 90, 45, 30, 88, 35, 20, 86, 82.

1. 70 – корень (первый узел);
2. $60 < 70$, следовательно новый узел будет расположен слева от 70;
3. $85 > 70$, следовательно новый узел будет расположен справа от 70;
4. $87 > 70$, следовательно 87 будет находиться в правой ветви; $87 > 85$, следовательно 87 будет расположен справа от 85. И т. д. (рисунок 10.6).

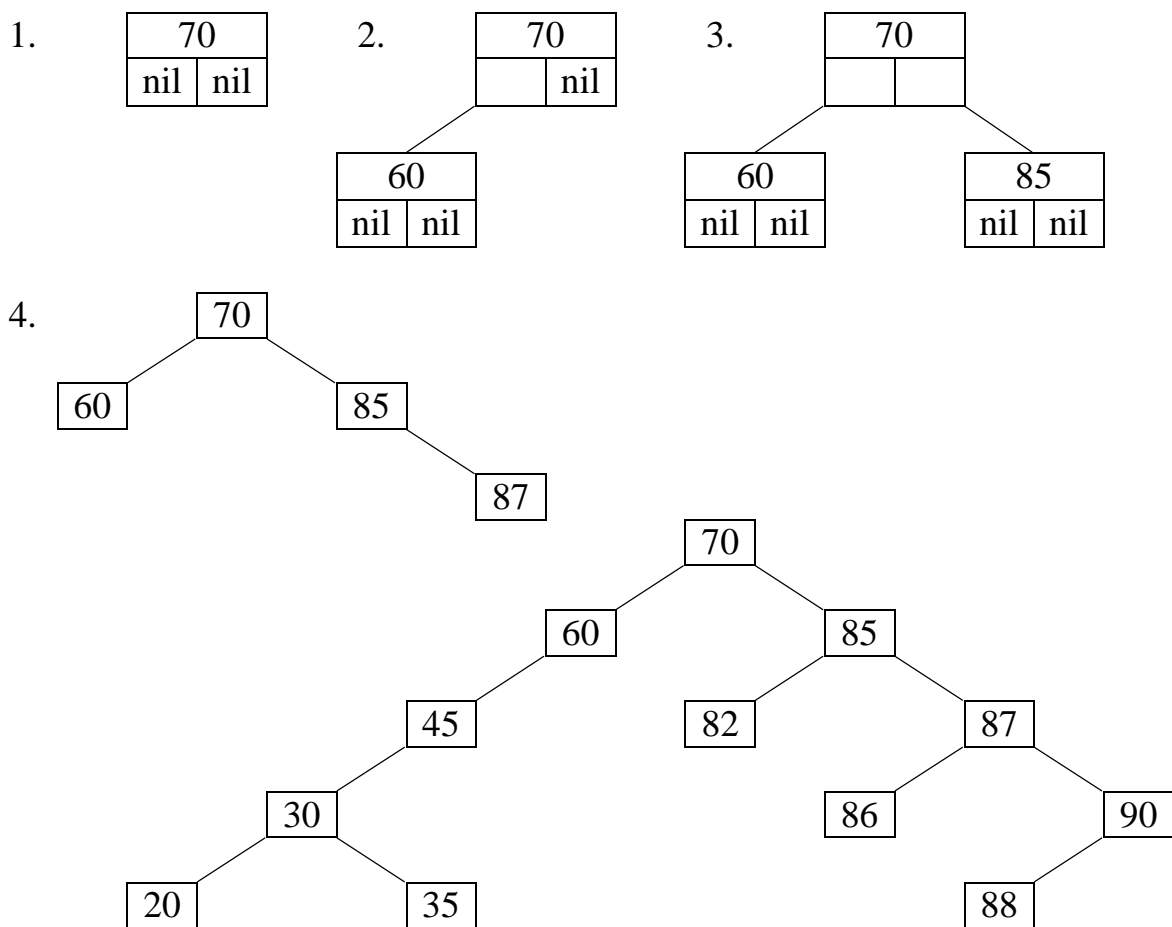


Рис. 10.6 – Процесс построения двоичного дерева поиска

При поступлении очередной записи с ключом k , этот ключ сравнивают, начиная с корня, с ключом очередного узла. В зависимости от результата сравнения, процесс продолжают в левой или правой ветви до тех пор, пока не будет достигнут один из узлов, с которым можно связать входящую запись. В зависимости от результата сравнения входящего ключа с ключом этого узла,

входящая запись будет размещена слева или справа от него и станет новым терминальным узлом.

Рекурсивная процедура добавления узла в дерево на языке Паскаль.

```
procedure addnode_r(r, prev: ptr; newkey: integer; newdata: info);
begin
  if r = nil then
  begin
    new(r);
    r^.left := nil;
    r^.right := nil;
    r^.key := newkey;
    r^.data := newdata;
    if tree = nil then
      tree := r;      (* Первый узел *)
    else
      begin
        if newkey < prev^.key then
          prev^.left := r
        else
          prev^.right := r
        end;
      exit
    end;
  if newkey < r^.key then
    addnode_r(r^.left, r, newkey)
  else
    addnode_r(r^.right, r, newkey)
  end;
```

Процедура добавляет информацию в двоичное дерево, отслеживая указатели на узлы и выбирая левые или правые ветви, в зависимости от ключа, пока не будет найдено соответствующее место в иерархии дерева.

Аргументы процедуры:

- указатель на корень дерева или ветви, в котором ищется место для нового узла;
- указатель на предыдущий узел;

- ключ;
- сохраняемые данные или указатель на них.

При первом вызове второй аргумент может быть равен `nil`.

Фактически этот процесс сортирует ключ, прежде чем добавить узел в дерево. Это вариант алгоритма сортировки методом простых вставок. При построении нового дерева или обслуживании уже упорядоченного дерева рекомендуется добавлять новые узлы именно такой процедурой.

Рассмотренная процедура использует ранее определённый указатель на дерево `tree`, которому необходимо присвоить значение `nil` перед первым вызовом этой процедуры (поскольку оно проверяется в строке `if tree = nil ...`). Приведём пример аналогичной функции на языке Си++, у которой такой же указатель является аргументом (это позволяет избежать использования глобальных структур данных).

```
void add_node(node*& tree, node*& r, node*&
prev, node& buf)
{
    if (r == NULL)
    {
        r = new node;
        r->left = NULL;
        r->right = NULL;
        r->key = buf.key;
        r->data = buf.data;
        if (tree != r)
            if (buf.key < prev->key)
                prev->left = r;
            else
                prev->right = r;
        return;
    }
    if (buf.key < r->key)
        add_node(tree, r->left, r, buf);
    else
        add_node(tree, r->right, r, buf);
}
```


Аргументы этой функции:

`tree` – указатель на всё дерево;

`r` – указатель на некоторый текущий узел;

`prev` – указатель на узел, «родительский» для `r`;

`buf` – буферная структура данных типа `node`, содержащая вводимые данные.

Перед первым вызовом указатель `tree` должен быть обнулён, а сам вызов может быть следующим:

```
tree = NULL;
node buf;
// ...
buf.key = ... // Ввод данных в буферную область памяти
buf.data = ...
// ...
add_node(tree, tree, tree, buf);
```

10.3.2. Прохождение дерева

Прохождение дерева, т.е. последовательное обращение ко всем его узлам может выполняться с разными целями: просмотр (т.е. вывод на дисплей), сохранение в файл, поиск. Наиболее просто реализуется прохождение с целью просмотра.

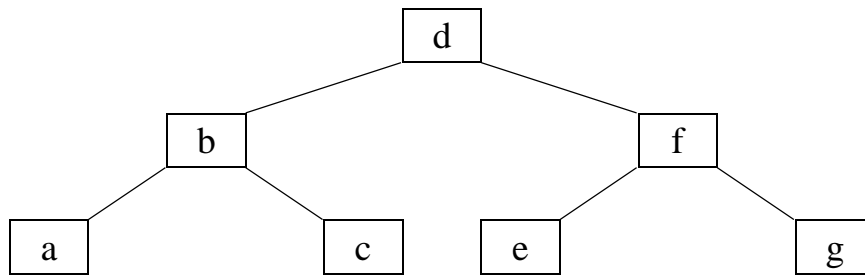
Порядок следования узлов дерева при его прохождении зависит от того, каким образом нужно организовать доступ к узлам. Процесс получения доступа ко всем узлам – прохождение дерева (или его обход). Прохождение дерева может выполняться по методам «*в глубину*» и «*в ширину*».

Существует три способа прохождения дерева *в глубину*:

1. Последовательный (он же инфиксный, симметричный, поперечный) – дерево проходится, начиная с левой ветви вверх к корню, затем к правой ветви;

2. Нисходящий (префиксный или прямой) – от корня к левой ветви, затем к правой;

3. Восходящий (постфиксный или обратный) – проходится левая ветвь, затем правая, затем корень.



Последовательный: a b c d e f g
 Нисходящий: d b a c f e g
 Восходящий: a c b e g f d

Рис. 10.7 – Обход дерева

Под корнем здесь понимается как корень всего дерева, так и корень текущей ветви (поддерева), т.е. текущий узел.

Последовательный способ удобен для сортировки данных. Хотя двоичное дерево не обязательно должно быть отсортировано (т.е. быть *двоичным деревом поиска*), во многих практических случаях это требуется. Порядок сортировки дерева определяется методом, которым дерево будет проходиться. Фактически сортировка как таковая для двоичного дерева поиска не требуется, поскольку при последовательном просмотре данные уже оказываются отсортированными по возрастанию.

Нисходящий способ удобен для сохранения дерева (т.е. данных, в нём хранящихся) в файл так, чтобы при последующем считывании была бы полностью восстановлена структура дерева.

Восходящий обход может использоваться при полном удалении всего дерева (или выбранной ветви).

Рекурсивные процедуры прохождения дерева в последовательном (inorder), нисходящем (preorder) и восходящем (postorder) порядках на языке Паскаль.

```

procedure inorder(r: ptr);
begin
  if r = nil then exit;
  inorder(r^.left);
  writeln( ... ); { Вывод информации }
  inorder(r^.right)
end;
  
```

```

procedure preorder(r: ptr);
begin
    if r = nil then exit;
    writeln( ... ); { Вывод информации }
    preorder(r^.left);
    preorder(r^.right)
end;

procedure postorder(r: ptr);
begin
    if r = nil then exit;
    postorder(r^.left);
    postorder(r^.right)
    writeln( ... ); { Вывод информации }
end;

```

Аргумент этих процедур – указатель на корень ветви, которую требуется найти. Если нужно найти все дерево, используется указатель на корень всего дерева. Выход из рекурсивной процедуры происходит, когда встречается терминальный узел.

Аналогичные процедуры на языке Си++.

```

void inorder(node* r)
{
    if (r == NULL) return;
    inorder(r->left);
    cout<<r->key<<" ";
    inorder(r->right);
}

void preorder(node* r)
{
    if (r == NULL) return;
    cout<<r->key<<" ";
    preorder(r->left);
    preorder(r->right);
}

```

```

void postorder(node* r)
{
    if (!r) return;
    postorder(r->left);
    postorder(r->right);
    cout<<r->key<<" ";
}

```

Обход дерева **в ширину** выполняется с помощью вспомогательной структуры данных – очереди или стека. Рассмотрим вариант с использованием очереди, в качестве которой возьмём двусвязный кольцевой список, что устраняет необходимость применения дополнительной переменной – указателя на хвост (или голову) очереди, т.к. в двусвязном списке с выделенным ведущим звеном голова и хвост находятся по обе стороны от ведущего звена. Применим список и операции с ним, рассмотренные в параграфах 9.2 и 9.5, дополнив их только функцией извлечения данных (узла дерева) из очереди.

```

struct Link2 // Список узлов
{
    node* n; // Узел дерева
    Link2* next, *prev;
};

void Insert2(Link2* Pred, node* data)
{
    ...
    Loc->n = data; // Ввод узла дерева в список
    ... // Обычные операции вставки в список
}

node* Retrieve2(Link2* Del) // Извлечение
{ // узла из очереди
    node* a=Del->n;
    Delete2(Del);
    return a;
}

```

```

void WidthTraverse(node *r)
{
    if(!r) return;
    Link2 *L2 = new Link2; // Локальная очередь
    L2->next = L2;
    L2->prev = L2;
    Insert2(L2->prev,r); // Занесение в очередь
    while(L2->prev!=L2) // Пока очередь не пуста
    {
        node* cur = Retrieve2(L2->next);
        printf("%d ", cur->key);
        if(cur->left)
            Insert2(L2->prev, cur->left);
        if(cur->right)
            Insert2(L2->prev, cur->right);
    }
    delete L2;
}

```

Результат обхода дерева, показанного на рисунке 10.7 с выводом содержимого узлов на экран, следующий:

d b f a c e g.

Фактически дерево проходит по уровням: верхний (корень), второй слева направо и т.д.

Следует отметить, что дерево может быть организовано не только как *динамическая связная структура данных*, но и как *массив*. В зависимости от способа обхода дерева данные в элементах такого массива могут размещаться по-разному, например, при нисходящем обходе начальный элемент массива оказывается корнем дерева, а при последовательном обходе – самым левым узлом (см рисунок 10.7). При использовании массива для построения дерева могут использоваться совсем другие операции для доступа к узлам, а также существенно усложняется процесс добавления в дерево узлов, количество которых превышает размер массива.

Одна из возможностей применения такого дерева – дерево с фиксированным или с заранее известным максимальным числом узлов, а преимущество – очень быстрый доступ к любому узлу.

Строго говоря, содержимое любого массива при желании может интерпретироваться как дерево с соответствующим корнем. В частности, именно массив является основной структурой данных в алгоритме быстрого доступа к данным с возможностью их модификации, подсчёта суммы, нахождения максимума или минимума. Такой алгоритм получил название *дерева отрезков*. Ещё один подобный алгоритм называется *деревом Фенвика*.

10.3.3. Поиск узла в дереве

Поиск узла в дереве (в том числе в *двоичном дереве поиска*) может выполняться по такому же принципу, что и просмотр дерева, но в этом случае теряется эффективность этой структуры данных, а поиск фактически приобретает линейный характер. Для сохранения эффективности поиска следует или использовать рекурсивную функцию, аналогичную показанным выше `inorder`, `preorder` и т.п., преобразовав её так, чтобы её работа (и все рекурсивные вызовы) завершалась после нахождения требуемых данных, или отказаться от рекурсии.

Рекурсивная функция поиска.

```
void poisk1(node* r, int skey, node*& f)
{
    if (r == NULL || f) return;
    if (r->key == skey)
    { f = r; return; } // Выход, если нашли
    else
        if (skey < r->key)
            poisk1(r->left, skey, f);
        else
            poisk1(r->right, skey, f);
}
```

Третий аргумент `node*& f` – одновременно узел, в котором находятся найденные данные, и признак успеха поиска. Перед вызовом этой функции соответствующий фактический аргумент должен быть равен нулю.

Рекурсивные процедуры и функции, при всех их достоинствах, обладают одним серьёзным недостатком – возможностью переполнения стека возврата из подпрограмм при большом количестве рекурсивных вызовов (а это может произойти при большом размере структуры данных), поэтому рассмотрим другой вариант реализации поиска – использование нерекурсивной процедуры.

Нерекурсивная функция поиска в двоичном дереве на языке Паскаль.

```
function search_tree(skey:integer; var tree,
fnode: ptr):boolean;
var
  p, q: ptr;
  b: boolean;
begin
  b := false;
  p := tree;
  if tree <> nil then
    repeat
      q := p;
      if p^.key = skey then
        b := true
      else
        begin
          if skey < p^.key then
            p := p^.left
          else
            p := p^.right
          end
        until (b) or (p = nil);
      search_tree := b;
      fnode := q;
    end;
```

Аналогичная функция на языке Си++.

```
void poisk2(node* r, int skey, node*& f)
{
  while (r != NULL && f == NULL)
```

```

{
  if (r->key == skey)
  { f = r; return; } // Выход, если нашли
  else
    if (skey < r->key)
      r = r->left;
    else
      r = r->right;
}
}

```

Эта функция практически идентична по общей структуре предыдущей рекурсивной функции. В этом примере виден и способ преобразования рекурсивных функций в нерекурсивные: рекурсивный вызов, находящийся в конце процедуры («*хвостовая*» рекурсия) заменяется на цикл (в этом примере – цикл с предусловием), а рекурсивный вызов, размещённый где-либо до конца процедуры, заменяется на оператор `if`.

Длительность поиска зависит от структуры дерева. Для сбалансированного дерева (изображено на рисунке 10.8) поиск аналогичен двоичному – то есть нужно просмотреть не более $\log_2 N$ узлов.

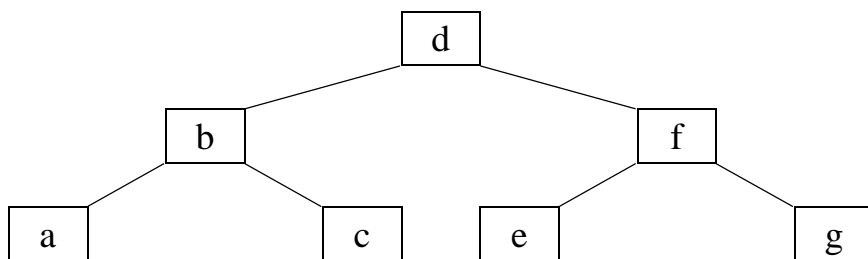


Рис. 10.8 – Сбалансированное дерево

Для вырожденного дерева (изображено на рисунке 10.9) поиск аналогичен поиску в односвязном списке – в среднем нужно просмотреть половину узлов, а в худшем случае – все N узлов.

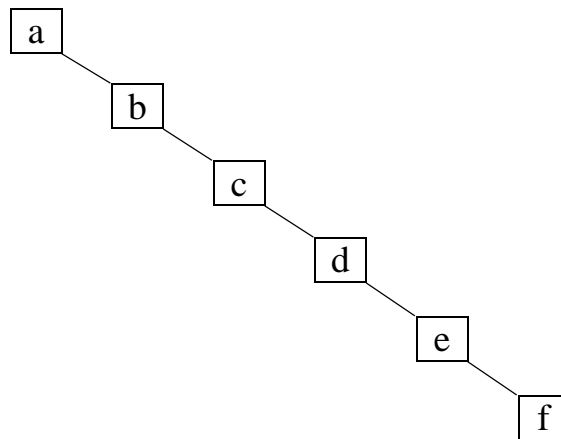


Рис. 10.9 – Вырожденное дерево

Используя поиск узла, можно заменить рекурсивную процедуру добавления узла на нерекурсивную.

Нерекурсивная процедура добавления нового узла в дерево на языке Паскаль.

```

procedure addnode2(skey:integer; var tree:ptr;
newdata:^info);
var
  r,s:ptr;
  t:^info;
begin
  if not search_tree(skey,tree,r) then
  begin
    new(t);           { занесение }
    t:=newdata;      { данных }
    new(s);           { новый }
    s^.key:=skey;
    s^.left:=nil;    { узел }
    s^.right:=nil;
    s^.data:=t;
    if tree=nil then tree:=s
  else
    if skey<r^.key then
      r^.left:=s
    else
      r^.right:=s
  end
end;

```

10.3.4. Удаление узла из дерева

Удаление узла из дерева – существенно более сложный процесс, чем поиск, так как удаляемый узел может быть корневым, левым или правым. Узел может давать начало ветвям (в двоичном дереве их может быть от 0 до двух).

Наиболее простым случаем является удаление терминального узла или узла, из которого выходит только одна ветвь. Для этого достаточно скорректировать соответствующий указатель у предшествующего узла.

Наиболее сложный случай – удаление корневого узла поддерева (или всего дерева) с двумя ветвями, поскольку приходится корректировать несколько указателей. Нужно найти подходящий узел, который можно было бы вставить на место удаляемого, причем этот подходящий узел должен просто перемещаться. Такой узел всегда существует. Это либо самый левый узел правой ветви, либо самый правый узел левой ветви.

Если это самый правый узел левой ветви, то для достижения этого узла нужно перейти в следующий от удаляемого узла по левой ветви, а затем переходить в очередные узлы только по правой ветви до тех пор, пока очередной правый указатель не будет равен `nil`. Такой узел может иметь не более одной ветви. Ситуации, возникающие при удалении узла из дерева, показаны на рисунке 10.10.

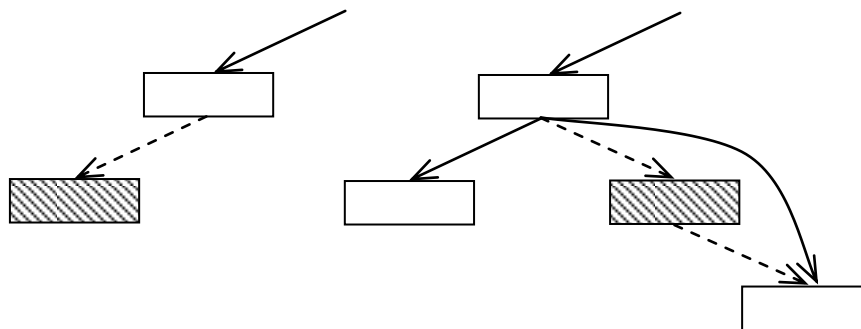


Рис. 10.10 – Удаление узла из дерева

Пример удаления из дерева узла с ключом 50.

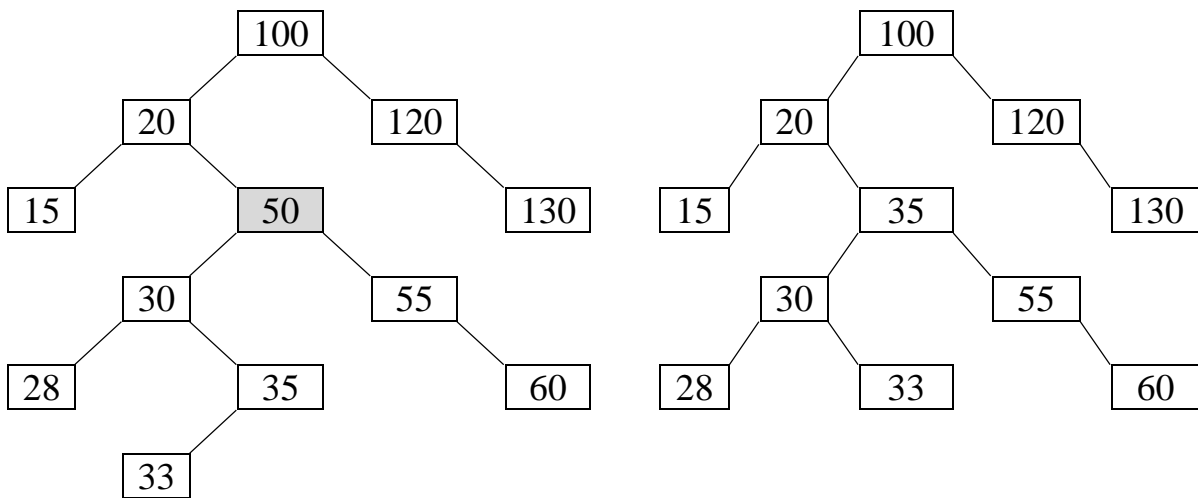


Рис. 10.11 – Пример удаления узла из дерева

Процедура удаления узла должна различать три случая:

- 1). узла с данным ключом в дереве нет;
- 2). узел с заданным ключом имеет не более одной ветви (рис. 10.10);
- 3). узел с заданным ключом имеет две ветви (рис. 10.11).

Процедура удаления узла двоичного дерева на языке Паскаль (автор процедуры – Никлаус Вирт).

```

procedure delnode(var d: ptr; key: integer);
var q: ptr;

procedure dell(var r:ptr); (* локальная вспомога-
тельная процедура *)
begin
  if r^.right = nil then
  begin
    q^.key := r^.key;
    q^.data := r^.data;
    q := r;
    r := r^.left
  end
  else
    dell(r^.right)
  end; (* конец локальной процедуры *)

```

```

begin  (* начало основной процедуры *)
  if d = nil then exit      (* Первый случай *)
  else
    if key < d^.key then
      delnode(d^.left, key)
    else
      if key > d^.key then
        delnode(d^.right, key)
      else
        begin
          q := d;          (* Второй случай *)
          if q^.right = nil then
            d := q^.left
          else
            if q^.left = nil then
              d := q^.right
            else          (* Третий случай *)
              dell(q^.left);
          dispose(q)      (* Собственно удаление *)
        end
      end
end;

```

Вспомогательная рекурсивная процедура `dell` вызывается только в третьем случае. Она проходит дерево до самого правого узла левого поддерева, начиная с удаляемого узла q^{\wedge} , и заменяет значения полей `key` и `data` в q^{\wedge} соответствующими записями полей узла r^{\wedge} . После этого узел, на который указывает `r` можно исключить (`r := r^.left`).

Аналогичные функции на языке Си++.

```

void dell(node*& r, node*& q)
{
  if (r->right == NULL)
  {
    q->key = r->key; // Перенос данных
    q->data = r->data;
    q = r;
    r = r->left;
  }
  else

```

```

        dell(r->right, q);
    }

void del_node(node*& d, int key)
{
    node* q;
    if (d == NULL)
        return;
    else
        if (key < d->key)
            del_node(d->left, key);
        else
            if (key > d->key)
                del_node(d->right, key);
            else
                {
                    q = d;
                    if (q->right == NULL)
                        d = q->left;
                    else
                        if (q->left == NULL)
                            d = q->right;
                        else
                            dell(q->left, q);
                    delete q;    // Удаление
                }
}

```

10.3.5. Удаление всех узлов дерева

Эта операция выполняется по тому же принципу, что и просмотр дерева. В этом случае удобно использовать восходящий обход дерева.

```

void del_all(node*& r)
{
    if (!r) return;
    del_all(r->left);
    del_all(r->right);
    delete r;
}

```

```

    r = NULL;
}

```

10.3.6. Подсчёт узлов

Это же справедливо и для подсчёта числа узлов в дереве (но используется нисходящий обход дерева).

```

void Nnodes(node* r, int& p)
{
    if (r == NULL) return;
    p++;
    Nnodes(r->left, p);
    Nnodes(r->right, p);
}

```

Число узлов сохраняется в аргументе `p`. Соответствующий ему фактический аргумент должен быть обнулён перед вызовом функции для получения правильного результата.

10.3.7. Определение высоты дерева

То же самое справедливо и при подсчёте числа уровней. После завершения работы функции высота дерева сохраняется в формальном аргументе `h` и в соответствующем фактическом аргументе.

```

void Height(node* r, int p, int& h)
{
    if (r == NULL) return;
    p++;
    if (r->left == NULL && r->right == NULL) //
Проверка на достижение терминального узла
        if (p > h)
            h = p;
    Height(r->left, p, h);
    Height(r->right, p, h);
}

```

Во многих алгоритмах обработки данных, хранящихся в деревьях, предполагается, что ключи узлов должны быть уникальными, т.е. неповторяющимися. На самом деле процедуры обработки дерева допускают существование повторяющихся ключей. Следует только помнить, что в этом случае при поиске по ключу может быть найден не тот узел, который требовался, а первый встреченный узел с указанным ключом. То же самое справедливо для удаления узлов.

10.4. Сбалансированные деревья

Максимальный эффект использования двоичного дерева поиска достигается, если оно сбалансировано – когда все узлы, кроме терминальных, имеют непустые и правый и левый соседние узлы; все поддеревья, начинающиеся с одного и того же уровня, имеют одинаковую высоту.

Сбалансированное бинарное дерево – максимально широкое и низкое. Именно такими являются деревья, показанные на рисунках 10.2, 10.5, 10.7 и 10.8. Фактически все эти деревья, кроме 10.5, являются *идеально сбалансированными*.

Менее строгое, но практически более удобное определение сбалансированности дерева – дерево является сбалансированным, если для каждого узла исходящие ветви *отличаются по высоте не более, чем на один уровень*.

Обратный случай – вырожденное дерево, – выродившееся в линейный односвязный список. Такое дерево получается, если заносимые в него данные упорядочены по возрастанию (рисунок 10.9) или по убыванию.

Вырожденные деревья также получили название *лево-* или *право-ассоциативных* или «*лоз*», а также являются частным случаем *ориентированных* деревьев.

Если данные случайны, то получается дерево, в той или иной степени приближающееся к сбалансированному (рисунки 10.6, 10.11).

Для двоичного идеально сбалансированного дерева с максимально возможным (для идеальной сбалансированности) числом узлов существуют простые соотношения между этим числом узлов N и высотой дерева (т.е. числом уровней) h :

$$N = 2^h - 1 \quad (10.1)$$

$$h = \log_2(N + 1) \quad (10.2)$$

Состояние сбалансированности (хотя бы в менее строгом смысле) часто оказывается настолько важным для тех областей, в которых деревья применяются, что для достижения этого состояния принимают специальные меры. Такими мерами являются либо та или иная операция **балансировки** (принудительной) дерева, в том числе включающая в себя упомянутые операции **поворотов**, либо использование специальных видов деревьев, обеспечивающих балансировку при каждой операции добавления или удаления узла. Основными видами таких деревьев являются **АВЛ-дерево** и **красно-чёрное** дерево.

10.4.1. АВЛ-дерево

АВЛ-дерево получило своё название по фамилиям его разработчиков – советских математиков Георгия Максимовича Адельсон-Вельского (родился 8 января 1922 г. в Самаре) и Евгения Михайловича Ландиса, которые предложили использовать такое дерево в 1962 году.

АВЛ-дерево полностью удовлетворяет менее строгому определению сбалансированности дерева. Сбалансированность достигается за счёт упомянутых выше операций поворотов (или вращений), а для сравнения высот ветвей в каждом узле двоичного дерева поиска используется **дополнительное поле** – признак сбалансированности ветвей (или разность их высот).

10.4.2. Красно-чёрное дерево

Красно-чёрное дерево (RB-tree) отличается от АВЛ-дерева смыслом признака сбалансированности: вместо разности высот ветвей используется абстрактный «цвет» (красный или чёрный) и дерево строится по следующим правилам:

1. Все **указатели** на терминальные узлы считаются **непустыми** (т.е. в дереве имеются **фиктивные терминальные узлы**).
2. Все такие **терминальные узлы** считаются «чёрными».
3. Все узлы, соседние с «красными» узлами, считаются

«чёрными» (т.е. запрещена ситуация с двумя «красными» узлами подряд).

4. В левой и правой ветвях дерева, ведущих от его корня к листьям, число «чёрных» узлов одинаково. Это число называется «чёрной высотой» дерева.

Красно-чёрное дерево, соответствующее перечисленным правилам, показано на рисунке 10.12.

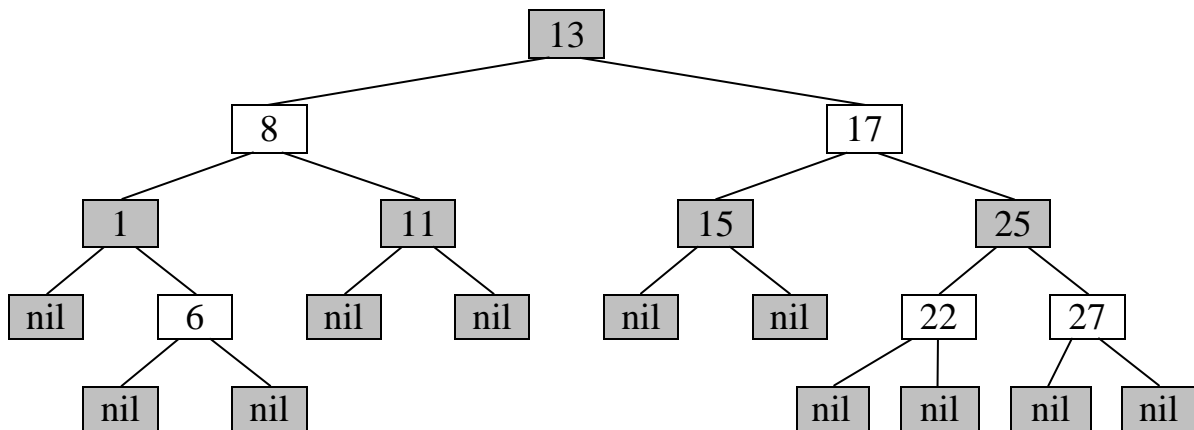


Рис. 10.12 – Пример красно-чёрного дерева с фиктивными чёрными терминальными узлами

Теоретически считается, что красно-чёрное дерево требует меньшего объёма памяти для хранения отдельного узла, чем AVL-дерево, т.к. для представления цвета достаточно всего одного бита. Но на практике это преимущество реализовать без потерь на дополнительные операции доступа к отдельным битам весьма сложно. Даже один из вариантов реализации красно-чёрного дерева – когда «красные» узлы обозначаются нечётными ключами, а «чёрные» узлы – чётными (или наоборот), не всегда пригоден на практике, т.к. решаемая задача может не допускать такого разделения узлов.

Используемые при балансировке операции *вращения* фактически представляют собой переприсвоения значений указателей в узлах дерева и, как следствие, перепроведение связей между узлами, после которого высоты левой и правой ветвей оказываются одинаковыми (или отличаются не более чем на один уровень) и дерево считается сбалансированным. Суть операции вращения иллюстрируется следующим рисунком:

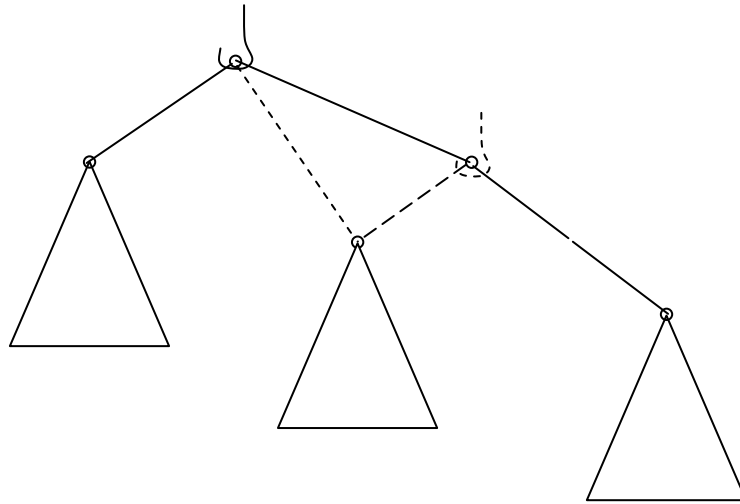


Рис. 10.13 – Операция вращения (поворота)

Здесь сплошными линиями показано исходное несбалансированное дерево (ветви показаны большими треугольниками, внутри этих ветвей узлы могут быть уже сбалансированы). Можно считать, что дерево «подвешено» за свой текущий корень на крюк, показанный сплошной линией. Поворот состоит в «подвеске» дерева за другой узел (который станет новым корнем) на крюк, показанный пунктирной линией, и коррекции связей: к левой ветви нового корня проводится новая связь (показана пунктирной линией) от старого корня (эта ветвь будет новой правой веткой старого корня), связь от нового корня к его старой левой ветви рвётся (штриховая линия), и связь между старым и новым корнями меняет направление (фактически существующая связь в одном направлении уже порвана, а в обратном направлении – проводится заново). В итоге дерево «поворачивается» вокруг новой точки подвески, откуда эта операция и получила своё название. После вращения в рассматриваемом случае высоты левой и правой ветвей нового корня будут одинаковыми.

Для выполнения операции вращения может потребоваться информация о текущем узле и связанных с ним узлах (его «родителе» и его «сыне»), т.е. адреса этих узлов в памяти. Или информация о текущем узле, его «отце» и «деде», или информация о текущем узле, его «сыне» и «внуке» (в зависимости от того, рассматривается дерево как ненаправленная структура данных, или как направленная).

Операция вращения может применяться не только для балансировки AVL- или красно-чёрных деревьев, но и обычных двоичных деревьев, не обязательно деревьев поиска. Например, один из вариантов принудительной балансировки обычного дерева заключается в преобразовании исходного дерева в *лево-ассоциативное* (т.е. в левую «лозу») с помощью последовательных операций вращения (см. рисунки).

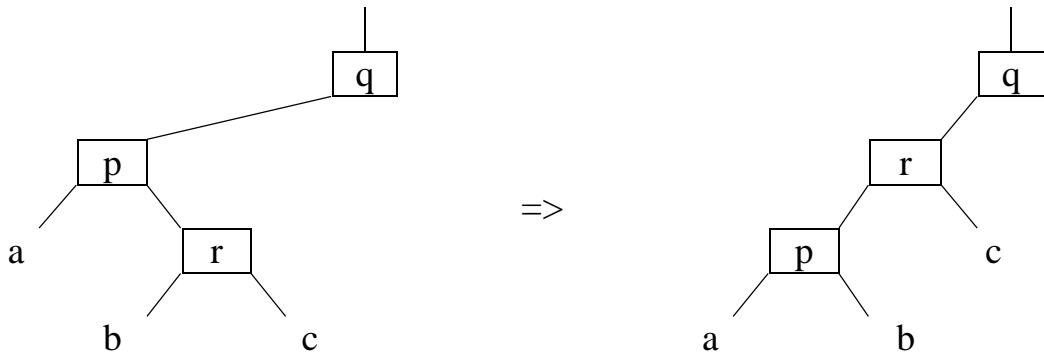


Рис. 10.14 – 1-й этап. Преобразование дерева в «лозу»

Затем «лоза» преобразуется в сбалансированное дерево при помощи той же операции.

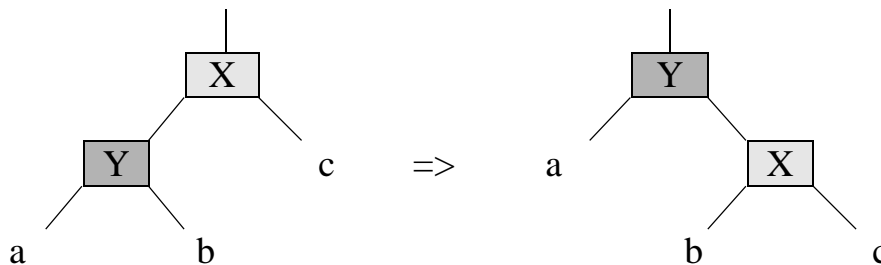


Рис. 10.15 – Однократный правый поворот

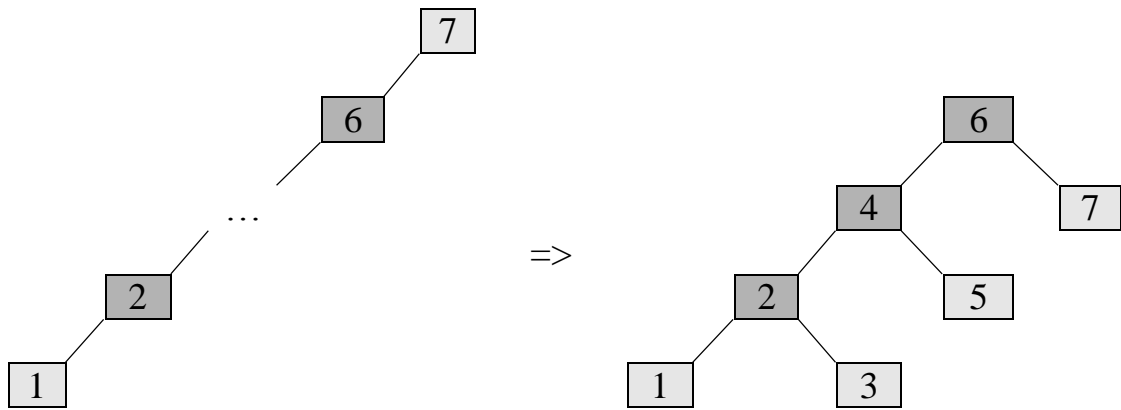


Рис. 10.16 – 2-й этап. Начало преобразования «лозы» в дерево

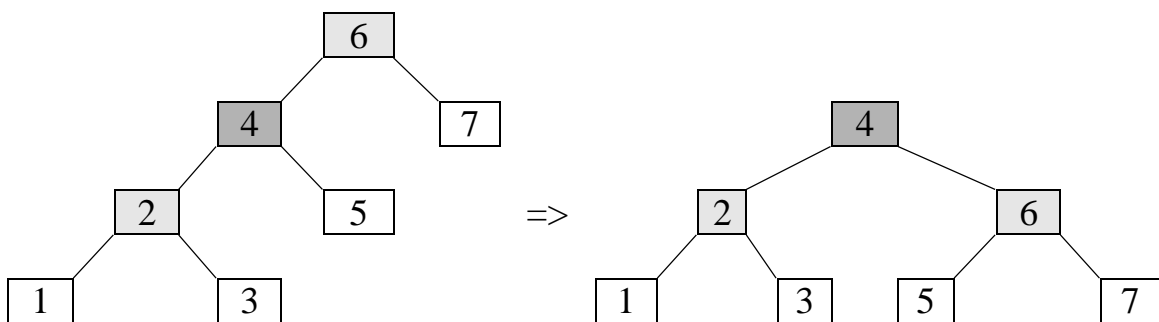


Рис. 10.17 – Завершение преобразования «лозы» в дерево.

Рекурсивная функция построения идеально сбалансированного дерева с заданным числом узлов на языке Паскаль.

```
function maketree(n_node:integer):ptr;
var
  newnode: ptr;
  newkey, nl, nr: integer;
begin
  if n_node = 0 then
    newnode := nil
  else
    begin
      nl := n_node div 2;
      nr := n_node - nl - 1; { Ввод ключа и данных }
      read(newkey);
      new(newnode);
      with newnode^ do
        begin
```

```

    key := newkey;
    left := maketree(nl);
    right := maketree(nr)
  end;
end;
maketree := newnode
end;
...
read(n);          (* Ввод числа узлов *)
tree := maketree(n); (* Первый вызов *)

```

10.4.3. Б-деревья

Б-деревья (Байера-МакКрейта) являются **сбалансированными** деревьями, у которых число ветвей, исходящих из узлов, может быть **два и более**. Узел, корневой для двух ветвей, содержит единственный ключ, а узел, корневой для нескольких ветвей, содержит составной ключ – несколько ключей, число которых на 1 меньше числа ветвей. Упрощенная архитектура Б-дерева показана на рисунке 10.18:

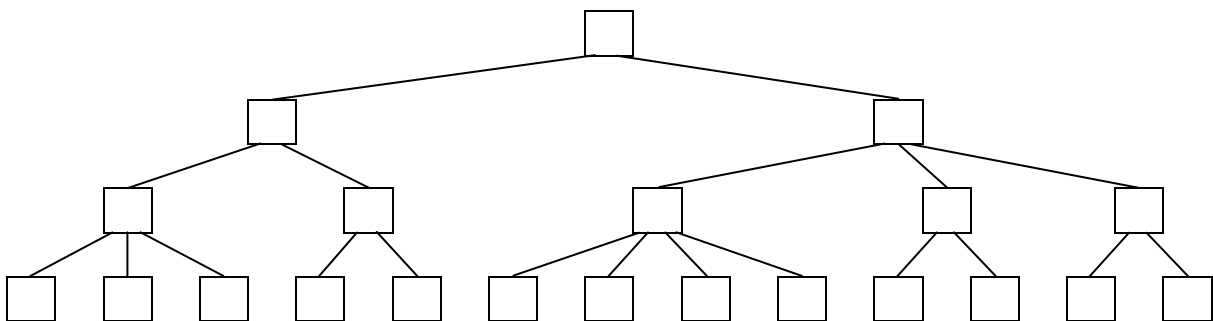


Рис. 10.18 – Б-дерево

В зависимости от области применения таких деревьев, полезные данные могут храниться только в их терминальных узлах (в этом случае узлы верхних уровней обеспечивают быстрый доступ по ключам к терминальным узлам), или во всех узлах, как в обычных деревьях.

Существуют способы преобразования двоичного дерева в Б-дерево и наоборот.

Модификацией Б-дерева является **Б+дерево** – Б-дерево, у которого терминальные узлы соединены в связный список (см. рисунок 10.19). Именно по такому принципу хранятся данные в базах данных по технологии Microsoft SQL Server.

Возможны и другие модификации Б-дерева, например Б++ дерево – Б+ дерево, у которого связный список формируется не только на самом нижнем уровне, но и на уровень выше.

В тех случаях, когда максимальное число ветвей, исходящих из узла, ограничено, получаются, например, **2-3-дерева** и **2-3-4-дерева**.

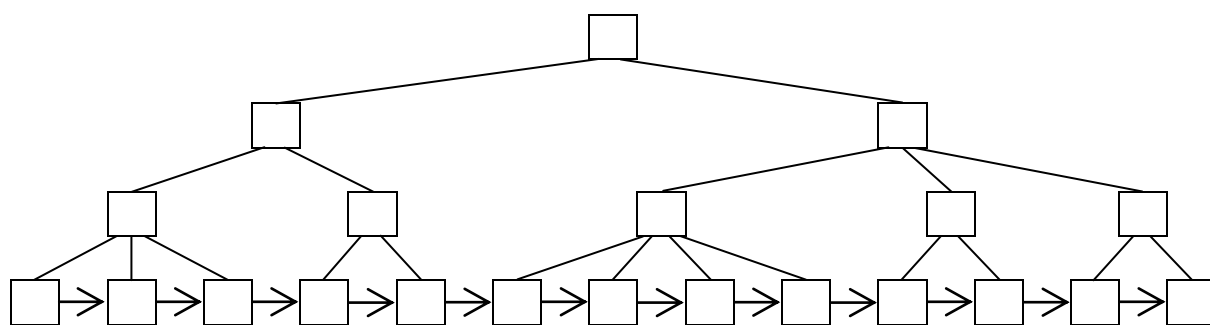


Рис. 10.19 – Б+ дерево

Кроме рассмотренных сбалансированных деревьев существует также **расширяемое дерево Тарьяна (splay-tree)**, являющееся двоичным деревом поиска и обеспечивающее балансировку без каких-либо дополнительных признаков. Балансировка выполняется в том числе с использованием операций поворотов, аналогичных показанной на рисунке 10.15.

При необходимости обычное двоичное дерево поиска можно принудительно балансировать, например, алгоритмом, аналогичным нисходящему обходу дерева. Но при этом не только не гарантируется эффективность алгоритма, но и существует возможность получить алгоритм с худшим временем работы, например, $O(N^2)$. Подобные алгоритмы относятся к категории «**наивных**», т.е. наиболее очевидных, но не всегда эффективных. То же самое справедливо для операции поиска **ближайшего общего предка**.

10.5. Многоключевые деревья

Б-деревья могут считаться частным случаем многоключевых деревьев (с переменным числом ключей). Ещё один такой частный случай – *Декартово дерево*. Это древовидная структура данных, в каждом узле которой содержится два ключа, по одному ключу она является двоичной пирамидой, а по второму ключу – двоичным деревом поиска.

Более простая ситуация – *двухключевое* дерево, которое является двоичным деревом поиска либо только по одному из ключей, либо по сумме ключей (именно в этом случае возможно повторение сумм ключей). Такое дерево может использоваться для описания рёбер графа. При этом также возможно повторение ключей в дереве, в зависимости от структуры графа.

Операции с такими двухключевыми деревьями ничем не отличаются от операций с обычными двоичными деревьями.

Вопросы и задания для самоконтроля

- 10.1. Что представляют собой древовидные структуры данных?
- 10.2. Какие существуют виды деревьев?
- 10.3. Что представляет собой двоичное дерево?
- 10.4. В чем заключается особенность дерева как структуры данных?
- 10.5. Опишите организацию связей в дереве.
- 10.6. Перечислите основные области применения деревьев.
- 10.7. Какие операции осуществляются над деревьями?
- 10.8. В чем преимущества использования деревьев?
- 10.9. Чем отличаются двоичные деревья от списков?
- 10.10. Изобразите структуру элемента двоичного дерева.
- 10.11. Процедуры какого характера наиболее эффективны при работе с деревьями?
- 10.12. Чем отличаются: сбалансированное, несбалансированное и вырожденное двоичные деревья. Проиллюстрируйте рисунками.
- 10.13. Что означают термины «сбалансированное дерево» и «идеально сбалансированное дерево»?

10.14. Чем вырожденное дерево отличается от односвязного списка?

10.15. Перечислите основные способы прохождения двоичного дерева.

10.16. Предложите процедуру обхода дерева в ширину с использованием стека вместо очереди.

10.17. Почему рекурсивные функции наиболее эффективны при работе с деревьями?

10.18. В чем заключается вставка узла в дерево?

10.19. Предложите нерекурсивную процедуру добавления узла в дерево без использования поиска.

10.20. В чем заключается удаление узла из дерева?

10.21. Проиллюстрируйте процесс удаления узла. Как действует алгоритм в каждом из трех возможных случаев?

10.22. Что такое высота дерева?

10.23. Как сохранить сбалансированность дерева при вставке и удалении узлов?

10.24. Чем отличается красно-чёрное дерево от AVL-дерева?

10.25. Каковы основные особенности красно-чёрного и AVL-дерева?

10.26. На каких структурах данных могут строиться деревья?

10.27. Используя приведённые в этом разделе функции, напишите программу построения и просмотра двоичного дерева поиска. Используя правильный порядок ввода данных, постройте с помощью этой программы 1-2-братское дерево.

10.28. В каком порядке должны вводиться данные, чтобы получилось сбалансированное дерево?

11. Элементы теории графов

11.1. Способы представления графов

Рассмотрим способы представления графов в памяти ЭВМ. Граф является топологической абстракцией, предназначенной для описания некоторых топологических свойств самых разных объектов и отношений между этими объектами. Существует большое число типов графов, отличающихся классификационными признаками. Такое многообразие обусловлено различными свойствами графов. Так, графы могут быть связными, слабосвязными, сильносвязными, полносвязными, несвязными, ориентированными, неориентированными, взвешенными, контурными, бесконтурными, содержащими петли или кратные рёбра (псевдографами), мультиграфами, гиперграфами и т.п. [2, 9].

Для графов имеется большое число алгоритмов обработки данных, например, поиск вершин в графе, поиск кратчайших путей, построение деревьев на графах, очередность обхода вершин и т.п.

Для работы с графом он должен быть каким-либо образом представлен в памяти ЭВМ. Способ представления зависит от характера решаемых на графе задач. Существует несколько способов отображения графа на память машины, каждый из которых обладает собственными достоинствами и недостатками [9]. Существуют также и алгоритмы преобразования этих способов описания графа друг в друга [10].

11.1.1. Список рёбер

Одним из наиболее простых способов является представление графа в виде списка рёбер, соединяющих вершины. Например, для графа, показанного на рисунке 11.1, этот список может выглядеть следующим образом:

(1, 2), (1, 3), (1, 4), (2, 4), (3, 4).

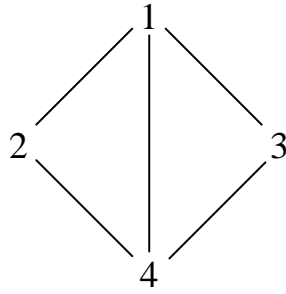
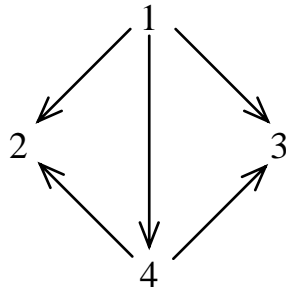


Рис.11.1 – Простой ненаправленный циклический граф

Достоинством такого способа является его простота, а также способность представлять графы практически любых видов. Например, в случае ориентированного графа направление ребёр задаётся порядком указания вершин. Такой граф показан на рисунке 11.2.



$(1, 2), (1, 3), (1, 4), (4, 2), (4, 3)$

Рис. 11.2 – Простой направленный циклический граф и его список ребёр

Этот же способ подходит и для описания взвешенных графов, графов с петлями, мультиграфов и т.п. Например, взвешенный направленный циклический граф, показанный на рисунке 11.3, описывается следующим списком ребёр:

$(1, 2, 0.20), (1, 3, 0.30), (1, 4, 0.10), (4, 2, 0.33), (4, 3, 0.15).$

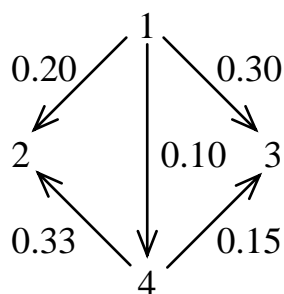


Рис. 11.3 – Взвешенный направленный циклический граф

Список рёбер для графа из M рёбер требует в расчёте на вершину объём памяти, пропорциональный $2M$. Для получения списка рёбер, непосредственно связанных с текущей вершиной требуется M шагов.

Под термином «список» во всех рассмотренных случаях следует понимать как массив однотипных элементов (в более простом случае), так и связный список в том числе, и даже *древовидную структуру данных*. Наряду с простотой, рассматриваемый способ описания графа обладает, по крайней мере, двумя особенностями, которые при определённых обстоятельствах можно посчитать недостатками. Эти особенности следующие:

- порядок перечисления рёбер в списке в общем случае может быть произвольным. Это следует из самой природы графа, когда точно расположение вершин не важно, важны лишь их взаимное расположение и связи между ними. В то же время возможны ситуации, когда отсутствует какой-либо логический порядок в перечислении рёбер, что может привести к ухудшению понимания структуры графа, а также к усложнению алгоритмов его обработки;
- в том случае, когда для хранения рёбер используется массив, список рёбер оказывается пригодным только для описания графа с неизменяющейся структурой или с одним и тем же числом рёбер. Для описания и обработки динамически изменяющегося графа потребуется применить связный список. В этом случае по мере добавления или удаления вершин и рёбер структура графа изменится, также как и порядок следования рёбер. Это также приве-

дѣт к усложнению алгоритмов обработки графа.

11.1.2. Список вершин

Рассмотрим другой способ описания графа. Представим граф, как список вершин, для каждой из которых указаны все другие вершины, с которыми текущая соединена рѣбрами. Для ненаправленного графа, показанного на рисунке 11.1, список вершин будет следующим:

(1, 2, 3, 4), (2, 1, 4), (3, 1, 4), (4, 1, 2, 3).

В каждом элементе этого списка первой указывается текущая вершина.

Направленный граф, показанный на рисунке 11.2, описывается следующим списком вершин:

(1, 2, 3, 4), (2), (3), (4, 2, 3).

В этом списке для каждой вершины графа указываются вершины, в которые входят исходящие из неё рѣбра. Если таких рѣбер для некоторой вершины нет, как, например, для вершин 2 и 3, то указываются только сами текущие вершины. Направление рѣбер устанавливается по взаимному указанию вершин в таком списке.

Список вершин легко адаптируется к случаю взвешенного графа (рисунок 11.3):

(1, [2, 0.20], [3, 0.30], [4, 0.10]), (2), (3), (4, [2, 0.33], [3, 0.15]).

Как видно из структуры этого списка, так же, как и списка рѣбер для направленного графа, их элементы содержат разнородную информацию, что ведѣт к усложнению структур данных отдельных элементов списка. Кроме того, количество исходящих рѣбер для каждой вершины может быть неодинаковым.

Такие списки вершин называют списками инцидентности. Теоретически, могут использоваться и простые списки вершин, в которых просто перечисляются по одной все имеющиеся вершины, но в изолированном виде такие списки пригодны для описания только полностью несвязных графов, у которых отсутствуют рѣбра. В противном случае списки вершин должны использоваться в паре со списками рѣбер.

Часто списки инцидентности строят на основе связанных списков. Такой способ построения показан на рисунке 11.4.

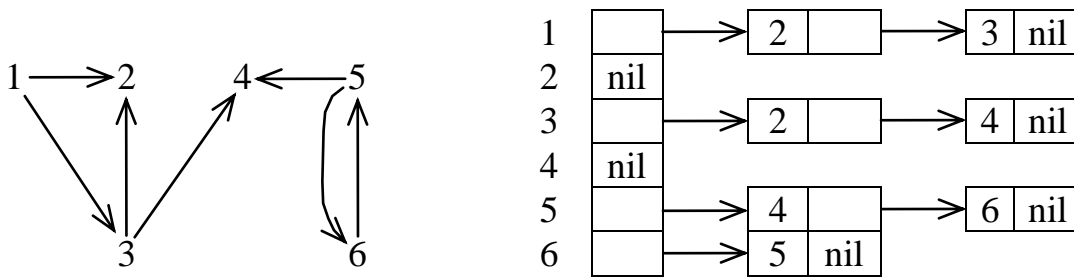


Рис. 11.4 – Список инцидентности направленного графа

Для списков инцидентности справедливо то же ограничение, что и для списков вершин – если исходной структурой данных является массив, то такой список подходит для описания только статических графов или графов с постоянным числом вершин. Для списка рёбер постоянным является число рёбер. Для описания динамического графа в качестве исходной структуры данных необходимо использовать связный список. Каждое звено такого списка будет точкой начала ещё одного списка, в котором указываются вершины, смежные с текущей. В этом случае требуется объём памяти, пропорциональный сумме $M+N$.

11.1.3. Матрица смежности

Матрица смежности представляет собой таблицу, в которой каждой вершине графа ставится в соответствие другая вершина. В результате, при анализе этой таблицы можно получить рёбра графа. Таблица смежности для графа, показанного на рисунке 11.1, приведена на рисунке 11.5.

	1	2	3	4
1	0	1	1	1
2	1	0	0	1
3	1	0	0	1
4	1	1	1	0

Рис. 11.5 – Матрица смежности ненаправленного графа

В матрице смежности ненаправленного невзвешенного графа каждое ребро обозначается, например, единицей. Для ненаправленного взвешенного графа вместо единицы указываются весовые коэффициенты соответствующего ребра.

Ребра направленного графа могут быть обозначены либо только одним направлением, например, единицей, либо обоими направлениями, тогда прямое направление обозначается как «1», а обратное – как «-1». Оба варианта матрицы смежности приведены на рисунке 11.6.

1	2	3	4	1	2	3	4
1	0	0	0	1	0	-1	-1
2	1	0	0	1	2	1	0
3	1	0	0	1	3	1	0
4	1	0	0	0	4	1	-1
а).				б).			

Рис. 11.6 – Матрицы смежности направленного графа

При использовании матрицы смежности, показанной на рисунке 11.6а, требуется оговаривать, что граф является ориентированным, а для случая, представленного на рисунке 11.6б это не требуется.

Аналогичным образом представляется взвешенный направленный граф. Его матрицы смежности приведены на рисунке 11.7.

1	2	3	4	1	2	3	4
1	0	0	0	1	0	-0.20	-0.30
2	0.20	0	0.33	2	0.20	0	0.33
3	0.30	0	0.15	3	0.30	0	0.15
4	0.10	0	0	4	0.10	-0.33	-0.15
а).				б).			

Рис. 11.7 – Матрицы смежности направленного взвешенного графа

Как видно на рисунках 11.5, 11.6б и 11.7б, матрицы смежности неориентированных графов всегда симметричны относительно

но диагонали (для ориентированных графов – с учётом знака). Матрица смежности позволяет получить ответ на вопрос о существовании конкретного ребра всего за один шаг просмотра, но при этом объём требуемой памяти для графа из N вершин пропорционален N^2 , независимо от количества рёбер. Матрица смежности с одной стороны, является достаточно наглядным представлением графа, а с другой обладает важным недостатком, который заключается в ещё большей сложности добавления новых вершин.

11.1.4. Матрица инцидентности

Ещё один способ описания графа, – в виде матрицы инцидентности, – часто используется в дискретной математике, но является менее эффективным, чем все рассмотренные выше. Матрица инцидентности отличается от матрицы смежности тем, что отдельный столбец используется для каждого ребра, а не вершины. Приведём матрицу инцидентности для графа, показанного на рисунке 11.1:

	(1, 2)	(1, 3)	(1, 4)	(2, 4)	(3, 4)
1	1	1	1	0	0
2	1	0	0	1	0
3	0	1	0	0	1
4	0	0	1	1	1

Рис. 11.8 – Матрица инцидентности ненаправленного графа

Для каждого ребра единицей помечаются вершины, с которыми это ребро соединено. Для направленного графа матрица инцидентности отличается только тем, что вершина, из которой выходит ребро, помечается как «-1». Для взвешенных графов вместо единицы указываются весовые коэффициенты рёбер.

Матрица инцидентности требует $N \times M$ элементов. Для получения информации о наличии некоторого ребра или о вершинах, смежных с некоторой выбранной вершиной, может потребоваться M шагов.

11.1.5. Граф как связная динамическая структура данных

Графы обычно применяются для моделирования или описания каких-либо систем или процессов – топографических карт, маршрутов транспорта или потоков грузов, сообщений и т.п., технологических процессов, отношений между объектами. При этом обычно требуется описывать вершины графа как элементы, хранящие некоторую информацию. В случае использования рассмотренных выше способов описания графов возможны два варианта:

- относящаяся к вершине информация хранится непосредственно в элементах списка или матрицы, вместе с информацией о наличии ребра, его направлении и весовых коэффициентах;
- относящаяся к вершине информация хранится отдельно, а соответствующие элементы списка или матрицы дополнительно к основным своим компонентам содержат ссылки на эту информацию.

Недостатком первого варианта для всех способов описания графа являются дополнительные затраты памяти при дублировании информации о звеньях, например, в списке инцидентности, недостатком второго варианта также являются дополнительные затраты памяти, но теперь на организацию ссылок в элементах списка или матрицы.

Ещё один недостаток, характерный как для списка инцидентности, так и для других рассмотренных способов описания графа, появляется при попытке устранить перечисленные недостатки вариантов хранения информации о вершинах. Он заключается в том, что информационные элементы имеют разную структуру – элементы базового списка вершин в списке инцидентности отличаются от элементов в списках смежных вершин.

Наконец, общий для всех рассмотренных способов описания графов недостаток – логическая структура описания графа, т.е. список или матрица, отделена и отличается от физической структуры, т.е. графа как такового. В некоторых простых случаях использования графов это можно не считать недостатком.

Представление графа как динамической структуры данных свободно от последнего недостатка, может использовать оба ва-

рианта хранения информации о вершинах и пригодно для описания практически любых графов – ориентированных и неориентированных, взвешенных, графов с петлями, мультиграфов и т.п. Структура данных показана на рисунке 11.9.

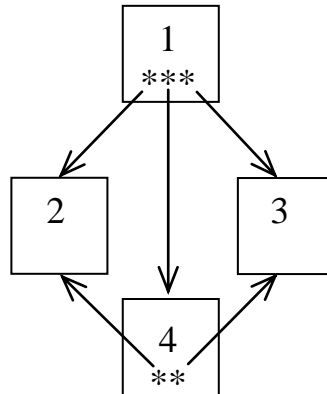


Рис. 11.9 – Направленный циклический граф как связная структура

Для иллюстрации этого способа описания направленный граф выбран не случайно. Рёбра между вершинами представляют собой связи между элементами, реализуемые при помощи ссылок или указателей. Вершина-источник ребра содержит указатель, хранящий адрес вершины, в которую входит ребро. В этом случае автоматически создаётся направленный граф. Для представления ненаправленного графа нужно создать связи в обоих направлениях. Такой вариант графа показан на рисунке 11.10.

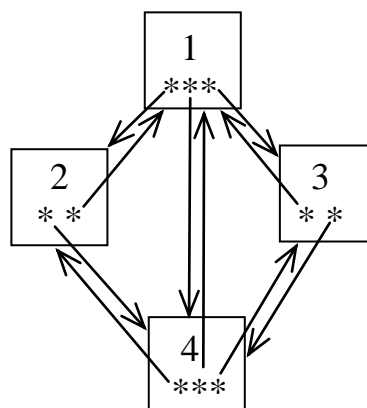


Рис. 11.10 – Ненаправленный циклический граф как связная структура

Следует указать, что такую же структуру имеет направленный граф, рёбра между вершинами которого образуют контуры.

Такая организация графа похожа на список инцидентности, за исключением того, что вершины в списках смежных вершин не повторяются, базовая структура данных не является линейной, а все элементы одинаковы. Для того чтобы можно было произвольно добавлять в граф неограниченное количество новых вершин, каждая вершина содержит внутреннюю динамическую структуру данных, например, связный список, каждый элемент которого является указателем на соседнюю вершину. Недостатком этого способа является алгоритмическая сложность некоторых операций, например, удаления вершины.

11.2. Алгоритмы на графах

Графы находят широчайшее применение при решении самых разнообразных вычислительных задач, в том числе потому, что являются структурами данных, наиболее богатыми операциями и алгоритмами обработки. Наряду с операциями, характерными и для списков и деревьев – добавлением и удалением вершин, – для графов существуют аналогичные операции для рёбер, а также поиск вершин, обход графа. Кроме того, имеются специфические для графов операции, например, построение *остовного дерева* (а для взвешенного графа – и *минимального остовного дерева*), поиск пути между заданными вершинами, в том числе кратчайшего пути или пути наименьшей стоимости, поиск эйлера или гамильтонова пути или цикла в графе, топологическая сортировка графа.

Многие алгоритмы на графах названы по фамилиям их создателей, например, алгоритмы Борувки, Прима (Ярника), Краскала (или Крускала) – построение минимального остовного дерева, Дейкстры, Беллмана–Форда, Флойда–Уоршалла, Джонсона – поиски кратчайших путей между вершинами, несколько алгоритмов Роберта Андре Тарьяна, в том числе для построения минимального остовного дерева и решения других задач на графах. Существуют также алгоритмы Форда–Фалькерсона, Диница, Карзанова, Фараджева, Касьянова и множество других [6].

Ситуации, возникающие при решении задач на графах, часто оказываются настолько сложными (учёт всех рёбер, инцидентных заданной вершине, и повторение этой ситуации для всех вершин), что трудоемкость алгоритмов на графах нередко составляет $O(N^2)$, что считается плохим для сортировок списков, и даже может достигать $O(N^3)$, например, для алгоритма Флойда–Уоршалла.

Одной из популярных задач, решаемых на графе, является **задача коммивояжёра (Traveling Salesman Problem, TSP)** – посещение всех вершин на взвешенном графе по маршруту минимальной стоимости (или по наиболее выгодному маршруту) с возвратом в исходную вершину. Эта задача относится к классу **NP-полных** – задач, не решаемых за **полиномиальное** время, т.е. время или число операций, которое можно обозначить достаточно простым и вычисляемым математическим выражением.

Многие алгоритмы на графах относятся к категории «**жадных**», т.е. выбирающих на каждом этапе наилучшее решение из оставшихся возможных, например, ребро с наименьшим весом при построении минимального остовного дерева. В итоге такая стратегия во многих случаях позволяет получить наилучшее решение задачи. «Жадными» являются, например, алгоритмы Прима и Краскала. В то же время доказано, что другие алгоритмы, в частности, задача коммивояжёра, не должны быть «жадными» для получения оптимального решения.

11.2.1 Операции добавления и удаления рёбер

В том случае, если граф является динамическим, т.е. допускающим добавление новых вершин и удаление существующих, а также проведение или разрыв рёбер между вершинами, для его организации удобно использовать какую-либо динамическую связную структуру данных – список или дерево. В этом случае для операций с вершинами используются обычные операции с выбранной структурой данных. То же самое справедливо и для рёбер – для их добавления и удаления можно использовать соответствующие операции со списком или деревом.

11.2.2 Поиск вершин в графе

Для поиска вершин в графе по их ключам применяются два основных алгоритма: поиск *в ширину* (**BFS – Breadth First Search**) и поиск *в глубину* (**DFS – Depth First Search**). Оба алгоритма обладают общей чертой – для своей работы они требуют дополнительной структуры данных, поиск в глубину использует *стек*, а поиск в ширину – *очередь*. Поиск в ширину обладает важным преимуществом – кроме нахождения искомой вершины он одновременно получает кратчайший маршрут к ней от точки начала поиска. Наиболее просто указанные виды поиска реализуются с помощью рекурсивных процедур, хотя возможны и их нерекурсивные варианты.

11.2.3 Построение остовного дерева

Остовное дерево графа (или *стягивающее, покрывающее* дерево, *каркас* графа, *скелет*) представляет собой дерево, включающее в себя все связанные вершины графа, но не содержащее циклов и контуров (рисунок 11.11).

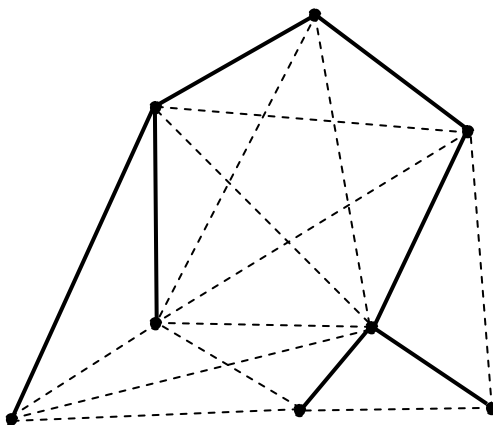


Рис. 11.11 – Остовное дерево на графе

На связном графе можно построить более чем одно остовное дерево, а для взвешенного графа существует по крайней мере одно *минимальное остовное дерево*. Создано не менее шести алгоритмов построения минимального остовного дерева, наиболее ранними и простыми из которых являются алгоритмы Отакара **Борувки** (1926 г., он же – алгоритм Соллина, Перкала, Дейкстры

и ещё четырёх исследователей), *Ярника-Прима* (1930 г.), и *Краскала* (1956 г.).

Все эти алгоритмы работают со связными взвешенными неориентированными графами. Алгоритм *Борувки-Соллина* заключается в последовательном добавлении рёбер к остовному лесу графа, до тех пор, пока лес не превратится в дерево, то есть, лес, состоящий из одной компоненты связности. На каждой итерации число деревьев в остовном лесу уменьшается по крайней мере в два раза, поэтому всего алгоритм совершает не более $O(\log V)$ итераций. Каждая итерация может быть реализована со сложностью $O(E)$, поэтому общее время работы алгоритмы составляет $O(E \log V)$ (где V и E – число вершин и рёбер в графе, соответственно). Для некоторых видов графов, в частности, планарных, оно может быть уменьшено до $O(E)$. Существует также рандомизированный алгоритм построения минимального остовного дерева, основанный на алгоритме Борувки, работающий в среднем за линейное время.

Для алгоритма *Ярника-Прима* берётся произвольная вершина и находится ребро, ей инцидентное и обладающее наименьшей стоимостью. Найденное ребро и соединяемые им две вершины образуют дерево. Затем, рассматриваются рёбра графа, одна из вершин которых уже принадлежит дереву, а вторая – нет; из этих рёбер выбирается ребро наименьшей стоимости («жадный» алгоритм). Выбираемое на каждом шаге ребро присоединяется к дереву. Построение дерева происходит до тех пор, пока не будут исчерпаны все вершины исходного графа.

До начала работы алгоритма *Краскала* необходимо отсортировать рёбра по весу, это требует $O(E \log E)$ времени. Текущее множество рёбер устанавливается пустым. Затем из всех рёбер, добавление которых к уже имеющемуся множеству не вызовет появление в нём цикла, выбирается ребро минимального веса («жадный» алгоритм) и добавляется к множеству. Когда таких рёбер больше нет, алгоритм завершён. Общее время работы алгоритма Краскала можно принять за $O(E \log E)$.

Остовные деревья находят применение в том числе в компьютерных сетях. В частности, по протоколу *STP (Spanning Tree Protocol)* – протокол остовного дерева) производится управление коммутаторами в локальных сетях Ethernet.

12. Поиск

Одно из наиболее часто встречающихся в программировании действий – поиск данных. Существует несколько основных вариантов поиска, и для них создано много различных алгоритмов. При дальнейшем рассмотрении делается принципиальное допущение: группа данных, в которой необходимо найти заданный элемент, фиксирована. Будем считать, что множество из N элементов задано в виде такого массива

```
var a: array [0..N-1] of Item
```

Обычно тип `Item` описывает запись с некоторым полем, играющим роль ключа. Задача заключается в поиске элемента, ключ которого равен заданному «аргументу поиска» x . Полученный в результате индекс i , удовлетворяющий условию $a[i].key = x$, обеспечивает доступ к другим полям обнаруженного элемента. Так как здесь рассматривается, прежде всего, сам процесс поиска, то мы будем считать, что тип `Item` включает только ключ.

С точки зрения *теории множеств* поиск можно рассматривать, как отображение множества $X = \{x_1, x_2, \dots, x_N\}$ на множество $X' = \{x_k\}$, X' является подмножеством X , и $x_k = x_i$. Если искомым данным в множестве X нет, то множество X' является пустым.

Поиск требуемой информации применяется ко всем основным структурам данных с произвольным доступом: массивам, спискам (одно- и двусвязным), деревьям, графам, таблицам.

12.1. Последовательный поиск

Нахождение информации в неотсортированной структуре данных, например в массиве, требует применения последовательного поиска.

Последовательный (или линейный) поиск – наиболее просто реализуемый метод поиска.

Последовательный поиск заключается в последовательном переборе элементов структуры данных (например, массива) от начального элемента до нахождения совпадения или до конца

структуры данных. Перебор элементов имеет линейный характер, поэтому такой поиск ещё называют линейным.

Рассмотрим примеры последовательного поиска с циклами `for` и `while`.

Функции линейного поиска на языке Паскаль.

```
var
  nums: array [0..N] of real;

function search_s1(item: ^real; n: integer;
key: real):integer;
var
  i: integer;
begin
  for i := 0 to N do
    if key = item[i] then
      begin
        search_s1 := i;
        break
      end
    else
      search_s1 := -1
    end;
end;

function search_s2(item: ^real; n: integer;
key: real):integer;
var
  i: integer;
begin
  i:=0;
  search_s2 := -1;
  while (key <> item[i]) or ( i < n) do
    begin
      if key = item[i] then
        search_s2 := i;
        i := i + 1
      end
    end;
end;
```

Аналогичные процедуры на языке Си++:

```
int search_s1(float* item, int n, float key)
{
    for (int i = 0; i < N; i++)
        if (key == item[i])
            return i;
    return -1;
}

int search_s2(float* item, int n, float key)
{
    int i=0;
    while (key != item[i] || i < n)
    {
        if (key == item[i])
            return i;
        i++;
    }
    return -1;
}
```

Последовательный поиск в среднем случае выполнит проверку $N/2$ элементов, в лучшем – 1 элемента, а в худшем – N элементов, таким образом, трудоёмкость алгоритма выражается как $O(N)$.

Недостаток этого поиска – медленное выполнение при большом объеме просматриваемого массива. Но если данные не отсортированы, то должен использоваться только последовательный поиск.

12.2. Двоичный поиск

Двоичный (или бинарный) поиск основан на итерационном сравнении ключа поиска с центральным элементом текущей части массива.

При каждой итерации находится середина текущей анализируемой части, т.е. интервал анализа делится пополам (на 2): $1/2$, $1/4$, $1/8$ и т.д., откуда этот метод поиска и получил свое название

(ещё один вариант названия – *дихотомический*). При неудачном сравнении ключа поиска с текущими данными, в зависимости от результата сравнения, выбирается нижний или верхний полуинтервал. Процесс продолжается до тех пор, пока не будет найдено совпадение или длина интервала анализа не станет равной единице, и если при этом всё ещё нет совпадения, то фиксируется неудача поиска. Процесс поиска числа 40 в упорядоченном массиве целых чисел от 1 до 127 показан на рисунке 12.1.

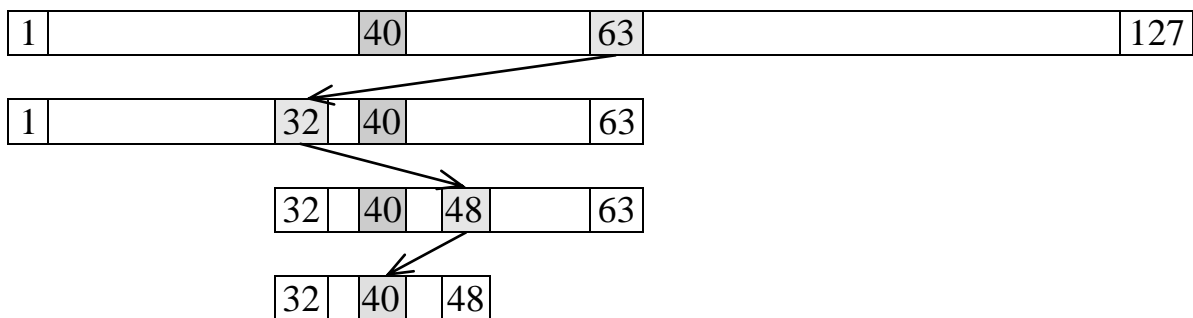


Рис. 12.1 – Поиск числа 40 при помощи двоичного алгоритма

Этот метод поиска значительно эффективнее чем последовательный поиск, но требует, чтобы данные были предварительно упорядочены (отсортированы). В худшем случае выполняется не более

$$\log_2 N + \xi' \quad (12.1)$$

сравнений (где $\xi' < 0,0861$), в связи с чем двоичный поиск ещё называется "*логарифмическим поиском*". Трудоёмкость его определяется как $O(\log_2 N)$.

Фактически упорядоченный массив используется как двоичное дерево поиска, каждый элемент массива является узлом дерева. Корень дерева – центральный элемент массива. В процессе поиска выполняется обход дерева в нисходящем порядке. Именно использованием дерева и объясняется высокая эффективность алгоритма.

Функция двоичного поиска на языке Паскаль.

```
function search_b(item: ^real; n: integer;
key: real):integer;
```

```

var
  low, high, mid: integer;
begin
  search_b := -1;
  low := 0;
  high := n - 1;
  while low <= high do
  begin
    mid := (low + high) / 2;
    if key < item[mid] then
      high := mid - 1
    else
      if key > item[mid] then
        low := mid + 1
      else
        search_b := mid
      end
    end
  end;
end;

```

Аналогичная функция на языке Си++:

```

int search_b(float* item, int n, float key)
{
  int low, high, mid;
  low = 0; high = n - 1;
  while (low <= high)
  {
    mid = (low + high) / 2;
    if (key < item[mid])
      high = mid - 1;
    else
      if (key > item[mid])
        low = mid + 1;
      else return mid;
    }
  return -1;
}

```

Существуют модификации алгоритма:

– с двумя переменными вместо low, high, mid (нижней,

- верхней границы и середины интервала анализа) – текущее положение и величина его изменения. Такой метод требует аккуратности;
- алгоритм со вспомогательными таблицами.

Поиск с использованием чисел Фибоначчи

Ряд чисел Фибоначчи определяется как

$$F(n) = F(n-1) + F(n-2), \quad (12.2)$$

$$F(0) = 0, F(1) = 1, F(2) = 1, \dots$$

$F = 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \dots$ Числа Фибоначчи используются вместо степеней двойки при делении интервала. В данном случае вместо деления используются только сложение и вычитание. Для этого требуется таблица чисел Фибоначчи, или процедура их вычисления, исходя из длины интервала. Данный метод проще реализуется при размере массива, на 1 меньше очередного числа Фибоначчи $N + 1 = F_k$.

12.3. Специальные виды поиска

Интерполяционный поиск произошел от естественного поиска данных в упорядоченном массиве человеком. Если известно, что искомый ключ k находится между некоторым «нижним» ключом k_l и некоторым «верхним» ключом k_h :

$$K_l < k < k_h, \quad (12.3)$$

то следующее сравнение делается на расстоянии

$$(l - h)(k - k_l) / (k_h - k_l) \quad (12.4)$$

от текущей позиции k_i . Предполагается что данные в массиве возрастают в арифметической прогрессии.

Именно по такому алгоритму выполняется поиск нужной карточки в "бумажном" библиотечном каталоге, или страницы в книге.

Этот алгоритм эффективнее бинарного поиска только в случае арифметической прогрессии, так как на каждом шаге уменьшает интервал анализа не до $n/2$ а до $\sqrt{n_i}$ и требует в среднем око-

ло $\log_2(\log_2(N))$ шагов, если упорядоченные данные имеют равномерное распределение.

Недостаток этого алгоритма – затраты времени на дополнительные операции при внутреннем поиске. Разница между $\log_2(\log_2(N))$ и $\log_2(N)$ становится весьма существенной при больших N , а типичные наборы данных (которыми для этого поиска часто являются файлы) обычно считаются недостаточно случайными.

Этот алгоритм может быть достаточно успешным при поиске во внешних запоминающих устройствах.

Ещё один специальный случай поиска – это поиск вхождения некоторой подстроки в строку. Для *поиска подстроки в строке* могут использоваться алгоритмы *Бойера–Мура* (и его несколько модификаций) и *Кнута–Морриса–Пратта*.

Вопросы и задания для самоконтроля

- 12.1. Что такое поиск?
- 12.2. Что называется ключом поиска?
- 12.3. Какие известны методы поиска?
- 12.4. К каким структурам данных может применяться операция поиска?
- 12.5. Какой из основных алгоритмов поиска является наиболее эффективным?
- 12.6. Какое требование предъявляется к структуре данных, в которой выполняется двоичный поиск?
- 12.7. Чем отличается поиск в массиве от поиска в списке?
- 12.8. Чем отличаются процедуры поиска в односвязном и двусвязном списках?
- 12.9. В чем заключается метод линейного поиска?
- 12.10. В чем заключается метод двоичного поиска?
- 12.11. Почему двоичный поиск получил такое название?
- 12.12. Какие известны варианты двоичного поиска?
- 12.13. Пригоден ли двоичный метод для поиска данных в неупорядоченной структуре?
- 12.14. Какой из методов поиска данных в массиве является более универсальным?

12.15. Существуют ли какие-нибудь недостатки у линейного поиска? Если да, то какие?

12.16. Какова трудоёмкость линейного поиска?

12.17. Перечислите особенности последовательного поиска.

12.18. В каких случаях применяется последовательный поиск?

12.19. Перечислите достоинства и недостатки последовательного поиска.

12.20. Соблюдение какого условия необходимо для применения к структуре данных двоичного поиска?

12.21. Какими достоинствами обладает двоичный поиск?

12.22. Какова трудоёмкость двоичного поиска?

12.23. Предложите эффективный алгоритм поиска в массиве упорядоченных неуникальных данных.

13. Сортировка

Чтобы воспользоваться эффективными алгоритмами поиска, данные должны быть упорядочены в том или ином порядке, иначе говоря, отсортированы.

Сортировка (или упорядочение), также как и поиск, является одной из часто используемых при обработке данных операций и применяется ко многим структурам данных: массивам, связным спискам, деревьям и графам.

Существуют свидетельства, что первой программой для ЭВМ с хранимой программой (с архитектурой Джона фон Неймана) была именно программа сортировки [8].

13.1. Классификация алгоритмов сортировки

Количество алгоритмов сортировки достаточно велико. В первом издании книги *Дональда Кнута «Искусство программирования для ЭВМ»* [8] упоминаются о существовании (на момент написания книги) около двадцати пяти алгоритмов сортировки. В настоящее время это число ещё больше увеличилось, и вместе с экзотическими и мало распространёнными алгоритмами приближается к 40 (и даже уже превышает 40, а с модификациями ещё больше – около 42-44, скорее всего и эти цифры заниженные). Все эти алгоритмы можно классифицировать по нескольким признакам.

1. В зависимости от того, сортируются данные в оперативной памяти машины (ОЗУ) или во внешнем ЗУ, сортировка бывает *внутренней* или *внешней*.

2. В зависимости от того, какая структура данных подвергается упорядочению, может быть сортировка массива, связного списка, дерева (пирамиды), графа.

3. В зависимости от того, как структура данных меняется в процессе сортировки может быть *сортировка списка* (если элементы структуры данных перемещаются), или *сортировка таблицы адресов* (если элементы не перемещаются, а сортируется имеющаяся или создаваемая таблица адресов). В таблицу адресов могут быть добавлены ключи, тогда получается *сортировка таблицы ключей*.

4. По особенностям функциональной реализации алгоритмы сортировки делятся на группы:

Первичные:

- сортировка перестановками (обменная);
- сортировка выбором (отбором);
- сортировка вставками;

Вторичные:

- сортировка подсчетом;
- сортировка слиянием;
- распределяющая сортировка.

5. По широте применения – универсальные (большинство перечисленных выше групп алгоритмов) и алгоритмы для конкретных случаев, не универсальные.

6. По признаку перестановки совпадающих данных в процессе сортировки – алгоритмы, не меняющие порядок следования совпадающих ключей (иногда такие алгоритмы не совсем удачно называют «устойчивыми» или «стабильными»), например, **сортировка вставками**, и меняющие порядок следования («неустойчивые», «нестабильные»).

7. По характеру зависимости времени работы от размера N структуры данных:

– $O(N^2)$ – N -квадратичные (это самые простые алгоритмы из первых трёх функциональных групп);

– $O(N^k)$, где $1 < k < 2$ (например, $k = 1,6667$ или $k = 1,5$ или $k = 1,27$ и т.п. – это **алгоритм Шелла**); возможен также случай, когда $k = 3$ – это один из самых неудачных и неэффективных алгоритмов, который получил название **глупая** (или **дурацкая**) **сортировка**;

– $O(\log_2 N)$ – логарифмические (например, **быстрая сортировка**, сортировка **слиянием** и **пирамидальная сортировка**), более точное выражение – $O(N \log_2 N)$;

– $O(N)$ – алгоритмы, требующие линейного времени, например, **«блинная» сортировка**.

Возможны также некоторые другие принципы разделения алгоритмов сортировки, в том числе по занимаемой дополнительной памяти.

Большинство алгоритмов имеет модификации. В книге Дональда Кнута упоминаются более двадцати алгоритмов внутренней сортировки и их комбинации.

При таком большом разнообразии важными оказываются **критерии выбора алгоритма**:

- средняя скорость сортировки (среднее время, удельная скорость);
- скорость в лучшем и худшем случаях (или минимальное и максимальное время);
- естественность поведения;
- переставляются ли элементы с совпадающими значениями (ключами).

Скорость (или время) сортировки определяется количеством операций сравнения и перестановки. У разных алгоритмов время работы находится в экспоненциальной или логарифмической зависимости от числа элементов структуры данных (массива) N .

Лучший случай – это ситуация, когда структура данных уже отсортирована в нужном порядке, **худший случай** – структура данных отсортирована в порядке, обратном требуемому. Время в лучшем и худшем случаях важны, если возможно частое повторение одной из этих ситуаций. Часто при хорошей средней скорости алгоритмы имеют неприемлемую скорость в худшем случае.

Естественность поведения означает, что если массив уже упорядочен, должно выполняться минимальное число операций (в идеале 0). Максимальное число операций выполняется, если массив отсортирован в обратном порядке.

Кроме перечисленных критериев при **выборе алгоритма** следует принимать во внимание:

- размер структуры данных, которую предстоит сортировать (некоторые алгоритмы, высокоэффективные в средних случаях, например **быстрая сортировка**, не показывают своей высокой эффективности для массивов малого размера);
- степень исходной упорядоченности сортируемых данных;
- требования к необходимой памяти (разным алгоритмам может требоваться $O(1)$, $O(N)$ или $O(N^2)$ памяти машины). Известно несколько примеров, когда эффективные алгоритмы тре-

буют таких больших объемов машинной памяти, что это сводит на нет все преимущества «эффективности» алгоритма;

– трудоёмкость реализации алгоритма в сравнении со сложностью решаемой в программе задачи (например, *пирамидальная сортировка* является одним из самых сложных алгоритмов сортировки). Эффективные, но сложные алгоритмы могут быть нежелательными, если готовые программы будут поддерживать люди, не участвовавшие в разработке этих программ. Необходимо предусмотреть возможность того, что эффективные, но достаточно сложные алгоритмы не будут востребованы из-за своей сложности и трудоёмкости их изучения и освоения.

13.2. Пузырьковая сортировка

Самый простой алгоритм группы *обменных* сортировок (или сортировок *перестановками*) получил название *пузырьковой* сортировки. Этот алгоритм считается самым простым из универсальных, одновременно являясь одним из самых малоэффективных (т.е. медленных). Пузырьковая сортировка заключается в *сравнении* соседних элементов и, при необходимости, в их *перестановке*. При неубывающем упорядочении элементы «всплывают» как пузырьки каждый до своего уровня (а другие элементы одновременно «тонут»).

Кроме пузырьковой сортировки к группе *обменных* относятся также *быстрая* сортировка и сортировка *«расчёской»*. Следует отметить, что операция перестановки используется не только в алгоритмах обменной сортировки, но и во многих других.

Описание алгоритма:

Используются *два цикла*. *Внешний* проходится $N - 1$ раз, это гарантирует, что даже в худшем случае каждый элемент будет находиться на своем месте. В этом цикле устанавливается (смещается) граница между отсортированной и неотсортированной частями структуры данных. Во *внутреннем* цикле выполняются непосредственно операции сравнения и перестановки. Перестановка элементов производится методом *«трёх вёдер»*: содержимое первого элемента помещается в буферную переменную, на его место помещается содержимое второго элемента, на

место которого помещается из буфера «старое» содержимое первого элемента. Количество проходов *внутреннего* цикла в каждом следующем проходе *внешнего* цикла уменьшается (от $N - 1$ до 1). Считается, что в среднем число проходов *внутреннего* цикла равно $N/2$. Процесс сортировки носит «треугольный» характер (рисунок 13.1).

Рассмотрим исходный массив символов, содержащий значения f, d, a, c, b, e.

В процессе работы программы массив будет изменяться следующим образом (показаны исходное состояние массива и его состояние после каждого прохода внешнего цикла):

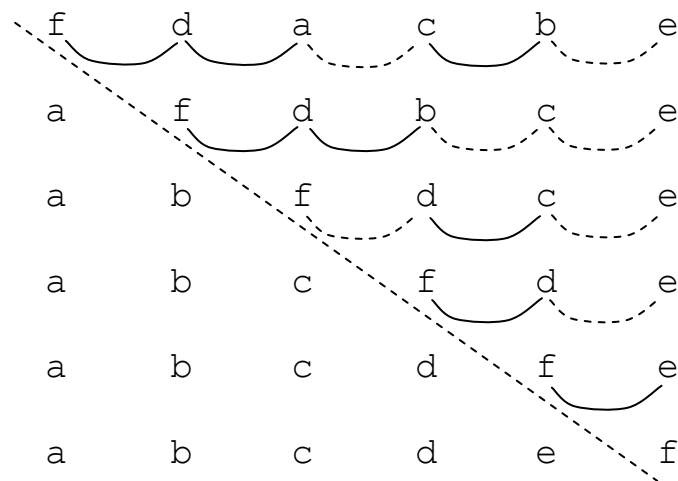


Рис. 13.1 – Изменение массива в процессе пузырьковой сортировки

Рассматриваемый вариант алгоритма всегда выполняет

$$N(N - 1)/2 \quad (13.1)$$

или

$$(N^2 - N)/2 \quad (13.2)$$

операций *сравнения*, так как внешний цикл выполняется $N - 1$ раз, а внутренний $N/2$. Это выражение соответствует площади прямоугольного треугольника с катетами N и $N - 1$.

Число *перестановок* зависит от степени предварительной упорядоченности массива: В *лучшем* случае перестановки не выполняются (массив уже отсортирован в нужном направлении и число перестановок равно 0);

В *среднем* случае выполняется

$$N(N - 1)/4 = (N^2 - N)/4 \quad (13.3)$$

перестановок (т.е. число сравнений делится пополам), число операций присваивания оказывается утроенным, т.е.

$$3N(N - 1)/4. \quad (13.4)$$

В *худшем* случае (массив отсортирован в обратном направлении) выполняется

$$N(N - 1)/2 = (N^2 - N)/2 \quad (13.5)$$

перестановок или

$$3N(N - 1)/2 \quad (13.6)$$

операций присваивания.

Из-за наличия в формулах компонента N^2 , пузырьковая сортировка относится к N -квадратичным алгоритмам (см. рисунок 13.2). При работе с большими структурами данных (массивами или связными списками) пузырьковая сортировка неэффективна.

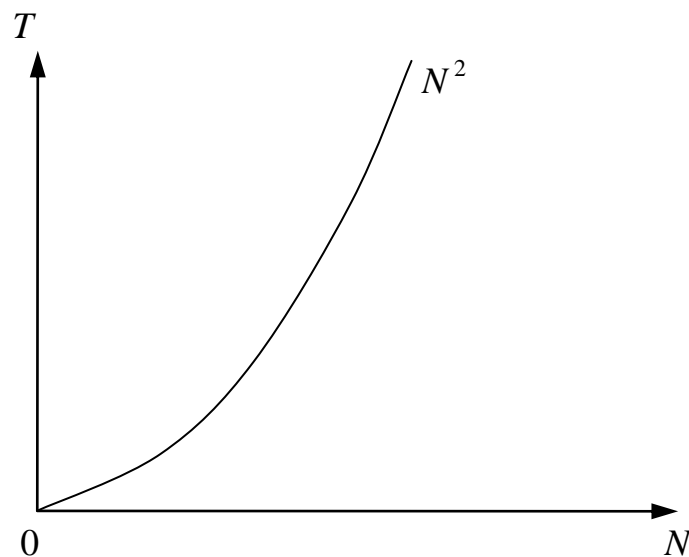


Рис. 13.2 – Зависимость времени работы от размера массива для N -квадратичных алгоритмов

Считается, что пузырьковая сортировка вообще является "учебной", но в реальности она может применяться для упорядочения массивов очень малого размера (3 – 7 элементов) в различных алгоритмах обработки сигналов.

Процедура пузырьковой сортировки на языке Паскаль.

```
var str: array[0..M] of char;
procedure bubble(item: array of char; n: integer);
var
  a, b: integer;
  t: char;
begin
  for a := 1 to n - 1 do
    for b := n - 1 downto a do
      if item[b-1] > item[b] then
        begin
          t := item[b-1]; { Перестановка }
          item[b-1] := item[b];
          item[b] := t
        end
      end;
    end;
  end;
```

Во втором варианте процедуры предпринимается попытка закончить процесс сортировки до завершения внешнего цикла, если все элементы уже заняли свои позиции (т.е. если в очередном проходе внутреннего цикла не было выполнено ни одной операции перестановки) – это соответствует *условию Айверсона* (или *скобке Айверсона*). Этот вариант процедуры длиннее, но позволяет при «удачном» расположении сортируемых данных делать меньше проходов.

```
procedure bubble1(item: array of char; n: integer);
var
  a, ssign, pass: integer;
  t: char;
begin
  ssign := 1;
  pass := 0;
  while ssign <> 0 do
    begin
      ssign := 0;
      pass := pass + 1;
```

```

    for a:=0 to n - pass - 1 do
        if item[a] > item[a + 1] then
            begin
                t := item[a - 1];
                item[a - 1] := item[a];
                item[a] := t;
                ssign := 1
            end
        end
    end
end;

```

Аналогичные функции на языках Си/Си++.

```

void bubble(char* item, int n)
{
    int a, b;
    char buf;
    for (a = 1; a < n; ++a)
        for (b = n - 1; b >= a; --b)
            if (item[b - 1] > item[b])
                {
                    buf = item[b - 1];    // Перестановка
                    item [b - 1] = item[b];
                    item[b] = buf;
                }
}

```

Вариант с условием Айверсона (перестановка не показана):

```

void bubble1(char* item, int n)
{
    int a, ssign = 1, pass = 0;
    char buf;
    while(ssign)
    {
        ssign = 0;
        pass++;
        for (a = 0; a < n - pass; a++)
            if (item[a] > item[a + 1])
                {

```

```

        // Перестановка
        ssign++;
    }
}

```

Условие Айверсона является одной из попыток оптимизации этого алгоритма для повышения его эффективности, но обычно все такие попытки, включая ручную оптимизацию машинных команд, не приводят к заметному повышению скорости работы алгоритма.

Существуют текстовые модификации приведённых процедуры, в которых, например, счётчики обоих циклов меняются на увеличение и т.п.

Тип упорядоченности – по возрастанию или по убыванию – зависит от знака операции сравнения текущего и последующего элементов.

13.3. Сортировка отбором

Другое название – сортировка ***выбором***.

В структуре данных (массиве или списке) ищется наименьший (или наибольший) элемент и меняется местами с первым (или с последним, в зависимости от направления упорядоченности). (Здесь сочетаются два процесса – поиск и обмен.) Затем из оставшихся $N - 1$ элементов ищется наименьший и меняется местами со вторым. Процесс повторяется до тех пор, пока нечего будет переставлять (т.е. пока структура данных не исчерпается).

При реализации алгоритма желательно оптимальным образом совместить процессы поиска минимума и обмена, для достижения максимальной эффективности.

Процедура сортировки простым или прямым выбором на языке Паскаль.

```

procedure select(item: array of char; n: integer);
var
    a, b, c, ssign: integer;
    t: char;

```

```

begin
  for a := 0 to n - 2 do
    begin
      ssign := 0;
      c := a;
      t := item[a];
      for b := a + 1 to n do
        if item[b] < t then
          begin
            c := b;
            t := item[b];
            ssign := 1
          end;
        if ssign <> 0 then
          begin
            item[c] := item[a];
            item[a] := t
          end
        end
      end
    end
  end;
end;

```

Такая же функция на языках Си/Си++.

```

void select_sort(char* item, int n)
{
  int a, b, c;
  char buf;
  int change;
  for(a=0; a < n - 1; ++a)
  {
    buf = item[a]; // Текущий минимум
    c = a; // Индекс минимума
    change=0; // Признак обмена
    for (b = a + 1; b < n; ++b)
      if (item[b] < buf)
      {
        buf = item[b]; // Начало обмена
        c = b; // Новый минимум
        change = 1; // Признак обмена
      }
  }
}

```

```

    if (change)
    {
        item[c] = item[a]; // Продолжение обмена
        item[a]=buf;
    }
}
}

```

Фактически в этих процедурах во *внешнем* цикле устанавливается нижняя граница интервала анализа и элемент с таким индексом принимается за опорный. Во *внутреннем* цикле каждый из оставшихся элементов сравнивается с опорным и если условие сравнения выполняется, найденный локальный минимум фиксируется в буфере (принимается за новый опорный) и устанавливается в «1» признак обмена. В продолжении *внешнего* цикла происходит обмен, если признак равен «1». Эффективность алгоритма повышается за счёт разнесения операции обмена по разным циклам (по сравнению с пузырьковой сортировкой).

Иллюстрация работы алгоритма показана на рисунке 13.3.

Здесь хорошо виден такой же «треугольный» характер, что и у пузырьковой сортировки.

Этот алгоритм так же является N -квадратичным. Внешний цикл выполняется $N - 1$ раз, внутренний – $N/2$ раз, каждая полная перестановка требует трех операций присваивания.

Число операций сравнения и перестановки (присваивания):

$(N^2 - N)/2$ сравнений;

$N - 1$ присваиваний в *лучшем* случае;

$N^2/4 + 3(N - 1)$ присваиваний в *худшем* случае;

$N(\log_2 N + y)$ присваиваний в *среднем* случае,

где $y \approx 0,577216$ – константа Эйлера.

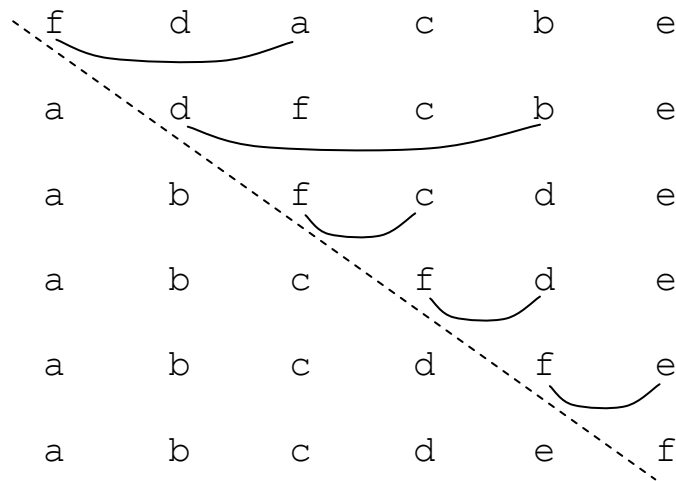


Рис. 13.3 – Обработка массива при сортировке отбором

При одинаковом числе сравнений, сортировка отбором эффективнее пузырьковой в среднем случае за счёт наличия логарифма в выражении для числа перестановок. Лучшая эффективность объясняется в том числе и тем, что в приведённых выше примерах реализации этой сортировки процессы поиска экстремума и обмена данными совмещаются, кроме того разные операции присваивания при обмене данными (метод «трёх вёдер») оказываются разнесёнными в разные части процедуры и выполняются неодинаковое количество раз.

К *модификациям* этого алгоритма относится *пирамидальная* сортировка. Существует также двунаправленный вариант сортировки отбором, в котором на каждом проходе отыскиваются и устанавливаются на свои места и минимальное, и максимальное значения.

13.4. Сортировка вставками

На первом шаге упорядочиваются один относительно другого два первых элемента структуры данных (массива или списка). Затем в упорядочивании участвуют три первых элемента (третий элемент вставляется в соответствующую позицию относительно первых двух элементов), потом – четыре и т. д., пока вся структура данных не будет упорядочена.

Ниже приведен пример алгоритма сортировки *простыми вставками* (просеиванием или погружением).

Процедура сортировки вставками на языке Паскаль.

```
procedure insert(item: array of char; n: integer);
var
  a, b: integer;
  t: char;
begin
  for a:=1 to n - 1 do
  begin
    t := item[a];
    b := a - 1;
    while (b >= 0) and (t < item[b]) do
    begin
      item[b+1] := item[b];
      b := b-1
    end;
    item[b + 1] := t
  end
end;
```

Процедура сортировки вставками на языках Си/Си++.

```
void insert_sort(char* item, int n)
{
  int a, b;
  char buf;
  for(a=1; a<n; ++a)
  {
    buf = item[a];
    for(b = a-1; b >= 0 && buf < item[b]; --b)
      item[b + 1] = item[b];
    item[b + 1] = buf;
  }
}
```

В отличие от предыдущих алгоритмов, число сравнений зависит от исходной упорядоченности массива. Если массив уже отсортирован в требуемом порядке, то за всё время работы процедуры выполняется $N - 1$ операций сравнения, если не отсорти-

рован, то $(N^2 - N)/2$ сравнений. Количество операций присваивания при перестановках определяется следующим образом [22]:

в лучшем случае – $2(N - 1)$;

в худшем случае оценивается в общем как $(N^2 + 2N)/2$.

Для среднего случая точную оценку дать достаточно сложно, но в общем виде время работы действительно оказывается средним между лучшим и худшим случаями.

В исходном массиве данный алгоритм произведет перестановки, показанные на рисунке 13.4.

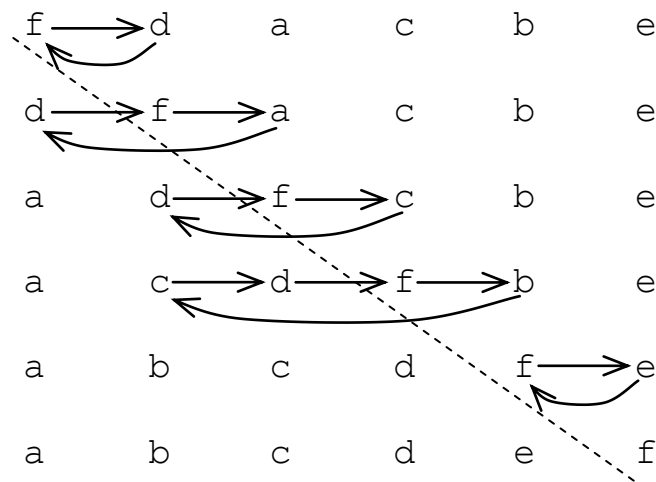


Рис. 13.4 – Обработка массива при сортировке вставками

Этот алгоритм в среднем лишь немного лучше предыдущих (также за счёт разнесения операций присваивания, выполняющих обмен, по разным циклам), но обладает несколькими преимуществами:

- его поведение естественно: если массив уже отсортирован в нужном порядке, алгоритм проводит минимальное количество вычислений, и максимальное, если массив отсортирован в порядке, обратном требуемому. Происходит быстрая обработка почти упорядоченных массивов;

- порядок следования одинаковых ключей не изменяется. Если список отсортирован по двум ключам, то после сортировки вставками он остается отсортированным по обоим ключам;

- одна из самых коротких процедур среди основных алгоритмов сортировки.

Недостаток: при каждой вставке производится сдвиг массива, следовательно даже при относительно малом числе сравнений число перестановок может быть довольно значительным.

Существуют варианты этого алгоритма: бинарные вставки, двухпутевые вставки.

Сортировка вставками хорошо подходит для связных списков, т.к. в этом случае не требуется перенос данных между различными элементами, в отличие от массива.

13.5 Алгоритм Шелла

Разработан Дональдом Л. Шеллом в 1959 году.

Этот алгоритм классифицируется как сортировка *вставками с убывающим шагом*.

В первом проходе попарно переставляются элементы, находящиеся друг от друга на некотором расстоянии, например, 9 позиций. Во втором проходе переставляются элементы, отстоящие на меньшее число позиций, например, 5. И т. д. На последнем шаге сортируются соседние элементы. На ранних этапах изучения алгоритма его исследователи отмечали: «Непонятно, как алгоритм работает».

Принцип убывающих шагов может быть применён для различных базовых сортировок, в том числе для *пузырьковой* сортировки. В результате такой модификации получается сортировка *расчёской*.

Эффективность алгоритма заключается в том, что на каждом из промежуточных шагов сортируется либо небольшое число элементов, либо уже достаточно хорошо упорядоченные наборы элементов. Упорядоченность массива возрастает после каждого прохода.

Поведение алгоритма Шелла до конца не исследовано до сих пор, в частности, не найдено универсальное решение проблемы выбора последовательности изменения шага. Исследование алгоритма привело к формулированию нескольких теорем.

Эффективность алгоритма проявляется даже в случае всего двух проходов с шагами h и 1. Время работы в этом случае зависит от размера массива как

$$t \approx \frac{2N^2}{h} + \sqrt{\pi N^3 h} . \quad (13.7)$$

А лучшее значение шага h оказывается равным

$$h \approx \sqrt[3]{\frac{16N}{\pi}} \approx 1,72\sqrt[3]{N} , \quad (13.8)$$

следствием этого является зависимость времени работы от размера массива, которую можно выразить как $N^{1,6667}$ или $N^{5/3}$.

При большем числе проходов эффективность повышается. Тем не менее, в процессе изучения алгоритма был обнаружен порог эффективности, который долго не удавалось преодолеть (барьер $N^{1,5}$). Этот барьер можно преодолеть подбором шагов.

Изменение массива при сортировке по алгоритму Шелла будет имеет вид, показанный на рисунке 13.5 (приведены исходное состояние массива, результат каждого прохода внешнего цикла и окончательное состояние).

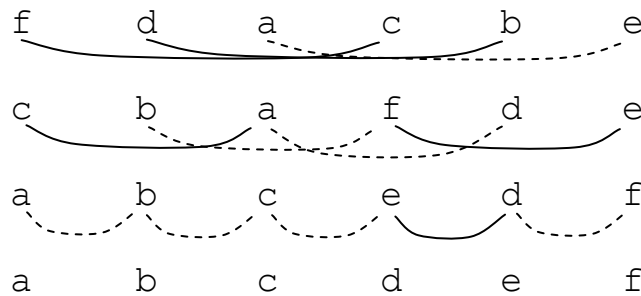


Рис. 13.5 – Обработка массива при сортировке Шелла

В ходе изучения алгоритма исследовалась зависимость среднего числа перестановок от размера массивов при N от 100 до 60000 для нескольких типов последовательностей шагов, для которых были получены соответствующие зависимости времени работ от размера массива:

$$2^{k+1}, \dots, 9, 5, 3, 1 \Rightarrow 1,09N^{1,27}$$

$$2^k - 1, \dots, 15, 7, 3, 1 \Rightarrow 1,22N^{1,26}$$

$$(2^k - (-1)^k)/3, \dots, 11, 5, 3, 1 \Rightarrow 1,12N^{1,28}$$

$$(3^k - 1)/2, \dots, 40, 13, 4, 1 \Rightarrow 1,66N^{1,25}$$

Качественное сравнение времени работы для N -квадратичных сортировок и метода Шелла приведено на рисунке 13.6.

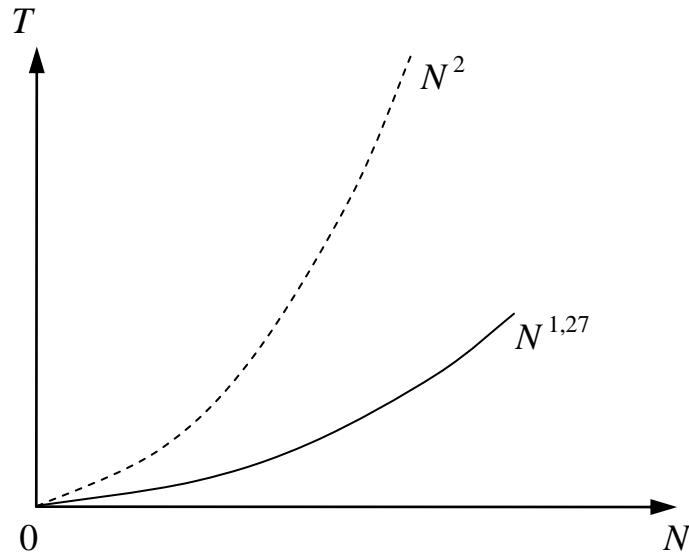


Рис. 13.6 – Сравнение зависимости времени работы от размера массива для алгоритма Шелла и N -квадратичных алгоритмов

Общую формулу можно выразить как

$$0,3N(\ln N)^2 - 1,35N(\ln N), \quad (13.9)$$

или, упростив, как

$$\alpha N(\ln N)^2 + \beta N \ln N \quad (13.10)$$

или αN^β .

Последний шаг обязательно должен быть единичным.

Распространённой (хотя и далеко не самой эффективной) является последовательность шагов 9, 5, 3, 2, 1.

Кроме указанных формул могут использоваться числа Фибоначчи.

Рекомендуется избегать последовательности шагов в виде степеней двойки (... , 16, 8, 4, 2, 1), так как доказано, что это снижает эффективность алгоритма (хотя сортировка выполняется и в этом случае).

Один из рекомендуемых вариантов выбора последовательности шагов: принять шаги последнего и каждого предыдущего этапов $h_1 = 1$, $h_{s+1} = 3h_s + 1$, и остановиться на некотором шаге h_t , когда $h_{t+2} \geq N$. В итоге получается последовательность шагов ..., 121, 40, 13, 4, 1 (такая же, как в случае $(3^k - 1)/2$), которая является одной из самых эффективных. Возможны и другие, эвристически подбираемые последовательности шагов, ещё более эффективные

(например, 120, 40, 12, 4, 1). Разумеется, большие шаги могут применяться только для массивов большого размера.

Возможен также расчёт шага первого этапа, исходя из размера массива, и уменьшение шага вдвое на каждом следующем этапе.

Процедура сортировки методом Шелла на языке Паскаль.

```
procedure shell(item: array of char; n: integer);
var
  i, j, k, h: Integer;
  x: char;
  a: array[0..4] of integer;
begin
  a[0] := 9; a[1] := 5;
  a[2] := 3; a[3] := 2;
  a[4] := 1;
  for k := 0 to 4 do
    begin
      h := a[k];
      for i := h to n-1 do
        begin
          x := item[i];
          j := i-h;
          while (x < item[j]) and (j >= 1) do
            begin
              item[j + h] := item[j];
              j := j-h;
            end;
            item[j + h] := x;
          end;
        end;
      end;
    end;
end;
```

Аналогичная функция на Си++.

```
const int ST = 5;
void Shell_sort(char* item, int n)
{
  int step[ST] = {120, 40, 12, 4, 1}; // Массив убывающих шагов
```

```

int i, j, k, h;
char buf;
for ( k = 0; k < ST; ++k)
{
    h = step[k];
    for (i = h; i < n; ++i)
    {
        buf = item[i];
        for (j = i - h; buf < item[j] && j >= 0;
j -= h)
            item[j + h] = item[j];
        item[j + h] = buf;
    }
}
}

```

Проверка условия $j \geq 1$ (для Паскаля) или $j \geq 0$ (для Си++) предотвращает выход за пределы массива. Считается, что такие дополнительные проверки слегка снижают производительность алгоритма.

13.6. Алгоритм быстрой сортировки

Классификационное название этого алгоритма – *обменная сортировка с разделением*.

Разработан Чарльзом Э. Р. Хоаром в 1962 г.

Один из лучших универсальных алгоритмов (одновременно не очень сложных), разработанных к настоящему времени. Относится к группе алгоритмов *обменной* сортировки (как и *пузырьковая* сортировка).

Сортируемый массив разбивается на два подмассива, для чего сначала выбирается пограничное или опорное значение – *компаранд* (т.е. значение, с которым будут *сравниваться* другие элементы массива). Все элементы, *большие* компаранда, переносятся в один подмассив, а *меньшие* – в другой. Весь процесс *повторяется* для *каждого* подмассива до тех пор, пока весь массив не будет упорядочен.

Процесс сортировки носит рекурсивный характер.

Перенос в подмассивы может происходить следующим образом:

Пусть имеются два индекса – i и j , причем сначала $i = 0$, $j = N - 1$.

Сравниваются элементы $k[i]$ и $k[j]$, если обмен не требуется, то j уменьшается на единицу и сравнения повторяются, j с каждым шагом уменьшается на 1.

После первого обмена i увеличивается на единицу и сравнения повторяются с увеличением i на 1, пока не произойдет следующий обмен, после которого опять начинает уменьшаться j . И так до тех пор, пока i и j не станут равны друг другу.

Представляет интерес вопрос о выборе компаранда. Существуют варианты:

- *случайный* выбор – иногда бывает наилучшим;
- выборка небольшого числа элементов из подмассива и использование в качестве компаранда медианы этой выборки. Широко распространенным и эффективным является метод выборки трех элементов и взятие среднего из них;
- один из *экстремумов* – наихудший вариант, хотя алгоритм работает и в этом случае;
- элемент, находящийся в середине каждого из подмассивов.

Процедура быстрой сортировки (метод Хоара) на языке Паскаль с компарандом в середине каждого текущего подмассива.

```
procedure quicksort(item: array of char; left,
right: integer);
var
  i, j: Integer;
  x, y: char;
begin
  i := left;
  j := right;
  x := item[int((left + right) / 2)];
  repeat
    while (item[i] < x) and (i < right) do
      i := i + 1;
    while (x < item[j]) and (j > left) do
```

```

    j := j - 1;
    if i <= j then
    begin
        y := item[i];
        item[i] := item[j];
        item[j] := y;
        i := i + 1; j := j - 1;
    end;
until i < j;
if left < j then
    quick(item, left, j);
if i < right then
    quick(item, i, right);
end;

```

Аналогичная функция на языках Си/Си++.

```

void quick_sort(char* item,int left,int right)
{
    int i, j;
    char comp, buf;
    i = left; j = right;
    comp = item[(left + right)/2]; // Компаранд
    do {
        while (item[i] < comp && i < right)
            i++;
        while (comp < item[j] && j > left)
            j--;
        if (i<=j)
        {
            buf = item[i]; // Обмен
            item[i] = item[j];
            item[j] = buf;
            i++;
            j--;
        }
    } while(i <= j);
    if (left < j)
        quick_sort(item, left, j); // Рекурсивный

```

ВЫЗОВ

```

    if (i < right)
        quick_sort(item, i, right);
}

```

Изменение массива при сортировке по алгоритму Хоара будет иметь вид, показанный на рисунке 13.7 (компаранды подчёркнуты, обрабатываемые подмассивы обведены).

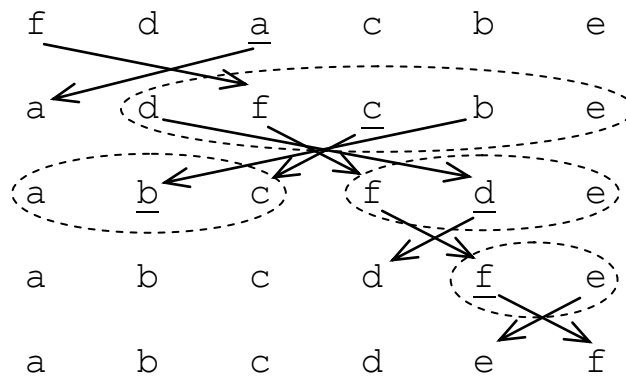


Рис. 13.7 – Обработка массива при быстрой сортировке

Для оптимальной сортировки в качестве компаранда следует выбрать элемент, расположенный точно в середине **диапазона** значений текущего подмассива. Однако для большинства наборов данных поиск такого элемента может вылиться в самостоятельную сортировку.

Характеристики: среднее число **сравнений** $M\log_2 N$, среднее число **операций присваивания** при перестановках $(1/6)M\log_2 N$ [22].

У этого алгоритма есть одна отрицательная особенность: если для каждого текущего подмассива компарандом является текущий экстремум (как на первом и последнем этапах сортировки, показанной на рисунке 13.7), то алгоритм становится N -квадратичным. Однако считается, что вероятность такого события обычно невысока. Возможность вырождения быстрой сортировки в N -квадратичную является недостатком этого алгоритма. От такого недостатка свободны сортировки **слиянием** и **пирамидальная**, также обладающие эффективностью $O(M\log_2 N)$, но существенно более сложные, чем алгоритм Хоара.

13.7. Параллельная сортировка Бэтчера

Обменная сортировка со слиянием. Разработана К. Э. Бэтчером в 1964 г.

Алгоритм напоминает *алгоритм Шелла* – также сравниваются пары несоседних ключей, но пары подбираются по-другому. Используется последовательность шагов 8, 4, 2, 1, но сравниваемые пары не перекрываются (т.е. в двух сравниваемых соседних парах нет общего элемента, как в алгоритме Шелла – 1-й и 9-й, 9-й и 17-й, 17-й и 26-й и т.д.). Далее происходит слияние пар отсортированных подмассивов.

Процедура параллельной сортировки Бэтчера на языке Паскаль.

```
procedure batcher (item: array of char;
  n: Integer);
var
  f, h, i, q, r, z: Integer;
  buf: char;
begin
  (* f:=Log2n; h:=2^(f-1); *) (* Библиотечные
  функции логарифма и возведения в степень *)
  while h > 0 do
  begin
    r := 0;
    z := h;
    (* q:=2^(f-1); *)
    while q >= h do
    begin
      for i:=0 to n-z-1 do
        if (i and h) = r then
          z := q - h; (* сравнение и обмен *)
          q := q / 2;
          r := h
        end;
      h := h div 2
    end
  end;
end;
```

Последовательность сравнений и перестановок, а также состояния переменных, показаны на рисунке 13.8.

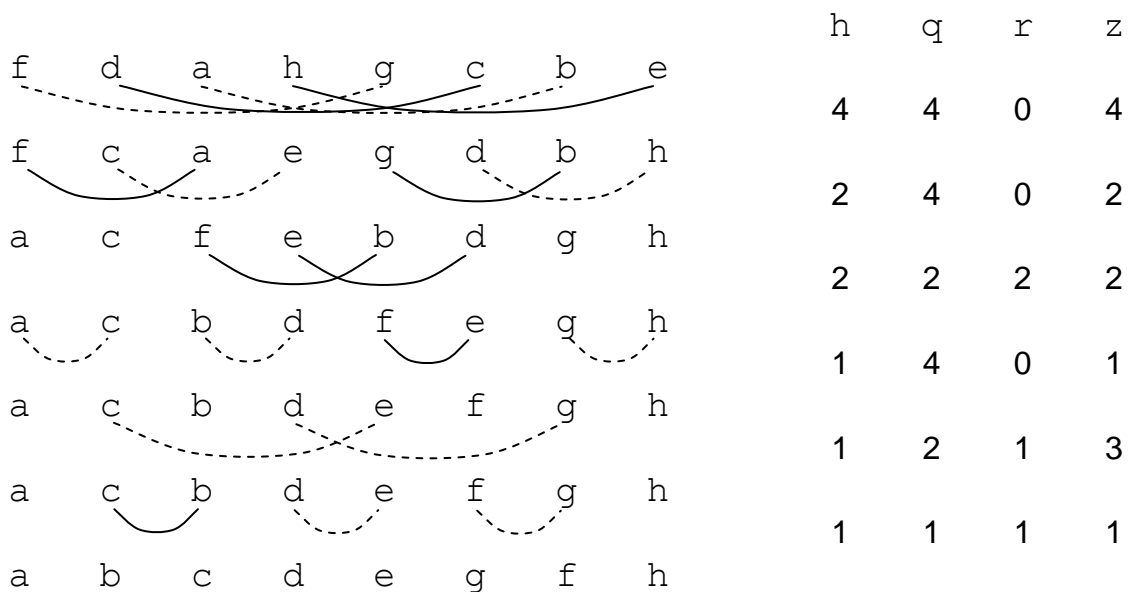


Рис. 13.8 – Сортировка массива алгоритмом Бэтчера

Выбирается начальный интервал сравнения пар h из условия $h = 2^{f-1}$, где $f = \log_2 N$ – наименьшее целое число, такое, что $2^f \geq N$. Перед первым проходом устанавливаются вспомогательные переменные

$$q = 2^{f-1}, r = 0, z = h.$$

Выполняется сравнение и обмен элементов для всех i , таких, что $0 \leq i < (N - z)$ и $i \& h = r$ (операция И над i и h), т.е. при необходимости меняются местами элементы с индексами $i + 1$ и $i + z + 1$, $i + k$ и $i + z + k$ и т.п.

К этому моменту z – нечетно кратно h (т.е. z / h – нечетное число), а h – степень двойки, следовательно $(i \& h) \neq ((i + z) \& h)$, значит, сравнение и обмен можно выполнять при всех нужных значениях i в любом порядке или даже одновременно. В связи с этим сортировка и названа **параллельной**.

Затем изменяются значения вспомогательных переменных $z = q - h$, $q = q / 2$, $r = h$ и процесс продолжается, пока $q \geq h$.

К этому моменту исходный массив будет упорядочен с шагом h .

Затем h уменьшается в два раза ($h = h / 2$) (с учетом целого типа) и весь процесс продолжается, пока $h > 0$.

Значительное количество вспомогательных операций, необходимых для управления последовательностью сравнений, снижает эффективность алгоритма.

Но все сравнения и обмены можно выполнять одновременно (если существует такая возможность, например на ЭВМ с распараллеливанием вычислений). В этом случае сортировка выполняется за $(\log_2 N(\log_2 N + 1))/2$ шагов.

Вопросы и задания для самоконтроля

- 13.1. Что такое «сортировка»?
- 13.2. Сколько существует групп алгоритмов сортировки?
- 13.3. Сколько существует алгоритмов сортировки?
- 13.4. По каким признакам характеризуются алгоритмы сортировки?
- 13.5. Что нужно учитывать при выборе алгоритма сортировки?
- 13.6. Какой алгоритм сортировки считается самым простым?
- 13.7. Какой алгоритм сортировки считается самым эффективным?
- 13.8. Что означает понятие «скорость сортировки»?
- 13.9. В чем заключается метод пузырьковой сортировки?
- 13.10. Модифицируйте рассмотренную в параграфе 13.2 функцию пузырьковой сортировки так, чтобы можно было контролировать состояние массива на каждом этапе работы алгоритма.
- 13.11. Предложите функцию пузырьковой сортировки для односвязного списка.
- 13.12. В чем заключается метод сортировки отбором?
- 13.13. Модифицируйте рассмотренную в параграфе 13.3 функцию сортировки отбором так, чтобы можно было контролировать состояние массива на каждом этапе работы алгоритма.
- 13.14. Изобразите диаграмму изменения состояния массива, аналогичную рисунку 13.1, для пузырьковой сортировки при худшем случае.

13.15. Изобразите диаграмму изменения состояния массива, аналогичную рисунку 13.1, для пузырьковой сортировки при лучшем случае.

13.16. Предложите функцию сортировки отбором для односвязного списка.

13.17. В чем заключается метод сортировки вставками?

13.18. Модифицируйте рассмотренную в параграфе 13.4 функцию сортировки вставками так, чтобы можно было контролировать состояние массива на каждом этапе работы алгоритма.

13.19. Модифицируйте функцию сортировки вставками так, чтобы можно было контролировать количество операций сравнения и присваивания.

13.20. Как будет изменяться при сортировке вставками односвязный линейный список?

13.21. Предложите функцию сортировки вставками для односвязного списка.

13.22. Изобразите диаграмму изменения состояния массива, аналогичную рисунку 13.4, для сортировки вставками при худшем случае.

13.23. В чем заключается метод сортировки Шелла?

13.24. Модифицируйте рассмотренную в параграфе 13.5 функцию сортировки методом Шелла так, чтобы можно было контролировать состояние массива на каждом этапе работы алгоритма.

13.25. Можно ли применить метод Шелла для сортировки связного списка?

13.26. Оправданно ли с точки зрения эффективности применение сортировки Шелла для связного списка?

13.27. В чем заключается метод быстрой сортировки?

13.28. Модифицируйте рассмотренную в параграфе 13.6 функцию быстрой сортировки так, чтобы можно было контролировать состояние массива на каждом этапе работы алгоритма.

13.29. В чем заключается метод сортировки Бэтчера?

13.30. Как зависит скорость сортировки от размера структуры данных для разных алгоритмов?

13.31. Почему метод Бэтчера называется параллельным?

13.32. Каковы критерии выбора алгоритма сортировки?

Заключение

В современных версиях основных объектно-ориентированных языков программирования – C++, Java и C#, – имеется стандартная библиотека, которая ещё недавно в языке C++ называлась стандартной библиотекой шаблонов (*STL* – Standard Template Library). Эта библиотека содержит в себе набор шаблонных классов, описывающих основные структуры данных – векторы, списки, множества, карты, стеки, очереди и деки (<http://www.cplusplus.com/reference/stl/>). Использование этих шаблонных классов обладает определенной спецификой и не рассмотрено в настоящем учебном пособии, в том числе по той причине, что все операции с перечисленными структурами данных полностью интегрированы в классы и в таком виде не подходят как примеры для изучения. Желающим изучить эти шаблоны и их практическое использование рекомендуется обратиться к соответствующим источникам.

То же самое можно порекомендовать при более подробном изучении алгоритмов, кратко упомянутых в учебном пособии или вообще в него не вошедших. В частности, операции с красно-чёрным и расширяемым деревьями хорошо рассмотрены в книгах Роберта Седжвика.

Для решения различных вычислительных задач могут не только использоваться известные структуры данных, но и создаваться новые. Примером такой относительно новой структуры данных является V-список. В своей работе квалифицированные программисты должны выбирать оптимальные структуры данных, основываясь на характере решаемой задачи и ограничениях на допустимые время работы и объём занимаемой памяти. залогом успеха являются знание теории структур данных и алгоритмов и постоянная практика в разработке программ.

Библиографический список

1. Алгоритмы: построение и анализ [Текст] : пер. с англ. / Т. Х. Кормен [и др.]. – 2-е изд. – М. : Издательский дом «Вильямс», 2005. – 1296 с.
2. Алексеев, В. Е. Графы и алгоритмы. Структуры данных. Модели вычислений [Текст] : учебник для вузов / В. Е. Алексеев, В. А. Таланов. – М. : Интернет Ун-т Информ. Технологий : БИНОМ, 2006. – 320 с.
3. Ахо, А. В. Структуры данных и алгоритмы [Текст] : пер. с англ. / А. В. Ахо, Д. Э. Хопкрофт, Д. Д. Ульман. – М. : Вильямс, 2003. – 382 с.
4. Берзтисс, А. Т. Структуры данных [Текст] : пер. с англ. / А. Т. Берзтисс. – М. : Статистика, 1974. – 406 с.
5. Вирт, Н. Алгоритмы и структуры данных с примерами на Паскале [Текст] / Н. Вирт. – 2-е изд. – СПб. : Невский диалект, 2005. – 352 с.
6. Касьянов, В. Н. Графы в программировании: обработка, визуализация и применение [Текст] / В. Н. Касьянов, В. А. Евстигнеев. – СПб. : БХВ-Петербург, 2003. – 1104 с.
7. Кнут, Д. Э. Искусство программирования для ЭВМ. Т. 1 : Основные алгоритмы [Текст] : пер. с англ. / Д. Э. Кнут. – М. : Мир, 1976. – 736 с.
8. Кнут, Д. Э. Искусство программирования для ЭВМ. Т. 3 : Сортировка и поиск [Текст] : пер. с англ. / Д. Э. Кнут. – М. : Мир, 1978. – 846 с.
9. Кондратьева, С. Д. Введение в структуры данных : лекции и упражнения по курсу [Текст] / С. Д. Кондратьева. – М. : Изд-во МГТУ им. Н. Э. Баумана, 2000. – 376 с.
10. Кубенский, А. А. Структуры и алгоритмы обработки данных: объектно-ориентированный подход и реализация на C++ [Текст] : учеб. пособие / А. А. Кубенский. – СПб. : БХВ-Петербург, 2004. – 464 с.
11. Лэнгсам, Й. Структуры данных для персональных ЭВМ [Текст] : пер. с англ. / Й. Лэнгсам, М. Огенстайн, А. Тененбаум. – М. : Мир, 1989. – 567с.

12. Мейн, М. Структуры данных и другие объекты в С++ [Текст] : пер. с англ. / М. Мейн, У. Савитч. – 2-е изд. – М. : Вильямс, 2002. – 832 с.
13. Подбельский, В. В. Язык Си++ [Текст] : учеб. пособие / В. В. Подбельский. – 5-е изд. – М. : Финансы и статистика, 2004. – 560 с.
14. Седжвик, Р. Фундаментальные алгоритмы на С. Ч. 1 – 4 : Анализ. Структуры данных. Сортировка. Поиск [Текст] : пер. с англ. / Р. Седжвик. – Киев : ДиаСофтЮП, 2003. – 672 с.
15. Седжвик, Р. Фундаментальные алгоритмы на С. Ч. 5 : Алгоритмы на графах [Текст] : пер. с англ. / Р. Седжвик. – Киев : ДиаСофтЮП, 2003. – 480 с.
16. Седжвик, Р. Фундаментальные алгоритмы на С++. Ч. 1 – 4 : Анализ. Структуры данных. Сортировка. Поиск [Текст] : пер. с англ. / Р. Седжвик. – Киев : ДиаСофт, 2002. – 688 с.
17. Седжвик, Р. Фундаментальные алгоритмы на С++. Ч. 5 : Алгоритмы на графах [Текст] : пер. с англ. / Р. Седжвик. – Киев : ДиаСофт, 2002. – 496 с.
18. Седжвик, Р. Фундаментальные алгоритмы на Java. Ч. 1 – 4 : Анализ. Структуры данных. Сортировка. Поиск [Текст] : пер. с англ. / Р. Седжвик. – Киев : ДиаСофт, 2002. – 688 с.
19. Сибуя, М. Алгоритмы обработки данных [Текст] / М. Сибуя, Т. Ямамото. – М. : Мир, 1986. – 218 с.
20. Топп, У. Структуры данных в С++ [Текст] : пер. с англ. / У. Топп, У. Форд. – М. : Бином, 2000. – 815 с.
21. Трамбле, Ж. Введение в структуры данных [Текст] : пер. с англ. / Ж. Трамбле, П. Соренсон. – М.: Машиностроение, 1982. – 784 с.
22. Шилдт, Д. Теория и практика С++ [Текст] / Г. Шилдт. – СПб. : ВНУ – Санкт-Петербург, 1996. – 416 с.
23. Хэзфилд, Р. Искусство программирования на С. Фундаментальные алгоритмы, структуры данных и примеры приложений [Текст] : пер. с англ. / Р. Хэзфилд, Л. Кирби. – Киев : ДиаСофт, 2001. – 736 с.
24. Алгоритмы и дискретные структуры [Электронный ресурс] / Национальный открытый университет «ИНТУИТ», 2015. – Режим доступа: http://www.intuit.ru/studies/courses?service=0&option_id=15, свободный. – Загл. с экрана.

25. Алгоритмы, методы, исходники [Электронный ресурс] / Илья Кантор, 2015. – Режим доступа: <http://algotlist.manual.ru/>, свободный. – Загл. с экрана.

26. Вычислительная сложность [Электронный ресурс] / Википедия – свободная энциклопедия, 2015. – Режим доступа https://ru.wikipedia.org/wiki/Вычислительная_сложность, свободный. – Загл. с экрана.

27. Дискретная математика: Алгоритмы [Электронный ресурс] / Санкт-Петербург. гос. ун-т информ. технологий, механики и оптики. Каф. компьютерных технологий, 2011. – Режим доступа: <http://rain.ifmo.ru/cat/view.php/vis>, свободный. – Загл. с экрана.

28. Реализация структур данных на Си [Электронный ресурс] / admin@learnс.info, 2015. – Режим доступа: <http://learnс.info/adt/>, свободный. – Загл. с экрана.

29. Язык программирования C++. Динамические структуры данных [Электронный ресурс] / Курган. гос. ун-т. Каф. информ. технологий. – Режим доступа: http://it.kgsu.ru/C_DIN/, свободный. – Загл. с экрана.

Федеральное государственное образовательное бюджетное
учреждение высшего профессионального образования
“Поволжский государственный университет
телекоммуникаций и информатики”
443010, г. Самара, ул. Льва Толстого 23

Подписано в печать 13.05.15 г. Формат 60 x 84/16
Бумага офсетная №1. Гарнитура Таймс.
Заказ 1003134. Печать оперативная. Усл. печ. л. 11,17 Тираж 100 экз.

Отпечатано в издательстве учебной и научной литературы
Поволжского государственного университета
телекоммуникаций и информатики
443090, г. Самара, Московское шоссе 77, т. (846) 228-00-44