МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
Государственное образовательное учреждение
высшего профессионального образования
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
АЭРОКОСМИЧЕСКОГО ПРИБОРОСТРОЕНИЯ

А. А. Ключарев, В. А. Матьяш, С. В. Щекин

СТРУКТУРЫ И АЛГОРИТМЫ ОБРАБОТКИ ДАННЫХ

Учебное пособие

Санкт-Петербург
2004

УДК 681.3.01
ББК 32.98
К52

Ключарев А. А., Матьяш В. А., Щекин С. В.

К52 Структуры и алгоритмы обработки данных: Учеб. пособие/ СПбГУАП. СПб., 2003. 172 с.: ил. ISBN 5-8088-0120-6

Приводятся описания различных форм организации данных в программах, методов обработки данных в различных классах задач и осуществляется их сравнительный анализ.

Учебное пособие предназначено для изучения теоретического материала дисциплин «Структуры и алгоритмы и обработки данных» и «Структуры и алгоритмы компьютерной обработки данных» студентами различных форм обучения, проходящих подготовку по специальностям 220400 и 351500 и полностью соответствует требованиям Государственых образовательных стандартов по этим специальностям.

Рецензенты:
кафедра математики и информатики Санкт-Петербургского ин-та Государственной противопожарной службы МЧС России;
доктор военных наук профессор *В. Ф. Волков*
(нач-к каф. автоматизированных систем управления войсками Военно-космической акад. им. А. Ф. Можайского)

Утверждено
редакционно-издательским советом университета
в качестве учебного пособия

ISBN 5-8088-0120-6

ПРЕДИСЛОВИЕ

В учебном пособии описаны структуры данных и алгоритмы, которые являются основой современного компьютерного программирования. Знание этих структур и алгоритмов позволяет осуществлять выбор наиболее оптимальных способов решения задач, возникающих при создании программного обеспечения различного назначения.

Учебное пособие состоит из трех разделов.

В первом разделе рассматриваются основные понятия алгоритмов и структур данных, а также основные подходы к анализу их сложности.

Во втором разделе приводятся описания различных структур данных и основных операций над ними. Рассмотрены элементарные типы данных, линейные и нелинейные структуры, а также файлы.

Третий раздел посвящен основным алгоритмам обработки рассмотренных ранее структур данных и анализу сложности этих алгоритмов. Приводятся различные алгоритмы поиска, сортировки, сжатия данных и алгоритмы на графах, а также обсуждаются методы разработки алгоритмов.

Материал учебного пособия базируется на следующих дисциплинах: «Программирование на языках высокого уровня», «Математическая логика и теория алгоритмов», «Дискретная математика», «Математическое обеспечение программных систем».

Понятия алгоритма и структуры данных

Алгоритм – это точное предписание, определяющее вычислительный процесс, ведущий от варьируемых начальных данных к искомому результату.

ЭВМ в настоящее время приходится не только считывать и выполнять определенные алгоритмы, но и хранить значительные объемы информации, к которой нужно быстро обращаться. Эта информация в некотором смысле представляет собой абстракцию тогог или иного фрагмента реального мира и состоит из определенного множества данных, относящихся к какой-либо проблеме.

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

Под *структурой данных* в общем случае понимают множество элементов данных и множество связей между ними. Такое определение охватывает все возможные подходы к структуризации данных, но в каждой конкретной задаче используются те или иные его аспекты. Поэтому вводится дополнительная классификация структур данных, которая соответствует различным аспектам их рассмотрения. Прежде чем приступать к изучению конкретных структур данных, дадим их общую классификацию по нескольким признакам (рис. 1).

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

Рассмотрение структуры данных без учета ее представления в машинной памяти называется а б с т р а к т н о й или л о г и ч е с к о й структурой. В общем случае между логической и соответствующей ей физической структурами существует различие, степень которого зависит от самой структуры и особенностей той среды, в которой она должна быть отражена. Вследствие этого различия существуют процедуры, осуществляющие отображение логической структуры в физическую и, наоборот, физической структуры в логическую. Эти процедуры обеспечивают, кроме того, доступ к физическим структурам и выполнение над ними различных операций, причем каждая операция рассматривается применительно к логической или физической структуре данных. Кроме того, в зависимости от размещения физических структур, а соответственно, и доступа к ним, различают внутренние (находятся в оперативной памяти) и внешние (на внешних устройствах) структуры данных.

Различаются элементарные (простые, базовые, примитивные) структуры данных и составные (интегрированные, композитные, сложные). Э л е м е н т а р н ы м и называются такие структуры данных, которые не могут быть расчленены на составные части, большие, чем биты. С точки зрения физической структуры важным является то обстоятельство, что в конкретной машинной архитектуре, в конкретной системе программирования всегда можно заранее сказать, каков будет размер элементарного данного и каково его размещение в памяти. С логической точки зрения элементарные данные являются неделимыми единицами.

С о с т а в н ы м и называются такие структуры данных, составными частями которых являются другие структуры данных – элементарные или в свою очередь составные. Составные структуры данных конструируются программистом с использованием средств интеграции данных, предоставляемых языками программирования.

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

Весьма важный признак структуры данных – ее изменчивость, т. е. изменение числа элементов и/или связей между составными частями структуры. В определении изменчивости структуры не отражен факт изменения значений элементов данных, поскольку в этом случае все структуры данных имели бы свойство изменчивости. По

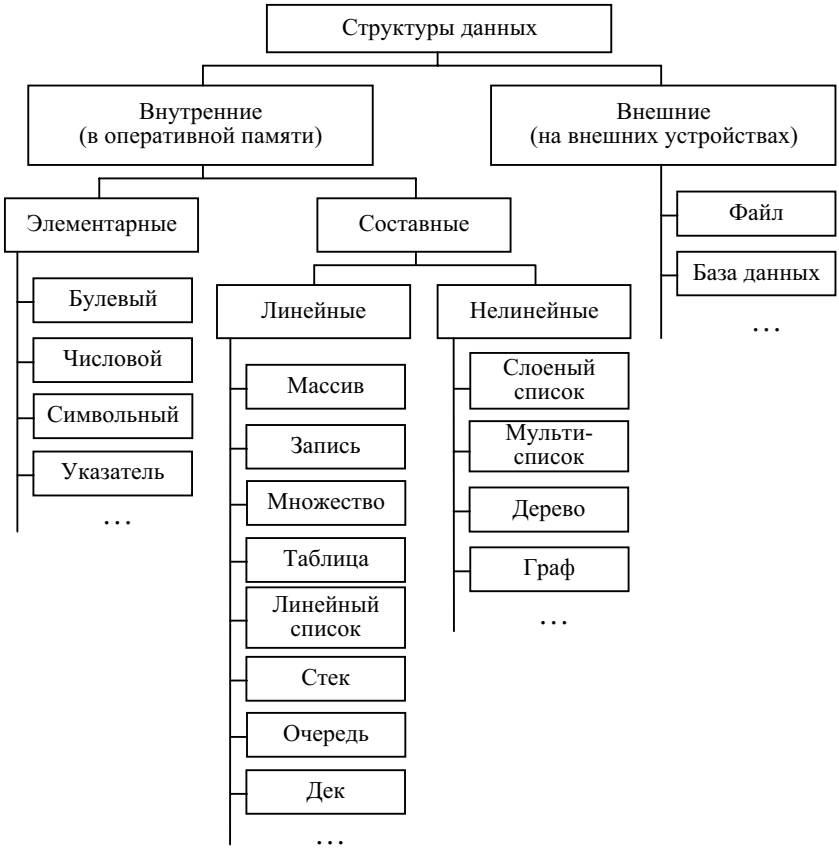


Рис. 1. Классификация структур данных

признаку изменчивости различают структуры статические и динамические.

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

Информация по каждому типу однозначно определяет:

– структуру хранения данных указанного типа, т. е. выделение памяти, представление данных в ней и метод доступа к данным;

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

– набор допустимых операций, которые применимы к объекту описываемого типа.

В последующих разделах рассматриваются структуры данных и соответствующие им типы данных. Базы данных детально изучаются в рамках отдельных дисциплин, и здесь рассматриваться не будут.

При описании элементарных типов и при конструировании составных типов использовался в основном на язык Паскаль. Этот язык используется и во всех примерах, поскольку он был создан специально для иллюстрирования структур данных и алгоритмов и традиционно используется для этих целей. В любом другом процедурном языке программирования высокого уровня (Си, Фортран и т. д.) без труда можно найти аналогичные средства.

Анализ сложности и эффективности алгоритмов и структур данных

В процессе решения прикладных задач выбор подходящего алгоритма вызывает определенные трудности. Алгоритм должен удовлетворять следующим противоречащим друг другу требованиям:

1) быть простым для понимания, перевода в программный код и отладки;

2) эффективно использовать вычислительные ресурсы и выполняться по возможности быстро.

Если разрабатываемая программа, реализующая некоторый алгоритм, должна выполняться только несколько раз, то первое требование наиболее важно. В этом случае стоимость программы оптимизируется по стоимости написания (а не выполнения) программы. Если решение задачи требует значительных вычислительных затрат, то стоимость выполнения программы может превысить стоимость написания программы, особенно если программа выполняется многократно. Поэтому более предпочтительным может стать сложный комплексный алгоритм (в надежде, что результирующая программа будет выполняться существенно быстрее). Таким образом, прежде чем принимать решение об использовании того или иного алгоритма, необходимо оценить сложность и эффективность этого алгоритма.

Сложность алгоритма – это величина, отражающая порядок величины требуемого ресурса (времени или дополнительной памяти) в зависимости от размерности задачи.

Таким образом, будем различать временную $T(n)$ и пространственную $V(n)$ сложности алгоритма. При рассмотрении оценок сложности

будем использовать только временную сложность. Пространственная сложность оценивается аналогично.

Самый простой способ оценки – экспериментальный, т. е. запрограммировать алгоритм и выполнить полученную программу на нескольких задачах, оценивая время выполнения программы. Однако этот способ имеет ряд недостатков. Во-первых, экспериментальное программирование – это, возможно, дорогостоящий процесс. Во-вторых, необходимо учитывать, что на время выполнения программ влияют следующие факторы:

1) временная сложность алгоритма программы;

2) качество скомпилированного кода исполняемой программы;

3) машинные инструкции, используемые для выполнения программы.

Наличие второго и третьего факторов не позволяют применять типовые единицы измерения временной сложности алгоритма (секунды, миллисекунды и т.п.), так как можно получить самые различные оценки для одного и того же алгоритма, если использовать разных программистов (которые программируют алгоритм каждый по-своему), разные компиляторы и разные вычислительные машины.

Существует метод, позволяющий теоретически оценить время выполнения алгоритма, который и рассмотрим далее.

Часто, временная сложность алгоритма зависит от количества входных данных. Обычно говорят, что временная сложность алгоритма имеет порядок $T(n)$ от входных данных размера n. Точно определить величину $T(n)$ на практике представляется довольно трудно. Поэтому прибегают к асимптотическим отношениям с использованием O-символики.

Например, если число тактов (действий), необходимое для работы алгоритма, выражается как $11n^2 + 19n \cdot \log n + 3n + 4$, то это алгоритм, для которого $T(n)$ имеет порядок $O(n^2)$. Фактически, из всех слагаемых оставляется только то, которое вносит наибольший вклад при больших n (в этом случае остальными слагаемыми можно пренебречь), и игнорируется коэффициент перед ним.

Когда используют обозначение $O()$, имеют в виду не точное время исполнения, а только его предел сверху, причем с точностью до постоянного множителя. Когда говорят, например, что алгоритму требуется время порядка $O(n^2)$, имеют в виду, что время исполнения задачи растет не быстрее, чем квадрат количества элементов.

Для примера приведем числа, иллюстрирующие скорость роста для нескольких функций, которые часто используются при оценке временной сложности алгоритмов (см. табл. 1).

Таблица 1

n	$\log n$	$n\log n$	n^2
1	0	0	1
16	4	64	256
256	8	2 048	65 536
4 096	12	49 152	16 777 216
65 536	16	1 048 565	4 294 967 296
1 048 476	20	20 969 520	1 099 301 922 576
16 775 616	24	402 614 784	281 421 292 179 456

Если считать, что числа соответствуют микросекундам, то для задачи с 1048476 элементами алгоритму со временем работы $T(\log n)$ потребуется 20 микросекунд, а алгоритму со временем работы $T(n^2)$ – более 12 дней.

Если операция выполняется за фиксированное число шагов, не зависящее от количества данных, то принято писать $O(1)$.

Следует обратить внимание, что основание логарифма здесь не пишется. Причина этого весьма проста. Пусть есть $O(\log_2 n)$. Но $\log_2 n = \log_3 n/\log_3 2$, а $\log_3 2$, как и любую константу, символ $O()$ не учитывает. Таким образом, $O(\log_2 n) = O(\log_3 n)$. К любому основанию можно перейти аналогично, а значит, и писать его не имеет смысла.

Практически время выполнения алгоритма зависит не только от количества входных данных, но и от их значений, например, время работы некоторых алгоритмов сортировки значительно сокращается, если первоначально данные частично упорядочены, тогда как другие методы оказываются нечувствительными к этому свойству. Чтобы учитывать этот факт, полностью сохраняя при этом возможность анализировать алгоритмы независимо от данных, различают:

– максимальную сложность $T_{\max}(n)$, или сложность наиболее неблагоприятного случая, когда алгоритм работает дольше всего;

– среднюю сложность $T_{\text{mid}}(n)$ – сложность алгоритма в среднем;

– минимальную сложность $T_{\min}(n)$ – сложность в наиболее благоприятном случае, когда алгоритм справляется быстрее всего.

Теоретическая оценка временной сложности алгоритма осуществляется с использованием следующих базовых принципов:

1) время выполнения операций присваивания, чтения, записи обычно имеют порядок $O(1)$. Исключением являются операторы присваивания, в которых операнды представляют собой массивы или вызовы функций;

2) время выполнения последовательности операций совпадает с наибольшим временем выполнения операции в данной последовательности (правило сумм: если $T_1(n)$ имеет порядок $O(f(n))$, а $T_2(n)$ – порядок $O(g(n))$, то $T_1(n) + T_2(n)$ имеет порядок $O(\max(f(n), g(n)))$);

3) время выполнения конструкции ветвления (`if-then-else`) состоит из времени вычисления логического выражения (обычно имеет порядок $O(1)$) и наибольшего из времени, необходимого для выполнения операций, исполняемых при истинном значении логического выражения и при ложном значении логического выражения;

4) время выполнения цикла состоит из времени вычисления условия прекращения цикла (обычно имеет порядок $O(1)$) и произведения количества выполненных итераций цикла на наибольшее возможное время выполнения операций тела цикла.

5) время выполнения операции вызова процедур определяется как время выполнения вызываемой процедуры;

6) при наличии в алгоритме операции безусловного перехода, необходимо учитывать изменения последовательности операций, осуществляемых с использованием этих операции безусловного перехода.

1. Структуры данных

1.1. Элементарные данные

Данные элементарных типов представляют собой единое и неделимое целое. В каждый момент времени они могут принимать только одно значение. Набор элементарных типов в разных языках программирования несколько различаются, однако есть типы, которые поддерживаются практически везде. Рассмотрим их на примере языка Паскаль:

```
var
  i, j: integer;
  x: real;
  s: char;
  b: boolean;
  p: pointer;
```

Здесь объявлены переменные *i*, *j* целочисленного типа, *x* – вещественного, *s* – символьного, *b* – логического типа и *p* – указатель.

К данным элементарных типов можно обращаться по их именам:

```
i := 46;
x := 3.14;
s := 'A';
b := true;
```

1.1.1. Данные числовых типов

1.1.1.1. Данные целочисленного типа

С помощью целых чисел может быть представлено количество объектов, являющихся дискретными по своей природе (т. е. счетное число объектов).

Диапазон возможных значений целых типов зависит от их внутреннего представления, которое может занимать 1, 2 или 4 байта. В табл. 2 приводится перечень целых типов, размер памяти для их внутреннего представления в битах, диапазон возможных значений.

Таблица 2

Тип	Диапазон значений	Машинное представление
shortint	–128…127	8 бит со знаком
integer	–32768…32767	16 бит со знаком
longint	–2147483648…2147483647	32 бита со знаком
byte	0...255	8 бит без знака
word	0…65535	16 бит без знака
comp	$-2^{63}+1...2^{63}-1$	64 бита со знаком

1.1.1.2. Данные вещественного типа

В отличие от целочисленных типов, значения которых всегда представляются в памяти ЭВМ абсолютно точно, значение вещественных типов определяет число лишь с некоторой конечной точностью, зависящей от внутреннего формата вещественного числа.

Суммарное количество байтов, диапазоны допустимых значений чисел вещественных типов, а также количество значащих цифр после запятой в представлении чисел приведены в табл. 3.

Таблица 3

Тип	Диапазон значений	Значащие цифры	Размер в байтах
real	$2{,}9\cdot10^{-39}...1{,}7\cdot10^{38}$	11–12	6
single	$1{,}4\cdot10^{-45}…3{,}4\cdot10^{38}$	7–8	4
double	$4{,}9\cdot10^{-324}…1{,}8\cdot10^{308}$	15–16	8
extended	$3{,}1\cdot10^{-4944}…1{,}2\cdot10^{4932}$	19–20	10

1.1.1.3. Операции над данными числовых типов

Над числовыми типами, как и над всеми другими возможны прежде всего, четыре основных операции: создание, уничтожение, выбор, обновление. Специфическими операциями над числовыми типами являются арифметические операции: сложение, вычитание, умножение, деление. Операция возведения в степень в некоторых языках также является базовой и обозначается специальным символом или комбинацией символов, в других – выполняется встроенными функциями.

Следует обратить внимание на то, что операция деления по-разному выполняется для целых и вещественных чисел. При делении целых чисел дробная часть результата отбрасывается, как бы близка к 1 она ни была. В

связи с этим в языке Паскаль имеются даже разные обозначения для деления вещественных и целых чисел – операции «/» и «div» соответственно. В других языках оба вида деления обозначаются одинаково, а тип деления определяется типом операндов. Для целых операндов возможна еще одна операция – остаток от деления (в языке Паскаль операция «mod»).

Еще одна группа операций над числовыми типами – операции сравнения >, <, ≥, ≤, =, <>. Существенно, что хотя операндами этих операций являются данные числовых типов, результат их имеет логический тип – «истина» или «ложь». Говоря об операциях сравнения, следует обратить внимание на особенность выполнения сравнений на равенство/неравенство вещественных чисел. Поскольку эти числа представляются в памяти с некоторой (не абсолютной) точностью, сравнения их не всегда могут быть абсолютно достоверны.

Поскольку одни и те же операции допустимы для разных числовых типов, возникает проблема арифметических выражений со смешением типов. В реальных задачах выражения со смешанными типами встречаются довольно часто. Поэтому большинство языков допускает выражения, операнды которых имеют разные числовые типы, но обрабатываются такие выражения в разных языках по-разному. В одних языках все операнды выражения приводятся к одному типу, а именно к типу той переменной, в которую будет записан результат, а затем уже выражение вычисляется. В других (например, язык Си) преобразование типов выполняется в процессе вычисления выражения, при выполнении каждой отдельной операции, без учета других операций; каждая операция вычисляется с точностью самого точного участвующего в ней операнда.

1.1.2. Данные символьного типа

Значением символьного типа char являются символы из некоторого предопределенного множества. В качестве примеров этих множеств можно назвать ASCII (American Standard Code for Information Interchange). Это множество состоит из 256 разных символов, упорядоченных определенным образом, и содержит символы заглавных и строчных букв, цифр и других символов, включая специальные управляющие символы.

Значение символьного типа char занимает в памяти 1 байт. Код от 0 до 255 в этом байте задает один из 256 возможных символов ASCII таблицы.

ASCII включает в себя буквенные символы только латинского алфавита. Символы национальных алфавитов занимают «свободные места»

в таблице кодов и, таким образом, одна таблица может поддерживать только один национальный алфавит. Этот недостаток преодолен во множестве UNICODE. В этом множестве каждый символ кодируется двумя байтами, что обеспечивает более 2^{16} возможных кодовых комбинаций и дает возможность иметь единую таблицу кодов, включающую в себя все национальные алфавиты. UNICODE, безусловно, является перспективным, однако, повсеместный переход к двухбайтным кодам символов может вызвать необходимость переделки значительной части существующего программного обеспечения.

Специфические операции над символьными типами – только операции сравнения. При сравнении коды символов рассматриваются как целые числа без знака. Кодовые таблицы строятся так, что результаты сравнения подчиняются лексикографическим правилам: символы, занимающие в алфавите места с меньшими номерами, имеют меньшие коды, чем символы, занимающие места с большими номерами.

1.1.3. Данные логического типа

Значениями логического типа может быть одна из предварительно объявленных констант `false` (ложь) или `true` (истина).

Данные логического типа занимают один байт памяти. При этом значению `false` соответствует нулевое значение байта, а значению `true` – любое ненулевое значение байта.

Над логическими типами возможны операции булевой алгебры – НЕ (`not`), ИЛИ (`or`), И (`and`), ИСКЛЮЧАЮЩЕЕ ИЛИ (`xor`). Последняя операция реализована для логического типа не во всех языках.

Кроме того, следует помнить, что результаты логического типа получаются при сравнении данных любых типов.

Интересно, что в языке Си данные логического типа отсутствуют, их функции выполняют данные числовых типов, чаще всего типа `int`. В логических выражениях операнд любого числового типа, имеющий нулевое значение, рассматривается как «ложь», а ненулевое – как «истина». Результатами выражений логического типа являются целые числа 0 (ложь) или 1 (истина).

1.1.4. Данные типа указатель

Тип указателя представляет собой адрес ячейки памяти. Физическое представление адреса существенно зависит от аппаратной архитектуры вычислительной системы.

В языках программирования высокого уровня определена специальная константа nil, которая означает пустой указатель или указатель, не содержащий какой-либо конкретный адрес.

При решении прикладных задач с использованием языков высокого уровня наиболее частые случаи, когда могут понадобиться указатели, следующие:

1) при необходимости представить одну и ту же область памяти, а следовательно, одни и те же физические данные как данные разной логической структуры. В этом случае вводятся два или более указателей, которые содержат адрес одной и той же области памяти, но имеют разный тип. Обращаясь к этой области памяти по тому или иному указателю, можно обрабатывать ее содержимое как данные того или иного типа;

2) при работе с динамическими структурами данных. Память под такие структуры выделяется в ходе выполнения программы, стандартные процедуры/функции выделения памяти возвращают адрес выделенной области памяти – указатель на нее. К содержимому динамически выделенной области памяти можно обращаться только через такой указатель.

В языках высокого уровня указатели могут быть типизированными и нетипизированными. При объявлении типизированного указателя определяется и тип данного в памяти, адресуемого этим указателем. Приведем пример объявления в языке Паскаль различных типизированных указателей:

```
var
  ipt: ^integer;
  cpt: ^char;
```

Здесь переменная ipt содержит адрес области памяти, в которой хранится целое число, а cpt – адрес области памяти, в которой хранится символ. Хотя физическая структура адреса не зависит от типа и значения данных, хранящихся по этому адресу, считается, что указатели ipt и cpt имеют разный тип.

Нетипизированный указатель служит для представления адреса, по которому содержатся данные неизвестного типа. В Паскале это тип pointer. Работа с нетипизированными указателями существенно ограничена, они могут использоваться только для сохранения адреса, а обращение по адресу, задаваемому нетипизированным указателем, невозможно.

Основными операциями, в которых участвуют указатели, являются присваивание, получение адреса, выборка.

Присваивание является двухместной операцией, оба операнда которой указатели. Как и для других типов, операция присваивания копирует значение одного указателя в другой, в результате оба указателя будут содержать один и тот же адрес памяти. Если оба указателя, участвующие в операции присваивания типизированные, то оба они должны указывать на данные одного и того же типа.

Операция получения адреса – одноместная, ее операнд может иметь любой тип. Результатом является типизированный (в соответствии с типом операнда) указатель, содержащий адрес операнда.

Операция выборки – одноместная, ее операндом является типизированный указатель. Результатом этой операции являются данные, выбранные из памяти по адресу, заданному операндом. Тип результата определяется типом указателя.

Перечисленных операций достаточно для решения задач прикладного программирования, поэтому набор операций над указателями, допустимых в языке Паскаль, этим и ограничивается. Системное программирование требует более гибкой работы с адресами, поэтому, например, в языке Си доступны также операции адресной арифметики.

1.2. Линейные структуры данных

Рассмотрим *статические структуры данных*: массивы, записи, множества. Цель описания типа данных и определения некоторых переменных, относящихся к статическим типам, состоит в том, чтобы зафиксировать диапазон значений, присваиваемых этим переменным, и соответственно размер выделяемой для них памяти. Поэтому такие переменные и называются статическими.

1.2.1. Массив

Массив – это поименованная совокупность однотипных элементов, упорядоченных по индексам, определяющих положение элемента в массиве.

Следующее объявление задает имя для массива, тип для индекса и тип элементов массива:

```
имя: array[ТипИндекса] of ТипЭлемента;
```

Тип индекса, в общем случае, может быть любым порядковым, но некоторые языки программирования поддерживают в качестве индексов массивов только последовательности целых чисел.

Количество используемых индексов определяет размерность массива. Массив может быть одномерным (вектор), двумерным (матрица), трехмерным (куб) и т. д.:

```
var
  Vector: array [1..100] of integer;
  Matrix: array [1..100, 1..100] of integer;
  Cube:   array [1..100, 1..100, 1..100] of integer;
```

В Паскале определены такие операции над массивами в целом, как сравнение на равенство и неравенство массивов, а также операция присвоения для массивов с одинаковым типом индексов и одинаковым типом элементов. Доступ к массивам в этих операциях осуществляется через имя массива без указания индексов. В некоторых языках программирования определен более мощный перечень операции, где в качестве операндов выступают целые массивы, это так называемые векторные вычисления.

Можно также выполнять операции над отдельными элементами массива. Перечень таких операций определяется типом элементов. Доступ к отдельным элементам массива осуществляется через имя массива и индекс (индексы) элемента:

```
Cube[0,0,10] := 25;
Matrix[10,30] := Cube[0,0,10] + 1;
```

В памяти ЭВМ элементы массива обычно располагаются непрерывно, в соседних ячейках. Размер памяти, занимаемой массивом, есть суммарный размер элементов массива.

1.2.2. Строка

Строка – это последовательность символов (элементов символьного типа).

В Паскале количество символов в строке (длина строки) может динамически меняться от 0 до 255.

Рассмотрим пример описания строк:

```
var
  TTxt: string;
  TWrd: string[10];
```

Здесь описаны строка TTxt, максимальная длина которой 255 символов (по умолчанию) и строка TWrd, максимальная длина которой ограничена 10 символами. Каждый символ строки имеет свой индекс, принимающий значение от 1 до заданной длины строки. Следует обратить внимание, что существует элемент строки с индексом 0, который не доступен с использованием индекса, и содержит текущее количество символов в строке. Доступ к этому специфическому элементу можно получить только с помощью специальных функций языка.

Благодаря индексам, строки очень похожи на одномерные массивы символов, и доступ к отдельным элементам строки можно получать с использованием этих индексов, выполняя операции, определенные для символьного типа данных. Так же как и для массивов, определена операция присвоения строк в целом.

Однако есть ряд отличий. Операций сравнения строк больше, чем аналогичных операций для массивов: <, >, ≥, ≤, =, <>. Существует операция сцепления (конкатенации) строк «+».

В памяти ЭВМ символы строки располагаются непрерывно, в соседних ячейках. Размер памяти, занимаемой строкой, есть суммарный размер элементов массива (включая элемент, содержащий длину строки).

1.2.3. Запись

Запись – это агрегат, составляющие которого (поля) имеют имя и могут быть различного типа.

Рассмотрим пример простейшей записи:

```
type
  TPerson = record
    Name:    string;
    Address: string;
    Index:   longint;
  end;
var
  Person1: TPerson;
```

Запись описанного типа объединяет три поля. Первые два из них символьного типа, а третье – целочисленного.

В Паскале определена операция присваивания для записей в целом (записи должны быть одного типа). Доступ к записи осуществляется через ее имя.

Можно также выполнять операции над отдельным полем записи. Перечень таких операций определяется типом поля.

Доступ к полям отдельной записи осуществляется через имя записи и имя поля:

```
Person1.Index := 190000;
Person1.Name  := 'Иванов';
Person1.Adress := 'Санкт-Петербург, ул. Б.Морская, д.67';
```

В памяти ЭВМ поля записи обычно располагаются непрерывно, в соседних ячейках. Размер памяти, занимаемой записью, есть суммарный размер полей, составляющих запись.

1.2.4. Множество

Наряду с массивами и записями существует еще один структурированный тип – множество. Этот тип используется не так часто, хотя его применение в некоторых случаях является вполне оправданным.

Множество – совокупность каких-либо однородных элементов, объединенных общим признаком и представляемых как единое целое.

Тип множество соответствует математическому понятию множества в смысле операций, которые допускаются над структурами такого типа. Множество допускает операции объединения множеств «+», пересечения множеств «*», разности множеств «–» и проверки элемента на принадлежность к множеству «in». Множества, так же как и массивы, объединяют однотипные элементы. Поэтому в описании множества обязательно должен быть указан тип его элементов:

```
var
  RGB, YIQ, CMY: set of char;
```

Здесь приведено описание трех множеств, элементами которых являются символы. Кроме того, определены операции сравнения множеств: ≥, ≤, =, <>. В отличие от массивов и записей здесь отсутствует возможность обращения к отдельным элементам. Операции выполняются по отношению ко всей совокупности элементов множества:

```
CMY := ['M', 'C', 'Y'];
RGB := ['R', 'G', 'B'];
YIQ := ['Y', 'Q', 'I'];
Writeln('Пересечение цветовых систем RGB и CMY ', RGB*CMY);
Writeln('Пересечение цветовых систем YIQ и CMY ', YIQ*CMY);
```

В Паскале в качестве типов элементов множества могут использоваться типы, максимальное количество значений которых не превышает 256. В памяти ЭВМ элементы множества обычно располагаются непрерывно, в соседних ячейках.

1.2.5. Таблица

Таблица представляет собой одномерный массив (вектор), элементами которого являются записи.

Отдельная запись массива называется строкой таблицы. Чаще всего используется простая запись, т. е. поля – элементарные данные. Совокупность одноименных полей всех строк называется столбцом, а конкретное поле отдельной строки – ячейкой:

```
type
  TPerson = record
    Name:    string;
    Address: string;
    Index:   longint;
  end;
  TTable = array[1..1000] of TPerson;
var
  PersonList: TTable;
```

Характерной особенностью таблиц является то, что доступ к элементам таблицы производится не по индексу, а по ключу, т. е. по значению одного из полей записи.

Ключ таблицы (основной, первичный) – поле, значение которого может быть использовано для однозначной идентификации каждой записи таблицы. Ключ таблицы может быть составным – образовываться не одним, а несколькими полями данной таблицы.

Вторичный ключ – поле таблицы с несколькими ключами, не обеспечивающий (в отличие от первичного ключа) однозначной идентификации записей таблицы. В этот ключ могут входить все поля таблицы за исключением полей, составляющих первичный ключ.

Если последовательность записей упорядочена относительно какого-либо столбца (поля), то такая таблица называется упорядоченной, иначе – таблица неупорядоченная.

Основной операцией при работе с таблицами является операция доступа к записи по ключу. Она реализуется процедурой поиска. Алго-

ритмы поиска рассматриваются в п. 2.3. Получив доступ к конкретной записи (строке таблицы), с ней можно работать как с записью в целом, так и с отдельными полями (ячейками). Перечень операций над отдельной ячейкой определяется типом ячейки:

```
PersonList[1].Index  := 190000;
PersonList[1].Name   := 'Иванов';
PersonList[1].Adress := 'Санкт-Петербург, ул. Б.Морская, д.67';
```

В памяти ЭВМ ячейки таблицы обычно располагаются построчно, непрерывно, в соседних ячейках. Размер памяти, занимаемой таблицей, есть суммарный размер ячеек.

1.2.6. Линейные списки

Список – это структура данных, представляющая собой логически связанную последовательность элементов списка.

Иногда бывают ситуации, когда невозможно на этапе разработки алгоритма определить диапазон значений переменной. В этом случае применяют динамические структуры данных.

Динамическая структура данных – это структура данных, определяющие характеристики которой могут изменяться на протяжении ее существования.

Обеспечиваемая такими структурами способность к адаптации часто достигается меньшей эффективностью доступа к их элементам.

Динамические структуры данных отличаются от статических двумя основными свойствами:

1) в них нельзя обеспечить хранение в заголовке всей информации о структуре, поскольку каждый элемент должен содержать информацию, логически связывающую его с другими элементами структуры;

2) для них зачастую не удобно использовать единый массив смежных элементов памяти, поэтому необходимо предусматривать ту или иную схему динамического управления памятью.

Для обращения к динамическим данным применяют указатели, рассмотренные выше.

Созданием динамических данных должна заниматься сама программа во время своего исполнения. В языке программирования Паскаль для этого существует специальная процедура:

```
New(Current);
```

После выполнения данной процедуры в оперативной памяти ЭВМ создается динамическая переменная, тип которой определяется типом указателя `Current`.

После использования динамического данного и при отсутствии необходимости его дальнейшего использования необходимо освободить оперативную память ЭВМ от этого данного с помощью соответствующей процедуры:

```
Dispose(Current);
```

Наиболее простой способ организовать структуру данных, состоящее из некоторого множества элементов – это организовать линейный список. При такой организации элементы некоторого типа образуют цепочку. Для связывания элементов в списке используют систему указателей, и в зависимости от их количества в элементах различают однонаправленные и двунаправленные линейные списки.

1.2.6.1. Линейный однонаправленный список

В этом списке любой элемент имеет один указатель, который указывает на следующий элемент в списке или является пустым указателем у последнего элемента (рис. 2).

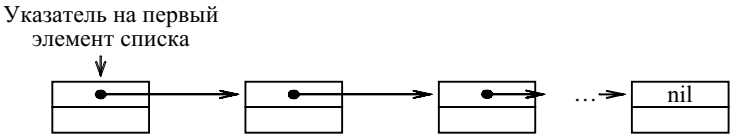


Рис. 2. Линейный однонаправленный список

Основные операции, осуществляемые с линейным однонаправленным списком:

– вставка элемента;
– просмотр;
– поиск;
– удаление элемента.

Следует обратить особое внимание на то, что при выполнении любых операций с линейным однонаправленным списком необходимо обеспечивать позиционирование какого-либо указателя на первый элемент. В противном случае часть или весь список будет недоступен.

Для описания алгоритмов этих основных операций используем следующие объявления:

```
type
  PElement = ^TypeElement; {указатель на тип элемента}
  TypeElement = record     {тип элемента списка}
    Data: TypeData;        {поле данных элемента}
    Next: PElement;        {поле указателя на следующий элемент}
  end;
var
  ptrHead: PElement;    {указатель на первый элемент списка}
  ptrCurrent: PElement; {указатель на текущий элемент}
```

Вставка первого и последующих элементов списка отличаются друг от друга. Поэтому приводится два алгоритма вставки, оформленных в виде процедур языка Паскаль: `InsFirst_LineSingleList` и `Ins_LineSingleList`. В качестве входных параметров передаются данное для заполнения создаваемого элемента, указатель на начало списка и указатель на текущий элемент в списке (при необходимости). Выходными параметрами процедур является указатель на начало списка (который возможно изменится) и указатель текущего элемента, который показывает на вновь созданный элемент (при вставке первого элемента указателем на него будет указатель на заголовок списка).

Для добавления элемента в конец списка используется процедура вставки последующего элемента для случая, когда текущий элемент является последним в списке:

```
procedure Ins_LineSingleList(DataElem: TypeData;
                             var ptrHead, ptrCurrent: PElement);
  {Вставка непервого элемента в линейный однонаправленный список}
  {справа от элемента, на который указывает ptrCurrent}
var
  ptrAddition: PElement;   {вспомогательный указатель}
begin
  New(ptrAddition);
  ptrAddition^.Data := DataElem;
  if ptrHead = nil then begin   {список пуст}
    {создаем первый элемент списка}
    ptrAddition^.Next := nil;
    ptrHead := ptrAddition;
  end else begin   {список не пуст}
    {вставляем элемент списка справа от элемента,}
    {на который указывает ptrCurrent}
    ptrAddition^.Next := ptrCurrent^.Next;
```

```
    ptrCurrent^.Next := ptrAddition;
  end;
  ptrCurrent := ptrAddition;
end;
procedure InsFirst_LineSingleList(DataElem: TypeData;
                                  var ptrHead: PElement);
  {Вставка первого элемента в линейный однонаправленный список}
var
  ptrAddition: PElement;   {вспомогательный указатель}
begin
  New(ptrAddition);
  ptrAddition^.Data := DataElem;
  if ptrHead = nil then begin   {список пуст}
    {создаем первый элемент списка}
    ptrAddition^.Next := nil;
  end else begin   {список не пуст}
    {вставляем новый элемент слева (перед) первым элементом}
    ptrAddition^.Next := ptrHead;
  end;
  ptrHead := ptrAddition;
end;
```

Порядок следования операторов присваивания обеих процедур очень важен. При неправильном переопределении указателей возможен разрыв списка или потери указателя на первый элемент, что приводит к потере доступа к части или всему списку.

Операция просмотра списка заключается в последовательном просмотре всех элементов списка:

```
procedure Scan_LineSingleList(ptrHead: PElement);
  {Просмотр линейного однонаправленного списка}
var
  ptrAddition: PElement;   {вспомогательный указатель}
begin
  ptrAddition := ptrHead;
  while ptrAddition <> nil do begin {пока не конец списка}
    writeln(ptrAddition^.Data);     {Вывод значения элемента}
    ptrAddition := ptrAddition^.Next;
  end;
end;
```

Операция поиска элемента в списке заключается в последовательном просмотре всех элементов списка до тех пор, пока текущий элемент не будет содержать заданное значение или пока не будет достигнут конец списка. В последнем случае фиксируется отсутствие искомого элемента в списке (функция принимает значение `false`). Входными параметрами являются значение, которое должен содержать искомый элемент и указатель на список. В качестве выходного параметра передается указатель, который устанавливается на найденный элемент или остается без изменений, если элемента в списке нет:

```
function Find_LineSingleList(DataElem: TypeData;
                    var ptrHead, ptrCurrent: PElement): boolean;
  {Поиск элемента в линейном однонаправленном списке}
var
  ptrAddition:PElement;  {Дополнительный указатель}
begin
  if (ptrHead <> nil)
  then begin  {Если существует список}
    ptrAddition := ptrHead;
    while (ptrAddition <> nil) and
          (ptrAddition^.Data <> DataElem) do
      {пока не конец списка и не найден элемент}
      ptrAddition := ptrAddition^.Next;
    {Если элемент найден,
     то результатом работы функции является – true}
    if ptrAddition <> nil then begin
      Find_LineSingleList := true;
      ptrCurrent := ptrAddition;
    end else begin
      Find_LineSingleList := false;
    end;
  end else begin
    Find_LineSingleList:= false;
  end;
end;
```

Можно отметить, что алгоритмы просмотра и поиска будут корректно работать без дополнительных проверок и в случае, когда список пуст.

Операция удаления элемента линейного однонаправленного списка осуществляет удаление элемента, на который установлен указатель текущего элемента. После удаления указатель текущего элемента ус-

танавливается на предшествующий элемент списка, или на новое начало списка, если удаляется первый.

Алгоритмы удаления первого и непервого элементов списка отличаются друг от друга. Поэтому в процедуре, реализующую данную операцию, осуществляется проверка, какой элемент удаляется, и далее реализуется соответствующий алгоритм удаления:

```
procedure Del_LineSingleList(var ptrHead,
                             ptrCurrent: PElement);
  {Удаление элемента из линейного однонаправленного списка}
var
  ptrAddition: PElement;   {вспомогательный указатель}
begin
  if ptrCurrent <> nil then begin   {вх.параметр корректен}
    if ptrCurrent = ptrHead then begin   {удаляем первый}
      ptrHead := ptrHead^.Next;
      dispose(ptrCurrent);
      ptrCurrent := ptrHead;
    end else begin                       {удаляем непервый}
        {устанавливаем вспомогательный указатель на элемент,
         предшествующий удаляемому}
        ptrAddition := ptrHead;
        while ptrAddition^.Next <> ptrCurrent do
          ptrAddition := ptrAddition^.Next;
        {непосредственное удаление элемента}
        ptrAddition^.Next := ptrCurrent^.Next;
        dispose(ptrCurrent);
        ptrCurrent := ptrAddition;
    end;
  end;
end;
```

Линейный однонаправленный список имеет только один указатель в элементах. Это позволяет минимизировать расход памяти на организацию такого списка. Одновременно, это позволяет осуществлять переходы между элементами только в одном направлении, что зачастую увеличивает время, затрачиваемое на обработку списка. Например, для перехода к предыдущему элементу необходимо осуществить просмотр списка с начала до элемента, указатель которого установлен на текущий элемент.

Для ускорения подобных операций целесообразно применять переходы между элементами списка в обоих направлениях. Это реализуется с помощью линейных двунаправленных списков.

1.2.6.2. Линейный двунаправленный список

В этом линейном списке любой элемент имеет два указателя, один из которых указывает на следующий элемент в списке или является пустым указателем у последнего элемента, а второй – на предыдущий элемент в списке или является пустым указателем у первого элемента (рис. 3).

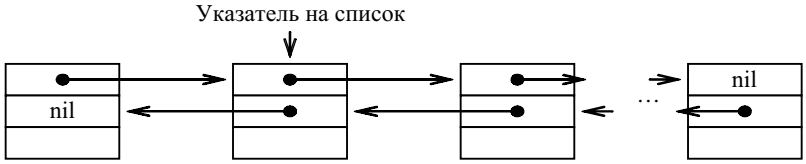


Рис. 3. Линейный двунаправленный список

Основные операции, осуществляемые с линейным двунаправленным списком те же, что и с однонаправленным линейным списком:

– вставка элемента;
– просмотр;
– поиск;
– удаление элемента.

Следует обратить внимание на то, что в отличие от однонаправленного списка здесь нет необходимости обеспечивать позиционирование какого-либо указателя именно на первый элемент списка, так как благодаря двум указателям в элементах можно получить доступ к любому элементу списка из любого другого элемента, осуществляя переходы в прямом или обратном направлении. Однако часто бывает полезно иметь указатель на заголовок списка.

Для описания алгоритмов этих основных операций используем следующие объявления:

```
type
  PElement = ^TypeElement;{указатель на тип элемента}
  TypeElement = record    {тип элемента списка}
    Data: TypeData;       {поле данных элемента}
    Next,                 {поле указателя на следующий элемент}
    Last: PElement;       {поле указателя на предыдущий элемент}
```

```
  end;
var
  ptrHead: PElement;       {указатель на первый элемент списка}
  ptrCurrent: PElement;    {указатель на текущий элемент}
```

Операция вставки реализовывается с помощью двух процедур, аналогичных процедурам вставки для линейного однонаправленного списка: `InsFirst_LineDubleList` и `Ins_LineDubleList`. Однако при вставке последующего элемента придется учитывать особенности добавления элемента в конец списка:

```
procedure Ins_LineDubleList(DataElem: TypeData;
                            var ptrHead, ptrCurrent: PElement);
  {Вставка непервого элемента в линейный двунаправленный список}
  {справа от элемента, на который указывает ptrCurrent}
var
  ptrAddition: PElement;   {вспомогательный указатель}
begin
  New(ptrAddition);
  ptrAddition^.Data := DataElem;
  if ptrHead = nil then begin   {список пуст}
    {создаем первый элемент списка}
    ptrAddition^.Next := nil;
    ptrAddition^.Last := nil;
    ptrHead := ptrAddition;
  end else begin                {список не пуст}
    {вставляем элемент списка справа от элемента,}
    {на который указывает ptrCurrent}
    if ptrCurrent^.Next <> nil then {вставляем не последний}
      ptrCurrent^.Next^.Last := ptrAddition;
    ptrAddition^.Next := ptrCurrent^.Next;
    ptrCurrent^.Next := ptrAddition;
    ptrAddition^.Last := ptrCurrent;
  end;
  ptrCurrent := ptrAddition;
end;
procedure InsFirst_LineDubleList(DataElem: TypeData;
                                 var ptrHead: PElement);
  {Вставка первого элемента в линейный двунаправленный список}
var
  ptrAddition: PElement;   {вспомогательный указатель}
begin
```

```
  New(ptrAddition);
  ptrAddition^.Data := DataElem;
  ptrAddition^.Last := nil;
  if ptrHead = nil then begin   {список пуст}
    {создаем первый элемент списка}
    ptrAddition^.Next := nil;
  end else begin   {список не пуст}
    {вставляем новый элемент слева (перед) первым элементом}
    ptrAddition^.Next := ptrHead;
    ptrHead^.Last := ptrAddition;
  end;
  ptrHead := ptrAddition;
end;
```

Порядок следования операторов присваивания обеих процедур здесь также очень важен. При неправильном переопределении указателей возможен разрыв списка или потери указателя на первый элемент, что приводит к потере доступа к части или всему списку.

Операция просмотра и операция поиска для линейного двунаправленного списка реализуются абсолютно аналогично соответствующим процедурам для линейного однонаправленного списка, и приводить их не будем. Отметим только, что просматривать и искать здесь можно в обоих направлениях.

Операция удаления элемента также осуществляется во многом аналогично удалению из линейного однонаправленного списка:

```
procedure Del_LineDubleList(var ptrHead,
                            ptrCurrent: PElement);
  {Удаление элемента из линейного двунаправленного списка}
var
  ptrAddition: PElement;   {вспомогательный указатель}
begin
  if ptrCurrent <> nil then begin   {входной параметр корректен}
    if ptrCurrent = ptrHead then begin   {удаляем первый}
      ptrHead := ptrHead^.Next;
      dispose(ptrCurrent);
      ptrHead^.Last := nil;
      ptrCurrent := ptrHead;
    end else begin   {удаляем непервый}
      if ptrCurrent^.Next = nil then begin   {удаляем последний}
        ptrCurrent^.Last^.Next := nil;
```

```
        dispose(ptrCurrent);
        ptrCurrent := ptrHead;
      end else begin   {удаляем непоследний и непервый}
        ptrAddition := ptrCurrent^.Next;
        ptrCurrent^.Last^.Next := ptrCurrent^.Next;
        ptrCurrent^.Next^.Last := ptrCurrent^.Last;
        dispose(ptrCurrent);
        ptrCurrent := ptrAddition;
      end;
    end;
  end;
end;
```

Использование двух указателей в линейном двунаправленном списке позволяет ускорить операции, связанные с передвижением по списку за счет двунаправленности этого движения. Однако элементы списка за счет дополнительного поля занимают больший объем памяти. Кроме того, усложнились операции вставки и удаления элементов за счет необходимости манипулирования большим числом указателей.

1.2.7. Циклические списки

Линейные списки характерны тем, что в них можно выделить первый и последний элементы (имеющие пустые указатели), причем для однонаправленного линейного списка обязательно нужно иметь указатель на первый элемент. Это приводило к тому, что алгоритмы вставки и удаления крайних и средних элементов списка отличались друг от друга, что, естественно, усложняло соответствующие операции.

Основное отличие циклического списка состоит в том, что в этом списке нет элементов, содержащих пустые указатели, и, следовательно, нельзя выделить крайние элементы. Таким образом, все элементы являются «средними».

Циклические списки, так же как и линейные, бывают однонаправленными и двунаправленными.

1.2.7.1. Циклический однонаправленный список

Циклический однонаправленный список похож на линейный однонаправленный список, но его последний элемент содержит указатель, связывающий его с первым элементом (рис. 4).

Для полного обхода такого списка достаточно иметь указатель на произвольный элемент, а не на первый, как в линейном однонаправ-

ленном списке. Понятие «первого» элемента здесь достаточно условно и не всегда требуется. Хотя иногда бывает полезно выделить некоторый элемент как «первый» путем установки на него специального указателя. Это требуется, например, для предотвращения «зацикливания» при просмотре списка.

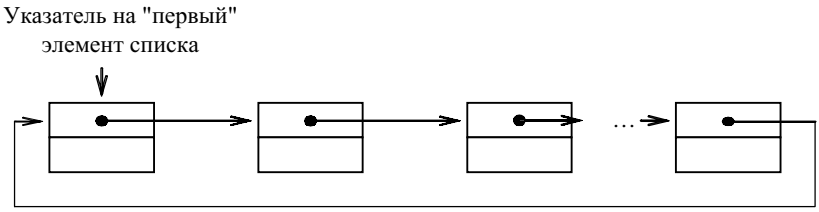


Рис. 4. Циклический однонаправленный список

Основные операции, осуществляемые с циклическим однонаправленным списком:

– вставка элемента;
– просмотр
– поиск;
– удаление элемента.

Для описания алгоритмов этих основных операций используем те же объявления данных, что и для линейного однонаправленного списка.

Вставка элемента в список, как уже говорилось, проще, чем для линейного однонаправленного списка и реализуется с помощью одной единственной процедуры: `Ins_CicleSingleList`. В качестве входных параметров передаются данные для заполнения создаваемого элемента, указатель на начало списка и указатель на текущий элемент в списке, после которого осуществляется вставка. Выходными параметрами процедур является указатель на начало списка (который возможно изменится) и указатель текущего элемента, который показывает на вновь созданный элемент:

```
procedure Ins_CicleSingleList(DataElem: TypeData;
                              var ptrHead, ptrCurrent: PElement);
  {Вставка элемента в циклический однонаправленный список}
  {справа от элемента, на который указывает ptrCurrent}
var
  ptrAddition: PElement;    {вспомогательный указатель}
begin
  New(ptrAddition);
```

```
  ptrAddition^.Data := DataElem;
  if ptrHead = nil then begin   {список пуст}
    {создаем первый элемент списка}
    ptrAddition^.Next := ptrAddition;  {цикл из 1 элемента}
    ptrHead := ptrAddition;
  end else begin   {список не пуст}
    {вставляем элемент списка справа от элемента,}
    {на который указывает ptrCurrent}
    ptrAddition^.Next := ptrCurrent^.Next;
    ptrCurrent^.Next := ptrAddition;
  end;
  ptrCurrent := ptrAddition;
end;
```

Порядок следования операторов присваивания процедуры очень важен. При неправильном переопределении указателей возможен разрыв списка или потери указателя на первый элемент, что приводит к потере доступа к части или всему списку.

Операция просмотра списка заключается в последовательном просмотре всех элементов списка. В отличие от линейного однонаправленного списка здесь признаком окончания просмотра списка будет возврат к элементу, выделенным как «первый»:

```
procedure Scan_CicleSingleList(ptrHead: PElement);
  {Просмотр циклического однонаправленного списка}
var
  ptrAddition: PElement;   {вспомогательный указатель}
begin
  if ptrHead <> nil do begin   {список не пуст}
    ptrAddition := ptrHead;
    repeat
      writeln(ptrAddition^.Data);  {Вывод значения элемента}
      ptrAddition := ptrAddition^.Next;
    until ptrAddition = ptrHead;
  end;
end;
```

Операция поиска элемента в списке заключается в последовательном просмотре всех элементов списка до тех пор, пока текущий элемент не будет содержать заданное значение или пока не достигнут «первый» элемент списка. В последнем случае фиксируется отсутствие ис-

комого элемента в списке (функция принимает значение `false`). Входными параметрами являются значение, которое должен содержать искомый элемент и указатель на список. В качестве выходного параметра передается указатель, который устанавливается на найденный элемент или остается без изменений, если элемента в списке нет:

```
function Find_CicleSingleList(DataElem: TypeData;
                              var ptrHead,
                              ptrCurrent: PElement): boolean;
  {Поиск в циклическом однонаправленном списке}
var
  ptrAddition: PElement;   {вспомогательный указатель}
begin
  if ptrHead <> nil do begin   {список не пуст}
    ptrAddition := ptrHead;
    repeat
      ptrAddition := ptrAddition^.Next;
    until (ptrAddition = ptrHead) or        {прошли весь список}
          (ptrAddition^.Data = DataElem)    {элемент найден}
    if ptrAddition^.Data = DataElem then begin
      Find_CicleSingleList := true;
      ptrCurrent := ptrAddition;
    end else begin
      Find_CicleSingleList := false;
    end;
  end else begin
    Find_CicleSingleList := false;
  end;
end;
```

Операция удаления элемента циклического однонаправленного списка осуществляет удаление элемента, на который установлен указатель текущего элемента. После удаления указатель текущего элемента устанавливается на следующий за удаляемым элемент списка. Здесь не требуется отдельных алгоритмов удаления для крайних элементов списка, как это было в линейных списках, но в случае удаления «первого» элемента необходимо соответствующий указатель переместить на следующий элемент:

```
procedure Del_CicleSingleList(var ptrHead, ptrCurrent:
PElement);
  {Удаление элемента из циклического однонаправленного списка}
```

```
var
  ptrAddition: PElement;  {дополнительный указатель}
begin
  if ptrCurrent <> nil then begin   {входной параметр корректен}
    if ptrHead^.Next <> ptrHead then begin
    {Если удаляемый элемент не единственный в списке}
    {устанавливаем вспомогательный указатель на элемент,
     предшествующий удаляемому}
      ptrAddition := ptrHead;
      while ptrAddition^.Next <> ptrCurrent do
        ptrAddition := ptrAddition^.Next;
      {непосредственное удаление элемента}
      ptrAddition^.Next := ptrCurrent^.Next;
      if ptrHead = ptrCurrent then     {удаляем первый}
        ptrHead := ptrCurrent^.Next;
      dispose(ptrCurrent);
      ptrCurrent := ptrAddition^.Next;
    end else begin
      ptrHead:=nil;
      dispose(ptrCurrent);
      ptrCurrent:=nil;
    end;
  end;
end;
```

Циклический однонаправленный список, так же как и линейный однонаправленный список, имеет только один указатель в элементах, что позволяет минимизировать расход памяти на организацию списка, но обеспечивает переходы между элементами только в одном направлении. Одновременно здесь упрощены операции вставки и удаления элементов.

Для ускорения доступа к элементам списка путем применения переходов между элементами в обоих направлениях в циклических списках применяют тот же подход, что и в линейных списках: циклический двунаправленный список.

1.2.7.2. Циклический двунаправленный список

В этом циклическом списке любой элемент имеет два указателя, один из которых указывает на следующий элемент в списке, а второй указывает на предыдущий элемент (рис. 5).

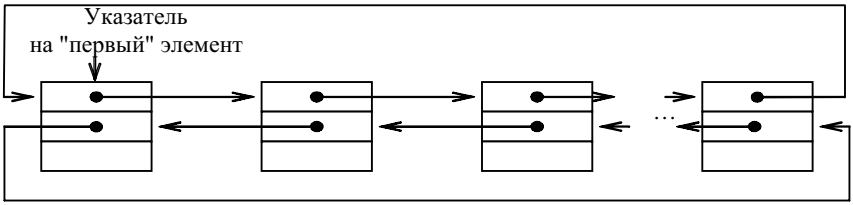


Рис. 5. Циклический двунаправленный список

Основные операции, осуществляемые с циклическим двунаправленным списком:

– вставка элемента;
– просмотр;
– поиск;
– удаление элемента.

Для описания алгоритмов этих основных операций используем те же объявления данных, что и для линейного двунаправленного списка.

Вставка элемента в список, как уже говорилось, проще, чем для линейного двунаправленного списка и реализуется с помощью одной единственной процедуры: `Ins_CicleDubleList`. В качестве входных параметров передаются: данные для заполнения создаваемого элемента, указатель на начало списка и указатель на текущий элемент в списке, после которого осуществляется вставка. Выходными параметрами процедур является указатель на начало списка (который возможно изменится) и указатель текущего элемента, который показывает на вновь созданный элемент:

```
procedure Ins_CicleDubleList(DataElem: TypeData;
                             var ptrHead, ptrCurrent: PElement);
  {Вставка элемента в циклический двунаправленный список
   справа от элемента, на который указывает ptrCurrent}
var
  ptrAddition: PElement;   {вспомогательный указатель}
begin
  New(ptrAddition);
  ptrAddition^.Data := DataElem;
  if ptrHead = nil then begin   {список пуст}
    {создаем первый элемент списка}
    ptrAddition^.Next := ptrAddition;  {петля из 1 элемента}
    ptrAddition^.Last := ptrAddition;
```

```
    ptrHead := ptrAddition;
  end else begin   {список не пуст}
    {вставляем элемент списка справа от элемента,}
    {на который указывает ptrCurrent}
    ptrAddition^.Next := ptrCurrent^.Next;
    ptrCurrent^.Next := ptrAddition;
    ptrAddition^.Last := ptrCurrent;
    ptrAddition^.Next^.Last := ptrAddition;
  end;
  ptrCurrent := ptrAddition;
end;
```

Порядок следования операторов присваивания процедуры очень важен. При неправильном переопределении указателей возможен разрыв списка или потери указателя на первый элемент, что приводит к потере доступа к части или всему списку.

Операция просмотра и операция поиска для циклического двунаправленного списка реализуются абсолютно аналогично соответствующим процедурам для циклического однонаправленного списка, и приводить их не будем. Отметим только, что просматривать и искать здесь можно в обоих направлениях.

Операция удаления элемента также осуществляется во многом аналогично удалению из циклического однонаправленного списка:

```
procedure Del_CicleDubleList(var ptrHead, ptrCurrent: PElement);
  {Удаление элемента из циклического двунаправленного списка}
var
  ptrAddition: PElement;  {дополнительный указатель}
begin
  if ptrCurrent <> nil then begin   {входной параметр корректен}
    if ptrHead^.Next <> ptrHead then begin
    {Если удаляемый элемент, не единственный в списке}
      ptrAddition := ptrCurrent^.Next;
      ptrCurrent^.Last^.Next := ptrCurrent^.Next;
      ptrCurrent^.Next^.Last := ptrCurrent^.Last;
      if ptrHead = ptrCurrent then     {удаляем первый}
        ptrHead := ptrCurrent^.Next;
      dispose(ptrCurrent);
      ptrCurrent := ptrAddition;
    end else begin
      ptrHead:=nil;
```

```
      dispose(ptrCurrent);
      ptrCurrent:=nil;
    end;
  end;
end;
```

Использование двух указателей в циклическом двунаправленном списке позволяет ускорить операции, связанные с передвижением по списку за счет двунаправленности этого движения. Однако элементы списка за счет дополнительного поля занимают больший объем памяти. Операции вставки и удаления элементов здесь осуществляются проще, чем в линейном двунаправленном списке, но сложнее, чем в циклическом однонаправленном списке (за счет необходимости манипулирования большим числом указателей).

1.2.8. Разреженные матрицы

Разреженная матрица – двухмерный массив, большинство элементов которого равны между собой, так что хранить в памяти достаточно лишь небольшое число значений, отличных от основного (фонового) значения остальных элементов.

Различают два типа разреженных матриц:

1) матрицы, в которых местоположения элементов со значениями, отличными от фонового, могут быть математически описаны;

2) матрицы со случайным расположением элементов.

В случае работы с разреженными матрицами вопросы размещения их в памяти реализуются с учетом их типа.

1.2.8.1. Матрицы с математическим описанием местоположения элементов

К данному типу матриц относятся матрицы, у которых местоположение элементов со значениями, отличными от фонового, может быть математически описано, т. е. в их расположении есть какая-либо закономерность.

Элементы, значения которых являются фоновыми, называют нулевыми, а элементы, значения которых отличны от фонового, называют ненулевыми. Но необходимо помнить, что фоновое значение не всегда равно нулю.

Ненулевые значения хранятся, как правило, в одномерном массиве (векторе), а связь между местоположением в разреженной матрице и в

новом, одномерном, описывается математически с помощью формулы, преобразующей индексы матрицы в индексы вектора.

На практике для работы с разреженной матрицей разрабатываются функции:

1) для преобразования индексов матрицы в индекс вектора;

2) для получения значения элемента матрицы из ее упакованного представления по двум индексам (строка, столбец);

3) для записи значения элемента матрицы в ее упакованное представление по двум индексам.

При таком подходе обращение к элементам матрицы выполняется с помощью указанных функций. Например, пусть имеется двумерная разреженная матрица, в которой все ненулевые элементы расположены в шахматном порядке, начиная со второго элемента. Для такой матрицы формула вычисления индекса элемента в линейном представлении будет следующей:

$$l = ((y - 1)\cdot X_m + x)/2,$$

где l – индекс в линейном представлении; x, y – индексы соответственно строки и столбца в двумерном представлении; X_m – количество элементов в строке исходной матрицы.

1.2.8.2. Матрицы со случайным расположением элементов

К данному типу относятся матрицы, у которых местоположение элементов со значениями, отличными от фонового, не могут быть математически описано, т. е. в их расположении нет какой-либо закономерности.

Пусть есть матрица **A** размерности 5×7, в которой из 35 элементов только 10 отличны от нуля:

0	0	6	0	9	0	0
2	0	0	7	8	0	4
10	0	0	0	0	0	0
0	0	12	0	0	0	0
0	0	0	3	0	0	5

Один из основных способов хранения подобных разреженных матриц заключается в запоминании ненулевых элементов в одномерном массиве записей с идентификацией каждого элемента массива ин-

дексами строки и столбца матрицы (рис. 6). Такой способ хранения называется последовательным представлением разреженных матриц.

Доступ к элементу матрицы **A** с индексами *i* и *j* выполняется выборкой индекса *i* из поля `Row`, индекса *j* из поля `Colum` и значения элемента из поля `Value`. Следует отметить, что элементы матрицы обязательно запоминаются в порядке возрастания номеров строк для ускорения поиска.

Такое представление матрицы **A** сокращает используемый объем памяти. Для больших матриц экономия памяти является очень актуальной проблемой.

Row	*Colum*	*Value*
1	3	6
1	5	9
2	1	2
2	4	7
2	5	8
2	7	4
3	1	10
4	3	12
5	4	3
5	7	5

Рис. 6. Последовательное представление разреженных матриц

Однако последовательное представление разреженных матриц имеет определенные недостатки. Так, включение и исключение новых элементов матрицы вызывает необходимость перемещения большого числа существующих элементов. Если включение новых элементов и их исключение осуществляется часто, то можно использовать описываемый ниже метод связанных структур.

Метод связанных структур переводит статическую структуру матрицы в динамическую. Эта динамическая структура реализована в виде циклических списков.

Для такого представления разреженных матриц требуется следующая структура элемента списка:

```
type
  PElement = ^TypeElement; {указатель на тип элемента}
  TypeElement = record     {тип элемента списка}
    Left: PElement;         {указатель на предыд. элемент в строке}
    Up: PElement;          {указатель на предыд. элемент в столбце}
    Value: TypeData;       {значение элемента матрицы}
    Row: integer;          {индекс строки матрицы}
    Colum: integer;        {индекс столбца матрицы}
  end;
```

Связанная структура для разреженной матрицы **A** показана на рис. 7.

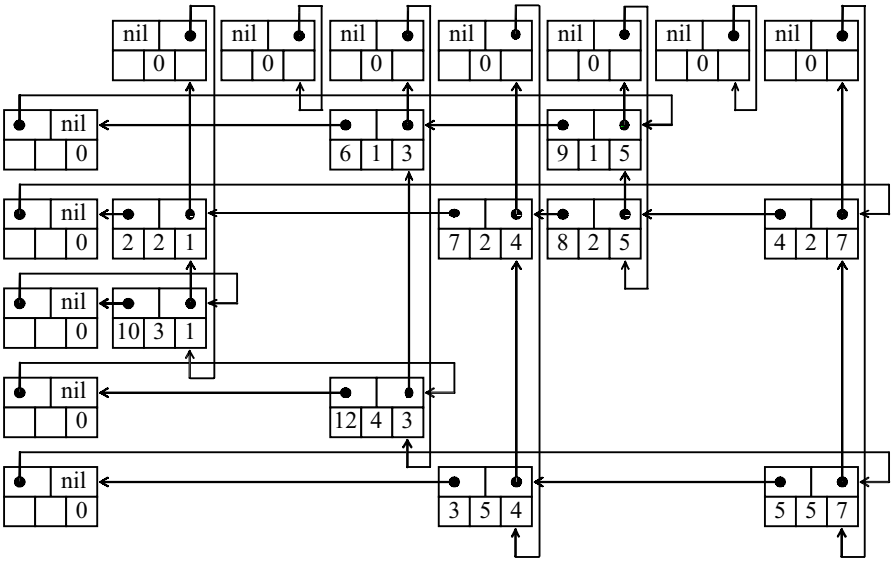


Рис. 7. Представление разреженных матриц в виде связанных структур

Циклический список представляет отдельную строку или столбец. Список столбца может содержать общие элементы с одним или более списком строки. Для того чтобы обеспечить использование более эффективных алгоритмов включения и исключения элементов, все списки строк и столбцов имеют головные элементы. Головной элемент каждого списка строки содержит ноль в поле `Colum`. Аналогично, каждый головной элемент в списке столбца имеет ноль в поле `Row`. Строка или столбец, содержащие только нулевые элементы, представлены головными вершинами, у которых поле `Left` или `Up` указывает само на себя.

Может показаться странным, что указатели в этой многосвязной структуре направлены вверх и влево, вследствие чего при сканировании циклического списка элементы матрицы встречаются в порядке убывания номеров строк и столбцов. Такой метод представления используется для упрощения включения новых элементов в структуру. Предполагается, что новые элементы, которые должны быть добавлены к матрице, обычно располагаются в порядке убывания индексов строк и индексов столбцов. Если это так, то новый элемент всегда добавляется после головного и не требуется никакого просмотра списка.

1.2.9. Стек

Стек – это структура данных, в которой новый элемент всегда записывается в ее начало (вершину) и очередной читаемый элемент также всегда выбирается из ее начала. Здесь используется принцип «последним пришел – первым вышел» (LIFO: Last Input – First Output).

Стек можно реализовывать как статическую структуру данных в виде одномерного массива, а можно как динамическую структуру – в виде линейного списка (рис. 8).

При реализации стека в виде статического массива необходимо резервировать массив, длина которого равна максимально возможной глубине стека, что приводит к неэффективному использованию памяти. Однако работать с такой реализацией проще и быстрее.

При такой реализации дно стека будет располагаться в первом элементе массива, а рост стека будет осуществляться в сторону увеличения индексов. Одновременно необходимо отдельно хранить значение индекса элемента массива, являющегося вершиной стека.

Можно обойтись без отдельного хранения индекса, если в качестве вершины стека всегда использовать первый элемент массива, но в этом случае, при записи или чтении из стека, необходимо будет осуществлять сдвиг всех остальных элементов, что приводит к дополнительным затратам вычислительных ресурсов.

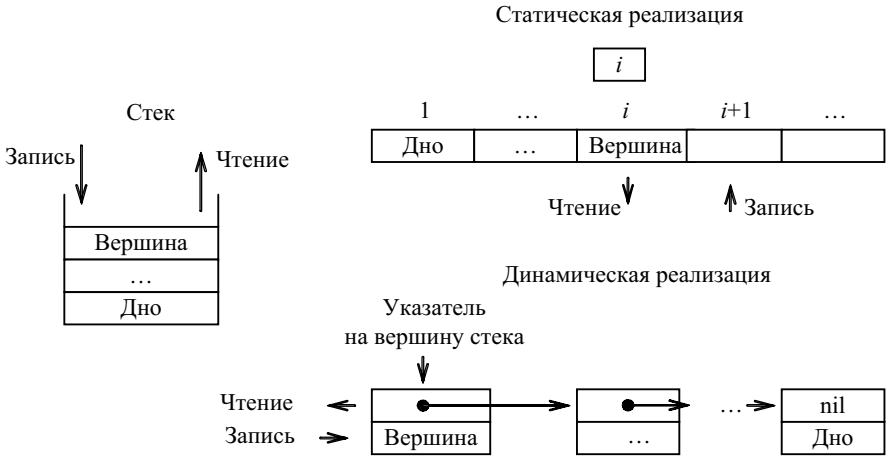


Рис. 8. Стек и его организация

Стек как динамическую структуру данных легко организовать на основе линейного списка. Поскольку работа всегда идет с заголовком стека, т. е. не требуется осуществлять просмотр элементов, удаление и вставку элементов в середину или конец списка, то достаточно использовать экономичный по памяти линейный однонаправленный список. Для такого списка достаточно хранить указатель вершины стека, который указывает на первый элемент списка. Если стек пуст, то списка не существует и указатель принимает значение `nil`.

Поскольку стек, по своей сути, является структурой с изменяемым количеством элементов, то основное внимание уделим динамической реализации стека. Как уже говорилось выше, для такой реализации целесообразно использовать линейный однонаправленный список. Поэтому при описании динамической реализации будем использовать определения и операции, приведенные в 1.2.6.1.

Описание элементов стека аналогично описанию элементов линейного однонаправленного списка, где `DataType` является типом элементов стека. Поэтому здесь приводить его не будем.

Основные операции, производимые со стеком:

– записать (положить в стек);

– прочитать (снять со стека);

– очистить стек;

– проверка пустоты стека.

Реализацию этих операций приведем в виде соответствующих процедур, которые, в свою очередь, используют процедуры операций с линейным однонаправленным списком:

```
procedure PushStack(NewElem: TypeData;
                    var ptrStack: PElement);
  {Запись элемента в стек (положить в стек)}
begin
  InsFirst_LineSingleList(NewElem, ptrStack);
end;
procedure PopStack(var NewElem: TypeData,
                    ptrStack: PElement);
  {Чтение элемента из стека (снять со стека)}
begin
  if ptrStack <> nil then begin
    NewElem := ptrStack^.Data;
    Del_LineSingleList(ptrStack, ptrStack);  {удаляем вершину}
```

```
  end;
end;
procedure ClearStack(var ptrStack: PElement);
  {Очистка стека}
begin
  while ptrStack <> nil do
    Del_LineSingleList(ptrStack, ptrStack);  {удаляем вершину}
end;
function EmptyStack(var ptrStack: PElement): boolean;
  {Проверка пустоты стека}
begin
  if ptrStack = nil then EmptyStack := true
                    else EmptyStack := false;
end;
```

1.2.10. Очередь

Очередь – это структура данных, представляющая собой последовательность элементов, образованная в порядке их поступления. Каждый новый элемент размещается в конце очереди; элемент, стоящий в начале очереди, выбирается из нее первым. Здесь используется принцип «первым пришел – первым вышел» (FIFO: First Input – First Output).

Очередь можно реализовывать как статическую структуру данных в виде одномерного массива, а можно как динамическую структуру – в виде линейного списка (рис. 9).

При реализации очереди в виде статического массива необходимо резервировать массив, длина которого равна максимально возможной длине очереди, что приводит к неэффективному использованию памяти.

При такой реализации начало очереди будет располагаться в первом элементе массива, а рост очереди будет осуществляться в сторону увеличения индексов. Однако, поскольку добавление элементов происходит в один конец, а выборка – из другого конца очереди, то с течением времени будет происходить миграция элементов очереди из начала массива в сторону его конца. Это может привести к быстрому исчерпанию массива и невозможности добавлении новых элементов в очередь при наличии свободных мест в начале массива. Предотвратить это можно двумя способами:

– после извлечения очередного элемента из начала очереди осуществлять сдвиг всей очереди на один элемент к началу массива. При этом необходимо отдельно хранить значение индекса элемента массива,

являющегося концом очереди (начало очереди всегда в первом элементе массива);

– представить массив в виде циклической структуры, где первый элемент массива следует за последним. Элементы очереди располагаются в «круге» элементов массива в последовательных позициях, конец очереди находится по часовой стрелке на некотором расстоянии от начала. При этом необходимо отдельно хранить значение индекса элемента массива, являющегося началом очереди, и значение индекса элемента массива, являющегося концом очереди. Когда происходит добавление в конец или извлечение из начала очереди, осуществляется смещение значений этих двух индексов по часовой стрелке.

С точки зрения экономии вычислительных ресурсов более предпочтителен второй способ. Однако здесь усложняется проверка на пустоту очереди и контроль переполнения очереди – индекс конца очереди не должен «набегать» на индекс начала.

Очередь как динамическую структуру данных легко организовать на основе линейного списка. Поскольку работа идет с обоими концами очереди, то предпочтительно будет использовать линейный двунаправленный список. Хотя, как уже говорилось при описании этого списка,

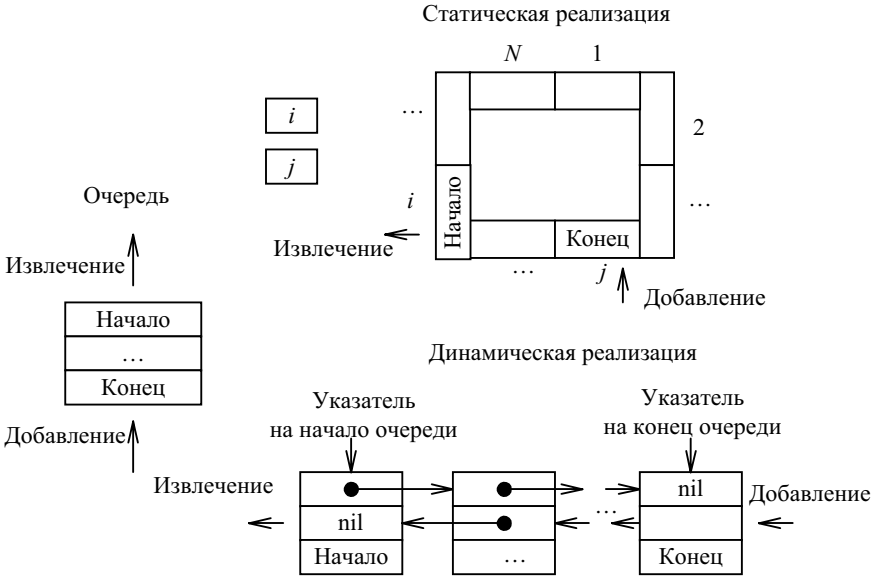


Рис. 9. Очередь и ее организация

для работы с ним достаточно иметь один указатель на любой элемент списка, здесь целесообразно хранить два указателя – один на начало списка (откуда извлекаем элементы) и один на конец списка (куда добавляем элементы). Если очередь пуста, то списка не существует и указатели принимают значение `nil`.

Поскольку очередь, по своей сути, является структурой с изменяемым количеством элементов, то основное внимание уделим динамической реализации очереди. Как уже говорилось выше, для такой реализации целесообразно использовать линейный двунаправленный список. Поэтому при описании динамической реализации будем использовать определения и операции, приведенные в 1.2.6.2.

Описание элементов очереди аналогично описанию элементов линейного двунаправленного списка, где `DataType` является типом элементов очереди. Поэтому здесь приводить его не будем, но введем дополнительно два указателя на начало и конец очереди:

```
var
  ptrBeginQueue,
  ptrEndQueue: PElement;
```

Основные операции, производимые с очередью:

– добавить элемент;

– извлечь элемент;

– очистить очередь;

– проверка пустоты очереди.

Реализацию этих операций приведем в виде соответствующих процедур, которые, в свою очередь, используют процедуры операций с линейным двунаправленным списком:

```
procedure InQueue(NewElem: TypeData;
                  var ptrBeginQueue, ptrEndQueue: PElement);
  {Добавление элемента в очередь}
begin
  Ins_LineDoubleList(NewElem, ptrBeginQueue, ptrEndQueue);
end;
procedure FromQueue(var NewElem: TypeData;
                    var ptrBeginQueue: PElement);
  {Извлечение элемента из очереди}
begin
  if ptrBeginQueue <> nil then begin
```

```
    NewElem := ptrEndQueue^.Data;
    Del_LineDoubleList(ptrBeginQueue, ptrBeginQueue);
  end;
end;
procedure ClearQueue(var ptrBeginQueue,
                         ptrEndQueue: PElement);
  {Очистка очереди}
begin
  while ptrBeginQueue <> nil do
    Del_LineDoubleList(ptrBeginQueue, ptrBeginQueue);
  ptrEndQueue := nil;
end;
function EmptyQueue(var ptrBeginQueue: PElement): boolean;
  {Проверка пустоты очереди}
begin
  if ptrBeginQueue = nil then EmptyQueue := true
                         else EmptyQueue := false;
end;
```

1.2.11. Дек

Дек – это структура данных, представляющая собой последовательность элементов, в которой можно добавлять и удалять в произвольном порядке элементы с двух сторон. Первый и последний элементы дека соответствуют входу и выходу дека.

Выделяют ограниченные деки:

– дек с ограниченным входом – из конца дека можно только извлекать элементы;

– дек с ограниченным выходом – в конец дека можно только добавлять элементы.

Данная структура является наиболее универсальной из рассмотренных выше линейных структур. Накладывая дополнительные ограничения на операции с началом и/или концом дека, можно осуществлять моделирование стека и очереди.

Дек также можно реализовывать как статическую структуру данных в виде одномерного массива, а можно как динамическую структуру – в виде линейного списка (рис. 10).

Поскольку в деке, как и в очереди, осуществляется работа с обоими концами структуры, то целесообразно использовать те же подходы к организации дека, что применялись и для очереди (см. 1.2.10).

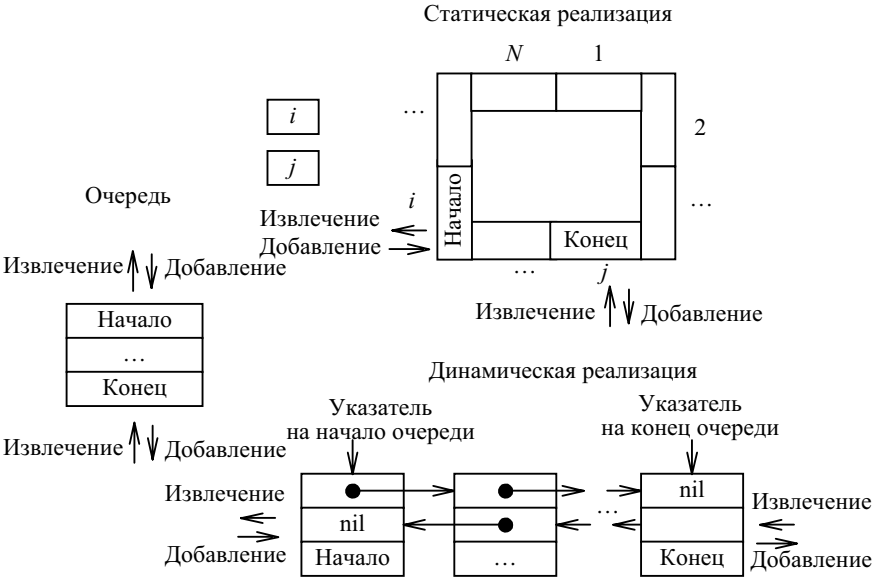


Рис. 10. Дек и его организация

Описание элементов дека аналогично описанию элементов линейного двунаправленного списка, где `DataType` является типом элементов дека. Поэтому здесь приводить его не будем. Но, как и для очереди, введем дополнительно два указателя на начало и конец дека:

```
var
  ptrBeginDeck,
  ptrEndDeck: PElement;
```

Основные операции, производимые с деком:

– добавить элемент в начало;
– добавить элемент в конец;
– извлечь элемент из начала;
– извлечь элемент из конца;
– очистить дек;
– проверка пустоты дека.

Реализацию этих операций приведем в виде соответствующих процедур, которые, в свою очередь, используют процедуры операций с линейным двунаправленным списком:

```
procedure InBeginDeck(NewElem: TypeData;
                      var ptrBeginDeck: PElement);
  {Добавление элемента в начало дека}
begin
  InsFirst_LineDoubleList(NewElem, ptrBeginDeck);
end;
procedure InEndDeck(NewElem: TypeData;
                    var ptrBeginDeck, ptrEndDeck: PElement);
  {Добавление элемента в конец дека}
begin
  Ins_LineDoubleList(NewElem, ptrBeginDeck, ptrEndDeck);
end;
procedure FromBeginDeck(NewElem: TypeData;
                        var ptrBeginDeck: PElement);
  {Извлечение элемента из начала дека}
begin
  if ptrBeginDeck <> nil then begin
    NewElem := ptrBeginDeck^.Data;
    Del_LineDoubleList(ptrBeginDeck, ptrBeginDeck); {удаляем первый}
  end;
end;
procedure FromEndDeck(NewElem: TypeData,
                      var ptrBeginDeck, ptrEndDeck: PElement);
  {Извлечение элемента из конца дека}
begin
  if ptrBeginDeck <> nil then begin
    NewElem := ptrEndDeck^.Data;
    Del_LineDoubleList(ptrBeginDeck, ptrEndDeck); {удаляем конец}
  end;
end;
procedure ClearDeck(var ptrBeginDeck: PElement);
  {Очистка дека}
begin
  while ptrBeginDeck <> nil do
    Del_LineDoubleList(ptrBeginDeck, ptrBeginDeck);
  ptrEndDeck := nil;
end;
function EmptyDeck(var ptrBeginDeck: PElement): boolean;
  {Проверка пустоты дека}
begin
  if ptrBeginDeck = nil then EmptyDeck := true
```

```
                    else EmptyDeck := false;
end;
```

1.3. Нелинейные структуры данных

1.3.1. Мультисписки

Мультисписок – это структура данных, состоящая из элементов, содержащих такое число указателей, которое позволяет организовать их одновременно в виде нескольких различных списков (рис. 11).

В элементах мультисписка важно различать поля указателей для разных списков, чтобы можно было проследить элементы одного списка, не вступая в противоречие с указателями другого списка.

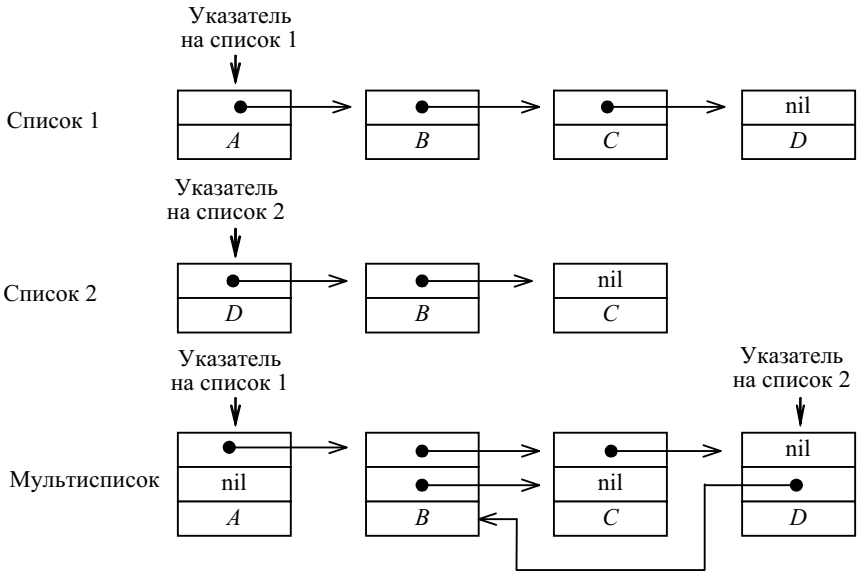


Рис. 11. Мультисписок

При использовании традиционных списков для представления повторяющихся данных происходит нерациональное использование памяти за счет дублирования динамических элементов, хранящих повторяющиеся данные. Использование мультисписков позволяет устранить этот недостаток.

Поиск в мультисписке происходит аналогично поиску в линейном списке, но при этом используется только один указатель, соответствующий списку, в котором осуществляется поиск.

Добавление элемента здесь сложнее. Добавление элемента, принадлежащего только одному из списков, осуществляется аналогично добавлению в линейный список, за исключением того, что поля указателей, относящиеся к другим спискам, устанавливаются в `nil`. При добавлении элемента, принадлежащего сразу нескольким спискам, необходимо аккуратно осуществлять определение значений соответствующих указателей. Алгоритм выполнения такой операции сильно зависит от количества списков и места вставки нового элемента.

1.3.2. Слоеные списки

Слоеные (skip), или разделенные, списки – это связные списки, которые позволяют перескакивать через некоторое количество элементов (рис. 12). Это позволяет преодолеть ограничения последовательного поиска, являющейся основной причиной низкой эффективности поиска в линейных списках. В то же время вставка и удаление остаются сравнительно эффективными.

Идея, лежащая в основе слоеных списков, очень напоминает метод, используемый при поиске имен в адресной книжке. Чтобы найти имя, ищут страницу, помеченную буквой, с которой начинается имя, а затем поиск осуществляют в пределах этой страницы.

В отличие от элементов обычных линейных списков, элементы этих списков имеют дополнительный указатель. Все элементы списка группируются по определенному признаку, и первый элемент каждой группы содержит указатель на первый элемент следующей группы. Если

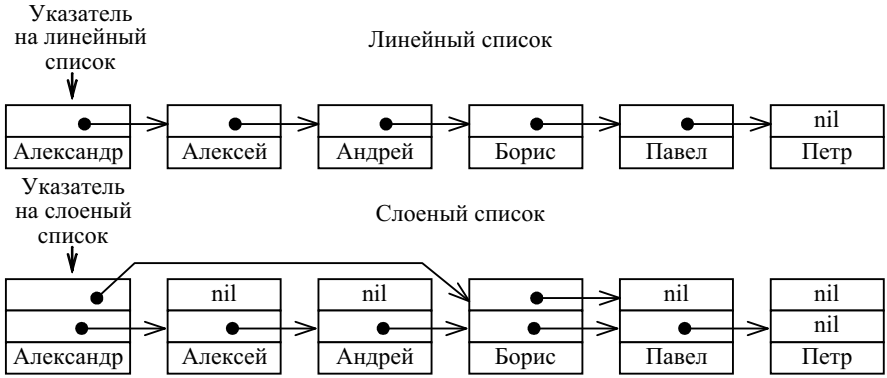


Рис. 12. Слоеный список и его организация

следующая группа отсутствует или элемент не является первым в группе, то этот дополнительный указатель принимает значение `nil`.

Эта простая идея может быть расширена – можно добавлять нужное число дополнительных указателей, группируя группы элементов и т. д. на нужном количестве уровней.

Если реализован только один уровень, то это, фактически, обычный список и время поиска пропорционально $O(n)$. Однако если имеется достаточное число уровней, то разделенный список можно считать деревом с корнем на высшем уровне, а для дерева время поиска, как будет показано ниже, пропорционально $O(\log n)$.

1.3.3. Графы

1.3.3.1. Спецификация

Граф G – это упорядоченная пара (V, E), где V – непустое множество вершин, E – множество пар элементов множества V, называемое множеством ребер.

Упорядоченная пара элементов из V называется *дугой*. Если все пары в E упорядочены, то граф называется о р и е н т и р о в а н н ы м (орграфом).

Если дуга e ведет от вершины v_1 к вершине v_2, то говорят, что дуга e инцидентна вершине v_2, а вершина v_2 являются смежной вершине v_1. В случае неориентированного графа ребро e является инцидентной обеим вершинам v_1 и v_2, а сами вершины – взаимно смежными.

Путь – это любая последовательность вершин орграфа такая, что в этой последовательности вершина b может следовать за вершиной a, только если существует дуга, следующая из a в b. Аналогично можно определить путь, состоящий из дуг.

Путь, начинающийся и заканчивающийся в одной и той же вершине, называется *циклом*. Граф, в котором отсутствуют циклы, называется а ц и к л и ч е с к и м .

Петля – дуга, соединяющая некоторую вершину сама с собой.

Теория графов является важной частью вычислительной математики. С помощью этой теории решаются большое количество задач из различных областей. Граф состоит из множества вершин и множества ребер, которые соединяют между собой вершины. С точки зрения теории графов не имеет значения, какой смысл вкладывается в вершины и ребра. Вершинами могут быть населенные пункты, а ребрами дороги, соединяющие их, или вершинами являться подпрограммы, а соедине-

ние вершин ребрами означает взаимодействие подпрограмм. Часто имеет значение направление дуги в графе.

1.3.3.2. Реализация

Выбор структуры данных для хранения графа в памяти имеет принципиальное значение при разработке эффективных алгоритмов. Рассмотрим два матричных и три списочных представления графа (см. рис. 13):

– матрица смежности;
– матрица инцидентности;
– списки смежных вершин;
– список ребер;
– списки вершин и ребер.

При дальнейшем изложении будем предполагать, что вершины графа обозначены символьной строкой и всего их до n, а ребер – до m. Каждое ребро и каждая вершина имеют вес – целое положительное число. Если граф не является помеченным, то считается, что вес равен единице.

Матрица смежности – это двумерный массив размером $n \times n$:

```
type
  TAdjacencyMatrix = array [1..n, 1..n] of integer;
var
  Graph: TAdjacencyMatrix;
```

При этом

$$\mathrm{Graf}[i, j] = \begin{cases} 0, \text{ если вершина } i \text{ не смежна вершине } j, \\ \text{вес ребра (дуги), если вершина } i \text{ смежна вершине } j, \\ \text{вес вершины, если } i = j. \end{cases}$$

Вес вершины указывается в элементах матрицы смежности, находящихся на главной диагонали, только в том случае, если в графе отсутствуют петли. В противном случае, в этих элементах указывается вес петли.

Пространственная сложность этого способа $V(n) \sim O(n^2)$. Способ очень хорош, когда надо часто проверять смежность или находить вес ребра по двум заданным вершинам.

Матрица инцидентности – это двумерный массив размером $n \times m$:

```
type
  TIncidenceMatrix = array [1..n, 1..m] of integer;
```

```
var
  Graph: TIncidenceMatrix;
```

При этом

$$\text{Graf}[i, j] = \begin{cases} 0, \text{ если вершина } i \text{ не инцидентна ребру } j, \\ \text{вес ребра (дуги в } i\text{), если вершина инцидентна ребру } j. \end{cases}$$

Пространственная сложность этого способа $V(n, m) \sim O(n \times m)$. Матрица инцидентности лучше всего подходит для операции «перечисление ребер, инцидентных вершине x».

Списки смежных вершин – это одномерный массив размером n, содержащий для каждой вершины указатели на списки смежных с ней вершин:

```
type
  PNode = ^Node;
  Node = record              {смежная вершина}
      Name: string;          {имя смежной вершины}
      Weight: integer;       {вес ребра}
      Next: PNode;           {следующая смежная вершина}
  end;
  TAdjacencyList = array [1..n] of record
    NodeWeight: integer;     {вес вершины}
    Name: string;            {имя вершины}
    List: PNode;             {указатель на список смежных}
    end;
var
  Graph: TAdjacencyList;
```

Пространственная сложность этого способа $V_{\max}(n) \sim O(n^2)$. Хранение списков в динамической памяти позволяет сократить объем расходуемой памяти, так как в этом случае не будет резервироваться место под n соседей для каждой вершины. Можно и сам массив представить в виде динамического списка, но это не имеет особого смысла, так как граф обычно является статичной (неизменяемой) структурой.

Этот способ хранения лучше всех других подходит для операции «перечисление всех вершин, смежных с x».

Список ребер – это одномерный массив размером m, содержащий список пар вершин, инцидентных с одним ребром графа:

```
type
  TBranchList = array [1..m] of record
```

```
    Node1: string;      {1-я вершина, инцидентная ребру}
    Node2: string;      {2-я вершина, инцидентная ребру}
    Weight: integer;    {вес ребра}
  end;
var
  Graph: TBranchList;
```

Пространственная сложность этого способа $V(m)$~$O(m)$. Этот способ хранения графа особенно удобен, если главная операция, которой чаще всего выполняется, это перечисление ребер или нахождение вершин и ребер, находящихся в отношениях инцидентности.

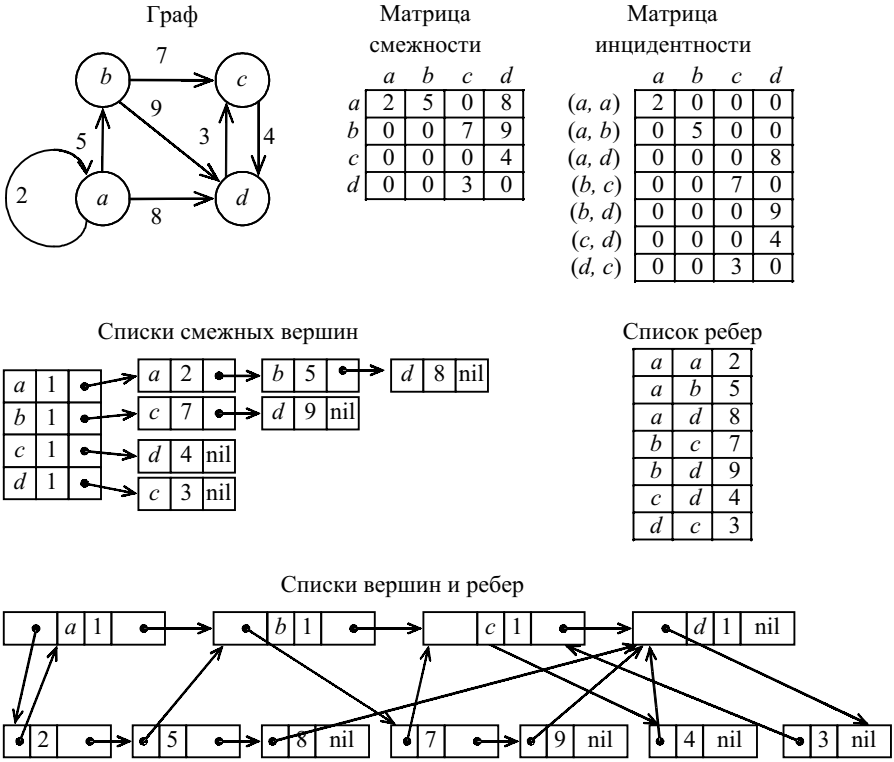


Рис. 13. Граф и его реализации

Граф можно представить также в виде списочной структуры, состоящей из списков двух типов – *списки вершин и списков ребер*:

```
type
  PNode = ^TNode;
  PBranch = ^TBranch;
  TNode = record       {вершина}
    Name: string;        {имя вершины}
    Weight: integer;     {вес вершины}
    Branch: PBranch;     {выходящее ребро}
    Next: PNode;         {следующая вершина}
  end;
  TBranch = record     {ребро}
    Node: PNode;         {вершина, в которую входит}
    Weight: integer;     {вес ребра}
    Next: PBranch;       {следующее выходящее ребро}
  end;
var
  Graph: PBranch;
```

Пространственная сложность этого способа $V(n, m) \sim O(n+m)$.

1.3.4. Деревья

1.3.4.1. Общие понятия

Дерево является одним из важных и интересных частных случаев графа, поэтому оно рассматривается отдельно от графов других видов.

Деревом называется орграф, для которого:

1) существует узел, в который не входит ни одной дуги (корень);

2) в каждую вершину, кроме корня, входит одна дуга.

Вершины дерева подразделяют на следующие три группы:

– корень – вершина, в которую не входит ни одной дуги;

– узлы – вершины, в которые входит одна дуга и выходит одна или более дуг;

– листья – вершины, в которые входит одна дуга и не выходит ни одной дуги.

Все вершины, в которые входят дуги, исходящей из одной вершины, называются *потомками* этой вершины, а сама вершина – *предком*. Корень не имеет предка, а листья не имеют потомков.

Выделяют уровни дерева. На первом уровне дерева может быть только одна вершина – корень, на втором – потомки корня, на третьем – потомки потомков корня и т. д.

Поддеревом называется вершина со всеми ее потомками.

Высотой поддерева будем считать максимальную длину цепи y_1, ..., y_n его вершин такую, что y_{i+1} – потомок y_i для всех i. Высота пустого дерева равна нулю, высота дерева из одного корня – единице.

Степенью вершины в дереве называется количество дуг, которое из нее выходит. Степень дерева равна максимальной степени вершины, входящей в дерево. При этом листьями в дереве являются вершины, имеющие степень нуль.

По величине степени дерева часто различают два типа деревьев:

– двоичные – степень дерева не более двух;

– сильноветвящиеся – степень дерева произвольная.

Дерево произвольной степени (сильноветвящееся дерево) можно реализовывать как любой граф (см. 1.3.3.2). Однако, учитывая специфику дерева как частного случая графа, можно рассмотреть отдельный способ реализации – как динамическую структуру в виде списка (рис. 14). Списочное представление деревьев произвольной степени основано на элементах, соответствующих вершинам дерева. Каждый элемент имеет поле данных и два поля указателей: указатель на начало списка потомков вершины и указатель на следующий элемент в списке потомков текущего уровня. При таком способе представления дерева обязательно следует сохранять указатель на вершину, являющуюся корнем дерева:

```
type
  PTree = ^TTree;
```

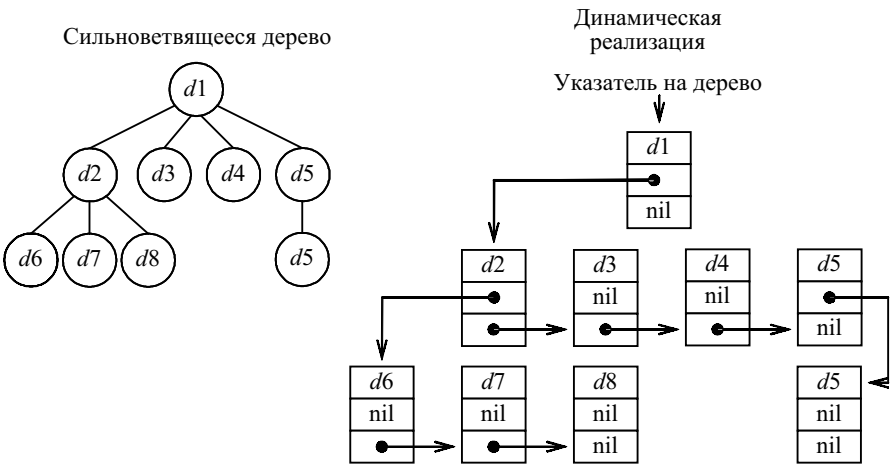


Рис. 14. Дерево произвольной степени и его динамическая организация

```
TTree = record
  Data: TypeElement; {поле данных}
  Childs, Next: PTree; {указатели на потомков и на следующий}
end;
```

1.3.4.2. Обходы деревьев

Существует несколько способов обхода (просмотра) всех вершин дерева. Три наиболее часто используемых способа обхода называются (рис. 15):

– в прямом порядке;

– в обратном порядке;

– в симметричном (внутреннем) порядке.

Все три способа обхода рекурсивно можно определить следующим образом:

1) если дерево *Tree* является пустым деревом, то в список обхода заносится пустая запись;

2) если дерево *Tree* состоит из одной вершины, то в список обхода записывается эта вершина;

3) если *Tree* – дерево с корнем *n* и поддеревьями $Tree_1$, $Tree_2$, …, $Tree_k$, то:

– при прохождении в прямом порядке сначала посещается корень *n*, затем в прямом порядке вершины поддерева $Tree_1$, далее в прямом порядке вершины поддерева $Tree_2$ и т. д. Последними в прямом порядке посещаются вершины поддерева $Tree_k$;

– при прохождении в обратном порядке сначала посещаются в обратном порядке вершины поддерева $Tree_1$, далее последовательно в обратном порядке посещаются вершины поддеревьев $Tree_2$, …, $Tree_k$. Последним посещается корень *n*;

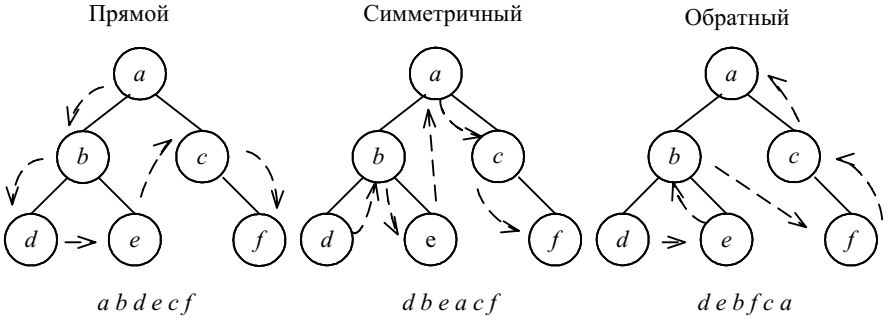


Рис. 15. Обходы деревьев

– при прохождении в симметричном порядке сначала посещаются в симметричном порядке вершины поддерева $Tree_1$, далее корень n, затем последовательно в симметричном порядке вершины поддеревьев $Tree_2$, …, $Tree_k$.

Далее приведены наброски процедур, реализующих указанные способы обходов деревьев. При доработке этих набросков необходимо учитывать конкретную реализацию деревьев:

```
procedure PreOrder(n: вершина);
  {Обход дерева в прямом порядке}
begin
  Занести в список обхода вершину n;
  for для каждого потомка s вершины n в порядке слева направо do
    PreOrder(s);
end;
procedure LastOrder(n: вершина);
  {Обход дерева в обратном порядке}
begin
  for для каждого потомка s вершины n в порядке слева направо do
    LastOrder(s);
  Занести в список обхода вершину n;
end;
procedure InOrder(n: вершина);
  {Обход дерева в симметричном порядке}
begin
  if n – лист then begin
    занести в список обхода узел n;
  end else begin
    InOrder(самый левый потомок вершины n);
    Занести в список обхода вершину n;
    for для каждого потомка s вершины n, исключая самый левый,
        в порядке слева направо do
      InOrder(s);
  end;
end;
```

1.3.4.3. Спецификация двоичных деревьев

Как уже говорилось выше, двоичные (бинарные) деревья – это деревья со степенью не более двух (рис. 16).

По степени вершин двоичные деревья бывают:

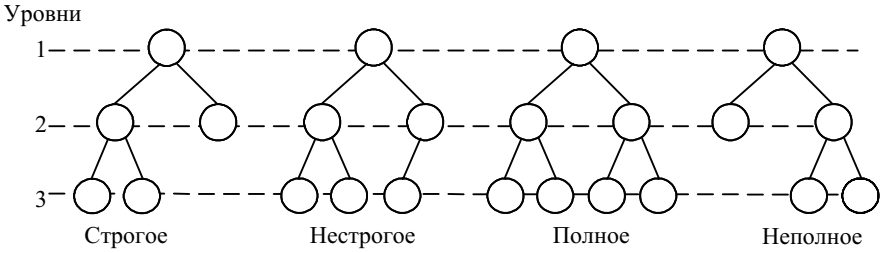


Рис. 16. Двоичное дерево

– с т р о г и е – вершины дерева имеют степень нуль (у листьев) или два (у узлов);

– н е с т р о г и е – вершины дерева имеют степень нуль (у листьев), один или два (у узлов).

В общем случае на k-м уровне двоичного дерева может быть до 2^{k-1} вершин.

Двоичное дерево, содержащее только полностью заполненные уровни (т. е. 2^{k-1} вершин на каждом k-м уровне), называется п о л н ы м .

1.3.4.4. Реализация

Двоичное дерево можно реализовывать как статическую структуру данных в виде одномерного массива, а можно как динамическую структуру – в виде списка (рис. 17).

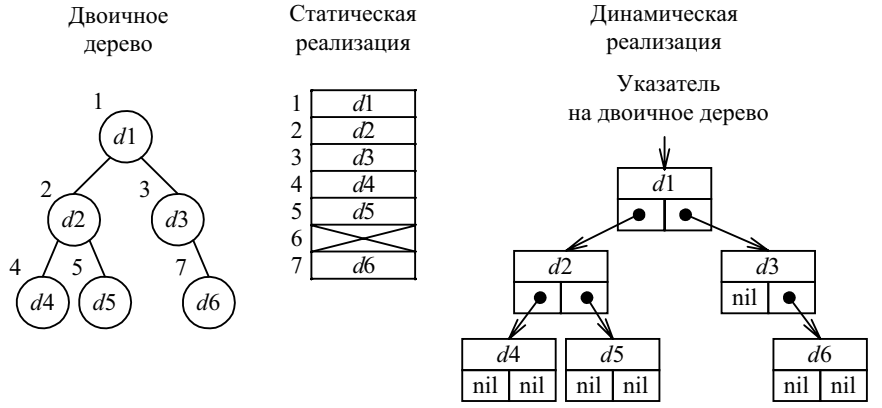


Рис. 17. Двоичное дерево и его организация

Списочное представление двоичных деревьев основано на элементах, соответствующих узлам дерева. Каждый элемент имеет поле данных и два поля указателей. Один указатель используется для связывания элемента с правым потомком, а другой – с левым. Листья имеют пустые указатели потомков. При таком способе представления дерева обязательно следует сохранять указатель на узел, являющийся корнем дерева:

```
type
  PTree = ^TTree;
  TTree = record
    Data: TypeElement; {поле данных}
    Left, Right: PTree; {указатели на левого и правого потомков}
  end;
```

В виде массива проще всего представляется полное двоичное дерево, так как оно всегда имеет строго определенное число вершин на каждом уровне. Вершины можно пронумеровать слева направо последовательно по уровням и использовать эти номера в качестве индексов в одномерном массиве. Если число уровней дерева в процессе обработки не будет существенно изменяться, то такой способ представления полного двоичного дерева будет значительно более экономичным, чем любая списковая структура:

```
type
  Tree: array[1..N] of TypeElement;
```

Адрес любой вершины в массиве вычисляется как

$$\text{адрес} = 2^{k-1}+i-1,$$

где k – номер уровня вершины; i – номер на уровне k в полном двоичном дереве. Адрес корня будет равен единице. Для любой вершины, имеющей индекс j в массиве, можно вычислить адреса левого и правого потомков:

$$\text{адрес\_левого} = 2*j$$

$$\text{адрес\_правого} = 2*j+1$$

Главным недостатком статического способа представления двоичного дерева является то, что массив имеет фиксированную длину. Размер массива выбирается исходя из максимально возможного количества уровней двоичного дерева, и чем менее полным является дерево, тем менее рационально используется память. Кроме того, недостатком являются

большие накладные расходы при изменении структуры дерева (например, при обмене местами двух поддеревьев).

1.3.4.5. Основные операции

Реализация операций будет рассматриваться для двоичных деревьев, представленных как динамическая структура.

В качестве основных операций с двоичными деревьями рассмотрим операцию прямого обхода двоичного дерева в рекурсивной и нерекурсивной форме. Реализация обратного и симметричного обходов аналогична. Операции добавления, поиска и удаления вершин дерева зависят от принятого порядка вершин, поэтому будут представлены в 2.3.4.1, посвященном упорядоченным деревьям.

```
procedure PreOrder_BinTree(Node: PTree);
  {Рекурсивный обход двоичного дерева в прямом порядке}
begin
  writeln(Node^.Data);
  if Node^.Left <> nil then PreOrder_BinTree(Node^.Left);
  if Node^.Right <> nil then PreOrder_BinTree(Node^.Right);
end;
```

В процедуре, реализующей нерекурсивный обход двоичного дерева, используется стек, хранящий путь от корня дерева до предка текущей вершины. Описание этого стека и операции с ним аналогичны тем, что приведены в 1.2.9 с одним уточнением – элементы стека хранят указатели на вершины дерева.

Процедура работает в двух режимах. В первом режиме осуществляется обход по направлению к левым потомкам до тех пор, пока не встретится лист, при этом выполняется печать значений вершин, и занесение указателей на них в стек. Во втором режиме осуществляется возврат по пройденному пути с поочередным извлечением указателей из стека до тех пор, пока не встретится вершина, имеющая еще не напечатанного правого потомка. Тогда процедура переходит в первый режим и исследует новый путь, начиная с этого потомка:

```
procedure NR_PreOrder_BinTree(Tree: PTree);
  {Нерекурсивный обход двоичного дерева в прямом порядке}
var
  Node: Ptree;       {Указатель на текущую вершину}
  S: ^TypeElement;   {Стек указателей вершин}
```

```
begin
  {Инициализация}
  ClearStack(S);
  Node := Tree;
  while true do
    if Node <> nil then begin
      writeln(Node^.Data);
      PushStack(Node, S);
      {Исследование левого потомка вершины Node}
      Node := Node^.Left;
    end else begin
      {Завершено исследование пути, содержащегося в стеке}
      if EmptyStack(S) then return;
      {Исследование правого потомка вершины Node}
      PopStack(Node, S);
      Node := Node^.Right;
    end;
end;
```

1.4. Файлы

Файл – это поименованная область во внешней памяти.

Ранее, при обсуждении структур данных, предполагалось, что объем данных позволяет обходиться исключительно основной (оперативной) памятью. Существуют задачи, в которых объем используемых данных намного превышает возможности основной памяти. В большинстве вычислительных систем предусмотрены устройства внешней памяти (диски, ленты), на которых можно хранить огромные объемы данных.

Во многих языках программирования предусмотрен файловый тип данных, предназначенный для представления данных, хранящихся во внешней памяти. Даже если в языке программирования файловый тип не определен, в операционной системе понятие файла, несомненно, поддерживается.

Операционная система делит внешнюю память на блоки одинакового размера. Размер блока зависит от конкретного типа операционной системы. Файлы хранятся в виде определенной последовательности блоков; каждый такой блок содержит целое число записей файла.

Базовыми операциями, выполняемыми по отношению к файлам, является перенос одного блока из внешней памяти в буфер и перенос

одного блока из буфера во внешнюю память. Буфер находится в основной памяти, и его размер соответствует размеру блока.

При осуществлении чтения из файла указатель считывания указывает на одну из записей в блоке, который в данный момент находится в буфере. Когда этот указатель должен переместиться на запись, отсутствующую в буфере, происходит чтение очередного блока из внешней памяти в буфер.

Аналогично, при осуществлении записи в файл фактически происходит внесение записей в буфер файла непосредственно за записями, которые уже находятся там. Если очередная запись не помещается в буфере, содержимое буфера переносится в свободный блок внешней памяти, который присоединяется к концу списка блоков данного файла. После этого буфер становится свободным для помещения в него очередной порции записей.

Рассматривая операции с файлами, в первом приближении можно считать, что файлы – это просто совокупности записей, над которыми можно выполнять операции, которые уже обсуждались выше. Однако имеются две важные особенности.

Природа устройств внешней памяти такова, что время, необходимое для поиска блока и чтения его в основную память, достаточно велико в сравнении со временем, которое требуется для относительно быстрой обработки данных, содержащихся в этом блоке. Процесс записи блока из буфера в определенное место внешней памяти занимает примерно столько же времени.

Оценивая эффективность структур данных и работы алгоритмов, в которых используются данные, хранящиеся в виде файлов, приходится в первую очередь учитывать количество обращений к блокам, т. е. сколько раз производится считывание в основную память или запись блока во внешнюю память. Предполагается, что размер блока фиксирован в операционной системе, поэтому нет возможности ускорить работу алгоритма, увеличив размер блока и сократив тем самым количество обращений к блокам.

Еще одной особенностью хранения данных во внешней памяти является наличие так называемых закрепленных записей. Иногда, например, в базах данных, используют указатели на записи, представляющие собой пару «физический адрес блока – смещение записи в блоке». Следствием применения подобных указателей является то, что записи, на которые имеются эти указатели, нельзя перемещать, поскольку не ис-

ключено, что какой-то неизвестный указатель после перемещений записи будет содержать неправильный адрес записи.

1.4.1. Организация

Существуют несколько способов организации данных в виде файлов:

– последовательный файл;
– хешированные файлы;
– индексированные файлы;
– B-деревья.

Рассмотрим кратко первые три способа использования (B-деревья рассмотрим более подробно далее).

При простой (и наименее эффективной) организации данных в виде *последовательных файлов* используются такие примитивы чтения и записи файлов, которые встречаются во многих языках программирования (например, `read`() и `write`() в языке Паскаль). В этом случае записи могут храниться в любом порядке.

Поиск записи осуществляется путем полного просмотра файла. Вставку в файл можно выполнять путем присоединения соответствующей записи в конец файла. В случае изменения записи необходимо осуществить поиск требуемой записи, а затем внести в нее изменения.

При удалении записи тоже необходимо найти удаляемую запись, а затем определенным вариантом удалить. Один из вариантов – сдвинуть все записи, следовавшие за удаленной записью, на одну позицию вперед (осуществляя при сдвиге перенос записей между блоками). Однако такой подход не годится, если записи являются закрепленными, поскольку указатель на i-ю запись в файле после выполнения такой операции будет указывать на (i+1)-ю запись. В этом случае необходимо определенным образом помечать уделенные записи, но не смещать оставшиеся на место удаленных (и не должны вставлять на их место новые записи). Существуют два способа помечать удаленные записи:

1) заменить значение записи на специальное значение, которое никогда не может стать значением неудаленной записи;

2) предусмотреть для каждой записи специальный бит удаления, который содержит, например, 1 в удаленных записях и 0 – в неудаленных записях.

Очевидным недостатком последовательного файла является то, что операции с такими файлами выполняются медленно. Выполнение каж-

дой операции требует, чтобы осуществлялось чтение всего файла. Однако существуют способы организации файлов, позволяющие обращаться к записи, считывая в основную память лишь небольшую часть файла. Такие способы предусматривают наличие у каждой записи файла так называемого ключа, т. е. поля (или совокупности полей), которое уникальным образом идентифицирует каждую запись. К сожалению, при отсутствии ключей, ускорения операций добиться не удается.

Хеширование – широко распространенный метод обеспечения быстрого доступа к информации, хранящейся во внешней памяти. Основная идея этого метода подобна методу цепочек, который рассматривается в 2.3.2. Только здесь, вместо записей таблицы организуется связный список блоков. Заголовок i-го блока содержит указатель на физический адрес (i+1)-го блока. Записи, хранящиеся в одном блоке, связывать друг с другом с помощью указателей не требуется. Сама таблица представляет собой таблицу указателей на блоки.

Такая структура оказывается вполне эффективной, если в выполняемой операции указывается значение ключа. В этом случае среднее количество обращений к блокам равно n/bk, где n – количество записей; b – количество записей в блоке; k – длина таблицы. Это в среднем в k раз меньше, чем в случае последовательного файла.

Чтобы вставить запись с ключом (запись с таким ключом отсутствует, так как значение ключа уникально), вычисляется хеш-функция по ключу, т. е. определяется строка таблицы указателей и просматривается соответствующая цепочка блоков. Для каждого блока осуществляется попытка вставки новой записи (при наличии свободного места в блоке). Если не удалось вставить ни в один блок цепочки, то у файловой системы запрашивается новый блок, который добавляется в конец цепочки и в него вставляется новая запись.

Чтобы удалить запись, также вычисляется строка таблицы указателей и находится запись в соответствующей цепочке блоков, а затем запись помечается как удаленная. Способы пометки записи здесь те же, что и в последовательных файлах. Если записи не являются закрепленными, то можно заменять удаляемую запись на последнюю запись в последнем блоке текущей цепочки. Если в результате такой замены последний блок стал пустым, то его можно вернуть файловой системе для повторного использования.

Еще одним распространенным способом эффективной организации файла записей, называемым *индексированным файлом*, является поддержание

файла в отсортированном (по значению ключа) порядке. Чтобы облегчить процедуру поиска, можно создать второй файл, называемый р а з р е ж е н н ы м индексом, который состоит из пар (x, b), где x – значение ключа, а b – физический адрес блока, в котором значение ключа первой записи равняется x. Этот индексный файл отсортирован по значению ключей.

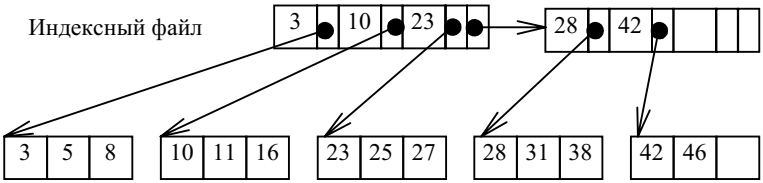


Рис. 18. Разреженный индекс

Чтобы отыскать запись с заданным ключом x, необходимо сначала просмотреть индексный файл, отыскивая в нем пару (x, b), а затем находят запись в блоке с физическим адресом b. Разработано несколько стратегий просмотра индексного файла. Простейшей из них является линейный поиск, более эффективным является двоичный поиск. Эти методы рассматриваются в 2.3.1. Для поиска записи необходимо считать один блок основного файла, и в зависимости от стратегии просмотра индексного файла просмотреть от n (при линейном поиске) до $\log_2(n + 1)$ (при двоичном поиске) блоков индексного файла, где n – общее количество блоков индексного файла.

Чтобы создать индексированный файл, записи сортируются по значениям их ключей, а затем распределяются по блокам в возрастающем порядке ключей. В каждый блок можно разместить столько записей, сколько в него помещается, но можно оставить место под записи, которые могут вставляться туда впоследствии (это уменьшает вероятность переполнения и, следовательно, обращение к смежным блокам). После распределения записей по блокам создается индексный файл. В нем также можно оставить место для новых индексов.

Чтобы вставить новую запись, с помощью индексного файла находят соответствующий блок основного файла. Если новая запись умещается в найденный блок, то она вставляется в него в правильной последовательности. Если новая запись становится первой записью в блоке, то необходима корректировка индексного файла.

Если новая запись не умещается в найденный блок, то возможно применение нескольких стратегий. Простейшая из них заключается в

том, чтобы перейти на следующий блок и узнать, можно ли последнюю запись найденного блока переместить в начало следующего. Если можно, то осуществляем перенос (освобождая место в найденном блоке), вставляем новую запись на подходящее место в найденный блок, корректируем индексный файл. Если следующий блок заполнен полностью или найденный блок является последним, то у файловой системы запрашиваем новый блок, помещаем его за найденным блоком, в новый блок вставляем новую запись и корректируем индексный файл.

Еще одним способом организации файла с использованием индексов является сохранение произвольного порядка записей в файле и создание другого файла, с помощью которого можно отыскивать требуемые записи. Этот файл называется плотным индексом. Плотный индекс состоит из пар (x, p), где p – указатель на запись с ключом x в основном файле. Эти пары отсортированы по значениям ключа. Поиск записи осуществляется подобно поиску с использованием разреженного индекса (рис. 18).

Если требуется вставить новую запись, отыскивают последний блок основного файла и туда вставляют новую запись. Если последний блок полностью заполнен, то запрашивают новый блок у файловой системы. Одновременно вставляют указатель на соответствующую запись в файле плотного индекса. Чтобы удалить запись, в ней просто устанавливают бит удаления и удаляют соответствующий указатель в плотном индексе.

1.4.2. B-деревья

1.4.2.1. Представление файлов B-деревьями

Как мы уже видели, очень эффективным является хранение множества данных в виде дерева. Поэтому в качестве типового способа организации внешней памяти стало B-дерево, которое обеспечивает при своем обслуживании относительно небольшое количество обращений к внешней памяти (рис. 19).

B-дерево представляет собой дерево поиска степени m, характеризующееся следующими свойствами:

1) корень либо является листом, либо имеет не менее двух потомков;

2) каждый узел, кроме корня и листьев, имеет от $(m/2)$ до m потомков;

3) все пути от корня до любого листа имеют одинаковую длину.

В каждой вершине будем хранить не более NumberOfItems записей. Также необходимо будет хранить текущее количество записей в вершине. Для удобства возврата назад к корню дерева будем запоминать для каждой вершины указатель на ее предка.

```
Type
  PBTreeNode = ^TBTreeNode;
  TBTreeNode = record                  {вершина дерева}
    Count: integer;                    {количество записей в вершине}
    PreviousNode: PBTreeNode;          {указатель на предка}
    Items: array[0..m+1] of record     {массив записей}
      Value: ItemType;
      NextNode: PBTreeNode;
    end;
  end;
TBTree = PBTreeNode;
```

У элемента Items[0] будет использоваться только поле NextNode. Дополнительный элемент Items[NumberOfItems+1] предназначен для обработки переполнения, о чем будет рассказано ниже, при описании алгоритма добавления элемента в B-дерево.

Поскольку дерево упорядочено, то

Items[1].Value<Items[2].Value<…<Items[Count].Value.

Указатель Items[i].NextNode указывает на поддерево элементов, больших Items[i].Value и меньших Items[i+1].Value. Понятно, что указатель Items[0].NextNode будет указывать на поддерево элементов, меньших Items[1].Value, а указатель

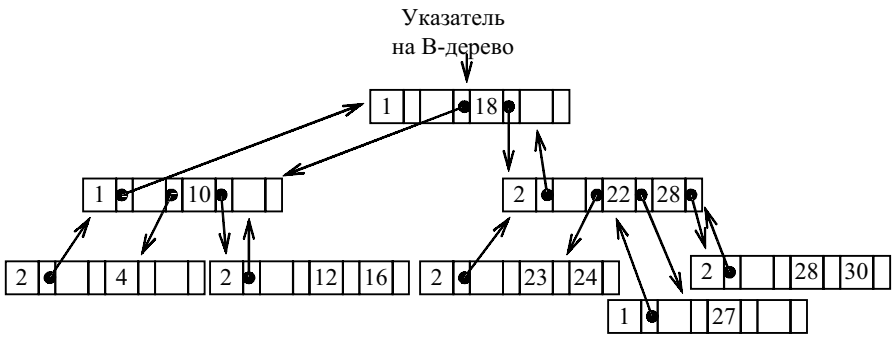


Рис. 19. B-дерево и его организация

`Items[Count].NextNode` – на поддерево элементов, больших `Items[Count].Value`.

У корневой вершины `PreviousNode` будет равен `nil`.

1.4.2.2. Основные операции

Основные операции, производимые с B-деревьями:

– поиск элемента;

– добавление элемента;

– удаление элемента.

При рассмотрении этих основных операций будут разбираться небольшие деревья, хотя в реальности B-деревья применяются при работе с большими массивами информации. Кроме того, для наглядности, на рисунках будут опускаться поля указателей.

Поиск элемента будем начинать с корневой вершины. Если искомый элемент присутствует в загруженной вершине, то завершаем поиск с положительным ответом, иначе загружаем следующую вершину, и так до тех пор, пока либо найдем искомый элемент, либо не окажется следующей вершины (пришли в лист B-дерева).

Посмотрим на примере, как это реализуется (рис. 20).

Будем искать элемент *11*. Сначала загрузим корневую вершину. Эта вершина содержит элементы *5* и *13*. Искомый элемент больше *5*, но меньше *13*. Значит, следует идти по ссылке, идущей от элемента *5*. Загружаем следующую вершину (с элементами *8* и *10*). Эта вершина тоже не содержит искомого элемента. Замечаем, что *11* больше *10*, следовательно, двигаемся по ссылке, идущей от элемента *10*. Загружаем соответствующую вершину (с элементами *11* и *12*), в которой и находим искомый элемент. Итак, в этом примере, чтобы найти элемент, потребовалось три раза обратиться к внешней памяти для чтения очередной вершины.

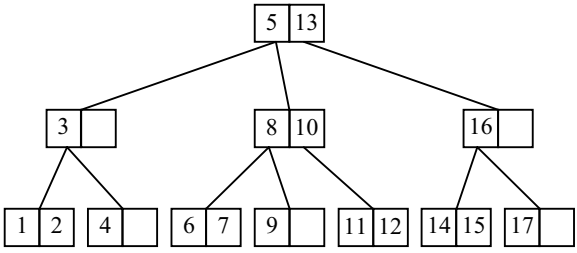


Рис. 20. Поиск элемента в B-дереве

Если бы в примере осуществлялся поиск, допустим, элемента *18*, то, просмотрев три вершины (последняя с элементом *17*), было бы обнаружено, что от элемента *17* нет ссылки на поддерево с элементами, большими, чем *17*, и на основании этого сделали бы вывод, что элемента 18 в дереве нет.

Теперь приведем точно сформулированный алгоритм поиска элемента `Item` в B-дереве, предположив, что функция `LookFor` возвращает номер первого большего или равного элемента вершины (фактически производит поиск в вершине).

```
function BTree_Search(Item: ItemType, BTree: PBTreeNode): boolean;
var
  CurrentNode: PBTreeNode;
  Position: integer;
begin
 BTree_Search := False;
  CurrentNode := BTree;
  repeat
    {Ищем в текущей вершине}
    Position := LookFor(CurrentNode, Item);
    if (CurrentNode.Count >= Position) and
       (CurrentNode.Items[Position].Value = Item) then begin
      {Элемент найден}
      BTree_Search := True;
      Exit;
    end;
    if CurrentNode.Items[Position-1].NextNode = nil then
      {Элемент не найден и дальше искать негде}
      break
    else
      {Элемент не найден, но продолжаем поиск дальше}
      CurrentNode := CurrentNode.Items[Position-1].NextNode;
  until False;
end;
```

Здесь пользуемся тем, что, если ключ лежит между `Items[i].Value` и `Items[i+1].Value`, то во внутреннюю память надо подкачать вершину, на которую указывает `Items[i].NextNode`.

Заметим, что для ускорения поиска ключа внутри вершины (функция `LookFor`), можно воспользоваться бинарным поиском (см. 2.3.1.2).

Учитывая то, что время обработки вершины есть величина постоянная, пропорциональная размеру вершины, временная сложность $T(n)$ алгоритма поиска в B-дереве будет пропорциональна $O(h)$, где h – глубина дерева.

Теперь рассмотрим *добавление элемента* в B-дерево

Для того чтобы B-дерево можно было считать эффективной структурой данных для хранения множества значений, необходимо, чтобы каждая вершина заполнялась хотя бы наполовину. Дерево строится снизу. Это означает, что любой новый элемент добавляется в лист. Если при этом произойдет переполнение (на этот случай в каждой вершине зарезервирован лишний элемент), т. е. число элементов в вершине превысит `NumberOfItems`, то надо будет разделить вершину на две вершины и вынести средний элемент на верхний уровень. Может случиться, что при этой операции на верхнем уровне тоже получится переполнение, что вызовет еще одно деление. В худшем случае эта волна делений докатится до корня дерева.

В общем виде алгоритм добавления элемента `Item` в B-дерево можно описать следующей последовательностью действий:

1. Поиск среди листьев вершины `Node`, в которую следует произвести добавление элемента `Item`.

2. Добавление элемента `Item` в вершину `Node`.

3. Если `Node` содержит больше, чем `NumberOfItems` элементов (произошло переполнение), то:

– делим `Node` на две вершины, не включая в них средний элемент;

– `Item` := средний элемент `Node`;

– `Node` := `Node.PreviousNode`;

– переходим к пункту 2.

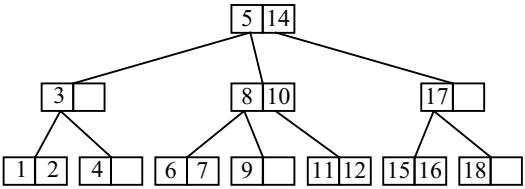
Заметим, что при обработке переполнения надо отдельно обработать случай, когда `Node` – корень, так как в этом случае `Node.PreviousNode = nil`.

Рассмотрим пример. Возьмем дерево (рис. 21, *а*) и добавим в него элемент *13*.

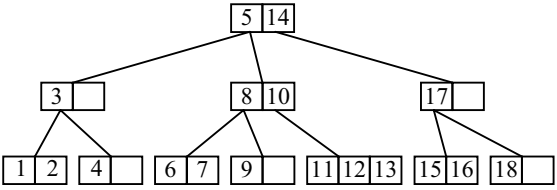
Двигаясь от корня, найдем лист, в который следует добавить искомый элемент. Таким узлом в нашем случае окажется лист, содержащий элементы *11* и *12*. Добавим в него элемент *13* (рис. 21, *б*).

Понятно, что при этом получается переполнение. При его обработке вершина, содержащая элементы *11*, *12* и *13*, разделится на две части: вершину с элементом *11* и вершину с элементом *13*, – а средний элемент *12* будет вынесен на верхний уровень (рис. 21, *в*).

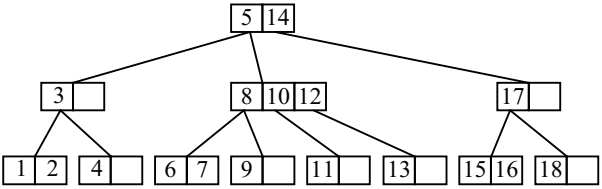
а)



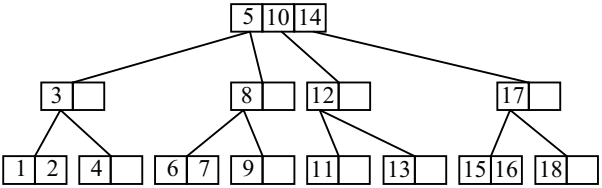
б)



в)



г)



д)

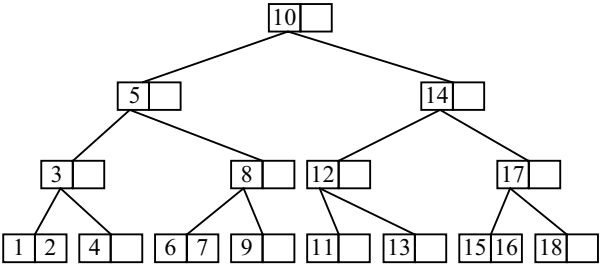


Рис. 21. Добавление элемента в B-дерево

Опять получилось переполнение, при обработке которого вершина, содержащая элементы *8*, *10* и *12* разделится на две вершины: вершину с элементом *8* и вершину с элементом *12*, – а средний элемент *10* будет вынесен на верхний уровень (рис. 21, *г*).

Получилось переполнение в корне дерева. Как оговаривалось ранее, этот случай надо обработать отдельно. Это связано с тем, что здесь необходимо создать новый корень, в который во время деления будет вынесен средний элемент. Теперь полученное дерево не имеет переполнения (рис. 21, *д*).

В этом случае, как и при поиске, время обработки вершины есть величина постоянная, пропорциональная размеру вершины, а значит, временная сложность $T(n)$ алгоритма добавления в B-дерево будет также пропорциональна $O(h)$, где h – глубина дерева.

Удаление элемента из B-дерева предполагает успешный предварительный поиск вершины, содержащей искомый элемент. Если такая вершина не найдена, то удаление не производится.

При удалении, так же как и при добавлении, необходимо делать так, чтобы число элементов в вершине лежало между `NumberOfItems/2` и `NumberOfItems`. Если при добавлении могла возникнуть ситуация переполнения вершины, то при удалении можно получить порожнюю вершину. Это означает, что число элементов в вершине оказалось меньше `NumberOfItems/2`. В этом случае надо посмотреть, нельзя ли занять у соседней вершины слева или справа («перелить» часть элементов от нее) некоторое количество элементов так, чтобы их (элементов) стало поровну и не было порожних вершин. Такое переливание возможно, если суммарное число элементов у данной вершины и соседней больше или равно `NumberOfItems`.

Если переливание невозможно, то объединяем данную вершину с соседней. При этом число элементов в родительской вершине уменьшится на единицу и может статься, что опять получаем порожнюю вершину. В худшем случае волна объединений дойдет до корня. Случай корня надо обрабатывать особо, потому что в корне не может быть менее одного элемента. Поэтому, если в корне не осталось ни одного элемента, надо сделать корнем ту единственную вершину, на который ссылается корень через ссылку `Node.Items[0].NextNode`, а старый корень удалить.

Приведем алгоритм удаления элемента `Item` из B-дерева:

1. Поиск вершины Node, которая содержит элемент Item. Если такой вершины нет, то удаление невозможно.

2. Если Node – лист, то удалить из Node элемент Item, иначе заменить элемент Item узла Node на самый левый элемент правого поддерева элемента Item (тем самым сохраняется упорядоченность дерева: аналогично можно заменять элемент Item на самый правый элемент левого поддерева), Node := вершина, из которой был взят элемент для замены.

3. Если в Node меньше NumberOfItems/2 элементов (получилась порожняя вершина), то:

– выбрать две соседние вершины Neighbor1 и Neighbor2: одна из них – Node, вторая – ее левая или правая ближайшие;

– если в Neighbor1 и Neighbor2 в сумме меньше чем NumberOfItems элементов, то слить вершины Neighbor1 и Neighbor2 иначе перераспределить элементы между Neighbor1 и Neighbor2 так, чтобы количество элементов в них не отличалось больше чем на единицу, Node := родительская вершина Neighbor1 и Neighbor2.

4. Если Node не корень дерева, то перейти к пункту 3.

5. Если корень дерева пуст, то удалить его. Новым корнем дерева будет та единственная вершина, на которую осталась ссылка в старом корне.

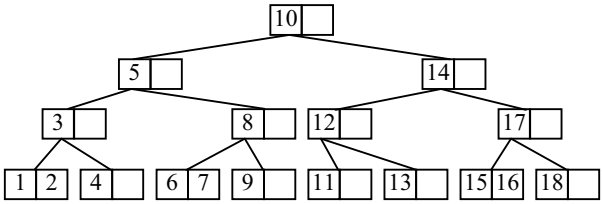
Рассмотрим работу алгоритма на примере. Возьмем дерево, получившееся в предыдущем примере после добавления элемента (рис. 22, *а*).

Будем удалять из этого дерева элемент *10*. Сначала надо, двигаясь от корня, найти вершину дерева, содержащую этот элемент. Он находится в корне. Теперь поскольку элемент находится не в листовом узле, надо заменить его, например, на самый левый элемент правого поддерева. Делаем один шаг вправо и попадаем в корень правого поддерева. Теперь пока не достигнем листа, переходим на самого левого потомка.

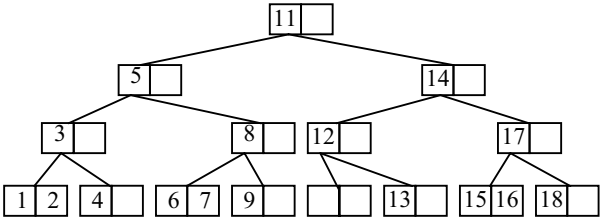
Таким образом, найдем вершину, из которой позаимствуем элемент для замены удаленного элемента *10*. В найденной вершине возьмем самый левый элемент. Таковым оказывается *11*. Произведем замену элемента *10* на *11* и удалим элемент *11* (рис. 22, *б*).

Теперь проверим вершину, из которой был удален элемент, не стала ли она порожней. Число элементов в вершине равно нулю. Это меньше половины размера вершины. Следовательно, вершина порожняя. Выберем две соседние вершины: одна – та, из которой было удалено *11*,

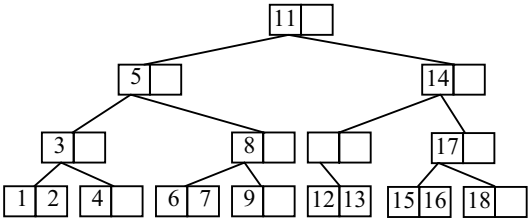
а)



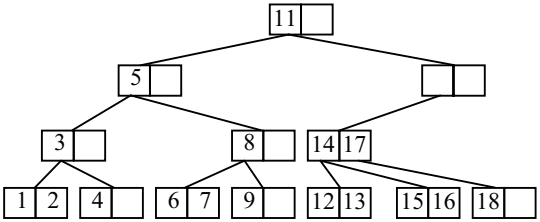
б)



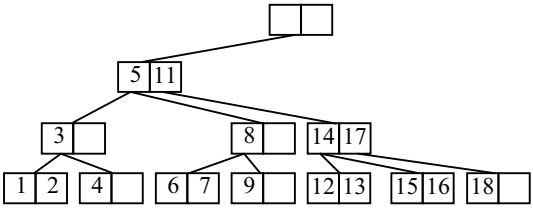
в)



г)



д)



е)

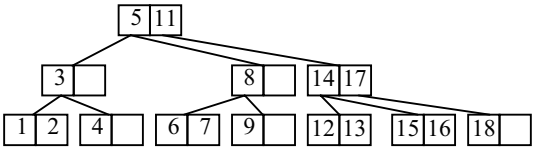


Рис. 22. Удаление элемента из B-дерева

вторая – соседняя вершина с числом *13*. Суммарное число элементов в этих вершинах не превышает максимального размера вершины, значит, нам следует произвести слияние вершин. При слиянии элемент *12* родителя попадает в результирующий узел и удаляется из родителя (рис. 22, *в*).

Теперь переходим к родителю, из которого был удален элемент, и проверяем, не является ли он порожним. Опять имеем порожнюю вершину, и необходимо выбрать две соседние вершины (в данном случае: вершину, из которой был удален элемент *12* и вершину с элементом *17*) и произвести слияние двух вершин. При слиянии получается вершина с элементами *14* и *17*, где *14* позаимствовано у родителя. Естественно, элемент *14* из родителя удаляется (рис. 22, *г*).

Поскольку опять был удален элемент родителя, то переходим к родителю и повторяем процесс проверки и слияния. Получаем вершину с элементами *5* и *11*, где *5* позаимствовано у родителя и удалено из него (рис. 22, *д*).

Опять переходим к родителю. Поскольку родитель оказывается корнем, то цикл обработки порожних вершин заканчивается. Осталось проверить, не пуст ли корень дерева. Корень дерева пуст, поэтому его можно удалить. Новым корнем дерева будет та единственная вершина, на которую есть ссылка в старом корне. Это вершина с элементами *5* и *11* (рис. 22, *е*).

В этом случае, как и при поиске, время обработки вершины есть величина постоянная, пропорциональная размеру вершины, а значит, временная сложность $T(n)$ алгоритма удаления из B-дерева будет также пропорциональна $O(h)$, где h – глубина дерева.

1.4.2.3. Общая оценка B-деревьев

Итак, как говорилось ранее, у B-деревьев есть своя сфера применения: хранение настолько больших массивов информации, что их невозможно целиком разместить в выделяемой оперативной памяти, но требуется обеспечить быстрый доступ к ним. В таких случаях B-деревья являются хорошим средством программно ускорить доступ к данным.

Ярким примером практического применения B-деревьев является файловая система NTFS, где B-деревья применяются для ускорения поиска имен в каталогах. Если сравнить скорость поиска в этой файловой системе и в обычной FAT на примере поиска на жестком диске большого объема или в каталоге, содержащем очень много файлов, то можно

будет констатировать превосходство NTFS. Поиск файла в каталоге всегда предшествует запуску программы или открытию документа.

B-деревья обладают прекрасным качеством: во всех трех операциях над данными (поиск/удаление/добавление) они обеспечивают сложность порядка $O(h)$, где h – глубина дерева. Это значит, что чем больше узлов в дереве и чем сильнее дерево ветвится, тем меньшую часть узлов надо будет просмотреть, чтобы найти нужный элемент. Попробуем оценить зависимость временной сложности операций $T(h)$ от высоты дерева h.

Число элементов в вершине есть величина вероятностная с постоянным математическим ожиданием MK. Математическое ожидание числа вершин равно $n/MK \sim n$, где n – число элементов, хранимых в B-дереве. Это дает сложность $T(h) \sim O(\log n)$, а это очень хороший результат.

Поскольку вершины могут заполняться не полностью (иметь менее `NumberOfItems` элементов), то можно говорить о коэффициенте использования памяти. Существуют доказательства, что память будет использоваться в среднем на $\ln 2 \cdot 100\% \approx 69{,}3\%$.

В отличие от сбалансированных деревьев, которые рассматриваются далее, B-деревья растут не вниз, а вверх. Поэтому (и из-за разной структуры узлов) алгоритмы включения или удаления принципиально различны, хотя цель их в обоих случаях одна – поддерживать сбалансированность дерева.

Идея внешнего поиска с использованием техники B-деревьев была предложена в 1970 году Р. Бэйером и Э. Мак-Крэйтом и независимо от них примерно в то же время М. Кауфманом. Естественно, что за это время было предложено ряд усовершенствований B-деревьев, связанных с увеличением коэффициента использования памяти и уменьшением общего количества расщеплений.

Одно из таких усовершенствований было предложено Р. Бэйером и Э. Мак-Крэйтом и заключалось в следующем. Если вершина дерева переполнена, то прежде чем расщеплять эту вершину, следует посмотреть, нельзя ли «перелить» часть элементов соседям слева и справа. При использовании такой методики уменьшается общее количество расщеплений и увеличивается коэффициент использования памяти.