МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

ГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«САМАРСКИЙ ГОСУДАРСТВЕННЫЙ АЭРОКОСМИЧЕСКИЙ
УНИВЕРСИТЕТ имени академика С.П. КОРОЛЕВА»
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

В. Д. ЕЛЕНЕВ, М. Ю. ГОГОЛЕВ

АЛГОРИТМИЧЕСКИЕ ЯЗЫКИ И ТЕХНОЛОГИИ ПРОГРАММИРОВАНИЯ НА ЯЗЫКАХ ВЫСОКОГО УРОВНЯ

Утверждено Редакционно-издательским советом университета
в качестве курса лекций

С А М А Р А
Издательство СГАУ
2010

УДК СГАУ: 004.43(075)
ББК 32.973я7
Е504

Еленев В. Д.
Е504 **Алгоритмические языки и технологии программирования на языках высокого уровня:** курс лекций / *В.Д. Еленев, М.Ю. Гоголев.* - Самара: Изд-во Самар. гос. аэрокосм. ун-та, 2010. - 224 с.

ISBN 978-5-7883-0762-6

Рекомендован для студентов высших учебных заведений, обучающихся по направлению 160400.68 «Ракетные комплексы и космонавтика» магистерская программа «Проектирование и конструирование космических мониторинговых и транспортных систем».

Разработан на кафедре летательных аппаратов СГАУ.

УДК СГАУ: 004.43(075)
ББК 32.973я7

ISBN 978-5-7883-0762-6

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Они служат базовыми элементами любой машинной программы. В организации структур данных и процедур их обработки заложена возможность проверки правильности работы программы.

Никлас Вирт.

Без понимания структур данных и алгоритмов невозможно создать сколько-нибудь серьезный программный продукт. И слова эпиграфа служат тому подтверждением. Поэтому главная задача данного учебного пособия заключалась в следующем:

- показать все разнообразие имеющихся структур данных, представление их в памяти на физическом уровне, т.е. "как это сделано внутри", и логическом уровне, или как эти структуры реализованы в языках программирования;
- выполняемые над ними операции физического и логического уровней;
- показать значение структурного подхода к разработке алгоритмов, продемонстрировать порядок разработки алгоритмов наиболее, по мнению авторов, интересных задач.

Нельзя сказать, что такие вопросы не рассматривались в литературе, но с полной уверенностью можно отметить, что так сконцентрировано, так подробно и в доступной для понимания форме, с таким количеством демонстрационных примеров ни в каком из известных изданиях не сделано.

В пособии приводится классификация структур данных, обширная информация о физическом и логическом представлении структур данных всех классов памяти ПВМ: простых, статических, полустатических, динамических; исчерпывающая информация об операциях над всеми перечисленными структурами. Приведено достаточно большое количество алгоритмов особенно важных операций, реализованных в виде процедур и функций, написанных на Turbo Pascal, которые могут быть применены как "заготовки" в самостоятельных разработках студентов и программистов.

1 СТРУКТУРЫ ДАННЫХ И АЛГОРИТМЫ

1.1 Понятие структур данных и алгоритмов

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

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

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

Имя константы или переменной помогает программисту, но компьютеру оно ни о чем не говорит. Компилятор же, транслирующий текст программы в двоичный код, связывает каждый идентификатор с определенным адресом памяти. Но для того чтобы компилятор смог это выполнить, нужно сообщить о "типе" каждой именованной величины. Человек, решающий какую-нибудь задачу "вручную", обладает интуитивной способностью быстро разобраться в типах данных и тех операциях, которые для каждого типа справедливы. Так, например, нельзя извлечь квадратный корень из слова или написать число с заглавной буквы. Одна из причин, позволяющих легко провести такое распознавание, состоит в том, что слова, числа и другие обозначения выглядят по-разному. Однако для компьютера все типы данных сводятся в конечном счете к последовательности битов,

поэтому различие в типах следует делать явным.

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

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

Структура данных относится, по существу, к "пространственным" понятиям: ее можно свести к схеме организации информации в памяти компьютера. Алгоритм же является соответствующим процедурным элементом в структуре программы - он служит рецептом расчета. Первые алгоритмы были придуманы для решения численных задач типа умножения чисел, нахождения наибольшего общего делителя, вычисления тригонометрических функций и других. Сегодня в равной степени важны и нечисленные алгоритмы; они разработаны для таких задач, как, например, поиск в тексте заданного слова, планирование событий, сортировка данных в указанном порядке и т.п. Нечисленные алгоритмы оперируют с данными, которые не обязательно являются числами; более того, не нужны никакие глубокие математические понятия, чтобы их конструировать или понимать. Из этого, однако, вовсе не следует, что в изучении таких алгоритмов математике нет места; напротив, точные, математические методы необходимы при поиске наилучших решений нечисленных задач при доказательстве правильности этих решений.

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

1.2 Информация и её представление в памяти

Начиная изучение структур данных, или информационных структур, необходимо ясно установить, что понимается под информацией, как информация передается и как она физически размещается в памяти вычислительной машины, с несколько упрощенной точки зрения для того, чтобы изучающий информационные структуры мог понять, что такое информация и какова ее физическая природа.

1.2.1 Природа информации

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

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

$$H = -\sum_{i=1}^{n} P_i * \log P_i .$$

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

1.2.2. Хранение информации

В цифровых вычислительных машинах можно выделить три основных вида запоминающих устройств: сверхоперативная, оперативная и внешняя память.

Сверхоперативная память строится обычно на регистрах. Регистры используются для временного хранения и преобразования информации.

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

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

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

Внешняя память служит прежде всего для долговременного хранения данных. Характерным для данных на внешней памяти является то, что они могут сохраняться там даже после завершения создавшей их программы, и могут быть впоследствии многократно использованы той же программой при повторных ее запусках или другими программами. Внешняя память используется также для хранения самих программ, когда они не выполняются. Поскольку стоимость внешней памяти значительно меньше оперативной, а объем значительно больше, то еще одно назначение внешней памяти временное хранение тех кодов и данных выполняемой программы, которые не используются на данном этапе ее выполнения. Активные коды выполняемой программы и обрабатываемые ею на данном этапе данные должны обязательно быть размещены в оперативной памяти, так как прямой обмен между внешней памятью и операционными устройствами (регистрами) невозможен.

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

1.3 Системы счисления

Чтобы обеспечить соответствующую основу для изучения структур данных следует обсудить существующие типы систем счислений: позиционные и непозиционные.

1.3.1. Непозиционные системы счисления

Числа используются для символического представления количества объектов. Очень простым методом представления количества является использование одинаковых значков. В такой системе между значками и пересчитываемыми объектами устанавливается взаимно однозначное соответствие. Например, шесть объектов могут быть представлены как ****** или 111111. Такая система становится очень неудобной, если попытаться с ее помощью представить большое количество объектов.

Системы счисления, подобные римской, обеспечивают частичное решение проблемы представления большого количества объектов. В римской системе дополнительные символы служат для представления групп значков. Например, если принять, что I=*, Y=IIII, X=YY, L=XXXXX и т.д. Заданная величина представляется с помощью комбинирования символов в соответствии с рядом правил, которые в некоторой степени зависят от положения символа в числе. Недостатком системы, которая с самого начала основывается на группировании некоторого множества символов с целью формирования нового символа, является то обстоятельство, что для представления очень больших количеств требуется очень много уникальных символов.

1.3.2 Позиционные системы счисления

В позиционной системе счисления используется конечное число R уникальных символов. Величину R часто называют основанием системы счисления. В позиционной системе количество представляется как самими символами, так и их позицией в записи числа. Система счисления с основанием десять, или десятичная система является позиционной. Рассмотрим, например, число 1303. Его можно представить в виде:

$$1*10^3 + 3*10^2 + 0*10^1 + 3*10^0.$$

В позиционной системе могут быть представлены и дробные числа. Например, одна четвертая записывается в виде 0.25, что интерпретируется как:

$$2*10^{-1} + 5*10^{-2}.$$

Другой пример позиционной системы счисления - двоичная система. Двоичное число 11001.101 представляет то же самое количество, что и десятичное число 26.625. Разложение данного двоичного числа в соответствии с его позиционным представлением следующее:

$1*2^4+1*2^3+0*2^1+1*2^0+1*2^{-1}+0*2^{-2}+1*2^{-3}=16+8+1+0.5+0.125=26.625.$

Наиболее часто встречаются системы счисления имеющие основание 2,8,10 и 16, которые обычно называют двоичной, восьмеричной, десятичной и шестнадцатеричной системами, соответственно. Вся вычислительная техника работает в двоичной системе счисления, так как базовые элементы вычислительной техники имеют два устойчивых состояния. Восьмеричная и шестнадцатеричная системы используются для удобства работы с большими двоичными числами.

1.3.3 Изображение чисел в позиционной системе счисления

Изображение чисел в любой позиционной системе счисления с натуральным основанием R (R >1) базируется на представлении их в виде произведения целочисленной степени m основания R на полином от этого основания :

$$Ar = R^m * \sum_{i=1}^{n} a[i] * R$$

где: a[i] { 0,1,..., R-1 } - цифры R - ичной системы счисления ;

n - количество разрядов (разрядность), используемых для представления числа;

R - основание системы счисления;

m {..., -2, -1, 0,+1,+2,...} - порядок числа;

R^{-1} - позиционный вес i - го разряда числа.

Так в десятичной (R=10) системе для представления чисел используются цифры a=(0,1,...9); в двоичной (R=2) - a=(0,1), в шестнадцатеричной (R=16), a=(0,1....9,A,B,C,D,E,F), где прописные латинские буквы A..F эквивалентны соответственно числам 10..15 в десятичной системе. Например,

1) $815=10^3*(8*10^{-1}+1*10^{-2}+5*10^{-3})=8*10^2+1*10^1+5*10^0$;

2) $8.15=10^1*(8*10^{-1}+1*10^{-2}+5*10^{-3})=8*10^0+1*10^{-1}+5*10^{-2}$;

3) $0.0815=10^{-1}*(8*10^{-1}+1*10^{-2}+5*10^{-3})=8*10^{-2}+1*10^{-3}+5*10^{-4}$.

1.3.4 Перевод чисел из одной системы счисления в другую

При переводе целого числа (целой части числа) из одной системы счисления в другую исходное число (или целую часть) надо разделить на основание системы счисления, в которую выполняется перевод. Деление выполнять, пока частное не станет меньше основания новой системы

счисления. Результат перевода определяется остатками от деления: первый остаток дает младшую цифру результирующего числа, последнее частное от деления дает старшую цифру.

При переводе правильной дроби из одной системы счисления в другую систему счисления дробь следует умножать на основание системы счисления, в которую выполняется перевод. Полученная после первого умножения целая часть является старшим разрядом результирующего числа. Умножение вести до тех пор пока произведение станет равным нулю или будет получено требуемое число знаков после разделительной точки.

Например,

1) Перевести дробное число 0.243 из десятичной системы счисления в двоичную.

   $0.243_{10} \rightarrow 0.0011111_2$.

Проверка: $0.0011111 = 0*2^{-1}+0*2^{-2}+1*2^{-3}+1*2^{-4}+1*2^{-5}+1*2^{-6}+1*2^{-7}= 0{,}2421875$

2) Перевести целое число 164 из десятичной системы счисления в двоичную систему.

   $164_{10} \rightarrow 10100100_2$

Проверка: $10100100 = 1*2^7+0*2^6+1*2^5+0*2^4+0*2^3+1*2^2+0*2^1+0*2^0= =128+32+4=164$

При переводе смешанных чисел целая и дробная части числа переводятся отдельно.

1.4 Классификация структур данных

Теперь можно дать более конкретное определение данного на машинном уровне представления информации. Независимо от содержания и сложности любые данные в памяти ЭВМ представляются последовательностью двоичных разрядов, или битов, а их значениями являются соответствующие двоичные числа.

Данные, рассматриваемые в виде последовательности битов, имеют очень простую организацию или, другими словами, слабо структурированы. Для человека описывать и исследовать сколько-нибудь сложные данные в терминах последовательностей битов весьма неудобно. Более крупные и содержательные, нежели бит, "строительные блоки" для организации произвольных данных получаются на основе понятия "структуры данных".

Под СТРУКТУРОЙ ДАННЫХ в общем случае понимают множество элементов данных и множество связей между ними. Такое определение охватывает все возможные подходы к структуризации данных, но в каждой конкретной задаче используются те или иные его аспекты.

Поэтому вводится дополнительная классификация структур данных, направления которой соответствуют различным аспектам их рассмотрения. Прежде чем приступать к изучению конкретных структур данных, дадим их общую классификацию по нескольким признакам.

Понятие "ФИЗИЧЕСКАЯ структура данных" отражает способ физического представления данных в памяти машины и называется еще структурой хранения, внутренней структурой или структурой памяти.

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

Различаются ПРОСТЫЕ (базовые, примитивные) структуры (типы) данных и ИНТЕГРИРОВАННЫЕ (структурированные, композитные, сложные). Простыми называются такие структуры данных, которые не могут быть расчленены на составные части, большие, чем биты. С точки зрения физической структуры важным является то обстоятельство, что в данной машинной архитектуре, в данной системе программирования мы всегда можем заранее сказать, каков будет размер данного простого типа и какова структура его размещения в памяти. С логической точки зрения простые данные являются неделимыми единицами.

Интегрированными называются такие структуры данных, составными частями которых являются другие структуры данных - простые или в свою очередь интегрированные. Интегрированные структуры данных конструируются программистом с использованием средств интеграции данных, предоставляемых языками программирования.

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

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

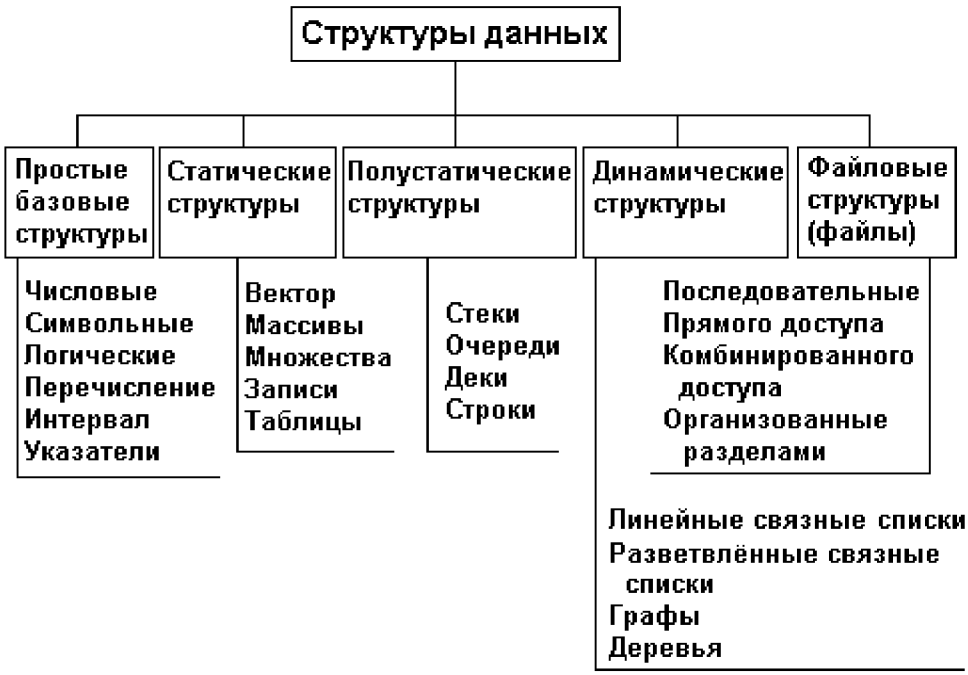


Рис. 1.1. Классификация структур данных

Важный признак структуры данных - характер упорядоченности ее элементов. По этому признаку структуры можно делить на ЛИНЕЙНЫЕ И НЕЛИНЕЙНЫЕ структуры.

В зависимости от характера взаимного расположения элементов в памяти линейные структуры можно разделить на структуры с ПОСЛЕДОВАТЕЛЬНЫМ распределением элементов в памяти (векторы, строки, массивы, стеки, очереди) и структуры с ПРОИЗВОЛЬНЫМ СВЯЗНЫМ распределением элементов в памяти (односвязные, двусвязные списки). Пример нелинейных структур - многосвязные списки, деревья, графы.

В языках программирования понятие "структуры данных" тесно связано с понятием "типы данных". Любые данные, т.е. константы, переменные, значения функций или выражения, характеризуются своими типами. Информация по каждому типу однозначно определяет :

1) структуру хранения данных указанного типа, т.е. выделение памяти и представление данных в ней, с одной стороны, и интерпретацию двоичного представления, с другой;
2) множество допустимых значений, которые может иметь тот или иной объект описываемого типа;
3) множество допустимых операций, которые применимы к объекту описываемого типа.

В последующих главах данного пособия рассматриваются структуры данных и соответствующие им типы данных. При описании базовых (простых) типов и при конструировании сложных типов ориентировались в основном на язык PASCAL. Этот язык использовался и во всех иллюстративных примерах. PASCAL был создан Н.Виртом специально для иллюстрирования структур данных и алгоритмов поэтому традиционно

используется для этих целей. Читатель, знакомый с любым другим процедурным языком программирования общего назначения (C, FORTRAN, ALGOL, PL/1 и т.д.), найдет аналогичные средства в известном ему языке.

1.5 Операции над структурами данных

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

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

Например, в PL/1 оператор DECLARE N FIXED DECIMAL приведет к выделению адресного пространства для переменной N во время выполнения программы. В FORTRAN (Integer I), в PASCAL (I:integer), в C (int I) в результате описания типа будет выделена память для соответствующих переменных. Для структур данных, объявленных в программе память выделяется автоматически средствами системы программирования либо на этапе компиляции, либо при активизации процедурного блока, в котором объявляются соответствующие переменные. Программист может и сам выделять память для структур данных, используя имеющиеся в системе программирования процедуры и функции для выделения и освобождения памяти. В объектно-ориентированных языках программирования при разработке нового объекта для него должны быть определены процедуры его создания и уничтожения.

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

Операция уничтожения структур данных противоположна по своему действию операции создания. Некоторые языки, такие как BASIC, FORTRAN, не дают возможности программисту уничтожать созданные структуры данных. В языках PL/1, C, PASCAL структуры данных, имеющиеся внутри блока, уничтожаются в процессе выполнения программы при выходе из этого блока. Операция уничтожения помогает эффективно использовать память.

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

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

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

1.6 Структурность данных и технология программирования

Большинство авторов публикаций, посвященных структурам и организации данных, делают основной акцент на том, что знание структуры данных позволяет организовать их хранение и обработку максимально эффективным образом - с точки зрения минимизации затрат как памяти, так и процессорного времени. Другим не менее, а может быть, и более важным преимуществом, которое обеспечивается структурным подходом к данным, является возможность структурирования сложного программного изделия. Современные промышленно выпускаемые программные пакеты - изделия чрезвычайно сложные, объем их исчисляется тысячами и миллионами строк кода, а трудоемкость разработки - сотнями человеко-лет. Естественно, что разработать такое программное изделие "все сразу" невозможно, оно должно быть представлено в виде какой-то структуры - составных частей и связей между ними. Правильное структурирование изделия дает возможность на каждом этапе разработки сосредоточить внимание разработчика на одной обозримой части изделия или поручить реализацию разных его частей разным исполнителям.

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

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

инструментальными средствами (например, одним оператором выбранного языка программирования).

Другой подход к структуризации основывается на данных. Программисту, который хочет, чтобы его программа имела реальное применение в некоторой прикладной области, не следует забывать о том, что программирование - это обработка данных. В программах можно изобретать сколь угодно замысловатые и изощренные алгоритмы, но у реального программного изделия всегда есть Заказчик. У Заказчика есть входные данные, и он хочет, чтобы по ним были получены выходные данные, а какими средствами это обеспечивается - его не интересует. Таким образом, задачей любого программного изделия является преобразование входных данных в выходные. Инструментальные средства программирования предоставляют набор базовых (простых, примитивных) типов данных и операции над ними. Интегрируя базовые типы, создаются более сложные типы данных и определяются новые операции над сложными типами. Можно здесь провести аналогию со строительными работами: базовые типы - "кирпичики", из которых создаются сложные типы - "строительные блоки". Полученные на первом шаге композиции "строительные блоки" используются в качестве базового набора для следующего шага, результатом которого будут еще более сложные конструкции данных и еще более мощные операции над ними и т.д. В идеале последний шаг композиции дает типы данных, соответствующие входным и выходным данным задачи, а операции над этими типами реализуют в полном объеме задачу проекта.

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

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

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

доступа пользователя ограничиваются лишь заведомо корректными входными точками, следовательно, снижается и вероятность ошибок.

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

Технология баз данных развивалась параллельно с технологией языков программирования и не всегда согласованно с ней. Отчасти этим, а отчасти и объективными различиями в природе задач, решаемых системами управления базами данных (СУБД) и системами программирования, вызваны некоторые терминологические и понятийные различия в подходе к данным в этих двух сферах. Ключевым понятием в СУБД является понятие модели данных, в основном тождественное понятию логической структуры данных. Отметим, что физическая структура данных в СУБД не рассматривается вообще. Но сами СУБД являются программными пакетами, выполняющими отображение физической структуры в логическую (в модель данных). Для реализации этих пакетов используются те или иные системы программирования, разработчики СУБД, следовательно, имеют дело со структурами данных в терминах систем программирования. Для пользователя же внутренняя структура СУБД и физическая структура данных совершенно прозрачна; он имеет дело только с моделью данных и с другими понятиями логического уровня.

2. ПРОСТЫЕ СТРУКТУРЫ ДАННЫХ

Простые структуры данных называют также примитивными или базовыми структурами. Эти структуры служат основой для построения более сложных структур. В языках программирования простые структуры описываются простыми (базовыми) типами. К таким типам относятся: числовые, битовые, логические, символьные, перечисляемые, интервальные, указатели. В дальнейшем изложении мы ориентируемся в основном на язык PASCAL и его реализации в среде MS DOS. Структура простых типов PASCAL приведена на рис 2.1 (через запятую указан размер памяти в байтах, требуемый для размещения данных соответствующего типа).

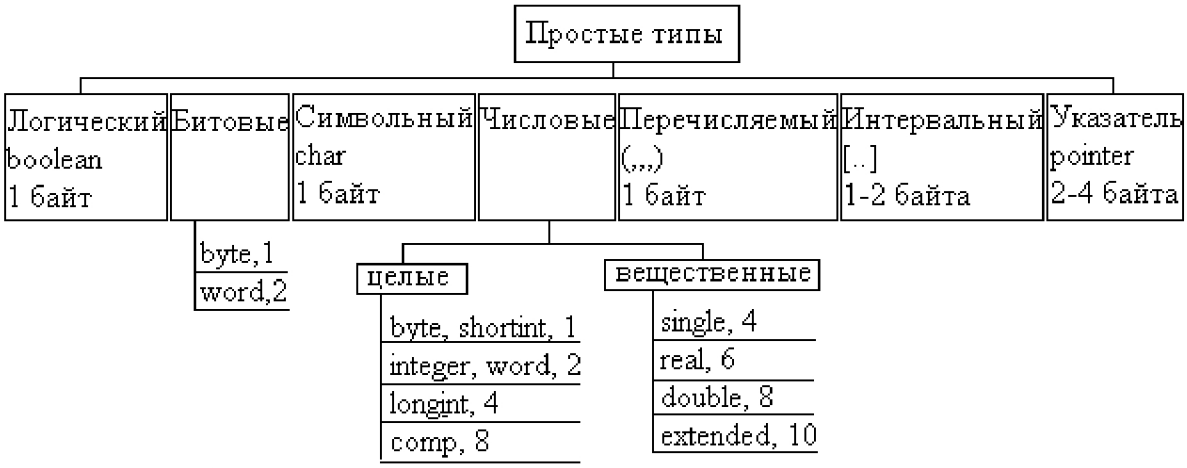


Рис. 2.1. Структура простых типов PASCAL

В других языках программирования набор простых типов может несколько отличаться от указанного. Размер же памяти, необходимый для данных того или иного типа может быть разным не только в разных языках программирования, но и в разных реализациях одного и того же языка, например, целый тип в языке CI.

2.1. Числовые типы

2.1.1.Целые типы

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

ПРЕДСТАВЛЕНИЕ В ПАМЯТИ. Для представления чисел со знаком в ряде компьютеров был использован метод, называемый методом знака и значения. Обычно для знака отводится первый (или самый левый) бит двоичного числа затем следует запись самого числа.

Например, +10 и -15 в двоичном виде можно представить так:

Число	Знаковый бит	Величина

+10	0	0001010
- 15	1	0001111

Отметим, что по соглашению 0 используется для представления знака плюс и 1 - для минуса. Такое представление удобно для программистов, т.к. легко воспринимается; но не является экономичным, поскольку при выполнении операций сложения и вычитания необходимо вначале определять знак каждого числа.

Например, сложение чисел +6 и -7 на самом деле подразумевает операцию вычитания, а вычитание -6 из +7 операцию сложения. Для анализа знакового бита требуется особая схема и, кроме того, при представлении числа в виде знака и величины необходимы отдельные устройства для сложения и вычитания, т.е., если положительное и отрицательные числа представлены в прямом коде, операции над кодами знаков выполняются раздельно. Поэтому представление чисел в виде знака и значения не нашло широкого применения.

В то же время, при помощи обратного и дополнительного кодов, используемых для представления отрицательных чисел, операция вычитания (алгебраического сложения) сводится к операции простого арифметического сложения. При этом операция сложения распространяется и на разряды знаков, рассматриваемых как разряды целой части числа. Именно поэтому для представления целых чисел со знаком применяется дополнительный код.

Дополнительный код отрицательного числа формируется по следующим правилам:

модуль отрицательного числа записать в прямом коде, в неиспользуемые старшие биты записать нули;

сформировать обратный код числа, для этого нуль заменить единицей, а единицу заменить нулем;

к обратному коду числа прибавить единицу.

Например: для числа -33 в формате integer

1000000000100001	прямой код
0111111111011110	обратный код
+_____1	
1111111111011111	дополнительный код

Для положительных чисел прямой, обратный и дополнительный коды одинаковы. Аналогично представляются целые числа в формате shortint, longint, comp.

При разработке программ на этапе выбора типа данных важно знать диапазон целых величин, которые могут храниться в n разрядах памяти. В соответствии с алгоритмом преобразования двоичных чисел в десятичные, формула суммирования для n разрядов имеет вид:

$2^0+2^1+2^2+\ldots+2^{n-1}$ или $\sum_{i=1}^{n} 2^i = 2^n - 1$.

При n-битовом хранении числа в дополнительном коде первый бит выражает знак целого числа. Поэтому положительные числа представляются в диапазоне от 0 до $1*2^0 + 1*2^1 +...+ 1*2^{n-2}$ или, что то же самое, от 0 до 2^{n-1}-1. Все другие конфигурации битов выражают отрицательные числа в диапазоне от -2^{n-1} до -1. Таким образом, можно сказать, что число N может храниться в n разрядах памяти, если его значение находится в диапазоне:

$$-2^{n-1} <= N <= 2^{n-1} - 1.$$

Иными словами, диапазон возможных значений целых типов зависит от их внутреннего представления, которое может занимать 1, 2 или 4 байта. В таблице 2.1 приводится перечень целых типов, размер памяти для их внутреннего представления в битах, диапазон возможных значений.

Таблица 2.1

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

МАШИННОЕ ПРЕДСТАВЛЕНИЕ БЕЗЗНАКОВЫХ ТИПОВ. К беззнаковым типам в PASCAL относятся типы BYTE и WORD.

Формат машинного представления чисел типа BYTE приведен на рис 2.2. а).

Например: 1). Машинное представление числа 45:

$45=2^5+2^3+2^2+2^0$ = 00101101

2). Машинное представление границ диапазона допустимых значений чисел 0 и 255:

0: 00000000; 255: 11111111.

Рис. 2.2. Формат машинного представления беззнаковых чисел

Формат машинного представления чисел типа WORD приведен на рис. 2.2. б).

Например: 1). Машинное представление числа 258:

$257=2^8+2^1$ = 00000010 00000001.

2). Машинное представление границ:

0: 00000000 00000000; 65535: 11111111 11111111.

МАШИННОЕ ПРЕДСТАВЛЕНИЕ ЧИСЕЛ СО ЗНАКОМ. Для представления чисел со знаком определены следующие типы SHORTINT, INTEGER, LONGINT. В приведенных типах числа хранятся в дополнительном коде. Напомним, что дополнительный код положительных чисел совпадает с прямым кодом.

Формат машинного представления чисел типа SHORTINT приведен на рис 2.3. а) где s-знаковый разряд числа. Для положительных чисел s=0, для отрицательных s=1.

Например, машинное представление чисел в формате shortint:

1). 0: 00000000;
2). +127: 01111111;
3). -128: 10000000.

Формат машинного представления чисел типа INTEGER приведен на рис 2.3. б). Например:

1). +32765: 11111101 01111111;
2). -32765: 00000011 10000000;
3). -47: 11010001 11111111.

Машинное представление границ диапазона допустимых значений:

4). -32768: 00000000 10000000;
5). 32767: 11111111 01111111.

Формат машинного представления чисел типа LONGINT приведен на рис 2.3. в). Например, представление чисел в формате longint:

1). +89 01011001 00000000 00000000 00000000;
2). -89 10100111 11111111 11111111 11111111.

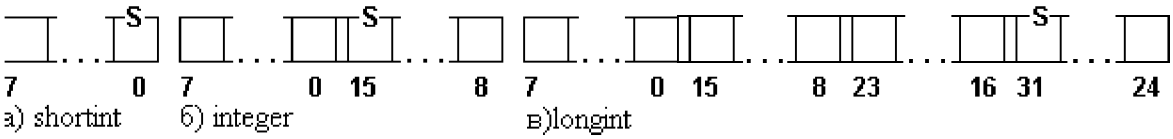


Рис. 2.3. Формат машинного представления чисел со знаком

На рис 2.3 s-знаковый бит числа. При этом, если s=0, то число положительное, если s=1 - число отрицательное. Цифры определяют номера разрядов памяти.

МАШИННОЕ ПРЕДСТАВЛЕНИЕ ДАННЫХ ТИПА COMP. Формат машинного представления данных типа COMP приведен на рисунке 2.4. Этот тип данных предназначен для работы с большими целыми числами (см. таблицу 2.1). Поэтому числа этого типа представляются в памяти в соответствии с правилами представления целых чисел со знаком - в дополнительном коде. Но для удобства пользователей при вводе и выводе значений чисел в этом формате допускается использование формы записи чисел характерных для вещественных чисел (в виде мантиссы и порядка).

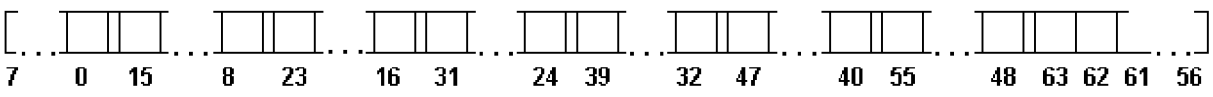


Рис.2.4. Формат машинного представления данных типа comp

Здесь s - знаковый разряд числа (если s=0, то число положительное, если s=1 - число отрицательное).

Например: машинное представление чисел в формате COMP:

+512 0..0 00000010 0..0 0..0 0..0 0..0 0..0 0..0
-512 0..0 11111110 1..1 1..1 1..1 1..1 1..1 1..1

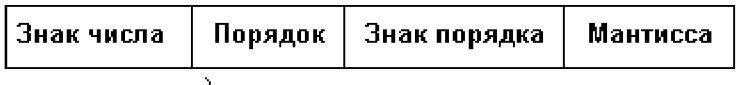
2.1.2. Вещественные типы

В отличии от порядковых типов (все целые, символьный, логический), значения которых всегда сопоставляются с рядом целых чисел и, следовательно, представляются в памяти машины абсолютно точно, значение вещественных типов определяет число лишь с некоторой конечной точностью, зависящей от внутреннего формата вещественного числа.

ПРЕДСТАВЛЕНИЕ ВЕЩЕСТВЕННЫХ ЧИСЕЛ В ПАМЯТИ. В некоторых областях вычислений требуются очень большие или весьма малые действительные числа. Для получения большей точности применяют запись чисел с плавающей точкой. Запись числа в формате с плавающей точкой является весьма эффективным средством представления очень больших и весьма малых вещественных чисел при условии, что они содержат ограниченное число значащих цифр, и, следовательно, не все вещественные числа могут быть представлены в памяти. Обычно число используемых при вычислениях значащих цифр таково, что для большинства задач ошибки округления пренебрежимо малы.

Формат для представления чисел с плавающей точкой содержит одно или два поля фиксированной длины для знаков. Количество позиций для значащих цифр различно в разных ЭВМ, но существует, тем не менее, общий формат, приведенный на рисунке 2.5 а). В соответствии с этой записью формат вещественного числа содержит в общем случае поля мантиссы, порядка и знаков мантиссы и порядка.

Однако, чаще вместо порядка используется характеристика, получающаяся прибавлением к порядку такого смещения, чтобы характеристика была всегда положительный. При этом имеет место формат представления вещественных чисел такой, как на рис 2.5 б).



а) с порядком

б) с характеристикой

Рис. 2.5. Формат представления вещественных чисел

Введение характеристики избавляет от необходимости выделять один

бит для знака порядка и упрощает выполнение операций сравнения (<,>, <=,>=) и арифметических операций над вещественными числами. Так, при сложении или вычитании чисел с плавающей точкой для того, чтобы выровнять операнды, требуется сдвиг влево или вправо мантиссы числа. Сдвиг можно осуществить с помощью единственного счетчика, в который сначала заносится положительное число, уменьшающееся затем до тех пор, пока не будет выполнено требуемое число сдвигов.

Таким образом, для представления вещественных чисел в памяти ЭВМ порядок **p** вещественного числа представляется в виде характеристики путем добавления смещения (старшего бита порядка):

$$X = 2^{n-1} + k + p, \qquad (2.1)$$

где n - число бит, отведенных для характеристики, p - порядок числа,
k - поправочный коэффициент фирмы IBM, равный +1 для real и
-1 для форматов single, double, extended.

Формулы для вычисления характеристики и количество бит, необходимых для ее хранения, приведены в таблице 2.2.

Таблица 2.2

Тип	Характеристика	Количествово бит на характеристику
real	$X=2^7+p+1$	8
single	$X=2^7+p-1$	8
double	$X=2^{10}+p-1$	11
extended	$X=2^{14}+p-1$	15

Следующим компонентом представляемого в машине числа с плавающей точкой является мантисса. Для увеличения количества значащих цифр в представлении числа и исключения переполнения при умножении мантиссу обычно подвергают нормализации. Нормализация означает, что мантисса (назовем ее F), кроме случая, когда F=0,должна находиться в интервале

$$R^{-1} <= F < 1.$$

Для двоичной системы счисления R=2. Тогда в связи с тем, что

$2^{-1} <= F < 1$, ненулевая мантисса любого хранимого числа с плавающей точкой должна начинаться с двоичной единицы. В этом и заключается одно из достоинств двоичной формы представления числа с плавающей точкой. Поскольку процесс нормализации создает дробь, первый бит которой равен 1, в структуре некоторых машин эта единица учитывается, однако не записывается в мантиссу. Эту единицу часто называют скрытой единицей, а получающийся дополнительный бит используют для увеличения точности представления чисел или их диапазона.

Приведенный метод нормализации является классическим методом, при котором результат нормализации представляется в виде правильной дроби, т.е. с единицей после точки и нулем в целой части числа. Но нормализацию мантиссы можно выполнить по-разному.

В IBM PC нормализованная мантисса содержит свой старший бит слева от точки. Иными словами нормализованная мантисса в IBM PC принадлежит интервалу 1 <= F < 2. В памяти машины для данных типа real, single, double этот бит не хранится, т.е. является "скрытым" и используется для увеличения порядка в форматах single или для хранения знака в формате real. Для положительных и отрицательных чисел нормализованная мантисса в памяти представлена в прямом коде.

Первый, старший, бит в представлении чисел в формате с плавающей точкой является знаковым, и по принятому соглашению нуль обозначает положительное число, а единица - отрицательное.

Число бит для хранения мантиссы и порядка зависит от типа вещественного числа. Суммарное количество байтов, диапазоны допустимых значений чисел вещественных типов, количество значащих цифр после запятой в представлении чисел приведены в таблице 2.3.

Таблица 2.3

Тип	Диапазон значений	Значащие цифры	Размер в байтах
real	$2.9*10^{-39}$…$1.7*10^{38}$	11-12	6
single	$1.4*10^{-45}$…$3.4*10^{38}$	7-8	4
double	$4.9*10^{-324}$…$1.8*10^{308}$	15-16	8
extended	$3.1*10^{-4944}$…$1.2*10^{4932}$	19-20	10

АЛГОРИТМ ФОРМИРОВАНИЯ МАШИННОГО ПРЕДСТАВЛЕНИЯ ВЕЩЕСТВЕННОГО ЧИСЛА В ПАМЯТИ ЭВМ. Алгоритм формирования состоит из следующих пунктов:

1). Число представляется в двоичном коде.

2). Двоичное число нормализуется. При этом для чисел, больших единицы, плавающая точка переносится влево, определяя положительный порядок. Для чисел, меньших единицы, точка переносится вправо, определяя отрицательный порядок.

3). По формуле из таблицы 2.2 с учетом типа вещественного числа определяется характеристика.

4). В отведенное в памяти поле в соответствии с типом числа записываются мантисса, характеристика и знак числа. При этом необходимо отметить следующее:

* для чисел типа real характеристика хранится в младшем байте памяти, для чисел типа single, double, extended - в старших байтах;
* знак числа находится всегда в старшем бите старшего байта;
* мантисса всегда хранится в прямом коде;
* целая часть мантиссы (для нормализованного числа всегда 1) для чисел типа real, single, double не хранится (является скрытой). В числах типа extended все разряды мантиссы хранятся в памяти ЭВМ.

МАШИННОЕ ПРЕДСТАВЛЕНИЕ ДАННЫХ ТИПА REAL. Формат машинного представления данных типа REAL следующий:

мл. байт ст. байт
а: 7 0 15 8 23 16 31 24 39 32 47 40
x....x м ... мм ... мм ... мм ... мsм ... м
б: 7 0 -32 -39 -24 -31 -16 -23 -8 -15 -1 -7
где а - номера разрядов памяти,
б - показатели степеней разрядов характеристики и мантиссы,
s - знаковый разряд числа,
м - нормализованная мантисса,
x - характеристика числа.

Например: 1). Десятичное число 15.375;
в двоичной системе счисления 1111.011;
результат нормализации $1.111011*2^3$; p=3.
Учитывая отбрасывание неявной единицы и сдвиг порядка, получаем:
s=0; $x=2^7+1+3=2^7+2^2=132$;
в двоичной системе счисления x=10000100; м=1110110...0;
машинное представление числа:
10000100 00000000 00000000 00000000 00000000 01110110

2). Десятичное число -0.5;
аналогичные выкладки дают: нормализованную мантиссу: 1.00...0;
машинное представление числа:
10000000 00000000 00000000 00000000 00000000 10000000

3). Десятичное число -25.75;
аналогично: нормализованная мантисса: 1.10011100...0;
машинное представление числа:
10000101 00000000 00000000 00000000 00000000 11001110

4). Число 0.0;
машинное представление числа:
00000000 00000000 00000000 00000000 00000000 00000000

5). Числа верхней и нижней границ положительного диапазона
$\sim 1.7*10^{38}$ - 11111111 11111111 11111111 11111111 11111111 01111111
$\sim 2.9*10^{-35}$ - 00000001 00000000 00000000 00000000 00000000 00000000

МАШИННОЕ ПРЕДСТАВЛЕНИЕ ДАННЫХ ТИПА SINGLE. Формат машинного представления данных типа SINGLE следующий:

мл. байт ст. байт
7 0 15 8 23 22 16 31 30 24 - номера разрядов памяти
м ... м м ... м x м ... м s x ... x
-16 -23 -8 -15 0 -1 -7 7 1 - показатели степеней разрядов мантиссы и характеристики,
где s - знаковый разряд,
x - характеристика числа,
м - нормализованная мантисса.

Например: 1). Число -15.375;

в двоичной системе счисления -1111.011;

нормализованное двоичное число $-1.111011*2^3$; p=3.

Учитывая отбрасывание неявной единицы и сдвиг порядка, получаем:

$s=1$; $x=2^7-1+3=2^7+2^1=130$;

в двоичной системе счисления x=10000010; м=1110110...0;

машинное представление числа в формате SINGLE:

00000000 00000000 01110110 11000001

2). Число -0.1875;

в двоичной системе счисления -0.0011;

нормализованное двоичное число $-1.1*2^{-3}$; p=-3.

Учитывая отбрасывание неявной единицы и сдвиг порядка, получаем:

$s=1$; $x=2^7-1-3=2^7-2^2$;

в двоичной системе счисления x=01111100; м=100...0;

машинное представление числа в формате SINGLE:

00000000 00000000 01000000 10111110

3). Десятичное число 4.5;

аналогичные выкладки дают нормализованную мантиссу: 1.00100...0;

машинное представление числа:

00000000 00000000 10010000 01000000

4). Значения верхней и нижней границ чисел отрицательного диапазона

$\sim -3.4*10^{38}$ - 11111111 11111111 01111111 11111111

$\sim -.4*10^{-45}$ - 00000001 00000000 00000000 10000000

МАШИННОЕ ПРЕДСТАВЛЕНИЕ ДАННЫХ ТИПА DOUBLE. Формат машинного представления данных типа DOUBLE следующий:

мл.байт ст.байт

7	0	15	8	23	16	31	24	39	32	47	40	55	52	51	48	63	56
м	...	м	м	...	м	м	...	м	м	...	м	м	...	м	м	...	м	s	x	...	x
-44	-50	-37	-43	-29	-36	-21	-28	-13	-20	-5	-12	3	0	-1	-4	10	4

где верхняя строка цифр от 0 до 63 - номера разрядов памяти;

нижняя строка цифр от -50 до -1 - показатели степеней разрядов мантиссы;

от 0 до 10 - разрядов характеристики;

s - знаковый разряд числа;

м - нормализованная мантисса;

x - характеристика числа ($x=2^{10}-1+p$,

где p - порядок нормализованного числа).

Например, 1). Число 15.375;

в двоичной системе счисления 1111.011;

результат нормализации $1.111011*2^3$; p=3.

Учитывая отбрасывание скрытой единицы и сдвиг порядка, получаем:

$s=0$; $x=2^{10}+3=2^{10}+2^1=1026$;

в двоичной системе счисления x=10000000010; m=1110110...0;
машинное представление числа в формате DOUBLE:
0 00000000 00000000 00000000 00000000 31
32 00000000 11000000 00101110 01000000 63

2). Десятичное число 0.0375;
в двоичной системе счисления 0.011;
результат нормализации $1.1*2^{-2}$; p=-2.
Учитывая отбрасывание скрытой единицы и сдвиг порядка, получаем:
s=0; $x=2^{10}-2^1-2^0=2^{10}-3$;
в двоичной системе счисления x=01111111101; m=100...0;
машинное представление числа в формате DOUBLE:
0 00000000 00000000 00000000 00000000 31
32 00000000 00000000 11011000 00111111 63

3). Десятичное число 2.5;
аналогичные выкладки дают нормализованную мантиссу: 1.0100...0;
машинное представление числа 2.5:
00000000 00000000 00000000 00000000
00000000 00000000 00000100 01000000

4). Значения верхней и нижней границ диапазона положительных чисел:
~1.8*10^308 - 11111111 11111111 11111111 11111111
11111111 11111111 11101111 01111111
~4.9*10^(-324) - 00000001 00000000 00000000 00000000
00000000 00000000 00000000 00000000

Символ ~ обозначает приближенное значение числа.

МАШИННОЕ ПРЕДСТАВЛЕНИЕ ДАННЫХ ТИПА EXTENDED. Формат машинного представления данных типа EXTENDED следующий:

мл.байт ст. байт
7 0 15 8 23 16 31 24 39 32 47 40 55 48 63 56 71 64 79 72
м..м м..м м..м м..м м..м м..м м..м м..м x..x sx..x
-56-63-48-55-40-47-32-39-24-31-16-23-8-15 0-7 7 0 14 8

где верхняя строка цифр - номера разрядов памяти;
нижняя строка цифр - показатели степеней разрядов мантиссы и характеристики;
s - знаковый разряд числа;
м - нормализованная мантисса;
x - характеристика числа.

Например: 1). Число -15.375;
в двоичной системе счисления -1111.011;
после нормализации -1.111011*2^3; p=3.
Учитывая присутствие скрытой единицы и сдвиг порядка, получаем:
s=1; x=2^14-1+3=2^14+2^1=16386;
в двоичной системе счисления x=100000000000010; м=11110110...0
(в мантиссе единица стоящая слева от запятой не отбрасывается).

Машинное представление данного числа в формате EXTENDED:
0..0 0..0 0..0 0..0 0..0 0..0 0..0 11110110 00000010 11000000

2). Число 1.0;

аналогичные выкладки дают
нормализованную мантиссу: 1.0...0;
машинное представление числа 1.0:
0..0 0..0 0..0 0..0 0..0 0..0 0..0 10000000 11111111 00111111

3). Значения верхней и нижней границ диапазона положительных чисел (символом * помечены разряды, значения которых при данной характеристике не идентифицируются т. е. их значения не влияют на значение мантиссы):

~1.2*10^4932 - ******** ******** 11111111 11111111 11111111
11111111 11111111 11111111 11111111 011111111

~3.1*10^4944 - ******** ******** 00000001 00000000 000000000
00000000 00000000 00000000 00000001 000000000

2.1.3. Десятичные типы

Десятичные типы не поддерживаются языком PASCAL, но имеются в некоторых других языках, например, COBOL, PL/1. Эти типы применяются для внутримашинного представления таких данных, которые в первую очередь должны храниться в вычислительной системе и выдаваться пользователю по требованию, и лишь во вторую очередь - обрабатываться (служить операндами вычислительных операций). Неслучайно эти типы впервые появились в языке COBOL, ориентированном на обработку экономической информации: в большинстве задач этой сферы важно прежде всего хранить и находить информацию, а ее преобразование выполняется сравнительно редко и сводится к простейшим арифметическим операциям.

Архитектура некоторых вычислительных систем (например, IBM System/390) предусматривает команды, работающие с десятичным представлением чисел, хотя эти команды и выполняются гораздо медленнее, чем команды двоичной арифметики. В других архитектурах операции с десятичными числами моделируются программно.

К десятичным типам относятся: десятичный тип с фиксированной точкой и тип шаблона.

ДЕСЯТИЧНЫЙ ТИП С ФИКСИРОВАННОЙ ТОЧКОЙ. В языке PL/1 десятичный тип с фиксированной точкой описывается в программе, как:

DECIMAL FIXED (m.d) или DECIMAL FIXED (m).

Первое описание означает, что данное представляется в виде числа, состоящего из m десятичных цифр, из которых d цифр расположены после десятичной точки. Второе - целое число из m десятичных цифр. Следует подчеркнуть, что в любом случае число десятичных цифр в числе фиксировано. Внутримашинное представление целых чисел и чисел с дробной частью одинаково. Для последних положение десятичной точки

запоминается компилятором и учитывается им при трансляции операций, в которых участвуют десятичные числа с фиксированной точкой.

Внутримашинное представление данного типа носит название десятичного упакованного формата. Примеры представления чисел 963 и -1534 в таком формате приведены на рис. 2.6.

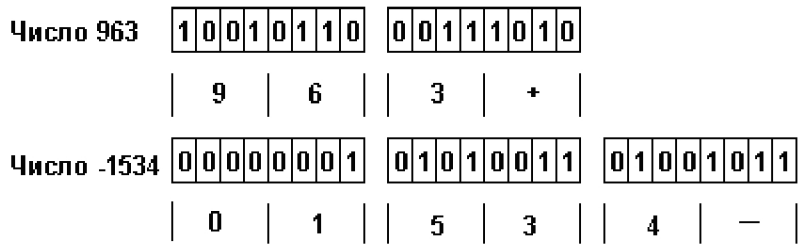


Рис. 2.6. Машинное представление десятичных чисел в упакованном формате

Каждая десятичная цифра числа занимает полбайта (4 двоичных разряда) и представляется в этом полубайте ее двоичным кодом. Еще полбайта занимает знак числа, который представляется двоичным кодом 1010 - знак "+" или 1011 - знак "-". Представление занимает целое число байт и при необходимости дополняется ведущим нулем.

ТИП ШАБЛОНА. В языке PL/1 тип шаблона описывается в программе, как: PICTURE '9...9'. Это означает, что данное представляет собой целое число, содержащее столько цифр, сколько девяток указано в описании. Внутримашинное представление этого типа, так называемый десятичный зонный формат, весьма близок к такому представлению данных, которое удобно пользователю: каждая десятичная цифра представляется байтом: содержащим код символа соответствующей цифры. В IBM System/390, которая аппаратно поддерживает зонный формат, применяется символьный код EBCDIC, в котором код символа цифры содержит в старшем полубайте код 1111, а в младшем - двоичный код цифры числа. Знак не входит в общее число цифр в числе, для представления знака в старшем полубайте последней цифры числа код 1111 заменяется на 1010 - знак "+" или 1011 - знак "-".

Примеры представления чисел в зонном формате приведены на рис.2.7.

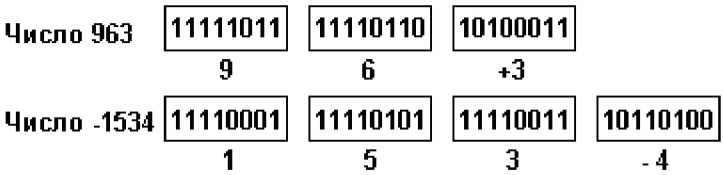


Рис.2.7. Машинное представление десятичных чисел в зонном формате

2.1.4. Операции над числовыми типами

Над числовыми типами, как и над всеми другими, возможны прежде всего четыре основных операции: создание, уничтожение, выбор, обновление. Специфические операции над числовыми типами - хорошо известные всем арифметические операции: сложение, вычитание, умножение, деление. Операция возведения в степень в некоторых языках также является базовой и обозначается специальным символом или комбинацией символов (^ - в BASIC, ** - в PL/1), в других - выполняется встроенными функциями (pow в C).

Обратим 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 PASCAL 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 - ("mod" - в PASCAL, "%" - в C).

Еще одна группа операций над числовыми типами - операции сравнения: "равно", "не равно", "больше", "меньше" и т.п. 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. Это создает некоторые неудобства для программистов, так как 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 языке PL/1, например, все операнды выражения приводятся к одному типу - к типу той переменной, в которую будет записан результат, а затем уже выражение вычисляется. В языке же C 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.

Программист, использующий выражения со смешением типов, должен точно знать правила их вычисления для выбранного языка.

2.2. Битовые типы

ПРЕДСТАВЛЕНИЕ БИТОВЫХ ТИПОВ. В ряде задач может потребоваться работа с отдельными двоичными разрядами данных. Чаще всего такие задачи возникают в системном программировании, когда, например, отдельный разряд связан с состоянием отдельного аппаратного

переключателя или отдельной шины передачи данных и т.п. Данные такого типа представляются в виде набора битов, упакованных в байты или слова, и не связанных друг с другом. Операции над такими данными обеспечивают доступ к выбранному биту данного. В языке PASCAL роль битовых типов выполняют беззнаковые целые типы byte и word. Над этими типами помимо операций, характерных для числовых типов, допускаются и побитовые операции. Аналогичным образом роль битовых типов играют беззнаковые целые и в языке С.

В языке PL/1 существует специальный тип данных - строка битов, объявляемый в программе, как: BIT(n).

Данные этого типа представляют собой последовательность бит длиною n. Строка битов занимает целое число байт в памяти и при необходимости дополняется справа нулями.

ОПЕРАЦИИ НАД БИТОВЫМИ ТИПАМИ. Над битовыми типами возможны три группы специфических операций: операции булевой алгебры, операции сдвигов, операции сравнения.

Операции булевой алгебры - НЕ (not), ИЛИ (or), И (and), исключающее ИЛИ (xor). Эти операции и по названию, и по смыслу похожи на операции над логическими операндами, но отличие в их применении к битовым операндам состоит в том, что операции выполняются над отдельными разрядами операндов.

Так операция НЕ состоит в том, что каждый разряд операнда изменяет значение на противоположный. Выполнение операции, например, ИЛИ над двумя битовыми операндами состоит в том, что выполняется ИЛИ между первым разрядом первого операнда и первым разрядом второго операнда, это дает первый разряд результата; затем выполняется ИЛИ между вторым разрядом первого операнда и вторым разрядом второго, получается второй разряд результата и т.д.

Ниже даны примеры выполнения побитовых логических операций:

а).x = 01101100
not x = 10010011

в). x = 01101100
y = 11001110
x and y = 01001100

б). x = 01101100
y = 11001110
x or y = 11101110

г). x = 01101100
y = 11001110
x xor y = 10100010

В некоторых языках (PASCAL) побитовые логические операции обозначаются так же, как и операции над логическими операндами и распознаются по типу операндов. В других языках (С) для побитовых и общих логических операций используются разные обозначения. В третьих (PL/1) - побитовые операции реализуются встроенными функциями языка.

Операции сдвигов выполняют смещение двоичного кода на заданное количество разрядов влево или вправо. Из трех возможных типов сдвига (арифметический, логический, циклический) в языках программирования обычно реализуется только логический (например, операциями shr, shl в PASCAL).

В операциях сравнения битовые данные интерпретируются как целые без знака, и сравнение выполняется как сравнение целых чисел. Битовые строки в языке PL/1 - более общий тип данных, к которому применимы также операции над строковыми данными.

2.3. Логический тип

Значениями логического типа BOOLEAN 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 соответствует любое ненулевое значение байта. Например: false всегда в машинном представлении: 00000000; true может выглядеть таким образом: 00000001 или 00010001 или 10000000.

Однако следует иметь в виду, что при выполнении операции присваивания переменной логического типа значения true, в соответствующее поле памяти всегда записывается код 00000001.

Над логическими типами возможны операции булевой алгебры - НЕ (not), ИЛИ (or), И (and), исключающее ИЛИ (xor) - последняя реализована для логического типа не во всех языках. В этих операциях операнды логического типа рассматриваются как единое целое - вне зависимости от битового состава их внутреннего представления.

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

Интересно, что в языке С 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 логического типа являются целые числа 0 (ложь) или 1 (истина).

2.4. Символьный тип

Значением символьного типа char являются символы из некоторого предопределенного множества. В большинстве современных персональных ЭВМ этим множеством является ASCII (American Standard Code for Information Intechange - американский стандартный код для обмена информацией). 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. Допускается некоторые отклонения от стандарта ASCII, в частности, при наличии соответствующей системной поддержки это множество может содержать буквы русского алфавита. Порядковые номера (кодировку) можно узнать в соответствующих разделах технических описаний.

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.

Например: символ "1" имеет ASCII код 49, следовательно машинное представление будет выглядеть следующим образом: 00110001.

ASCII, однако, не является единственно возможным множеством. Другим достаточно широко используемым множеством является код EBCDIC (Extended Binary Coded Decimal Interchange Code - расширенный двоично-кодированный десятичный код обмена), применяемый в системах IBM средней и большой мощности. В EBCDIC код символа также занимает один байт, но с иной кодировкой, чем в ASCII.

И ASCII, и EBCDIC включают в себя буквенные символы только латинского алфавита. Символы национальных алфавитов занимают "свободные места" в таблицах кодов и, таким образом, одна таблица может поддерживать только один национальный алфавит. Этот недостаток преодолен во множестве UNICODE, которое находит все большее распространение прежде всего в UNIX-ориентированных системах. В UNICODE каждый символ кодируется двумя байтами, что обеспечивает более 64 тыс. 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. В основном символьный тип данных используется как базовый для построения интегрированного типа "строка символов".

2.5. Перечислимый тип

ЛОГИЧЕСКАЯ СТРУКТУРА. Перечислимый тип представляет собой упорядоченный тип данных, определяемый программистом, т.е. программист перечисляет все значения, которые может принимать переменная этого типа. Значения являются неповторяющимися в пределах программы идентификаторами, количество которых не может быть больше 256, например,

```
type      color=(red,blue,green);
          work_day=(mo,tu,we,th,fr);
          winter_day=(december,january,february);
```

МАШИННОЕ ПРЕДСТАВЛЕНИЕ. Для переменной перечислимого типа выделяет-ся один байт, в который записывается порядковый номер присваиваемого значения. Порядковый номер определяется из описанного типа, причём нумерация начинается с 0. Имена из списка перечислимого типа являются константами, например,

```
var   B,C:color;
```

```
begin  B:=blue;          (*  B=1  *)
       C:=green;         (*  C=2  *)
       Write(ord(B):4,ord(C):4);
       end.
```

После выполнения данного фрагмента программы на экран будут выданы цифры 1 и 2. Содержимое памяти для переменных B И C при этом следующее: B - 00000001; C - 00000010.

ОПЕРАЦИИ. На физическом уровне над переменными перечислимого типа определены операции создания, уничтожения, выбора, обновления. При этом выполняется определение порядкового номера идентификатора по его значению и, наоборот, по номеру идентификатора определяется его значение.

На логическом уровне переменные перечислимого типа могут быть использованы только в выражениях булевского типа и в операциях сравнения; при этом сравниваются порядковые номера значений.

2.6. Интервальный тип

ЛОГИЧЕСКАЯ СТРУКТУРА. Один из способов образования новых типов из уже существующих - ограничение допустимого диапазона значений некоторого стандартного скалярного типа или рамок описанного перечислимого типа. Это ограничение определяется заданием минимального и максимального значений диапазона. При этом изменяется диапазон допустимых значений по отношению к базовому типу, но представление в памяти полностью соответствует базовому типу.

МАШИННОЕ ПРЕДСТАВЛЕНИЕ. Данные интервального типа могут храниться в зависимости от верхней и нижней границ интервала независимо от входящего в этот предел количества значений в виде, представленном в таблице 2.4. Для данных интервального типа требуется память размером один, два или четыре байта, например,

```
var    A: 220..250;      (*  Занимает 1 байт   *)
       B: 2221..2226;    (*  Занимает 2 байта  *)
       C: 'A'..'K';      (*  Занимает 1 байт   *)
begin        A:=240;     C:='C';       B:=2222;    end.
```

После выполнения данной программы содержимое памяти будет следующим: A - 11110000; C - 01000011; B - 10101110 00001000.

Таблица 2.4

Базовый тип	Максимально допустимый диапазон	Размер требуемой памяти
Shortint	-128 … 127	1 байт
Integer	-32768 … 32767	2 байта
Longint	-2147483648 …	4 байта
Byte	2147483647	1 байт
Word	0 … 255	2 байта
Char	0 … 65535	1 байт

Boolean	chr(ord(0)) … chr(ord(255)) false … true	1 байт

Примечание: запись chr(ord(0)) в таблице следует понимать как: символ с кодом 0.

ОПЕРАЦИИ. На физическом уровне над переменными интервального типа определены операции создания, уничтожения, выбора, обновления. Дополнительные операции определены базовым типом элементов интервального типа.

А) Интервальный тип от символьного: определение кода символа и, наоборот, символа по его коду.

Пусть задана переменная типа tz:'d'..'h'. Данной переменной присвоено значение 'e'. Байт памяти, отведенный под эту переменную, будет хранить ASCII-код буквы 'e' т.е. 01100101 (в 10-ом представлении 101).

Б) Интервальный тип от перечислимого: определение порядкового номера идентификатора по его значению и, наоборот, по номеру идентификатора - его значение.

На логическом уровне все операции, разрешенные для данных базового типа, возможны и для данных соответствующих интервальных типов.

2.7. Указатели

Тип указателя представляет собой адрес ячейки памяти (в подавляющем большинстве современных вычислительных систем размер ячейки - минимальной адресуемой единицы памяти - составляет один байт). При программировании на низком уровне - в машинных кодах, на языке Ассемблера и на языке С, который специально ориентирован на системных программистов, работа с адресами составляет значительную часть программных кодов. При решении прикладных задач с использованием языков высокого уровня наиболее частые случаи, когда программисту 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 в программе вводятся два или более указателей, которые содержат адрес одной и той же области памяти, но имеют разный тип (см.ниже). Обращаясь к этой области памяти по тому или иному указателю, программист обрабатывает ее содержимое как данные того или иного типа.

2) При работе с динамическими структурами данных, что более важно. 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 программист может обращаться только через такой указатель.

2.7.1. Физическая структура указателя

Физическое представление адреса существенно зависит от аппаратной архитектуры вычислительной системы. Рассмотрим в качестве примера структуру адреса в микропроцессоре i8086.

Машинное слово этого процессора имеет размер 16 двоичных разрядов. Если использовать представление адреса в одном слове, то можно адресовать 64 Кбайт памяти, что недостаточно для сколько-нибудь серьезного программного изделия. Поэтому адрес представляется в виде двух 16-разрядных слов - сегмента и смещения. Сегментная часть адреса загружается в один из специальных сегментных регистров (в i8086 таких регистров 4). При обращении по адресу задается идентификатор сегментного регистра и 16-битное смещение. Полный физический (эффективный) адрес получается следующим образом. Сегментная часть адреса сдвигается на 4 разряда влево, освободившиеся слева разряды заполняются нулями, к полученному таким образом коду прибавляется смещение, как показано на рис. 2.8. Полученный эффективный адрес имеет размер 20 двоичных разрядов, таким образом, он позволяет адресовать до 1 Мбайт памяти.

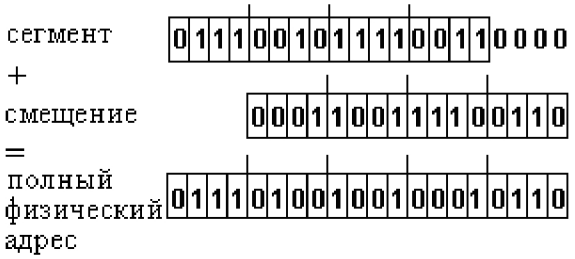


Рис. 2.8. Вычисление полного адреса в микропроцессоре i8086

Следует подчеркнуть, что физическая структура адреса принципиально различна для разных аппаратных архитектур. Так, например, в микропроцессоре i386 обе компоненты адреса 32-разрядные; в процессорах семейства S/390 адрес представляется в виде 31-разрядного смещения в одном из 19 адресных пространств, в процессоре Power PC 620 одним 64-разрядным словом может адресоваться вся как оперативная, так и внешняя память.

Операционная система MS DOS была разработана именно для процессора i8086 и использует описанную структуру адреса даже, когда выполняется на более совершенных процессорах. Однако, это сегодня единственная операционная система, в среде которой программист может работать с адресами в реальной памяти и с физической структурой адреса. Все без исключения современные модели процессоров аппаратно выполняют так называемую динамическую трансляцию адресов и совместно с современными операционными системами обеспечивают работу программ в виртуальной (кажущейся) памяти. Программа разрабатывается и выполняется в некоторой виртуальной памяти, адреса в которой линейно изменяются от 0 до некоторого максимального значения. Виртуальный адрес представляет собой число - номер ячейки в виртуальном адресном

пространстве. Преобразование виртуального адреса в реальный производится аппаратно при каждом обращении по виртуальному адресу. Это преобразование выполняется совершенно незаметно (прозрачно) для программиста, поэтому в современных системах программист может считать физической структурой адреса структуру виртуального адреса. Виртуальный же адрес представляет собой целое число без знака. В разных вычислительных системах может различаться разрядность этого числа. Большинство современных систем обеспечивают 32-разрядный адрес, позволяющий адресовать до 4 Гбайт памяти, но уже существуют системы с 48 и даже 64-разрядными адресами.

2.7.2. Представление указателей в языках программирования

В программе на языке высокого уровня указатели могут быть типизированными и нетипизированными.

При объявлении типизированного указателя определяется и тип объекта в памяти, адресуемого этим указателем. Так, например, объявления в языке PASCAL:

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

или в языке C:

```
int  *ipt;  char *cpt;
```

означают, что переменная ipt представляет собой 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, компилятор считает указатели ipt и cpt имеющими разный тип, и в Pascal оператор: `cpt := ipt;` будет расценен компилятором как ошибочный (компилятор C для аналогичного оператора присваивания ограничится предупреждением).

Таким образом, когда речь идет об указателях типизированных, правильнее говорить не о едином типе данных "указатель", а о целом семействе типов: "указатель на целое", "указатель на символ" и т.д. Могут быть указатели и на более сложные, интегрированные структуры данных, и указатели на указатели.

Нетипизированный указатель, тип pointer в Pascal или void * в C, служит для представления адреса, по которому содержатся данные неизвестного типа. Работа с нетипизированными указателями существенно ограничена, они могут использоваться только для сохранения адреса, обращение по адресу, задаваемому нетипизированным указателем, невозможно.

2.7.3. Операции над указателями

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 объекты 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 объекта-операнда.

Операция выборки - одноместная, ее операндом является типизированный (обязательно!) указатель, результат - данные, выбранные из памяти по адресу, заданному операндом. Тип результата определяется типом указателя-операнда.

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

К указателю можно прибавить целое число или вычесть из него целое число. Поскольку память имеет линейную структуру, прибавление к адресу числа даст адрес памяти, смещенный на это число байт (или других единиц измерения) относительно исходного адреса. Результат операций "указатель + целое", "указатель - целое" имеет тип "указатель". Можно вычесть один указатель из другого (оба указателя-операнда при этом должны иметь одинаковый тип). Результат такого вычитания будет иметь тип целого числа со знаком. Его значение показывает, на сколько байт (или других единиц измерения) один адрес отстоит от другого в памяти.

Следует отметить, что сложение указателей не имеет смысла. Поскольку программа разрабатывается в относительных адресах и при разных своих выполнениях может размещаться в разных областях памяти, сумма двух адресов в программе будет давать разные результаты при разных выполнениях. Смещение же объектов внутри программы друг относительно друга не зависит от адреса загрузки программы, поэтому результат операции вычитания указателей будет постоянным, и такая операция является допустимой.

Операции адресной арифметики выполняются только над типизированными указателями. Единицей измерения в адресной арифметике является размер объекта, который указателем адресуется. Так, если переменная ipt определена как указатель на целое число (int *ipt), то выражение ipt+1 даст адрес, больший не на 1, а на количество байт в целом числе (в MS DOS - 2). Вычитание указателей также дает в результате не количество байт, а количество объектов данного типа, помещающихся в памяти между двумя адресами. Это справедливо как для указателей на простые типы, так и для указателей на сложные объекты, размеры которых составляют десятки, сотни и более байт.

В связи с имеющимися в языке С расширенными средствами работы с указателями, следует упомянуть и о разных представлениях указателей в этом языке. В С указатели любого типа могут быть ближними (near) и дальними (far) или (huge). Эта дифференциация связана с физической

структурой адреса в i8086, которая была рассмотрена выше. Ближние указатели представляют собой смещение в текущем сегменте, для представления такого указателя достаточно одного 16-разрядного слова. Дальние указатели представляются двумя 16-разрядными словами - сегментом и смещением. Разница между far и huge указателями состоит в том, что для первых адресная арифметика работает только со смещением, не затрагивая сегментную часть адреса, таким образом, операции адресной арифметики могут изменять адрес в диапазоне не более 64 Кбайт; для вторых - в адресной арифметике участвует и сегментная часть, таким образом, предел изменения адреса - 1 Мбайт.

Впрочем, это различие в представлении указателей имеется только в системах программирования, работающих в среде MS DOS, в современных же операционных системах, поддерживающих виртуальную адресацию, различий между указателями нет, все указатели можно считать гигантскими.

3. Статические структуры данных

Статические структуры относятся к разряду непримитивных структур, которые, фактически, представляют собой структурированное множество примитивных, базовых, структур. Например, вектор может быть представлен упорядоченным множеством чисел. Поскольку по определению статические структуры отличаются отсутствием изменчивости, память для них выделяется автоматически - как правило, на этапе компиляции или при выполнении - в момент активизации того программного блока, в котором они описаны. Ряд языков программирования (PL/1, ALGOL-60) допускают размещение статических структур в памяти на этапе выполнения по явному требованию программиста, но и в этом случае объем выделенной памяти остается неизменным до уничтожения структуры. Выделение памяти на этапе компиляции является столь удобным свойством статических структур, что в ряде задач программисты используют их даже для представления объектов, обладающих изменчивостью. Например, когда размер массива неизвестен заранее, для него резервируется максимально возможный размер.

Каждую структуру данных характеризуют её логическим и физическим представлениями. Очень часто говоря о той или иной структуре данных, имеют в виду её логическое представление. Физическое представление обычно не соответствует логическому, и кроме того, может существенно различаться в разных программных системах. Нередко физической структуре ставится в соответствие дескриптор, или заголовок, который содержит общие сведения о физической структуре. Дескриптор необходим, например, в том случае, когда граничные значения индексов элементов массива неизвестны на этапе компиляции, и, следовательно, выделение памяти для массива может быть выполнено только на этапе выполнения программы (как в языке PL/1, ALGOL-60). Дескриптор хранится как и сама физическая структура, в памяти и состоит из полей, характер, число и размеры которых зависят от той структуры, которую он описывает и от принятых способов ее обработки. В ряде случаев дескриптор является совершенно необходимым, так как выполнение операции доступа к структуре требует обязательного знания каких-то ее параметров, и эти параметры хранятся в дескрипторе. Другие хранимые в дескрипторе параметры не являются совершенно необходимыми, но их использование позволяет сократить время доступа или обеспечить контроль правильности доступа к структуре. Дескриптор структуры данных, поддерживаемый языками программирования, является "невидимым" для программиста; он создается компилятором и компилятор же, формируя объектные коды для доступа к структуре, включает в эти коды команды, обращающиеся к дескриптору. Статические структуры в языках программирования связаны со структурированными типами. Структурированные типы в языках программирования являются теми средствами интеграции, которые позволяют строить структуры данных сколь угодно большой сложности. К таким типам относятся: массивы, записи (в некоторых языках - структуры) и множества (этот тип реализован не во всех

языках).

3.1. Векторы

ЛОГИЧЕСКАЯ СТРУКТУРА. Вектор (одномерный массив) - структура данных с фиксированным числом элементов одного и того же типа. Каждый элемент вектора имеет уникальный в рамках заданного вектора номер. Обращение к элементу вектора выполняется по имени вектора и номеру требуемого элемента.

МАШИННОЕ ПРЕДСТАВЛЕНИЕ. АДРЕСАЦИЯ ЭЛЕМЕНТОВ СТРУКТУР. Элементы вектора размещаются в памяти в подряд расположенных ячейках памяти. Под элемент вектора выделяется количество байт памяти, определяемое базовым типом элемента этого вектора. Необходимое число байтов памяти для хранения одного элемента вектора называется **слотом**. Размер памяти для хранения вектора определяется произведением длины слота на число элементов.

В языках программирования вектор представляется одномерным массивом с синтаксисом описания вида (PASCAL):

<Имя> : array [n..k] of <тип>;

где n-номер первого элемента, k-номер последнего элемента. Представление в памяти вектора будет такое, как показано на рис. 3.1.

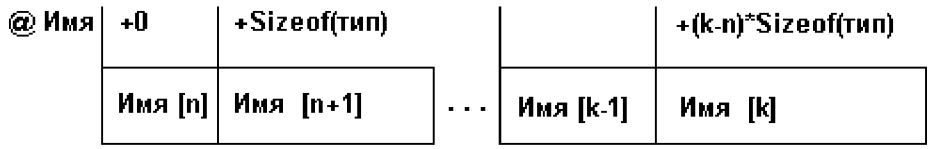


Рис. 3.1. Представление вектора в памяти

Здесь: @ Имя -адрес вектора или, что тоже самое, адрес первого элемента вектора, Sizeof(тип)-размер слота (количество байтов памяти для записи одного элемента вектора),

(k-n)*Sizeof(тип) - относительный адрес элемента с номером k, или, что тоже самое, смещение элемента с номером k.

Например: var m1:array[-2..2] of real;

представление данного вектора в памяти будет как на рис. 3.2.

Смещение элемента относит. адреса m1 (байт)	+0	+6	+12	+18	+24
Значения элем. массива	m1[-2]	m1[-1]	m1[0]	m1[1]	m1[2]

Рис. 3.2. Представление вектора m1 в памяти

В языках, где память распределяется до выполнения программы на этапе компиляции (C, PASCAL, FORTRAN), при описании типа вектора граничные значения индексов должны определены. В языках, где память может распределяться динамически (ALGOL, PL/1), значения индексов могут быть заданы во время выполнения программы.

Количество байтов непрерывной области памяти, занятых одновременно вектором, определяется по формуле:

ByteSise = (k - n + 1) * Sizeof (тип)

Обращение к i-тому элементу вектора выполняется по адресу вектора плюс смещение к данному элементу. Смещение i-ого элемента вектора определяется по формуле:

ByteNumer = (i- n) * Sizeof (тип),

а адрес его: @ ByteNumber = @ имя + ByteNumber.

где @ имя - адрес первого элемента вектора.

Например: var MAS: array [5..10] of word.

Базовый тип элемента вектора - Word требует 2 байта, поэтому на каждый элемент вектора выделяется по два байта. Тогда таблица 3.1 смещений элементов вектора относительно @Mas выглядит так:

Таблица 3.1

Смещение, байт	+0	+2	+4	+6	+6	+6
Элемент массива	mas[5]	mas[5]	mas[5]	mas[5]	mas[5]	mas[5]

Этот вектор будет занимать в памяти: (10-5+1)*2 = 12 байт.

Смещение к элементу вектора с номером 8: (8-5)*2 = 6

Адрес элемента с номером 8: @ MAS + 6.

При доступе к вектору задается имя вектора и номер элемента вектора. Таким образом, адрес i-го элемента может быть вычислен как:

@Имя[i] = @Имя + i*Sizeof(тип) - n*Sizeof(тип) (3.1)

Это вычисление не может быть выполнено на этапе компиляции, так как значение переменной i в это время еще неизвестно. Следовательно, вычисление адреса элемента должно производиться на этапе выполнения программы при каждом обращении к элементу вектора.

Но для этого на этапе выполнения, во-первых, должны быть известны параметры формулы (3.1): @Имя Sizeof(тип), n, а во-вторых, при каждом обращении должны выполняться две операции умножения и две - сложения. Преобразовав формулу (3.1) в формулу (3.2), получим:

$$\begin{matrix} @Имя[i] = A0 + i*Sizeof(тип) \\ A0 = @Имя - n*Sizeof(тип) \end{matrix} \quad (3.2)$$

Таким образом сокращается число хранимых параметров до двух, а число операций - до одного умножения и одного сложения, так как значение A0 может быть вычислено на этапе компиляции и сохранено вместе с Sizeof(тип) в дескрипторе вектора. Обычно в дескрипторе вектора сохраняются и граничные значения индексов. При каждом обращении к элементу вектора заданное значение сравнивается с граничными и программа аварийно завершается, если заданный индекс выходит за допустимые пределы.

Таким образом, информация, содержащаяся в дескрипторе вектора, позволяет, во-первых, сократить время доступа, а во-вторых, обеспечивает проверку правильности обращения. Но за эти преимущества приходится платить, во-первых, быстродействием, так как обращения к дескриптору - это команды, во-вторых, памятью как для размещения самого дескриптора, так и для команд, с ним работающих.

Можно ли обойтись без дескриптора вектора?

В языке С, например, дескриптор вектора отсутствует, точнее, не сохраняется на этапе выполнения. Индексация массивов в С обязательно начинается с нуля. Компилятор каждое обращение к элементу массива заменяет на последовательность команд, реализующую частный случай формулы (3.1) при n = 0:

@Имя[i] = @Имя + i*Sizeof(тип)

Программисты, привыкшие работать на С, часто вместо выражения вида: Имя[i] употребляют выражение вида: *(Имя+i).

Но во-первых, ограничение в выборе начального индекса само по себе может являться неудобством для программиста, во-вторых, отсутствие граничных значений индексов делает невозможным контроль выхода за пределы массива. Программисты, работающие с С, хорошо знают, что именно такие ошибки часто являются причиной "зависания" С-программы при ее отладке.

3.2. Массивы

3.2.1. Логическая структура

Массив – такая структура данных, которая характеризуется:

- фиксированным набором элементов одного и того же типа;
- каждый элемент имеет уникальный набор значений индексов;
- количество индексов определяют мерность массива, например, два индекса – двумерный массив, три индекса – трехмерный массив, один индекс – одномерный массив или вектор;
- обращение к элементу массива выполняется по имени массива и значениям индексов для данного элемента.

Другое определение: массив – это вектор, каждый элемент которого – вектор.

Синтаксис описания массива представляется в виде:

<Имя> : Array [n1..k1] [n2..k2] .. [nn..kn] of <Тип>.

Для случая двумерного массива:

Mas2D : Array [n1..k1] [n2..k2] of <Тип>, или

Mas2D : Array [n1..k1 , n2..k2] of <Тип>

Наглядно двумерный массив можно представить в виде таблицы из (k1-n1+1) строк и (k2-n2+1) столбцов.

3.2.2. Физическая структура

Физическая структура - это размещение элементов массива в памяти ЭВМ. Для случая двумерного массива, состоящего из (k1-n1+1) строк и (k2-n2+1) столбцов физическая структура представлена на рис. 3.3.

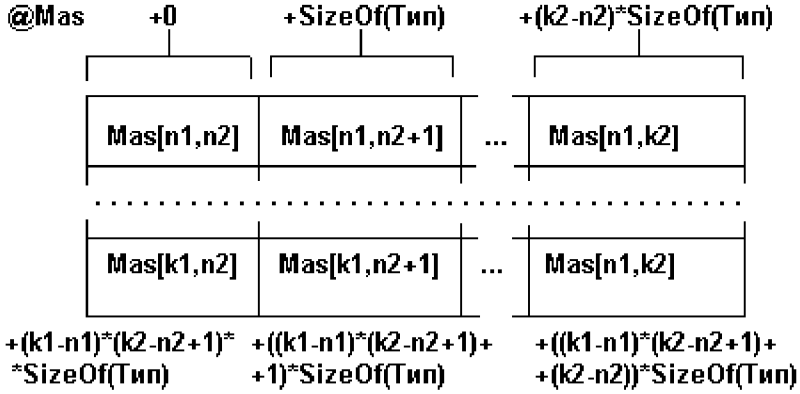


Рис. 3.3. Физическая структура двумерного массива из (k1-n1+1) строк и (k2-n2+1) столбцов

Многомерные массивы хранятся в непрерывной области памяти. Размер слота определяется базовым типом элемента массива. Количество элементов массива и размер слота определяют размер памяти для хранения массива. Принцип распределения элементов массива в памяти определен языком программирования. Так в FORTRAN элементы распределяются по столбцам - так, что быстрее меняется левые индексы, в PASCAL - по строкам - изменение индексов выполняется в направлении справа налево.

Количество байтов памяти, занятых двумерным массивом, определяется по формуле :

ByteSize = (k1-n1+1)*(k2-n2+1)*SizeOf(Тип) (3.3)

Адресом массива является адрес первого байта начального компонента массива. Смещение к элементу массива Mas[i1,i2] определяется по формуле:

ByteNumber = [(i1-n1)*(k2-n2+1)+(i2-n2)]*SizeOf(Тип) (3.4)

его адрес : @ByteNumber = @mas + ByteNumber.

Например:

var Mas : Array [3..5] [7..8] of Word;

Базовый тип элемента Word требует два байта памяти, тогда таблица 3.2 смещений элементов массива относительно @Mas будет следующей:

Таблица 3.2

Смещение, байт	Элемент массива	Смещениие, байт	Элемент массива
+0	Mas[3,7]	+2	Mas[3,8]
+4	Mas[4,7]	+6	Mas[4,8]
+8	Mas[5,7]	+10	Mas[5,8]

Этот массив будет занимать в памяти: (5-3+1)*(8-7+1)*2=12 байт; а адрес элемента Mas[4,8]: @Mas+((4-3)*(8-7+1)+(8-7)*2 = @Mas+6

3.2.3. Операции

Важнейшая операция физического уровня над массивом – доступ к заданному элементу. Как только реализован доступ к элементу, над ним может быть выполнена любая операция, имеющая смысл для того типа данных, которому соответствует элемент. Преобразование логической структуры в физическую, в ходе которого многомерная логическая структура массива преобразуется в одномерную физическую структуру, называется процессом линеаризации.

В соответствии с формулами (3.3), (3.4) и по аналогии с вектором (3.1), (3.2)для двумерного массива с границами изменения индексов:
[B(1)..E(1)][B(2)..E(2)], размещенного в памяти по строкам, адрес элемента с индексами [I(1),I(2)] может быть вычислен как:

Addr[I(1),I(2)] = Addr[B(1),B(2)] +
+ { [I(1)-B(1)] * [E(2)-B(2)+1] + [I(2)-B(2)] }*SizeOf(Тип) (3.5)

Обобщая (3.5) для массива произвольной размерности:

Mas[B(1)..E(2)][B(2)..E(2)]...[B(n)..E(n)]

получим:

$$Addr[I(1), I(2),..., I(n)] = Addr[B(1), B(2),...B(n)] -$$

$$Sizeof(\text{ТИП}) * \sum_{m=1}^{n}[B(m) * D(m)] + Sizeof(\text{ТИП}) * \sum_{m=1}^{n}[I(m) * D(m)] \quad (3.6)$$

где Dm зависит от способа размещения массива. При размещении по строкам:

D(m)=[E(m+1)-B(m+1)+1]*D(m+1), где m = n-1,...,1 и D(n)=1

при размещении по столбцам:

D(m)=[E(m-1)-B(m-1)+1]*D(m-1), где m = 2,...,n и D(1)=1

При вычислении адреса элемента наиболее сложным является вычисление третьей составляющей формулы (3.6), т.к. первые две не зависят от индексов и могут быть вычислены заранее. Для ускорения вычислений множители D(m) также могут быть вычислены заранее и сохраняться в дескрипторе массива. Дескриптор массива, таким образом, содержит:

- начальный адрес массива - Addr[I(1),I(2),...,I(n)];
- число измерений в массиве - n;
- постоянную составляющую формулы линеаризации (первые две составляющие формулы (3.6);
- для каждого из n измерений массива:
  - значения граничных индексов - B(i), E(i);
  - множитель формулы линеаризации - D(i).

Одно из определений массива гласит, что это вектор, каждый элемент которого - вектор. Некоторые языки программирования позволяют выделить из многомерного массива одно или несколько измерений и рассматривать их как массив меньшей мерности.

Например, если в PL/1-программе объявлен двумерный массив:

DECLARE A(10,10) BINARY FIXED;

то выражение: A[*,I] - будет обращаться к одномерному массиву, состоящему из элементов: A(1,I), A(2,I),...,A(10,I).

Символ-джокер "*" означает, что выбираются все элементы массива по тому измерению, которому соответствует заданный джокером индекс. Использование джокера позволяет также задавать групповые операции над всеми элементами массива или над выбранным его измерением, например: A(*,I) = A(*,I) + 1

К операциям логического уровня над массивами необходимо отнести такие как сортировка массива, поиск элемента по ключу. Наиболее распространенные алгоритмы поиска и сортировок будут рассмотрены в данной главе ниже.

3.2.4. Адресация массивов с помощью векторов Айлиффа

Из выше приведенных формул видно, что вычисление адреса элемента многомерного массива может потребовать много времени, поскольку при этом должны выполняться операции сложения и умножения, число которых пропорционально размерности массива. Операцию умножения можно исключить, если применять следующий метод.

Для массива любой мерности формируется набор дескрипторов: основного и несколько уровней вспомогательных дескрипторов, называемых векторами Айлиффа. Каждый вектор Айлиффа определённого уровня содержит указатель на нулевые компоненты векторов Айлиффа следующего, более низкого уровня, а векторы Айлиффа самого нижнего уровня содержат указатели групп элементов отображаемого массива. Основной дескриптор массива хранит указатель вектора Айлиффа первого уровня. При такой организации к произвольному элементу B(j1,j2,...,jn) многомерного массива можно обратиться пройдя по цепочке от основного дескриптора через соответствующие компоненты векторов Айлиффа.

На рис. 3.4 приведена физическая структура трёхмерного массива B[4..5,-1..1,0..1], представленная по методу Айлиффа. Из этого рисунка видно, что метод Айлиффа, увеличивая скорость доступа к элементам массива, приводит в то же время к увеличению суммарного объёма памяти, требуемого для представления массива. В этом заключается основной недостаток представления массивов с помощью векторов Айлиффа.

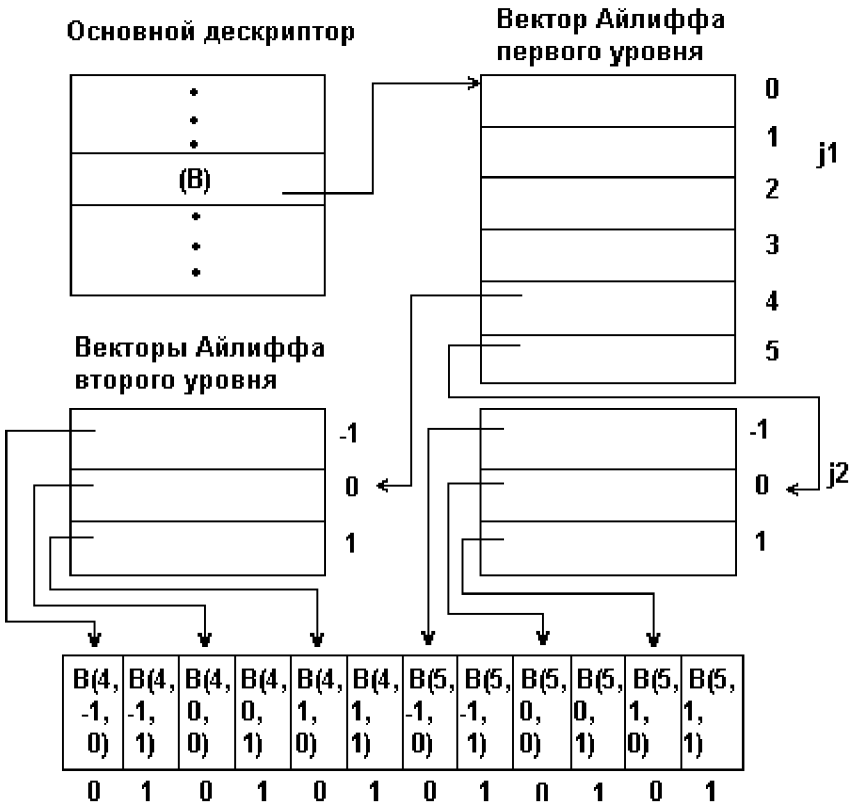


Рис. 3. 4. Представление массивов с помощью векторов Айлиффа

3.2.5. Специальные массивы

На практике встречаются массивы, которые в силу определенных причин могут записываться в память не полностью, а частично. Это особенно актуально для массивов настолько больших размеров, что для их хранения в полном объеме памяти может быть недостаточно. К таким массивам относятся симметричные и разреженные массивы.

СИММЕТРИЧНЫЕ МАССИВЫ. Двумерный массив, в котором количество строк равно количеству столбцов называется квадратной матрицей.

Квадратная матрица, у которой элементы, расположенные симметрично относительно главной диагонали, попарно равны друг другу, называется симметричной. Если матрица порядка n симметрична, то в ее физической структуре достаточно отобразить не n^2, а лишь n*(n+1)/2 её элементов. Иными словами, в памяти необходимо представить только верхний (включая и диагональ) треугольник квадратной логической структуры. Доступ к треугольному массиву организуется таким образом, чтобы можно было обращаться к любому элементу исходной логической структуры, в том числе и к элементам, значения которых хотя и не представлены в памяти, но могут быть определены на основе значений симметричных им элементов.

На практике для работы с симметричной матрицей разрабатываются следующие процедуры для:

а) преобразования индексов матрицы в индекс вектора,

б) формирования вектора и записи в него элементов верхнего треугольника элементов исходной матрицы,
в) получения значения элемента матрицы из ее упакованного представления. При таком подходе обращение к элементам исходной матрицы выполняется опосредованно, через указанные функции.

РАЗРЕЖЕННЫЕ МАССИВЫ. Разреженный массив - 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 массивов:
1) массивы, в которых местоположения элементов со значениями, отличными от фонового, могут быть математически описаны;
2) массивы со случайным расположением элементов.

В случае работы с разреженными массивами вопросы размещения их в памяти реализуются на логическом уровне с учетом их типа.

МАССИВЫ С МАТЕМАТИЧЕСКИМ ОПИСАНИЕМ МЕСТОПОЛОЖЕНИЯ НЕФОНОВЫХ ЭЛЕМЕНТОВ. К данному типу массивов относятся массивы, у которых местоположения элементов со значениями, отличными от фонового, могут быть математически описаны, т. е. в их расположении есть какая-либо закономерность.

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

Ненулевые значения хранятся, как правило, в одномерном массиве, а связь между местоположением в исходном, разреженном, массиве и в новом, одномерном, описывается математически с помощью формулы, преобразующей индексы массива в индексы вектора.

На практике для работы с разреженным массивом разрабатываются такие функции:
а) для преобразования индексов массива в индекс вектора;
б) для получения значения элемента массива из ее упакованного представления по двум индексам (строка, столбец);
в) для записи значения элемента массива в ее упакованное представление по двум индексам.

При таком подходе обращение к элементам исходного массива 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)*XM+x)/2),
где L - индекс в линейном представлении;
x, y - соответственно строка и столбец в двумерном представлении;
XM - количество элементов в строке исходной матрицы.

В программном примере 3.1 приведен модуль, обеспечивающий работу с такой матрицей (предполагается, что размер матрицы XM заранее известен).

```
{===== Программный пример 3.1 =====}
Unit ComprMatr;
Interface
 Function PutTab(y,x,value : integer) : boolean;
 Function GetTab(x,y: integer) : integer;
Implementation
 Const XM=...;
 Var arrp: array[1..XM*XM div 2] of integer;
 Function NewIndex(y, x : integer) : integer;
   var  i: integer;
   begin   NewIndex:=((y-1)*XM+x) div 2);  end;
 Function PutTab(y,x,value : integer) : boolean;
   begin
   if NOT ((x mod 2<>0) and (y mod 2<>0)) or
     NOT ((x mod 2=0) and (y mod 2=0))  then begin
       arrp[NewIndex(y,x)]:=value;  PutTab:=true; end
   else PutTab:=false;
   end;
 Function GetTab(x,y: integer) : integer;
   begin
   if ((x mod 2<>0) and (y mod 2<>0)) or
     ((x mod 2=0) and (y mod 2=0)) then GetTab:=0
   else GetTab:=arrp[NewIndex(y,x)];
   end;
end.
```

Сжатое представление матрицы хранится в векторе arrp.

Функция NewIndex выполняет пересчет индексов по вышеприведенной формуле и возвращает индекс элемента в векторе arrp.

Функция PutTab выполняет сохранение в сжатом представлении одного элемента с индексами x, y и значением value. Сохранение выполняется только в том случае, если индексы x, y адресуют не заведомо нулевой элемент. Если сохранение выполнено, функция возвращает true, иначе - false.

Для доступа к элементу по индексам двумерной матрицы используется функция GetTab, которая по индексам x, y возвращает выбранное значение. Если индексы адресуют заведомо нулевой элемент матрицы, функция возвращает 0.

Обратите внимание на то, что массив arrp, а также функция NewIndex не описаны в секции IMPLEMENTATION модуля. Доступ к содержимому матрицы извне возможен только через входные точки PutTab, GetTab с заданием двух индексов.

В программном примере 3.2 та же задача решается несколько иным способом: для матрицы создается дескриптор - массив desc, который заполняется при инициализации матрицы таким образом, что i-ый элемент массива desc содержит индекс первого элемента i-ой строки матрицы в ее линейном представлении. Процедура инициализации InitTab включена в число входных точек модуля и должна вызываться перед началом работы с матрицей. Но доступ к каждому элементу матрицы (функция NewIndex) упрощается и выполняется быстрее: по номеру строки y из дескриптора сразу выбирается индекс начала строки и к нему прибавляется смещение элемента из столбца x. Процедуры PutTab и GetTab - такие же, как и в примере 3.1 поэтому здесь не приводятся.

```
{===== Программный пример 3.2 =====}
Unit ComprMatr;
Interface
 Function PutTab(y,x,value : integer) : boolean;
 Function GetTab(x,y: integer) : integer;
 Procedure InitTab;
Implementation
 Const XM=...;
 Var arrp: array[1..XM*XM div 2] of integer;
    desc: array[1..XM] of integer;
 Procedure InitTab;
 var i : integer;
  begin
     desc[1]:=0;  for i:=1 to XM do desc[i]:=desc[i-1]+XM;      end;
 Function NewIndex(y, x : integer) : integer;
 var  i: integer;
 begin  NewIndex:=desc[y]+x div 2;  end;
end.
```

РАЗРЕЖЕННЫЕ МАССИВЫ СО СЛУЧАЙНЫМ РАСПОЛОЖЕНИЕМ ЭЛЕМЕНТОВ. К данному типу массивов относятся массивы, у которых местоположения элементов со значениями отличными от фонового, не могут быть математически описаны, т. е. в их расположении нет какой-либо закономерности.

```
| 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 |
```

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

ПРЕДСТАВЛЕНИЕ РАЗРЕЖЕННЫХ МАТРИЦ МЕТОДОМ ПОСЛЕДОВАТЕЛЬНОГО РАЗМЕЩЕНИЯ. Один из основных способов

хранения подобных разреженных матриц заключается в запоминании ненулевых элементов в одномерном массиве и идентификации каждого элемента массива индексами строки и столбца, как на рис. 3.5 а).

Доступ к элементу массива A с индексами i и j выполняется выборкой индекса i из вектора ROW, индекса j из вектора COLUM и значения элемента из вектора A. Слева указан индекс k векторов наибольшее значение, которого определяется количеством нефоновых элементов. Отметим, что элементы массива обязательно запоминаются в порядке возрастания номеров строк.

Более эффективное представление, с точки зрения требований к памяти и времени доступа к строкам матрицы, показано на рис. 3.5.б). Вектор ROW уменьшен, количество его элементов соответствует числу строк исходного массива A, содержащих нефоновые элементы. Этот вектор получен из вектора ROW рис. 3.5.а) так, что его i-й элемент является индексом k для первого нефонового элемента i-ой строки.

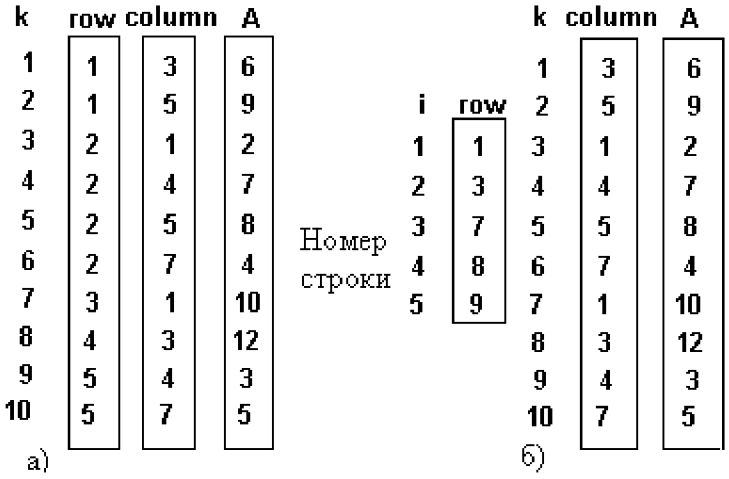


Рис. 3.5. Последовательное представление разреженных матриц

Представление матрицы A, данное на рис. 3.5, сокращает требования к объему памяти более чем в 2 раза. Для больших матриц экономия памяти очень важна. Способ последовательного распределения имеет также то преимущество, что операции над матрицами могут быть выполнены быстрее, чем это возможно при представлении в виде последовательного двумерного массива, особенно если размер матрицы велик.

ПРЕДСТАВЛЕНИЕ РАЗРЕЖЕННЫХ МАТРИЦ МЕТОДОМ СВЯЗАННЫХ СТРУКТУР. Методы последовательного размещения для представления разреженных матриц обычно позволяют быстрее выполнять операции над матрицами и более эффективно использовать память, чем методы со связанными структурами. Однако последовательное представление матриц имеет определенные недостатки. Так включение и исключение новых элементов матрицы вызывает необходимость перемещения большого числа других элементов. Если включение новых элементов и их исключение осуществляется часто, то должен быть выбран описываемый ниже метод связанных структур.

Метод связанных структур, однако, переводит представляемую структуру данных в другой раздел классификации. При том, что логическая структура данных остается статической, физическая структура становится динамической.

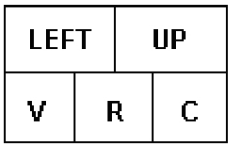


Рис. 3.6. Формат вершины для представления разреженных матриц

Для представления разреженных матриц требуется базовая структура вершины (рис.3.6), называемая MATRIX_ELEMENT ("элемент матрицы"). Поля V, R и C каждой из этих вершин содержат соответственно значение, индексы строки и столбца элемента матрицы. Поля LEFT и UP являются указателями на следующий элемент для строки и столбца в циклическом списке, содержащем элементы матрицы. Поле LEFT указывает на вершину со следующим наименьшим номером строки.

На рис. 3.7 приведена многосвязная структура, в которой используются вершины такого типа для представления матрицы A, описанной ранее в данном пункте. Циклический список представляет все строки и столбцы. Список столбца может содержать общие вершины с одним списком строки или более. Для того, чтобы обеспечить использование более эффективных алгоритмов включения и исключения элементов, все списки строк и столбцов имеют головные вершины.

Головная вершина каждого списка строки содержит нуль в поле C; аналогично каждая головная вершина в списке столбца имеет нуль в поле R. 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 вершин в структуру.

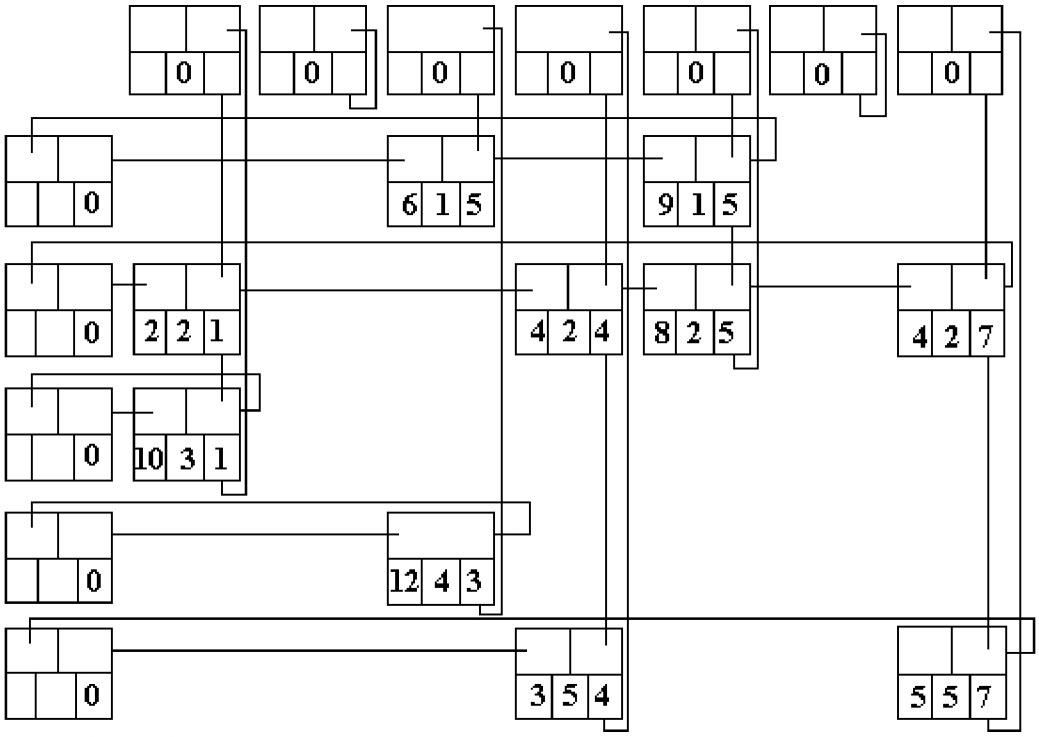


Рис. 3.7. Многосвязная структура для представления матрицы A

Предполагается, что новые вершины, которые должны быть добавлены к матрице, обычно располагаются в порядке убывания индексов строк и индексов столбцов. Если это так, то новая вершина всегда добавляется после головной и не требуется никакого просмотра списка.

3.3. Множества

ЛОГИЧЕСКАЯ СТРУКТУРА. Множество - такая структура, которая представляет собой набор неповторяющихся данных одного и того же типа. Множество может принимать все значения базового типа. Базовый тип не должен превышать 256 возможных значений. Поэтому базовым типом множества могут быть byte, char и производные от них типы.

ФИЗИЧЕСКАЯ СТРУКТУРА. Множество в памяти хранится как массив битов, в котором каждый бит указывает, является ли элемент принадлежащим объявленному множеству или нет. Т.о. максимальное число элементов множества 256, а данные типа множество могут занимать не более 32-ух байт.

Число байтов, выделяемых для данных типа множество, вычисляется по формуле:

ByteSize = (max div 8)-(min div 8) + 1,

где max и min - верхняя и нижняя границы базового типа данного множества.

Номер байта для конкретного элемента E вычисляется по формуле:

ByteNumber = (E div 8)-(min div 8),

номер бита внутри этого байта по формуле:

BitNumber = E mod 8

{===== Программный пример 3.3 =====}

```
const  max=255; min=0; E=13;
var    S : set of byte;
       ByteSize, ByteNumb, BitNumb  : byte;
       begin
            S:=[];              {  обнуление множества }
            S:=S+[E];           {  запись числа в множество }
            ByteSize:=(max div 8)-(min div 8)+1;
            Bytenumb:=(E div 8)-(min div 8); BitNumb:=E mod 8;
            writeln(bytesize);        { на экране 32  }
            writeln(bytenumb);        { на экране 1   }
            writeln(bitnumb);         { на экране 5   }
end.
```

3.3.1. Числовые множества

Стандартный числовой тип, который может быть базовым для формирования множества – тип byte.

Множество хранится в памяти как показано в таблице 3.3.

Таблица 3.3

Смещение (байт)	Представление байта в машинной памяти (номера разрядов)
@S+0	7 6 5 4 3 2 1 0
. . . . .	. . . . .
@S+31	255 254 253 252 251 250 249 248
	7 0

В таблице : @S – адрес данного типа множества.

Бит поля установлен в 1, если элемент входит в множество, и в 0 - если не входит.

Например, S: set of byte; S:=[15,19];

Содержимое памяти при этом будет следующим:

@S+0 - 00000000 @S+2 - 00001000

@S+1 - 10000000 . . . . .

@S+31 – 00000000

3.3.2. Символьные множества

Символьные множества хранятся в памяти также как и числовые множества. Разница лишь в том, что хранятся не числа, а коды ASCII символов.

Например, S : set of char; S:=['A','C'];

В этом случае представление множества S в памяти выглядит следующим образом :

@S+0 - 00000000 . . . . .

. . . . . @S+31 - 00000000

@S+8 - 00001010

3.3.3. Множество из элементов перечислимого типа

Множество, базовым типом которого есть перечислимый тип, хранится также, как множество, базовым типом которого является тип byte. Однако, в памяти занимает место, которое зависит от количества элементов в перечислимом типе.

Пример:

```
Type
    Video=(MDA,CGA,HGC,EGA,EGAm,VGA,VGAm,SVGA,PGA,XGA);
Var  S : set of Video;
```

В памяти будет занимать: ByteSize = (9 div 8)-(0 div 8)+1=2 байта

При этом память для переменной S будет распределена как показано на рис. 3.8.

	SVGA	VGAm	VGA	EGAm	EGA	HGC	CGA	MDA
@S+0								
@S+1	0	0	0	0	0	0	XGA	PGA
	7							0

Рис. 3.8. Распределение памяти для переменной типа set of Video.

Если выполнить оператор S:=[CGA,SVGA], содержимое памяти при этом будет: @S+0 - 10000010 @S+1 - 00000000.

3.3.4. Множество от интервального типа

Множество, базовым типом которого есть интервальный тип, хранится также, как множество, базовым типом которого является тип byte. Однако, в памяти занимает место, которое зависит от количества элементов, входящих в объявленный интервал.

Например,

```
type  S=10..17;
var   I:set of S;
```

Это не значит, что первый элемент будет начинаться с 10-того или 0-го бита, как может показаться на первый взгляд. Как видно из формулы вычисления смещения внутри байта 10 mod 8 = 2, смещение первого элемента множества I начнётся со второго бита. И, хотя множество этого интервала свободно могло поместиться в один байт, оно займёт

(17 div 8)-(10 div 8)+1 = 2 байта.

В памяти это множество имеет представление как на рис. 3.9.

	15	14	13	12	11	10	0	0
@S+0								
@S+1	0	0	0	0	0	0	17	16
	7							0

Рис. 3.9. Представление переменной типа set of S.

Для конструирования множеств интервальный тип самый экономичный, т.к. занимает память в зависимости от заданных границ.

Например,

```
type  S = 510..520;
var   I : S;
        begin  I:=[512];  end.
```

Представление в памяти переменной I будет:

@i+0 - 00000000 @i+1 - 00000001

3.3.5. Операции над множествами

Пусть S1, S2, S3 : set of byte , Над этими множествами определены следующие специфические операции.

1) Объединение множеств: S2+S3. Результатом является множество, содержащее элементы обоих исходных множеств.

2) Пересечение множеств: S2*S3. Результатом является множество, содержащее общие элементы обоих исходных множеств.

3) Проверка на вхождение элемента в множество: a in S1. Результатом этой операции является значение логического типа - true, если элемент a входит в множество S1, false - в противном случае.

3.4. Записи

3.4.1. Логическое и машинное представление записей

Запись – конечное упорядоченное множество полей, характеризующихся различным типом данных. Записи являются чрезвычайно удобным средством для представления программных моделей реальных объектов предметной области, ибо, как правило, каждый такой объект обладает набором свойств, характеризуемых данными различных типов. Пример записи – совокупность сведений о некотором студенте. В данном примере объектом является "студент" и обладает такими свойствами: личный номер" – характеризуется целым положительным числом, "фамилия-имя-отчество" – характеризуется строкой символов и т.д. Пример:

```
var  rec: record
        num  :byte;              {   личный номер студента        }
        name :string[20];        {   Ф.И.О.                       }
        fac, group:string[7];    {  факультет,  группа            }
        math,comp,lang :byte;{оценки по математике, выч.тех-  }
end;                             {нике, ин. языку                 }
```

В памяти эта структура может быть представлена в одном из двух видов:

а) в виде последовательности полей, занимающих непрерывную область памяти (рис. 3.10). При такой организации достаточно иметь один указатель на начало области и смещение относительно начала. Это дает экономию памяти, но лишнюю трату времени на вычисление адресов полей записи.

@rec	+0	+1	+21	+29	+37	+38	+39
	24	Иванов В.И.	АП	54	4	5	5

Рис. 3.10. Представление в памяти переменной типа record в виде последовательности полей

б) в виде связного списка с указателями на значения полей записи. При такой организации имеет место быстрое обращение к элементам, но очень неэкономичный расход памяти для хранения.

Структура хранения в памяти связного списка с указателями на элементы приведена на рис. 3.11. При этом для экономии объема памяти, отводимой под запись, значения некоторых ее полей хранятся в самом дескрипторе, вместо указателей, тогда в дескрипторе должны быть записаны соответствующие признаки.

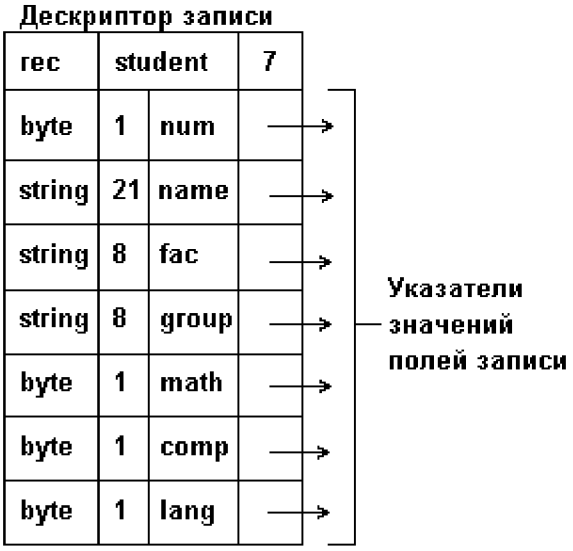


Рис. 3.11. Представление в памяти переменной типа record в виде связного списка

В соответствии с общим подходом языка С дескриптор записи (в этом языке записи называются структурами) не сохраняется до выполнения программы. Поля структуры просто располагаются в смежных слотах памяти, обращения к отдельным полям заменяются на их адреса еще на этапе компиляции.

Полем записи может быть в свою очередь интегрированная структура данных – вектор, массив или другая запись. В некоторых языках программирования (COBOL, PL/1) при описании вложенных записей указывается уровень вложенности, в других (PASCAL, C) – уровень вложенности определяется автоматически.

Полем записи может быть другая запись, но ни в коем случае не такая же. Это связано прежде всего с тем, что компилятор должен выделить для размещения записи память. Предположим, описана запись вида:

```
type   rec = record
                f1 : integer;
```

```
        f2 : char[2];
        f3 : rec;
end;
```

Как компилятор будет выделять память для такой записи? Для поля f1 будет выделено 2 байта, для поля f2 – 2 байта, а поле f3 – запись, которая в свою очередь состоит из f1 (2 байта), f2 (2 байта) и f3, которое... и т.д. Недаром компилятор С, встретив подобное описание, выдает сообщение о нехватке памяти.

Однако, полем записи вполне может быть указатель на другую такую же запись: размер памяти, занимаемой указателем известен и проблем с выделением памяти не возникает. Этот прием широко используется в программировании для установления связей между однотипными записями (см. главу 5).

3.4.2. Операции над записями

Важнейшей операцией для записи является операция доступа к выбранному полю записи – операция квалификации. Практически во всех языках программирования обозначение этой операции имеет вид:

<имя переменной-записи>.<имя поля>

Так, для записи, описанной в начале п.3.4.1, конструкции:
stud1.num и stud1.math будут обеспечивать обращения к полям num и math соответственно.

Над выбранным полем записи возможны любые операции, допустимые для типа этого поля. Большинство языков программирования поддерживает некоторые операции, работающие с записью, как с единым целым, а не с отдельными ее полями. Это операции присваивания одной записи значения другой однотипной записи и сравнения двух однотипных записей на равенство/неравенство. В тех же случаях, когда такие операции не поддерживаются языком явно (язык С), они могут выполняться над отдельными полями записей или же записи могут копироваться и сравниваться как неструктурированные области памяти.

3.5. Записи с вариантами

В ряде прикладных задач программист может столкнуться с группами объектов, чьи наборы свойств перекрываются лишь частично. Обработка таких объектов производится по одним и тем же алгоритмам, если обрабатываются общие свойства объектов, или по разным - если обрабатываются специфические свойства. Можно описать все группы единообразно, включив в описание все наборы свойств для всех групп, но такое описание будет неэкономичным с точки зрения расходуемой памяти и неудобным с логической точки зрения.

Если же описать каждую группу собственной структурой, теряется возможность обрабатывать общие свойства по единым алгоритмам.

Для задач подобного рода развитые языки программирования (С,

PASCAL) предоставляют в распоряжение программиста записи с вариантами. Запись с вариантами состоит из двух частей. В первой части описываются поля, общие для всех групп объектов, моделируемых записью. Среди этих полей обычно бывает поле, значение которого позволяет идентифицировать группу, к которой данный объект принадлежит и, следовательно, какой из вариантов второй части записи должен быть использован при обработке. Вторая часть записи содержит описания непересекающихся свойств – для каждого подмножества таких свойств – отдельное описание. Язык программирования может требовать, чтобы имена полей-свойств не повторялись в разных вариантах (PASCAL), или же требовать именования каждого варианта (C). В первом случае идентификация поля, находящегося в вариантной части записи при обращении к нему ничем не отличается от случая простой записи:

<имя переменной-записи>.<имя поля>

Во втором случае идентификация немного усложняется:

<имя переменной-записи>.<имя варианта>.<имя поля>

Рассмотрим использование записей с вариантами на примере. Пусть требуется размещать на экране видеотерминала простые геометрические фигуры – круги, прямоугольники, треугольники. Для "базы данных", которая будет описывать состояние экрана, удобно представлять все фигуры однотипными записями. Для любой фигуры описание ее должно включать в себя координаты некоторой опорной точки (центра, правого верхнего угла, одной из вершин) и код цвета. Другие же параметры построения будут разными для разных фигур. Так для круга – радиус; для прямоугольника – длины непараллельных сторон; для треугольника – координаты двух других вершин.

Запись с вариантами для такой задачи в языке PASCAL выглядит, как:

```
type   figure = record
                  fig_type : char;      { тип фигуры }
                  x0, y0   : word;      { координаты опорной точки }
                  color    : byte;      { цвет }
                  case fig_t : char of
                        'C': ( radius : word);        { радиус окружности }
                        'R': (len1, len2 : word);   {          длины          сторон
                  прямоугольника }
                        'T': (x1,y1,x2,y2 : word); { координаты двух вершин }
            end;
```

а в языке C, как:

```
typedef struct
      { char fig_type;      /* тип фигуры */
      unsigned int x0, y0; /* координаты опорной точки */
      unsigned char color; /* цвет */
      union
      { struct
            { unsigned int radius;    /* радиус окружности */
```

```
        } cyrcle;
struct
        { unsigned int len1, len2;  /* длины сторон прямоугольн.*/
        } rectangle;
struct
{ unsigned int x1,y1,x2,y2; /* координаты двух вершин */
} triangle;   } fig_t;        } figure;
```

И если в программе определена переменная fig1 типа figure, в которой хранится описание окружности, то обращение к радиусу этой окружности в языке PASCAL будет иметь вид: fig1.radius, а в C: fig1.circle.radius

Поле с именем fig_type введено для представления идентификатора вида фигуры, который, например, может кодироваться символами: "C" – окружность или "R" – прямоу-гольник, или "T" – треугольник.

Выделение памяти для записи с вариантами показано на рис. 3.12.

Как видно из рисунка, под запись с вариантами выделяется в любом случае объем памяти, достаточный для размещения самого большого варианта. Если же выделенная память используется для меньшего варианта, часть ее остается неиспользуемой. Общая для всех вариантов часть записи размещается так, чтобы смещения всех полей относительно начала записи были одинаковыми для всех вариантов. Очевидно, что наиболее просто это достигается размещением общих полей в начале записи, но это не строго обязательно. Вариантная часть может и "вклиниваться" между полями общей части. Поскольку в любом случае вариантная часть имеет фиксированный максимальный) размер, смещения полей общей части также останутся фиксированными.

Рис. 3.12 Выделение памяти для записи с вариантами

3.6. Таблицы

Когда речь шла о записях, было отмечено, что полями записи могут быть интегрированные структуры данных – векторы, массивы, другие записи. Аналогично и элементами векторов и массивов могут быть также интегрированные структуры. Одна из таких сложных структур – таблица. С физической точки зрения таблица представляет собой вектор, элементами которого являются записи. Характерной логической особенностью таблиц, которая и определила их рассмотрение в отдельном разделе, является то, что

доступ к элементам таблицы производится не по номеру (индексу), а по ключу – по значению одного из свойств объекта, описываемого записью-элементом таблицы. Ключ – это свойство, идентифицирующее данную запись во множестве однотипных записей. Как правило, к ключу предъявляется требование уникальности в данной таблице. Ключ может включаться в состав записи и быть одним из ее полей, но может и не включаться в запись, а вычисляться по положению записи. Таблица может иметь один или несколько ключей. Например, при интеграции в таблицу записей о студентах (описание записи приведено в п.3.4.1) выборка может производиться как по личному номеру студента, так и по фамилии.

Основной операцией при работе с таблицами является операция доступа к записи по ключу. Она реализуется процедурой поиска. Поскольку поиск может быть значительно более эффективным в таблицах, упорядоченных по значениям ключей, довольно часто над таблицами необходимо выполнять операции сортировки.

Иногда различают таблицы с фиксированной и с переменной длиной записи. Очевидно, что таблицы, объединяющие записи совершенно идентичных типов, будут иметь фиксированные длины записей. Необходимость в переменной длине может возникнуть в задачах, подобных тем, которые рассматривались для записей с вариантами. Как правило таблицы для таких задач и составляются из записей с вариантами, т.е. сводятся к фиксированной (максимальной) длине записи. Значительно реже встречаются таблицы с действительно переменной длиной записи. Хотя в таких таблицах и экономится память, но возможности работы с такими таблицами ограничены, так как по номеру записи невозможно определить ее адрес. Таблицы с записями переменной длины обрабатываются только последовательно – в порядке возрастания номеров записей. Доступ к элементу такой таблицы обычно осуществляется в два шага. На первом шаге выбирается постоянная часть записи, в которой содержится – в явном или неявном виде – длина записи. На втором шаге выбирается переменная часть записи в соответствии с ее длиной. Прибавив к адресу текущей записи ее длину, получают адрес следующей записи.

Так таблица с записями переменной длины может, например, рассматриваться в некоторых задачах программируемых в машинных кодах. Каждая машинная команда – запись, состоит из одного или нескольких байт. Первый байт – всегда код операции, количество и формат остальных байтов определяется типом команды. Процессор выбирает байт по адресу, задаваемому программным счетчиком, и определяет тип команды. По типу команды процессор определяет ее длину и выбирает остальные ее байты. Содержимое программного счетчика увеличивается на длину команды.

3.7. Операции логического уровня над статическими структурами. Поиск

В этом и следующих разделах представлен ряд алгоритмов поиска данных и сортировок, выполняемых на статических структурах данных, так как это характерные операции логического уровня для таких структур. Однако, те же операции и те же алгоритмы применимы и к данным, имеющим логическую структуру таблицы, но физически размещенным в динамической памяти или на внешней памяти, а также к логическим таблицам любого физического представления, обладающим изменчивостью.

Объективным критерием, позволяющим оценить эффективность того или иного алгоритма, является, так называемый, порядок алгоритма. Порядком алгоритма называется функция O(N), позволяющая оценить зависимость времени выполнения алгоритма от объема перерабатываемых данных (N – количество элементов в массиве или таблице). Эффективность алгоритма тем выше, чем меньше время его выполнения зависит от объема данных. Большинство алгоритмов с точки зрения порядка сводятся к трем основным типам:

- степенные – $O(N^a)$;
- линейные – O(N);
- логарифмические – $O(\log_a(N))$.

Эффективность степенных алгоритмов обычно считается плохой, линейных – удовлетворительной, логарифмических – хорошей.

Аналитическое определение порядка алгоритма, хотя часто и сложно, но возможно в большинстве случаев. Возникает вопрос: зачем тогда нужно такое разнообразие алгоритмов, например, сортировок, если есть возможность раз и навсегда определить алгоритм с наилучшим аналитическим показателем эффективности и оставить "право на жизнь" исключительно за ним? Ответ прост: в реальных задачах имеются ограничения, определяемые как логикой задачи, так и свойствами конкретной вычислительной среды, которые могут помогать или мешать программисту, и которые могут существенно влиять на эффективность данной конкретной реализации алгоритма. Поэтому выбор того или иного алгоритма всегда остается за программистом.

В последующем изложении все описания алгоритмов даны для работы с таблицей, состоящей из записей R[1], R[2], ..., R[N] с ключами K[1], K[2], ..., K[N]. Во всех случаях N – количество элементов таблицы. Программные примеры для сокращения их объема работают с массивами целых чисел. Такой массив можно рассматривать как вырожденный случай таблицы, каждая запись которой состоит из единственного поля, которое является также и ключом. Во всех программных примерах следует считать уже определенными:

- константу N – целое положительное число, число элементов в массиве;
- константу EMPTY – целое число, признак "пусто" (EMPTY=-1);

- тип – type SEQ = array[1..N] of integer; сортируемые последовательности.

3.7.1. Последовательный или линейный поиск

Простейшим методом поиска элемента, находящегося в неупорядоченном наборе данных, по значению его ключа является последовательный просмотр каждого элемента набора, который продолжается до тех пор, пока не будет найден желаемый элемент. Если просмотрен весь набор, но элемент не найден – значит, искомый ключ отсутствует в наборе.

Для последовательного поиска в среднем требуется (N+1)/2 сравнений. Таким образом, порядок алгоритма – линейный – O(N).

Программная иллюстрация линейного поиска в неупорядоченном массиве приведена в следующем примере, где a – исходный массив, key – ключ, который ищется; функция возвращает индекс найденного элемента или EMPTY – если элемент отсутствует в массиве.

```
{===== Программный пример 3.4 =====}
Function LinSearch( a : SEQ; key : integer) : integer;
  var i : integer;
  for i:=1 to N do                  { перебор эл-тов массива        }
    if a[i]=key then begin          {ключ найден - возврат индекса }
      LinSearch:=i; Exit;   end;
  LinSearch:=EMPTY;   {просмотрен весь массив, но ключ не найден }
end;
```

3.7.2. Бинарный поиск

Другим относительно простым методом доступа к элементу является метод бинарного (дихотомического, двоичного) поиска, который выполняется в заведомо упорядоченной последовательности элементов. Записи в таблицу заносятся в лексикографическом (символьные ключи) или численно (числовые ключи) возрастающем порядке. Для достижения упорядоченности может быть использован какой-либо из методов сортировки.

В рассматриваемом методе поиск отдельной записи с определенным значением ключа напоминает поиск фамилии в телефонном справочнике. Сначала приближенно определяется запись в середине таблицы и анализируется значение ее ключа. Если оно слишком велико, то анализируется значение ключа, соответствующего записи в середине первой половины таблицы, и указанная процедура повторяется в этой половине до тех пор, пока не будет найдена требуемая запись.

Если значение ключа слишком мало, испытывается ключ, соответствующий записи в середине второй половины таблицы, и процедура повторяется в этой половине. Этот процесс продолжается до тех пор, пока не

будет найден требуемый ключ или не станет пустым интервал, в котором осуществляется поиск.

Для того, чтобы найти нужную запись в таблице, в худшем случае требуется $\log_2(N)$ сравнений. Это значительно лучше, чем при последовательном поиске.

Программная иллюстрация бинарного поиска в упорядоченном массиве приведена в следующем примере, где a – исходный массив, key – ключ, который ищется; функция возвращает индекс найденного элемента или EMPTY – если элемент отсутствует в массиве.

```
{===== Программный пример 3.5 =====}
Function BinSearch(a : SEQ; key : integer) : integer;
Var b, e, i : integer;
begin
  b:=1; e:=N;       { начальные значения границ  }
  while b<=e do   { цикл, пока интервал поиска не сузится до 0 }
  begin      i:=(b+e) div 2;       {середина интервала }
   if a[i]=key then
   begin BinSearch:=i; Exit;   {ключ найден - возврат индекса }
   end       else
      if a[i]<key then b:=i+1 { поиск в правом подинтервале }
      else e:=i-1;           { поиск в левом подинтервале }
   end; BinSearch:=EMPTY; { ключ не найден  }
end;
```

Трассировка бинарного поиска ключа 275 в исходной последовательности:

75, 151, 203, 275, 318, 489, 524, 519, 647, 777

представлена в таблице 3.4.

Таблица 3.4

Итерация	b	e	i	K[i]
1	1	10	5	318
2	1	4	2	151
3	3	4	3	203
4	4	4	4	275

Алгоритм бинарного поиска можно представить и несколько иначе, используя рекурсивное описание. В этом случае граничные индексы интервала b и e являются параметрами алгоритма.

Рекурсивная процедура бинарного поиска представлена в программном примере 3.6. Для выполнения поиска необходимо при вызове процедуры задать значения ее формальных параметров b и e - 1 и N соответственно, где b, e - граничные индексы области поиска.

```
{===== Программный пример 3.6 =====}
```

```
Function BinSearch( a: SEQ; key, b, e : integer) : integer;
Var i : integer;
begin
  if b>e then BinSearch:=EMPTY      { проверка ширины  интервала  }
  else begin
    i:=(b+e) div 2;                 { середина интервала          }
    if a[i]=key then BinSearch:=I   {ключ найден, возврат индекса }
       else   if a[i]<key then      { поиск в правом подинтервале }
                BinSearch:=BinSearch(a,key,i+1,e)
                           else     { поиск в левом подинтервале  }
       BinSearch:=BinSearch(a,key,b,i-1);
 end; end;
```

Известно несколько модификаций алгоритма бинарного поиска, выполняемых на деревьях, которые будут рассмотрены ниже.

3.8 Операции логического уровня над статическими структурами. Сортировка.

Для самого общего случая сформулируем задачу сортировки таким образом: имеется некоторое неупорядоченное входное множество ключей и должны получить выходное множество тех же ключей, упорядоченных по возрастанию или убыванию в численном или лексикографическом порядке.

Из всех задач программирования сортировка, возможно, имеет самый богатый выбор алгоритмов решения. Назовем некоторые факторы, которые влияют на выбор алгоритма (помимо порядка алгоритма).

1). Имеющийся ресурс памяти: должны ли входное и выходное множества располагаться в разных областях памяти или выходное множество может быть сформировано на месте входного. В последнем случае имеющаяся область памяти должна в ходе сортировки динамически перераспределяться между входным и выходным множествами; для одних алгоритмов это связано с большими затратами, для других - с меньшими.

2). Исходная упорядоченность входного множества: во входном множестве (даже если оно сгенерировано датчиком случайных величин) могут попадаться упорядоченные участки. В предельном случае входное множество может оказаться уже упорядоченным. Одни алгоритмы не учитывают исходной упорядоченности и требуют одного и того же времени для сортировки любого (в том числе и уже упорядоченного) множества данного объема, другие выполняются тем быстрее, чем лучше упорядоченность на входе.

3). Временные характеристики операций: при определении порядка алгоритма время выполнения считается обычно пропорциональным числу сравнений ключей. Ясно, однако, что сравнение числовых ключей выполняется быстрее, чем строковых, операции пересылки, характерные для некоторых алгоритмов, выполняются тем быстрее, чем меньше объем записи,

и т.п. В зависимости от характеристик записи таблицы может быть выбран алгоритм, обеспечивающий минимизацию числа тех или иных операций.

4). Сложность алгоритма является не последним соображением при его выборе. Простой алгоритм требует меньшего времени для его реализации и вероятность ошибки в реализации его меньше. При промышленном изготовлении программного продукта требования соблюдения сроков разработки и надежности продукта могут даже превалировать над требованиями эффективности функционирования.

Разнообразие алгоритмов сортировки требует некоторой их классификации. Выбран один из применяемых для классификации подходов, ориентированный прежде всего на логические характеристики применяемых алгоритмов. Согласно этому подходу любой алгоритм сортировки использует одну из следующих четырех стратегий (или их комбинацию).

1). Стратегия выборки. Из входного множества выбирается следующий по критерию упорядоченности элемент и включается в выходное множество на место, следующее по номеру.

2). Стратегия включения. Из входного множества выбирается следующий по номеру элемент и включается в выходное множество на то место, которое он должен занимать в соответствии с критерием упорядоченности.

3). Стратегия распределения. Входное множество разбивается на ряд подмножеств (возможно, меньшего объема) и сортировка ведется внутри каждого такого подмножества.

4). Стратегия слияния. Выходное множество получается путем слияния маленьких упорядоченных подмножеств.

Далее приводится обзор (далеко не полный) методов сортировки, сгруппированных по стратегиям, применяемым в их алгоритмах.

Все алгоритмы рассмотрены для случая упорядочения по возрастанию ключей.

3.8.1. Сортировки выборкой

СОРТИРОВКА ПРОСТОЙ ВЫБОРКОЙ. Данный метод реализует практически "дословно" сформулированную выше стратегию выборки. Порядок алгоритма простой выборки – $O(N^2)$. Количество пересылок – N.

Алгоритм сортировки простой выборкой иллюстрируется программным примером 3.7.

В программной реализации алгоритма возникает проблема значения ключа "пусто". Довольно часто программисты используют в качестве такового некоторое заведомо отсутствующее во входной последовательности значение ключа, например, максимальное из теоретически возможных значений. Другой, более строгий подход – создание отдельного вектора, каждый элемент которого имеет логический тип и отражает состояние соответствующего элемента входного множества ("истина" – "непусто", "ложь" – "пусто"). Именно такой подход реализован в нашем программном примере. Роль входной последовательности здесь выполняет параметр a,

роль выходной – параметр b, роль вектора состояний – массив c. Алгоритм несколько усложняется за счет того, что для установки начального значения при поиске минимума приходится отбрасывать уже "пустые" элементы.

```
{===== Программный пример 3.7 =====}
Procedure Sort( a : SEQ; var b : SEQ);
Var  i, j, m : integer;
    c: array[1..N] of boolean;    {состояние эл-тов вх.множества}
begin
  for i:=1 to N do c[i]:=true;     { сброс отметок }
  for i:=1 to N do           {поиск 1-го невыбранного эл. во вх.множестве}
   begin j:=1;
        while not c[j] do j:=j+1;
        m:=j;          { поиск минимального элемента }
    for j:=2 to N do
        if c[j] and (a[j]<a[m]) then m:=j;
    b[i]:=a[m];       { запись в выходное множество }
    c[m]:=false;      { во входное множество - "пусто"}
end; end;
```

ОБМЕННАЯ СОРТИРОВКА ПРОСТОЙ ВЫБОРКОЙ. Алгоритм сортировки простой выборкой, однако, редко применяется в том варианте, в каком он описан выше. Гораздо чаще применяется его, так называемый, обменный вариант. При обменной сортировке выборкой входное и выходное множество располагаются в одной и той же области памяти; выходное - в начале области, входное – в оставшейся ее части. В исходном состоянии входное множество занимает всю область, а выходное множество – пустое. По мере выполнения сортировки входное множество сужается, а выходное – расширяется.

Обменная сортировка простой выборкой показана в программном примере 3.8. Процедура имеет только один параметр – сортируемый массив.

```
{===== Программный пример 3.8 =====}
Procedure Sort(var a : SEQ);
Var  x, i, j, m : integer;
begin
  for i:=1 to N-1 do        { перебор элементов выходного множества}
   { входное множество - [i:N]; выходное - [1:i-1] }
    begin  m:=i;
     for j:=i+1 to N do      { поиск минимума во входном множестве  }
       if (a[j]<a[m]) then m:=j;
         { обмен 1-го элемента вх. множества с минимальным }
     if i<>m then begin
       x:=a[i]; a[i]:=a[m]; a[m]:=x;
 end;end; end;
```

Результаты трассировки программного примера 3.8 представлены в таблице 3.5. Двоеточием показана граница между входным и выходным множествами.

Очевидно, что обменный вариант обеспечивает экономию памяти. Очевидно также, что здесь не возникает проблемы "пустого" значения. Общее число сравнений уменьшается вдвое – N*(N-1)/2, но порядок алгоритма остается степенным – $O(N^2)$. Количество перестановок N-1, но перестановка, по-видимому, вдвое более времяемкая операция, чем пересылка в предыдущем алгоритме.

Таблица 3.5

Шаг	Содержимое массива a
исходный	: 242 447 286 708 24 11 192 860 937 561
1	11:447 286 708 24 242 192 860 937 561
2	11 24:286 708 447 242 192 860 937 561
3	11 24 192:708 447 242 286 860 937 561
4	11 24 192 242:447 708 286 860 937 561
5	11 24 192 242 286:708 447 860 937 561
6	11 24 192 242 286 447:708 860 937 561
7	11 24 192 242 286 447 561:860 937 708
8	11 24 192 242 286 447 561 708:937 860
9	11 24 192 242 286 447 561 708 860:937
результат	11 24 192 242 286 447 561 708 860 937

Довольно простая модификация обменной сортировки выборкой предусматривает поиск в одном цикле просмотра входного множества сразу и минимума, и максимума и обмен их с первым и с последним элементами множества соответственно. Хотя итоговое количество сравнений и пересылок в этой модификации не уменьшается, достигается экономия на количестве итераций внешнего цикла.

Приведенные выше алгоритмы сортировки выборкой практически нечувствительны к исходной упорядоченности. В любом случае поиск минимума требует полного просмотра входного множества. В обменном варианте исходная упорядоченность может дать некоторую экономию на перестановках для случаев, когда минимальный элемент найден на первом месте во входном множестве.

ПУЗЫРЬКОВАЯ СОРТИРОВКА. Входное множество просматривается, при этом попарно сравниваются соседние элементы множества. Если порядок их следования не соответствует заданному критерию упорядоченности, то элементы меняются местами. В результате одного такого просмотра при сортировке по возрастанию элемент с самым большим значением ключа переместится ("всплывет") на последнее место в множестве. При следующем проходе на свое место "всплывет" второй по величине ключа элемент и т.д. Для постановки на свои места N элементов следует сделать N-1 проходов. Выходное множество, таким образом,

формируется в конце сортируемой последовательности, при каждом следующем проходе его объем увеличивается на 1, а объем входного множества уменьшается на 1.

Порядок пузырьковой сортировки - $O(N^2)$. Среднее число сравнений – N*(N-1)/2 и таково же среднее число перестановок, что значительно хуже, чем для обменной сортировки простым выбором.

Однако, то обстоятельство, что здесь всегда сравниваются и перемещаются только соседние элементы, делает пузырьковую сортировку удобной для обработки связных списков. Перестановка в связных списках также получается более экономной.

Еще одно достоинство пузырьковой сортировки заключается в том, что при незначительных модификациях ее можно сделать чувствительной к исходной упорядоченности входного множества. Рассмотрим некоторые их таких модификаций.

Во-первых, можно ввести некоторую логическую переменную, которая будет сбрасываться в false перед началом каждого прохода и устанавливаться в true при любой перестановке. Если по окончании прохода значение этой переменной останется false, это означает, что менять местами больше нечего, сортировка закончена. При такой модификации поступление на вход алгоритма уже упорядоченного множества потребует только одного просмотра.

Во-вторых, может быть учтено то обстоятельство, что за один просмотр входного множества на свое место могут "всплыть" не один, а два и более элементов. Это легко учесть, запоминая в каждом просмотре позицию последней перестановки и установки этой позиции в качестве границы между множествами для следующего просмотра. Именно эта модификация реализована в программной иллюстрации пузырьковой сортировке в примере 3.9. Переменная nn в каждом проходе устанавливает верхнюю границу входного множества.

В переменной x запоминается позиция перестановок и в конце просмотра последнее запомненное значение вносится в nn. Сортировка закончена, когда верхняя граница входного множества станет равной 1.

```
{===== Программный пример 3.9 =====}
Procedure Sort( var a : seq);
Var nn, i, x : integer;
begin   nn:=N;              { граница входного множества   }
  repeat x:=1;              { признак перестановок         }
    for i:=2 to nn do       { перебор входного множества   }
    if a[i-1]>a[i] then begin       { сравнение соседних эл-в }
      x:=a[i-1]; a[i-1]:=a[i]; a[i]:=x      { перестановка   }
      x:=i-1;               { запоминание позиции          }
    end;  nn:=x;            { сдвиг границы                }
  until (nn=1);             {цикл пока вых. множество не захватит весь мас.}
end;
```

Результаты трассировки программного примера 3.9 представлены в таблице 3.6.

Таблица 3.6

Шаг	nn	Содержимое a
исходный	10	717 473 313 160 949 764 34 467 757 800:
1	9	473 313 160 717 764 34 467 757 800:949
2	7	313 160 473 717 34 467 757:764 800 949
3	5	160 313 773 34 467:717 757 764 800 949
4	4	160 313 34 467:473 717 757 764 800 949
5	2	160 34:313 467 473 717 757 764 800 949
6	1	34:160 313 467 473 717 757 764 800 949
Результат		: 34 160 313 467 473 717 757 764 800 949

Еще одна модификация пузырьковой сортировки носит название шейкер-сортировки. Суть ее состоит в том, что направления просмотров чередуются: за просмотром от начала к концу следует просмотр от конца к началу входного множества. При просмотре в прямом направлении запись с самым большим ключом ставится на свое место в последовательности, при просмотре в обратном направлении - запись с самым маленьким. Этот алгоритм весьма эффективен для задач восстановления упорядоченности, когда исходная последовательность уже была упорядочена, но подверглась не очень значительным изменениям. Упорядоченность в последовательности с одиночным изменением будет гарантированно восстановлена всего за два прохода.

СОРТИРОВКА ШЕЛЛА. Это еще одна модификация пузырьковой сортировки. Здесь выполняется сравнение ключей, отстоящих один от другого на некотором расстоянии d. Исходный размер d обычно выбирается соизмеримым с половиной общего размера сортируемой последовательности. Выполняется пузырьковая сортировка с интервалом сравнения d. Затем величина d уменьшается вдвое и вновь выполняется пузырьковая сортировка, далее d уменьшается еще вдвое и т.д.

Последняя пузырьковая сортировка выполняется при d=1. Качественный порядок сортировки Шелла остается $O(N^2)$, среднее же число сравнений, определенное эмпирическим путем - $\log_2(N^2*N)$. Ускорение достигается за счет того, что выявленные "не на месте" элементы при d>1, быстрее "всплывают" на свои места.

Пример 3.10 иллюстрирует сортировку Шелла.

```
{===== Программный пример 3.10 =====}
Procedure Sort( var a : seq);
Var d, i, t : integer;
      k : boolean;          { признак перестановки  }
begin   d:=N div 2;         { начальное значение интервала  }
  while d>0 do begin        { цикл с уменьшением интервала до 1 }
```

```
  k:=true;                { пузырьковая сортировка с интервалом d }
  while k do              { цикл, пока есть перестановки }
   begin  k:=false; i:=1;
      for i:=1 to N-d do  {сравнение эл-тов на интервале d
      begin  if a[i]>a[i+d] then
             begin  t:=a[i]; a[i]:=a[i+d]; a[i+d]:=t;  {перестановка }
             k:=true;       { признак перестановки  }
      end; { if ... }  end; { for ... }   end; { while k }
      d:=d div 2;           { уменьшение интервала }
  end;  { while d>0 }  end;
```

Результаты трассировки программного примера 3.10 представлены в таблице 3.7.

Таблица 3.7

Шаг	d	Содержимое массива a
исходный		76 22 4 17 13 49 4 18 32 40 96 57 77 20 1 52
1	8	32 22 4 17 13 20 1 18 76 40 96 57 77 49 4 52
2	8	32 22 4 17 13 20 1 18 76 40 96 57 77 49 4 52
3	4	13 20 1 17 32 22 4 18 76 40 4 52 77 49 96 57
4	4	13 20 1 17 32 22 4 18 76 40 4 52 77 49 96 57
5	2	1 17 13 20 4 18 32 22 4 40 76 49 77 52 96 57
6	2	1 17 4 18 13 20 4 22 32 40 76 49 77 52 96 57
7	2	1 17 4 18 4 20 13 22 32 40 76 49 77 52 96 57
8	2	1 17 4 18 4 20 13 22 32 40 76 49 77 52 96 57
9	1	1 4 17 4 18 13 20 22 32 40 49 76 52 77 57 96
10	1	1 4 4 17 13 18 20 22 32 40 49 52 76 57 77 96
11	1	1 4 4 13 17 18 20 22 32 40 49 52 57 76 77 96
12	1	1 4 4 13 17 18 20 22 32 40 49 52 57 76 77 96
результат		1 4 4 13 17 18 20 22 32 40 49 52 57 76 77 96

3.8.2. Сортировки включением

СОРТИРОВКА ПРОСТЫМИ ВСТАВКАМИ. Этот метод – "дословная" реализации стратегии включения. Порядок алгоритма сортировки простыми вставками - O(N^2), если учитывать только операции сравнения.

Но сортировка требует еще и в среднем $N^{2./4}$ перемещений, что делает ее в таком варианте значительно менее эффективной, чем сортировка выборкой.

Алгоритм сортировки простыми вставками иллюстрируется программным примером 3.11.

```
 {===== Программный пример 3.11 =====}
 Procedure Sort(a : Seq; var b : Seq);
   Var i, j, k : integer;
   begin
```

```
for i:=1 to N do          { перебор входного массива  }
                          { поиск места для a[i] в выходном массиве }
begin  j:=1; while (j<i) and (b[j]<=a[i]) do j:=j+1;
                          { освобождение места для нового эл-та}
  for k:=i downto j+1 do b[k]:=b[k-1];
  b[j]:=a[i];             { запись в выходной массив  }
end;   end;
```

Эффективность алгоритма может быть несколько улучшена при применении не линейного, а дихотомического поиска. Однако, следует иметь в виду, что такое увеличение эффективности может быть достигнуто только на множествах значительного по количеству элементов объема. Но поскольку алгоритм требует большого числа пересылок, то при значительном объеме одной записи эффективность может определяться не количеством операций сравнения, а количеством пересылок. Алгоритм обменной сортировки простыми вставками отличается от базового алгоритма только тем, что входное и выходное множество будут разделять одну область памяти.

ПУЗЫРЬКОВАЯ СОРТИРОВКА ВСТАВКАМИ. Это модификация обменного варианта сортировки. При такой сортировке входное и выходное множества находятся в одной последовательности, причем выходное - в начальной ее части. В исходном состоянии можно считать, что первый элемент последовательности уже принадлежит упорядоченному выходному множеству, остальная часть последовательности - неупорядоченное входное. Первый элемент входного множества примыкает к концу выходного множества. На каждом шаге сортировки происходит перераспределение последовательности: выходное множество увеличивается на один элемент, а входное - уменьшается. Это происходит за счет того, что первый элемент входного множества теперь считается последним элементом выходного. Затем выполняется просмотр выходного множества от конца к началу с перестановкой соседних элементов, которые не соответствуют критерию упорядоченности.

Просмотр прекращается, когда прекращаются перестановки. Это приводит к тому, что последний элемент выходного множества "выплывает" на свое место в множестве. Поскольку при этом перестановка приводит к сдвигу нового в выходном множестве элемента на одну позицию влево, нет смысла всякий раз производить полный обмен между соседними элементами - достаточно сдвигать старый элемент вправо, а новый элемент записать в выходное множество, когда его место будет установлено. Именно так и построен программный пример пузырьковой сортировки вставками - 3.12.

```
{===== Программный пример 3.12 =====}
Procedure Sort (var a : Seq);
Var i, j, k, t : integer;
begin   for i:=2 to N do  { перебор входного массива }
```

```
                {*** вх.множество - [i..N], вых.множество - [1..i]    }
  begin  t:=a[i];     { запоминается значение нового эл-та }
       j:=i-1;                {поиск места для эл. в вых. множестве со сдвигом  }
                      {конец цикла при достижении начала или, если найден
              эл.меньший}
нового} while (j>=1) and (a[j]>t) do
   begin a[j+1]:=a[j];       { все эл-ты, большие нового сдвигаются}
       j:=j-1;                { цикл от конца к началу выходного множества  }
   end; a[j+1]:=t;   { новый эл-т ставится на свое место }
  end;  end;
```

Результаты трассировки программного примера 3.12 представлены в таблице 3.8.

Таблица 3.8

Шаг	Содержимое массива a
исходный	48:43 90 39 9 56 40 41 75 72
1	43 48:90 39 9 56 40 41 75 72
2	43 48 90:39 9 56 40 41 75 72
3	39 43 48 90: 9 56 40 41 75 72
4	9 39 43 48 90:56 40 41 75 72
5	9 39 43 48 56 90:40 41 75 72
6	9 39 40 43 48 56 90:41 75 72
7	9 39 40 41 43 48 56 90:75 72
8	9 39 40 41 43 48 56 75 90:72
результат	9 39 40 41 43 48 56 72 75 90:

Хотя обменные алгоритмы стратегии включения и позволяют сократить число сравнений при наличии некоторой исходной упорядоченности входного множества, значительное число пересылок существенно снижает эффективность этих алгоритмов. Поэтому алгоритмы включения целесообразно применять к связным структурам данных, когда операция перестановки элементов структуры требует не пересылки данных в памяти, а выполняется способом коррекции указателей (см. главу 5).

Еще одна группа включающих алгоритмов сортировки использует структуру дерева. Рассмотрение последующих алгоритмов будет полезно после ознакомления с главой 6.

СОРТИРОВКА УПОРЯДОЧЕННЫМ ДВОИЧНЫМ ДЕРЕВОМ. Алгоритм складывается из построения упорядоченного двоичного дерева и последующего его обхода. Если нет необходимости в построении всего линейного упорядоченного списка значений, то нет необходимости и в обходе дерева, в этом случае применяется поиск в упорядоченном двоичном дереве. Алгоритмы работы с упорядоченными двоичными деревьями подробно рассмотрены в главе 6. Отметим, что порядок алгоритма - $O(N*\log_2 N)$), но в конкретных случаях все зависит от упорядоченности

исходной последовательности, который влияет на степень сбалансированности дерева и в конечном счете - на эффективность поиска.

ТУРНИРНАЯ СОРТИРОВКА. Этот метод сортировки получил свое название из-за сходства с кубковой системой проведения спортивных соревнований: участники соревнований разбиваются на пары, в которых разыгрывается первый тур; из победителей первого тура составляются пары для розыгрыша второго тура и т.д. Алгоритм сортировки состоит из двух этапов. На первом этапе строится дерево: аналогичное схеме розыгрыша кубка.

Например, для последовательности чисел: 16 21 8 14 26 94 30 1 такое дерево будет иметь вид пирамиды, показанной на рис. 3.13.

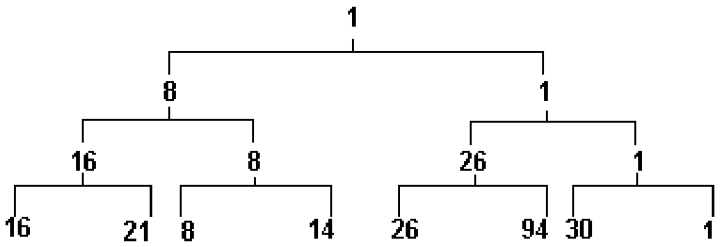


Рис.3.13. Пирамида турнирной сортировки

В примере 3.13 приведена программная иллюстрация алгоритма турнирной сортировки. Она нуждается в некоторых пояснениях. Построение пирамиды выполняется функцией Create_Heap. Пирамида строится от основания к вершине. Элементы, составляющие каждый уровень, связываются в линейный список, поэтому каждый узел дерева помимо обычных указателей на потомков - left и right - содержит и указатель на "брата" - next. При работе с каждым уровнем указатель содержит начальный адрес списка элементов в данном уровне. В первой фазе строится линейный список для нижнего уровня пирамиды, в элементы которого заносятся ключи из исходной последовательности. Следующий цикл while в каждой своей итерации надстраивает следующий уровень пирамиды. Условием завершения этого цикла является получение списка, состоящего из единственного элемента, то есть, вершины пирамиды. Построение очередного уровня состоит в попарном переборе элементов списка, составляющего предыдущий (нижний) уровень. В новый уровень переносится наименьшее значение ключа из каждой пары.

Следующий этап состоит в выборке значений из пирамиды и формирования из них упорядоченной последовательности (процедура Heap_Sort и функция Competit). В каждой итерации цикла процедуры Heap_Sort выбирается значение из вершины пирамиды - это наименьшее из имеющихся в пирамиде значений ключа. Узел-вершина при этом освобождается, освобождаются также и все узлы, занимаемые выбранным значением на более низких уровнях пирамиды. За освободившиеся узлы устраивается (снизу вверх) состязание между их потомками.

Так, для пирамиды, исходное состояние которой было показано на рис 3.13, при выборке первых трех ключей (1, 8, 14) пирамида будет последовательно принимать вид, показанный на рис.3.14 (символом x помечены пустые места в пирамиде).

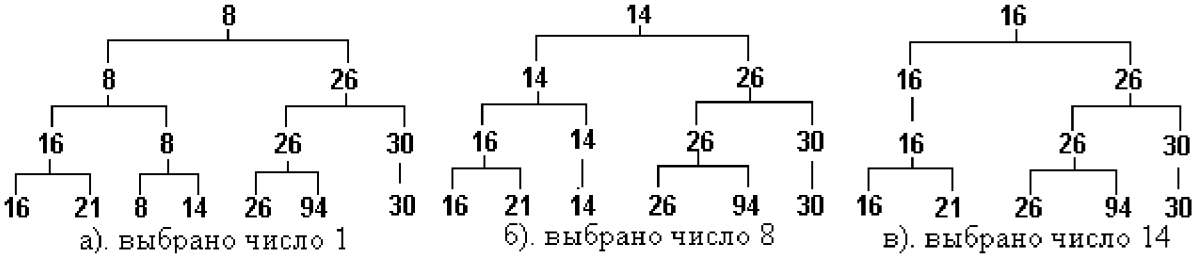


Рис.3.14. Пирамида после последовательных выборок

I а). выбрано число зходной парам б). выбрано число вершину пирамиды и формирует выходной параме массив чисел. Вся процедура Heap_Sort состоит из цикла, в каждой итерации которого значение из вершины переносится в массив a, а затем вызывается функция Competit, которая обеспечивает реорганизацию пирамиды в связи с удалением значения из вершины.

Функция Competet рекурсивная, ее параметром является указатель на вершину того поддерева, которое подлежит реорганизации. В первой фазе функции устанавливается, есть ли у узла, составляющего вершину заданного поддерева, потомок, значение данных в котором совпадает со значением данных в вершине. Если такой потомок есть, то функция Competit вызывает сама се(в). выбрано число 1 ерева, вершиной которого является обнаруженный потомок. После реорганизации адрес потомка в узле заменяется тем адресом, который вернул рекурсивный вызов Competit. Если после реорганизации оказывается, что у узла нет потомков (или он не имел потомков с самого начала), то узел уничтожается и функция возвращает пустой указатель. Если же у узла еще остаются потомки, то в поле данных узла заносится значение данных из того потомка, в котором это значение наименьшее, и функция возвращает прежний адрес узла.

```
{===== Программный пример 3.13 =====}
{ Турнирная сортировка }
type nptr = ^node;              { указатель на узел        }
   node = record          { узел дерева                }
     key : integer;       { данные                     }
     left, right : nptr;  { указатели на потомков }
     next : nptr;         { указатель на "брата"     }
     end;
{ Создание дерева - функция возвращает указатель на вершину
                созданного дерева }
Function Heap_Create(a : Seq) : nptr;
```

```
 var i : integer;
    ph2 : nptr;              { адрес начала списка уровня    }
    p1 : nptr;               { адрес нового элемента         }
    p2 : nptr;               { адрес предыдущего элемента }
    pp1, pp2 : nptr;         { адреса соревнующейся пары   }
 begin
 ph2:=nil;  {Фаза 1- построение самого нижнего уровня пирамиды}
 for i:=1 to n do
 begin  New(p1);             { выделение памяти для нового эл-та }
       p1^.key:=a[i];        { запись данных из массива }
       p1^.left:=nil; p1^.right:=nil;      { потомков нет  }
              { связывание в линейный список по уровню }
  if ph2=nil then ph2:=p1 else p2^.next:=p1; p2:=p1;
 end; { for }  p1^.next:=nil;
 { Фаза 2 - построение других уровней }
 while ph2^.next<>nil do         { цикл до вершины пирамиды }
 begin  pp1:=ph2; ph2:=nil;      { начало нового уровня  }
  while pp1<>nil do              { цикл по очередному уровню  }
  begin  pp2:=pp1^.next;  New(p1);
        { адреса потомков из предыдущего уровня }
    p1^.left:=pp1; p1^.right:=pp2; p1^.next:=nil;
       { связывание в линейный список по уровню }
    if ph2=nil then ph2:=p1 else p2^.next:=p1; p2:=p1;
       { состязание данных за выход на уровень }
    if (pp2=nil)or(pp2^.key>pp1^.key) then p1^.key:=pp1^.key
    else p1^.key:=pp2^.key;  { переход к следующей паре }
    if pp2<>nil then pp1:=pp2^.next else pp1:=nil;
  end; { while pp1<>nil }
 end; { while ph2^.next<>nil }
 Heap_Create:=ph2;   end;
{ Реорганизация поддерева - функция возвращает указатель на вершину
 реорганизован-ного дерева }
Function Competit(ph : nptr) : nptr;
begin
 { определение наличия потомков, выбор потомка для реорганизации,
   реорганизация его }
 if (ph^.left<>nil)and(ph^.left^.key=ph^.key) then
  ph^.left:=Competit(ph^.left)
 else if (ph^.right<>nil) then
  ph^.right:=Competit(ph^.right);
 if (ph^.left=nil)and(ph^.right=nil) then
       { освобождение пустого узла }
 begin  Dispose(ph); ph:=nil;  end;
 else { состязание данных потомков }
  if (ph^.left=nil) or
```

```
   ((ph^.right<>nil)and(ph^.left^.key>ph^.right^.key)) then
     ph^.key:=ph^.right^.key
    else ph^.key:=ph^.left^.key;
  Competit:=ph;  end;
 Procedure Heap_Sort(var a : Seq);     { Сортировка  }
  var ph : nptr;                       { адрес вершины дерева }
     i : integer;
  begin
  ph:=Heap_Create(a);                  { создание дерева   }
  for i:=1 to N do                     { выборка из дерева }
  begin  a[i]:=ph^.key;    { перенос данных из вершины в массив }
   ph:=Competit(ph);       { реорганизация дерева   }
  end;  end;
```

Построение дерева требует N-1 сравнений, выборка - $N*\log_2(N)$ сравнений. Порядок алгоритма, таким образом, $O(N*\log_2(N))$. Сложность операций над связными структурами данных, однако, значительно выше, чем над статическими структурами. Кроме того, алгоритм неэкономичен в отношении памяти: дублирование данных на разных уровнях пирамиды приводит к тому, что рабочая область памяти содержит примерно 2*N узлов.

СОРТИРОВКА ЧАСТИЧНО УПОРЯДОЧЕННЫМ ДЕРЕВОМ. В двоичном дереве, которое строится в этом методе сортировки для каждого узла справедливо следующее утверждение: значения ключа, записанное в узле, меньше, чем ключи его потомков. Для полностью упорядоченного дерева имеются требования к соотношению между ключами потомков. Для данного дерева таких требований нет, поэтому такое дерево и называется частично упорядоченным. Кроме того, дерево должно быть абсолютно сбалансированным. Это означает не только то, что длины путей к любым двум листьям различаются не более, чем на 1, но и то, что при добавлении нового элемента в дерево предпочтение всегда отдается левой ветви/подветви, пока это не нарушает сбалансированность. Более подробно деревья рассматриваются в гл.6.

Например, последовательность чисел: 3 20 12 58 35 30 32 28 будет представлена в виде дерева, показанного на рис. 3.15.

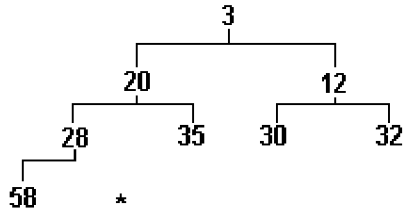


Рис.3.15. Частично упорядоченное дерево

Представление дерева в виде пирамиды наглядно показывает, что для такого дерева можно ввести понятия "начала" и "конца". Началом,

естественно, будет считаться вершина пирамиды, а концом - крайний левый элемент в самом нижнем ряду (на рис.3.15 это 58).

Для сортировки этим методом должны быть определены две операции: вставка в дерево нового элемента и выборка из дерева минимального элемента; причем выполнение любой из этих не должно нарушать ни сформулированной выше частичной упорядоченности дерева, ни его сбалансированности.

Алгоритм вставки состоит в следующем. Новый элемент вставляется на первое свободное место за концом дерева (на рис.3.15 это место обозначено символом "*"). Если ключ вставленного элемента меньше, чем ключ его предка, то предок и вставленный элемент меняются местами. Ключ вставленного элемента теперь сравнивается с ключом его предка на новом месте и т.д. Сравнения заканчиваются, когда ключ нового элемента окажется больше ключа предка или когда новый элемент "выплывет" в вершину пирамиды. Пирамида, показанная на рис.3.15, построена именно последовательным включением в нее чисел из приведенного ряда. Если мы включим в нее, например, еще число 16, то пирамида примет вид, представленный на рис.3.16. (Символом "*" помечены элементы, перемещенные при этой операции.)

Процедура выборки элемента несколько сложнее. Очевидно, что минимальный элемент находится в вершине. После выборки за освободившееся место устраивается состязание между потомками, и в вершину перемещается потомок с наименьшим значением ключа. За освободившееся место перемешенного потомка состязаются его потомки и т.д., пока свободное место не опустится до листа пирамиды. Состояние дерева после выборки из него минимального числа (3) показано на рис.3.17. а).

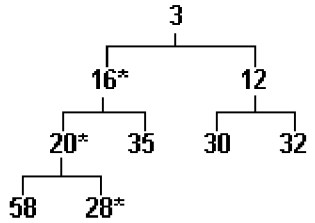


Рис.3.16. Частично упорядоченное дерево, включение элемента

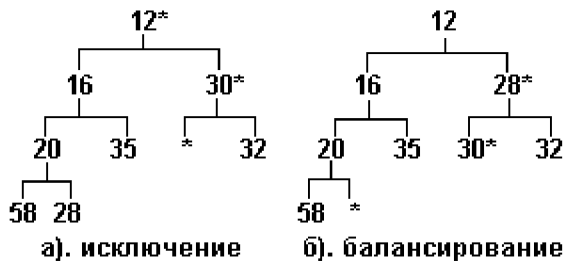


Рис.3.17. Частично упорядоченное дерево, исключение элемента

Упорядоченность дерева восстановлена, но нарушено условие его сбалансированности, так как свободное место находится не в конце дерева.

Для восстановления сбалансированности последний элемент дерева переносится на освободившееся место, а затем "всплывает" по тому же алгоритму, который применялся при вставке. Результат такой балансировки показан на рис.3.17.б.

Прежде, чем описывать программный пример, иллюстрирующий сортировку частично упорядоченным деревом - пример 3.14, рассмотрим способ представления дерева в памяти. Это способ представления двоичных деревьев в статической памяти (в одномерном массиве), который может быть применен и в других задачах. Элементы дерева располагаются в соседних слотах памяти по уровням. Самый первый слот выделенной памяти занимает вершина. Следующие 2 слота - элементы второго уровня, следующие 4 слота - третьего и т.д.

Дерево с рис.3.17.б, например, будет линеаризовано таким образом:

12 16 28 20 35 30 32 58

В таком представлении отпадает необходимость хранить в составе узла дерева указатели, так как адреса потомков могут быть вычислены. Для узла, представленного элементом массива с индексом i индексы его левого и правого потомков будут 2*i и 2*i+1 соответственно. Для узла с индексом i индекс его предка будет i div 2.

После всего вышесказанного алгоритм программного примера 3.14 не нуждается в особых пояснениях. Поясним только структуру примера. Пример оформлен в виде законченного программного модуля, который будет использован и в следующем примере. Само дерево представлено в массиве tree, переменная nt является индексом первого свободного элемента в массиве. Входные точки модуля:

- процедура InitST - инициализация модуля, установка начального значения nt;
- функция InsertST - вставка в дерево нового элемента; функция возвращает false, если в дереве нет свободного места, иначе - true;
- функция DeleteST - выборка из дерева минимального элемента;
- функция возвращает false, если дерево пустое, иначе - true;
- функция CheckST - проверка состояния дерева: ключ минимального элемента возвращается в выходном параметре, но элемент не исключается из дерева; а возвращаемое значение функции - 0 – если дерево пустое, 1 - если дерево заполнено не до конца, 2 – если дерево заполнено до конца.

Кроме того в модуле определены внутренние программные единицы:

- функция Down - обеспечивает спуск свободного места из вершины пирамиды в ее основание, функция возвращает индекс свободного места после спуска;
- процедура Up - обеспечивающая всплытие элемента с заданного места.

```
{===== Программный пример 3.14 =====}
Unit SortTree;     { Сортировка частично упорядоченным деревом }
Interface
```

```
Procedure InitSt;
Function CheckST(var a : integer) : integer;
Function DeleteST(var a : integer) : boolean;
Function InsertST(a : integer) : boolean;
Implementation
Const NN=16;
var tr : array[1..NN] of integer;                { дерево }
   nt : integer;      { индек последнего эл-та в дереве  }
Procedure Up(l : integer);       {Всплытие эл-та с места с индексом l }
  var h : integer;                {l - индекс узла, h - индекс его предка      }
      x : integer;
begin    h:=l div 2;                    { индекс предка  }
  while h>0 do                   { до начала  дерева  }
    if tr[l]<tr[h] then          { ключ узла меньше, чем у предка }
    begin  x:=tr[l]; tr[l]:=tr[h]; tr[h]:=x;  { перестановка }
     l:=h; h:=l div 2;           { предок становится текущим узлом  }
    end  else h:=0;              { конец всплытия  }
end;                         { Procedure Up }
{** Спуск свободного места из начала дерева **}
Function Down : integer;
  var h, l : integer;            { h - индекс узла, l - индекс его потомка   }
begin  h:=1;                     { начальный узел - начало дерева  }
  while true do
  begin   l:=h*2;          { вычисление индекса 1-го потомка }
    if l+1<=nt then        { у узла есть 2-й потомок  }
    begin  if tr[l]<=tr[l+1] then  { 1-й потомок меньше 2-го  }
       begin tr[h]:=tr[l];    {1-й потомок переносится в тек. узел  }
        h:=l;  end           { 1-й потомок становится текущим узлом  }
      else                   { 2-й потомок меньше 1-го }
    begin tr[h]:=tr[l+1];    {2-й потомок переносится в текущ.узел     }
       h:=l+1;  end;                 {2-й потомок становится текущим узлом   }
  end else
     if l=nt then            { 1-й потомок есть, 2-го нет  }
    begin  tr[h]:=tr[l];     {1-й потомок переносится в текущ.узел     }
       Down:=l;  Exit;       { спуск закончен }
    end  else                { потомков нет - спуск закончен }
    begin    Down:=h; Exit;  end;
 end;          { while }
end;           { Function Down }
Procedure InitSt;  {** Инициализация сортировки деревом **}
begin  nt:=0;          { дерево пустое }
end;                   { Procedure InitSt  }
{** Проверка состояния дерева **}
Function CheckST(var a : integer) : integer;
begin  a:=tr[1];     { выборка эл-та из начала  }
```

```
  case nt of  { формирование возвращаемого значения функции }
    0:   { дерево пустое } CheckSt:=0;
    NN:  { дерево полное } CheckSt:=2;
    else { дерево частично заполнено } CheckSt:=1;
  end;
end;            {  Function CheckST  }
{** Вставка эл-та a в дерево **}
 Function InsertST(a : integer) : boolean;
 begin
  if nt=NN then              { дерево заполнено - отказ  }
     InsertST:=false else          { в дереве есть место }
  begin  nt:=nt+1; tr[nt]:=a;      { запись в конец дерева  }
    Up(nt);  InsertSt:=true;       { всплытие  }
  end;   end;         { Function InsertST  }
{** Выборка эл-та из дерева **}
 Function DeleteST(var a : integer) : boolean;
 var n : integer;
 begin
  if nt=0 then               { дерево пустое - отказ  }
    DeleteST:=false else  { дерево не пустое         }
    begin   a:=tr[1];          { выборка эл-та из начала  }
        n:=Down;               { спуск свободного места в позицию n  }
        if n<nt then begin
{ если свободное место спустилось не в конец дерева }
tr[n]:=tr[nt];          { эл-т из конца переносится на своб.место        }
        Up(n);  end;           { всплытие }
    nt:=nt-1;   DeleteSt:=true;
  end;   end;          { Function DeleteST       }
 END.
```

Если применять сортировку частично упорядоченным деревом для упорядочения уже готовой последовательности размером N, то необходимо N раз выполнить вставку, а затем N раз - выборку. Порядок алгоритма - O(N*log2(N)), но среднее значение количества сравнений примерно в 3 раза больше, чем для турнирной сортировки. Но сортировка частично упорядоченным деревом имеет одно существенное преимущество перед всеми другими алгоритмами. Дело в том, что это самый удобный алгоритм для "сортировки on-line", когда сортируемая последовательность не зафиксирована до начала сортировки, а меняется в процессе работы и вставки чередуются с выборками. Каждое изменение (добавление/удаление элемента) сортируемой последовательности потребует здесь не более, чем 2*log2(N) сравнений и перестановок, в то время, как другие алгоритмы потребуют при единичном изменении переупорядочивания всей последовательности "по полной программе".

Типичная задача, которая требует такой сортировки, возникает при сортировке данных на внешней памяти (файлов). Первым этапом такой сортировки является формирование из данных файла упорядоченных последовательностей максимально возможной длины при ограниченном объёме оперативной памяти. Приведенный ниже программный пример (пример 3.15) показывает решение этой задачи.

Последовательность чисел, записанная во входном файле, поэлементно считывается, и числа по мере считывания включаются в дерево. Когда дерево оказывается заполненным, очередное считанное из файла число сравнивается с последним числом, выведенным в выходной файл. Если считанное число не меньше последнего выведенного, но меньше числа, находящегося в вершине дерева, то в выходной файл выводится считанное число. Если считанное число не меньше последнего выведенного, и не меньше числа, находящегося в вершине дерева, то в выходной файл выводится число, выбираемое из дерева, а считанное число заносится в дерево. Наконец, если считанное число меньше последнего выведенного, то поэлементно выбирается и выводится все содержимое дерева, и формирование новой последовательности начинается с записи в пустое дерево считанного числа.

```
{===== Программный пример 3.15 =====}
{ Формирование отсортированных последовательностей в файле }
Uses SortTree;
var x : integar;        { считанное число                                  }
    y : integer;        { число в вершине дерева                           }
    old : integer;      { последнее выведенное число                       }
    inp : text;         { входной файл                                     }
    out : text;         { выходной файл                                    }
    bf : boolean;       { признак начала вывода последовательности }
    bx : boolean;       { рабочая переменная                               }
begin
Assign(inp,'STX.INP'); Reset(inp);
Assign(out,'STX.OUT'); Rewrite(out);
InitST;                         { инициализация сортировки }
bf:=false;                      { вывод последовательности еще не начат }
while not Eof(inp) do
begin   ReadLn(inp,x);          { считывание числа из файла          }
  { если в дереве есть свободное место - включить в дерево }
  if CheckST(y)<=1 then bx:=InsertST(x)
  else                          { в дереве нет свободного места   }
    if (bf and (x<old)) or (not bf and (x<y))  then
      { вывод содержимого дерева }
    begin  while DeleteST(y) do Write(out,y:3,' ');
       WriteLn(out);
       bf:=false;                       { начало новой последовательности }
       bx:=InsertST(x);         { занесение считанного числа в дерево }
```

```
   end  else          {продолжение формирования последовательности }
   begin  if x<y then              { вывод считанного числа }
     begin Write(out,x:3,' '); old:=x; end;
             else                  { вывод числа из вершины дерева }
    begin   bx:=DeleteST(y);
      Write(out,y:3,' '); old:=y;
      bx:=InsertST(x);             { занесение считанного в дерево }
    end;   bf:=true;               { вывод последовательности начался }
  end;  end;
 Close(inp);                       { вывод остатка }
 while DeleteST(y) do Write(out,y:3,' ');
 WriteLn(out);  Close(out);
end.
```

3.8.3. Сортировки распределением

ПОРАЗРЯДНАЯ ЦИФРОВАЯ СОРТИРОВКА. Алгоритм требует представления ключей сортируемой последовательности в виде чисел в некоторой системе счисления P. Число проходов сортировка равно максимальному числу значащих цифр в числе - D. В каждом проходе анализируется значащая цифра в очередном разряде ключа, начиная с младшего разряда. Все ключи с одинаковым значением этой цифры объединяются в одну группу. Ключи в группе располагаются в порядке их поступления. После того, как вся исходная последовательность распределена по группам, группы располагаются в порядке возрастания связанных с группами цифр. Процесс повторяется для второй цифры и т.д., пока не будут исчерпаны значащие цифры в ключе. Основание системы счисления P может быть любым, в частном случае 2 или 10. Для системы счисления с основанием P требуется P групп.

Порядок алгоритма качественно линейный - O(N), для сортировки требуется D*N операций анализа цифры. Однако, в такой оценке порядка не учитывается ряд обстоятельств.

Во-первых, операция выделения значащей цифры будет простой и быстрой только при P=2, для других систем счисления эта операция может оказаться значительно более времяемкой, чем операция сравнения.

Во-вторых, в оценке алгоритма не учитываются расходы времени и памяти на создание и ведение групп. Размещение групп в статической рабочей памяти потребует памяти для P*N элементов, так как в предельном случае все элементы могут попасть в какую-то одну группу. Если же формировать группы внутри той же последовательности по принципу обменных алгоритмов, то возникает необходимость перераспределения последовательности между группами и все проблемы и недостатки, присущие алгоритмам включения. Наиболее рациональным является формирование групп в виде связных списков с динамическим выделением памяти.

В программном примере 3.16 применена поразрядная сортировка к статической структуре данных и формируются группы на том же месте, где расположена исходная последовательность. Пример требует некоторых пояснений.

Область памяти, занимаемая массивом, перераспределяется между входным и выходным множествами, как это делалось и в ряде предыдущих примеров. Выходное множество (оно размещается в начале массива) разбивается на группы. Разбиение отслеживается в массиве b.

Элемент массива b[i] содержит индекс в массиве a, с которого начинается i+1-ая группа. Номер группы определяется значением анализируемой цифры числа, поэтому индексация в массиве b начинается с 0. Когда очередное число выбирается из входного множества и должно быть занесено в i-ую группу выходного множества, оно будет записано в позицию, определяемую значением b[i]. Но предварительно эта позиция должна быть освобождена: участок массива от b[i] до конца выходного множества включительно сдвигается вправо. После записи числа в i-ую группу i-ое и все последующие значения в массиве b корректируются - увеличиваются на 1.

```
{===== Программный пример 3.16 =====}
{ Цифровая сортировка (распределение) }
const D=...;                { максимальное количество цифр в числе }
    P=...;                  { основание системы счисления }
Function Digit(v, n : integer) : integer;
begin                  { возвращает значение n-ой цифры в числе v }
 for n:=n downto 2 do v:=v div P;
 Digit:=v mod P;
end;
Procedure Sort(var a : Seq);
 Var b : array[0..P-2] of integer; { индекс элемента, следующего }
                                   { за последним в i-ой группе }
    i, j, k, m, x : integer;
 begin
  for m:=1 to D do       { перебор цифр, начиная с младшей }
  begin for i:=0 to P-2 do b[i]:=1;    { нач. значения индексов }
  for i:=1 to N do                { перебор массива  }
  begin
    k:=Digit(a[i],m);      { определение m-ой цифры }
    x:=a[i];                {сдвиг - освобождение}
                            {места в конце k-ой группы   }
    for j:=i downto b[k]+1 do a[j]:=a[j-1];
    a[b[k]]:=x;             { запись в конец k-ой группы}
{ модификация k-го индекса и всех больших }
    for j:=k to P-2 do b[j]:=b[j]+1;
  end;
end;  end;
```

Результаты трассировки программного примера 3.16 при P=10 и D=4 представлены в таблице 3.9.

БЫСТРАЯ СОРТИРОВКА ХОАРА. Данный алгоритм относится к распределительным и обеспечивает показатели эффективности O(N*log2(N)) даже при наихудшем исходном распределении.

Используются два индекса - i и j - с начальными значениями 1 и N соответственно. Ключ K[i] сравнивается с ключом K[j]. Если ключи удовлетворяют критерию упорядоченности, то индекс j уменьшается на 1 и производится следующее сравнение. Если ключи не удовлетворяют критерию, то записи R[i] и R[j] меняются местами.

Таблица 3.9

Цифра	содержимое массивов a и b
исх.	220 8390 9524 9510 462 2124 7970 4572 4418 12383
1	220 8390 9510 7970 462 4572 1283 9524 2124 4418 b=(5,5,7,8,10,10,10,10,11,11)
2	9510 4418 220 9524 2124 462 7970 4572 1283 8390 b=(1,3,6,6,6,6,7,9,10,11)
3	2124 220 1283 8390 4418 462 9510 9524 4572 7970 b=(1,2,4,5,7,10,10,10,10,11)
4	220 462 1283 2124 4418 4572 7970 8390 9510 9524 b=(3,4,5,5,7,7,7,8,9,11)

При этом индекс j фиксируется и начинает меняться индекс i (увеличиваться на 1 после каждого сравнения). После следующей перестановки фиксируется i и начинает изменяться j и т.д. Проход заканчивается, когда индексы i и j становятся равными. Запись, находящаяся на позиции встречи индексов, стоит на своем месте в последовательности. Эта запись делит последовательность на два подмножества. Все записи, расположенные слева от нее имеют ключи, меньшие чем ключ этой записи, все записи справа - большие. Тот же самый алгоритм применяется к левому подмножеству, а затем к правому. Записи подмножества распределяются по двум еще меньшим подмножествам и т.д., и т.д. Распределение заканчивается, когда полученное подмножество будет состоять из единственного элемента - такое подмножество уже является упорядоченным.

Процедура сортировки в примере 3.17 рекурсивная. При ее вызове должны быть заданы значения границ сортируемого участка от 1 до N

```
{===== Программный пример 3.17 =====}
{Быстрая сортировка Хоара; i0, j0 - границы сортируемого участка}
Procedure Sort(var a : Seq; i0,j0 : integer);
Var i, j : integer;                { текущие индексы в массиве                }
   flag : boolean;                 { признак меняющегося индекса: если
```

```
          flag=true - уменьшается j, иначе - увеличивается i          }
    x : integer;
begin   if j0<=i0 Exit;    { подмножество пустое или из 1 эл-та     }
  i:=i0; j:=j0; flag:=true;          { вначале будет изменяться j          }
  while i<j do
  begin   if a[i]>a[j] then
     begin x:=a[i]; a[i]:=a[j]; a[j]:=x;   { перестановка                    }
      flag:= not flag;                { после перестановки меняется  индекс   }
     end;                             { реально изменяется только один индекс }
     if flag then j:=j-1 else i:=i+1;
  end;
  Sort(a,i0,i-1);                     { сортировка левого подмассива          }
  Sort(a,i+1,j0);                     { сортировка правого подмассива   }
end;
```

Таблица 3.10

проход	содержимое массива a
1	42i 79 39 65 60 29 86 95 25 37j 37 79i 39 65 60 29 86 95 25 42j 37 42i 39 65 60 29 86 95 25j 79 37 25 39i 65 60 29 86 95 42j 79 37 25 39 65i 60 29 86 95 42j 79 37 25 39 42i 60 29 86 95j 65 79 37 25 39 42i 60 29 86j 95 65 79 37 25 39 42i 60 29j 86 95 65 79 37 25 39 29 60i 42j 86 95 65 79
2	29i 25 39 37j 42* 60 86 95 65 79 29 25i 39 37j 42 60 86 95 65 79 29 25 37i 39j 42 60 86 95 65 79
3	25i 29j 37* 39 42 60 86 95 65 79
4	25* 29 37* 39 42 60 86 95 65 79
5	25* 29* 37* 39 42 60 86 95 65 79
6	25* 29* 37* 39* 42* 60i 86 95 65 79j 25* 29* 37* 39* 42* 60i 86 95 65i 79 25* 29* 37* 39* 42* 60i 86 95j 65 79 25* 29* 37* 39* 42* 60i 86j 95 65 79
7	25* 29* 37* 39* 42* 60* 79i 95 65 86j 25* 29* 37* 39* 42* 60* 79i 86 65j 95 25* 29* 37* 39* 42* 60* 65 86i 79j 95
8	25* 29* 37* 39* 42* 60* 65 79* 86 95
9	25* 29* 37* 39* 42* 60* 65* 79* 86i 95j
10	25* 29* 37* 39* 42* 60* 65* 79* 86* 95

Результаты трассировки примера приведены в таблице 3.10. В каждой строке таблицы показаны текущие положения индексов i и j, звездочками

отмечены элементы, ставшие на свои места. Для каждого прохода показаны границы подмножества, в котором ведется сортировка.

СОРТИРОВКА СЛИЯНИЕМ. Алгоритмы сортировки слиянием, как правило, имеют порядок O(N*log2(N)), но отличаются от других алгоритмов большей сложностью и требуют большого числа пересылок. Алгоритмы слияния применяются в основном, как составная часть внешней сортировки. Здесь же для понимания принципа слияния приведен простейший алгоритм слияния в оперативной памяти.

СОРТИРОВКА ПОПАРНЫМ СЛИЯНИЕМ. Входное множество рассматривается, как последовательность подмножеств, каждое из которых состоит из единственного элемента и, следовательно, является уже упорядоченным. На первом проходе каждые два соседних одноэлементных множества сливаются в одно двухэлементное упорядо-ченное множество. На втором проходе двухэлементные множества сливаютс в 4-элементные упорядоченные множества и т.д. В конце концов получается одно большое упорядоченное множество.

Программный пример 3.18 иллюстрирует сортировку попарным слиянием в ее обменном варианте - выходные множества формируются на месте входных.

```
{===== Программный пример 3.18 =====}
Procedure Sort(var a :Seq);
Var i0,j0,i,j,si,sj,k,ke,t,m : integer;
begin  si:=1;            { начальный размер одного множества }
  while si<N do{цикл пока одно множество не составит весь массив}
  begin  i0:=1;          { нач. индекс 1-го множества пары }
    while i0<N do        { цикл пока не пересмотрим весь массив }
    begin   j0:=i0+si;     { нач. индекс 2-го множества пары }
      i:=i0; j:=j0;
{размер 2-го множества пары может ограничиваться концом массива }
      if si>N-j0+1 then sj:=N-j0+1 else sj:=si;
      if sj>0 then
      begin  k:=i0;        { нач. индекс слитого множества }
        while (i<i0+si+sj) and (j<j0+sj) do
          { цикл пока не исчерпаются оба входные множества }
        begin   if a[i]>a[j] then
{ если эл-т 1-го <= элемента 2-го, он остается на своем месте, но вых. множество расширяется иначе - освобождается место в вых.множестве и туда заносится эл-т из 2-го множества }
          begin t:=a[j];
            for m:=j-1 downto k do a[m+1]:=a[m];
            a[k]:=t; j:=j+1;  {к след. эл-ту во 2-м множестве}
          end;  { if a[i]>a[j] }
        k:=k+1; { вых. множество увеличилось }
        i:=i+1; { если был перенос - за счет сдвига, если не было - за счет
```

```
                перехода эл-та в вых. }
        end; { while }    end; { if sj>0 }
      i0:=i0+si*2;  { начало следующей пары }
     end;      { while i0<N }
    si:=si*2;  { размер эл-тов пары увеличивается вдвое }
   end; { while si<N }
 end;
```

Результаты трассировки примера приведены в таблице 3.11. Для каждого прохода показаны множества, которые на этом проходе сливаются. Обратите внимание на обработку последнего множества, оставшегося без пары.

Таблица 3.11

проход	содержимое массива a
1	40 5 76 86 90 25 29 96 54 15
2	5 40 76 86 25 90 29 96 15 54
3	5 40 76 86 25 29 90 96 15 54
4	5 25 29 40 76 86 90 96 15 54
результат	5 15 25 29 40 54 76 86 90 96

3.9. Прямой доступ и хеширование

В рассмотренных выше методах поиска число проб при поиске в лучшем случае было пропорционально log2(N). Естественно, возникает желание найти такой метод поиска, при котором число проб не зависело бы от размера таблицы, а в идеальном случае поиск сводился бы к одной пробе.

3.9.1. Таблицы прямого доступа

Простейшей организацией таблицы, обеспечивающей идеально быстрый поиск, является таблица прямого доступа. В такой таблице ключ является адресом записи в таблице или может быть преобразован в адрес, причем таким образом, что никакие два разных ключа не преобразуются в один и тот же адрес. При создании таблицы выделяется память для хранения всей таблицы и заполняется пустыми записями. Затем записи вносятся в таблицу - каждая на свое место, определяемое ее ключом. При поиске ключ используется как адрес и по этому адресу выбирается запись, если выбранная запись пустая, то записи с таким ключом вообще нет в таблице.

Таблицы прямого доступа очень эффективны в использовании, но, к сожалению, область их применения весьма ограничена. Назовем пространством ключей множество всех теоретически возможных значений ключей записи. Назовем пространством записей множество тех слотов в

памяти, которые мы выделяем для хранения таблицы. Таблицы прямого доступа применимы только для таких задач, в которых размер пространства записей может быть равен размеру пространства ключей. В большинстве реальных задач, однако, размер пространства записей много меньше, чем пространства ключей. Так, если в качестве ключа используется фамилия, то даже ограничив длину ключа 10 символами мы получаем 33^10 возможных значений ключей. Ни в какой вычислительной системе не может быть выделено пространство записей такого размера. Даже если ресурсы вычислительной системы и позволят это, то значительная часть этого пространства будет заполнена пустыми записями, так как в каждом конкретном заполнении таблицы факти-ческое множество ключей не будет полностью покрывать пространство ключей.

3.9.2. Таблицы со справочниками

Одним из способов устранения этого недостатка является метод справочников. Основная таблица содержит записи в произвольном порядке. В дополнение к основной строится справочная или индексная таблица, записи которой состоят всего из двух полей: ключа и адреса в основной таблице. Поиск по ключу производится в справочной таблице. Если справочная таблица является таблицей прямого доступа, то потери памяти на пустые записи уменьшаются. Однако, очевидно, что в случае ключа-фамилии справочная таблица нас не спасет. Поэтому, обычно справочные таблицы содержат только фактические ключи и к ним применяются методы сортировки и поиска, описанные выше. При сортировке справочных таблиц, конечно, достигается некоторая экономия на пересылках, но в целом применение справочников было бы нецелесообразно, если бы не два их важных свойства:

- во-первых, если основная таблица расположена на внешней памяти, то справочная таблица (или значительная часть ее) может быть размещена в оперативной памяти и поиск ключа, таким образом, будет выполняться в оперативной памяти, что гораздо быстрее;
- во-вторых, для одной основной таблицы могут быть построены несколько справочников, обеспечивающих использование в качестве ключа разных полей записи основной таблицы.

Заметим, что для таблиц прямого доступа и для таблиц со справочниками нет необходимости хранить ключ в составе записи основной таблицы, так как ключ может быть восстановлен по адресу записи либо по справочнику.

3.9.3. Хешированные таблицы и функции хеширования

Как отмечалось выше, в каждой реальной таблице фактическое множество ключей является лишь небольшим подмножеством множества всех теоретически возможных ключей. Поскольку память является одним из самых дорогостоящих ресурсов вычислительной системы, из соображений

ее экономии целесообразно назначать размер пространства записей равным размеру фактического множества записей или превосходящим его незначительно. В этом случае мы должны иметь некоторую функцию, обеспечивающую отображение точки из пространства ключей в точку в пространстве записей, т.е., преобразование ключа в адрес записи: r = H(k), где – r адрес записи, k – ключ.

Такая функция называется функцией хеширования (другие ее названия – функция перемешивания, функция рандомизации).

При попытке отображения точек из некоторого широкого пространства в узкое неизбежны ситуации, когда разные точки в исходном пространстве отобразятся в одну и ту же точку в целевом пространстве. Ситуация, при которой разные ключи отображаются в один и тот же адрес записи, называется коллизией или переполнением, а такие ключи называются синонимами. Коллизии – основная проблема для хешированных таблиц, решение которой будет рассмотрено далее.

Если функция H, преобразующая ключ в адрес, может порождать коллизии, то однозначной обратной функции: k = H`(r), позволяющей восстановить ключ по известному адресу, существовать не может.

Поэтому ключ должен обязательно входить в состав записи хешированной таблицы как одно из ее полей.

К функции хеширования в общем случае предъявляются следующие требования:

- она должна обеспечивать равномерное распределение отображений фактических ключей по пространству записей;
- она должна порождать как можно меньше коллизий для данного фактического множества записей;
- она не должна отображать какую-либо связь между значениями ключей в связь между значениями адресов;
- она должна быть простой и быстрой для вычисления.

Простейшей функцией хеширования является деление по модулю числового значения ключа на размер пространства записи. Результат интерпретируется как адрес записи. Хотя эта функция и применяется во всех приводимых ниже примерах данного раздела, следует иметь в виду, что такая функция плохо соответствует первым трем требованиям к функции хеширования и сама по себе может быть применена лишь в очень ограниченном диапазоне реальных задач. Однако, операция деления по модулю обычно применяется как последний шаг в более сложных функциях хеширования, обеспечивая приведение результата к размеру пространства записей.

Функция середины квадрата. Значение ключа преобразуется в число, это число затем возводится в квадрат, из него выбираются несколько средних цифр и интерпретируются как адрес записи.

Функция свертки. Цифровое представление ключа разбивается на части, каждая из которых имеет длину, равную длине требуемого адреса. Над частями производятся какие-то арифметические или поразрядные логические

операции, результат которых интерпретируется как адрес. Например, для сравнительно небольших таблиц с ключами – символьными строками неплохие результаты дает функция хеширования, в которой адрес записи получается в результате сложения кодов символов, составляющих строку-ключ.

Функция преобразования системы счисления. Ключ, записанный как число в некоторой системе счисления P, интерпретируется как число в системе счисления Q>P. Обычно выбирают Q=P+1. Это число переводится из системы Q обратно в систему P, приводится к размеру пространства записей и интерпретируется как адрес.

3.9.4. Проблема коллизий в хешированных таблицах

Удачно подобранная функция хеширования может минимизировать число коллизий, но не может гарантировать их полного отсутствия.

Ниже мы рассмотрим методы разрешения проблемы коллизий в хешированных таблицах.

ПОВТОРНОЕ ХЕШИРОВАНИЕ. Повторное хеширование, известное также под названием открытой таблицы, предусматривает следующее: если при попытке записи в таблицу оказывается, что требуемое место в таблице уже занято, то значение записывается в ту же таблицу на какое-то другое место. Другое место определяется при помощи вторичной функции хеширования H2, аргументом которой в общем случае может быть и исходное значение ключа и результат предыдущего хеширования: r = H2(k,r`), где r` – адрес, полученный при предыдущем применении функции хеширования. Если полученный в результате применения функции H2 адрес также оказывается занятым, функция H2 применяется повторно - до тех пор, пока не будет найдено свободное место. Простейшей функцией вторичного хеширования является функция: r = r` + 1. Эту функцию иногда называют функцией линейного опробования. Фактически при применении линейного опробования, если "законное" место записи (т.е. слот, расположенный по адресу, получаемому из первичной функции хеширования) уже занято, то запись занимает первое свободное место за "законным" (таблица при этом рассматривается как кольцо). Выборка элемента по ключу производится аналогичным образом: адрес записи вычисляется по первичной функции хеширования и ключ записи, расположенной по этому адресу, сравнивается с искомым. Если запись непуста, и ключи не совпадают, то продолжается поиск с применением вторичной функции хеширования. Поиск заканчивается, когда найдена запись с искомым ключом (успешное завершение) или перебрана вся таблица (неуспешное завершение).

Приведенный ниже программный пример иллюстрирует применение метода линейного опробования для разрешения коллизий. В составляющем этот пример модуле определены процедуры/функции инициализации таблицы, вставки элемента в таблицу и поиска элемента в таблице. Процедура инициализации является обязательной для хешированных таблиц, так как перед началом работы с таблицей для нее должна быть выделена

память и заполнена "пустыми" (свободными) записями. В качестве признака пустой записи значение ключа использована константа EMPTY, которая при отладке была определена как -1. Функция первичного хеширования – Hash – выполняет деление по модулю.

```
{==== Программный пример 3.19 ====}
{ Хешированная таблица с повторным перемешиванием }
Unit HashTbl;
Interface
Procedure Init;
Function Insert(key : integer) : boolean;
Function Fetch(key : integer) : integer;
Implementation
const N=...;                    { число записей в таблице }
type Seq = array[1..N] of integer;  { тип таблицы }
var tabl : Seq;                  { таблица }
                        { Хеширование - деление по модулю }
Function Hash(key : integer) : integer;
begin  Hash:= key mod N+1;    end;
       { Инициализация таблицы - заполнение пустыми записями }
Procedure Init;
 var i : integer;
 begin    for i:=1 to N do tabl[i]:=EMPTY;    end;
                        {  Добавление элемента в таблицу  }
Function Insert(key : integer) : boolean;
  Var addr, a1 : integer;
  begin     addr:=Hash(key);        { вычисление адреса }
  if tabl[addr]<>EMPTY then         { если адрес занят }
  begin  a1:=addr;
    repeat                        { поиск свободного места }
      addr:=addr mod N+1;
      until (addr=a1) or (tabl[addr]=EMPTY);
    if tabl[addr]<>EMPTY then       { нет свободного места }
    begin  Insert:=false; Exit;  end;
  end;   tabl[addr]:=key;           { запись в таблицу }
  Insert:=true;   end;
{ Выборка из таблицы - возвращает адрес найденного ключа
  или EMPTY - если ключ не найден }
Function Fetch(key : integer) : integer;
Var addr, a1 : integer;
begin   addr:=Hash(key);
   if tabl[addr]=EMPTY then
     Fetch:=EMPTY      { место свободно - ключа нет в таблице }
     else if tabl[addr]=key then
     Fetch:=addr               { ключ найден на своем месте }
```

```
            else begin     { место занято другим ключом }
    a1:=(addr+1) mod N;
     { Поиск, пока не найден ключ или не сделан полный оборот }
    while (tabl[a1]<>key) and (a1<>addr) do addr:=(a1+1) mod N;
   if tabl[a1]<>key then Fetch:=EMPTY  else Fetch:=a1;
end;
end.
```

Повторное хеширование обладает существенным недостатком: число коллизий зависит от порядка заполнения таблицы. Ниже приведен пример работы программы примера 3.19 для двух случаев. В обоих случаях размер таблицы задавался равным 15. В первом случае в таблицу заносилась следующая последовательность из 14 чисел-ключей: 58 0 19 96 38 52 62 77 4 15 79 75 81 66

Результирующая таблица имела такой вид:

0 15* 62 77 19 4* 96 52 38 79* 75* 81* 66* 58 E

Буквой "E" обозначено свободное место в таблице. Значком "*" помечены элементы, стоящие не на своих "законных" местах. Во втором случае те же ключи заносились в таблицу в иной последовательности, а именно:

0 75 15 62 77 19 4 79 96 81 66 52 38 58

Результирующая таблица имела вид:

0 75* 15* 62* 77* 19* 4* 79* 96* 81* 66* 52* 38* 58 E

Большее число коллизий во втором случае объясняется тем, что если ключ не может быть записан по тому адресу, который вычислен для него первичной функцией хеширования, он записывается на свободное место, а это пока свободное место принадлежит (по первичной функции хеширования другому ключу, который впоследствии тоже может поступить на вход таблицы.

ПАКЕТИРОВАНИЕ. Сущность метода пакетирования состоит в том, что записи таблицы объединяются в пакеты фиксированного, относительно небольшого размера. Функция хеширования дает на выходе не адрес записи, а адрес пакета. После нахождения пакета, в пакете выполняется линейный поиск по ключу. Пакетирование позволяет сгладить нарушения равномерности распределения ключей по пространству пакетов и, следовательно, уменьшить число коллизий, но не может гарантированно их предотвратить. Пакеты также могут переполняться. Поэтому пакетирование применяется как дополнение к более радикальным методам - к методу повторного хеширования или к методам, описанным ниже. В программном примере 3.20, применен метод пакетирования без комбинации с другими методами. При общем размере таблицы - 15 и размере пакета - 3 уже ранее опробованная последовательность:

58 0 75 19 96 38 81 52 66 62 77 4 15 79

записалась в результирующую таблицу без коллизий (значком "|" обозначены границы пакетов):

0 75 15| 96 81 66| 52 62 77| 58 38 E| 19 4 79

```
{==== Программный пример 3.20 ====}
{ Хешированная таблица с пакетами }
Unit HashTbl;
Interface
Procedure Init;
Function Insert(key : integer) : boolean;
Function Fetch(key : integer) : integer;
Implementation
const N=...;                    { число записей в таблице }
const NB=...;                    { размер пакета }
type Seq = array[1..N] of integer;  { тип таблицы }
var tabl : Seq;                  { таблица }
{ Инициализация таблицы - заполнение пустыми записями }
Procedure Init;
 var i : integer;
 begin for i:=1 to N do tabl[i]:=EMPTY; end;
        { Хеширование - деление по модулю на число пакетов }
Function Hash(key : integer) : integer;
  begin Hash:= key mod (N div NB); end;
        { Добавление элемента в таблицу  }
Function Insert(key : integer) : boolean;
  Var addr, a1, pa : integer;
  begin   pa:=Hash(key);         { вычисление номера пакета }
  addr:=pa*NB+1;                { номер 1-го эл-та в пакете }
  Insert:=true;
  a1:=addr;               { поиск свободного места в пакете }
  while (a1<addr+NB) and (tabl[a1]<>EMPTY) do a1:=a1+1;
  if a1<addr+NB then { своб.место найдено } tabl[a1]:=key
          else { своб.место не найдено } Insert:=false;
end;
                          { Выборка из таблицы }
Function Fetch(key : integer) : integer;
  Var addr, a1 : integer;
  begin
  addr:=Hash(key)*NB+1;          { номер 1-го эл-та в пакете }
  a1:=addr;                    { поиск в пакете }
  while (a1<addr+NB) and (tabl[a1]<>key) do a1:=a1+1;
  if a1<addr+NB then Fetch:=a1   else Fetch:=EMPTY;
  end;
END.
```

ОБЩАЯ ОБЛАСТЬ ПЕРЕПОЛНЕНИЙ. Для таблицы выделяются две области памяти: основная область и область переполнений. Функция хеширования на выходе дает адрес записи или пакета в основной области.

При вставке записи, если ее "законное" место в основной области уже занято, запись заносится на первое свободное место в области переполнения. При поиске, если "законное" место в основной занято записью с другим ключом, выполняется линейный поиск в области переполнения. Программная иллюстрация приведена в примере 3.21.

При размере таблицы N=15 и размере области переполнения NPP=6 запись последовательности чисел:

58 0 75 82 96 38 88 52 66 62 78 4 15 79

дает такой вид таблицы (значком "|" показана граница между основной областью и областью переполнения):

0 -1 62 78 4 -1 96 82 38 -1 -1 -1 -1 58 -1 | 75 88 52 66 15 79

```
{==== Программный пример 3.21 ====}
{ Хешированная таблица с областью переполнения }
Unit HashTbl;
Interface
Procedure Init;
Function Insert(key : integer) : boolean;
Function Fetch(key : integer) : integer;
Implementation
const N=...;                    { число записей в таблице }
const NPP=...;                  { размер области переполнения }
type Seq = array[1..N+NPP] of integer;  { тип таблицы - массив,
в котором первые N эл.  составляют основную область,  а следующие
NPP эл.тов - область переполнения }
var tabl : Seq;                 { таблица }
Procedure Init;{Инициализация таблицы-заполнение пустыми записями}
var i : integer;
begin for i:=1 to N+NPP do tabl[i]:=EMPTY; end;
                        { Хеширование - деление по модулю }
Function Hash(key : integer) : integer;
begin Hash:= key mod N+1; end;
                        { Добавление элемента в таблицу  }
Function Insert(key : integer) : boolean;
Var addr : integer;
begin
  addr:=Hash(key);              { вычисление адреса }
  Insert:=true;
  if tabl[addr]=EMPTY then
        { если место в основной табл.свободно - пишем на него }
    tabl[addr]:=key
  else begin            { если место в основной таблице занято }
             { поиск свободного места в таблице переполнения }
    addr:=N+1;                  { нач.адрес табл.переполнения }
    while (tabl[addr]<>EMPTY) and (addr<N+NPP) do addr:=addr+1;
```

```
    if tabl[addr]<>EMPTY then Insert:=false { нет места }
    else tabl[addr]:=key;      { запись в обл.переполнения }
  end;
end;
Function Fetch(key : integer) : integer;  { Выборка из таблицы }
Var addr : integer;
begin
  addr:=Hash(key);
  if tabl[addr]=key then          { найден в основной таблице }
    Fetch:=addr
  else if tabl[addr]=EMPTY then    { отсутствует в таблице }
    Fetch:=EMPTY
  else              { линейный поиск в таблице переполнения }
  begin  addr:=N+1;              { начало табл.переполнения }
    while (addr<=N+NPP) and (tabl[addr]<>key) do  addr:=addr+1;
   if tabl[addr]<>key then {отсутствует в таблице} Fetch:=EMPTY
   else  { найден в таблице переполнения } Fetch:=addr;
   end;
end;
END.
```

Общая область переполнений требует больше памяти, чем открытые таблицы: если размер открытой таблицы может не превышать размера фактического множества записей, то здесь еще требуется дополнительная память для переполнений. Однако, эффективность доступа к таблице с областью переполнения выше, чем к таблице с повторным хешированием. Если в таблице с повторным хешированием при неудачной первой пробе приходится продолжать поиск во всей таблице, то в таблице с областью переполнения продолжение поиска ограничивается только областью переполнения, размер которой значительно меньше размера основной таблицы.

РАЗДЕЛЬНЫЕ ЦЕПОЧКИ ПЕРЕПОЛНЕНИЙ. Естественным представляется желание ограничить продолжение поиска лишь множеством тех значений ключей, которые претендуют на данное место в основной таблице. Эта идея реализуется в таблицах с цепочками переполнения. В структуру каждой записи добавляется еще одно поле - указатель на следующую запись. Через эти указатели записи с ключами-синонимами связываются в линейный список, начало которого находится в основной таблице, а продолжение - вне ее. При вставке записи в таблицу по функции хеширования вычисляется адрес записи (или пакета) в основной таблице. Если это место в основной таблице свободно, то запись заносится в основную таблицу. Если же место в основной таблице занято, то запись располагается вне ее. Память для такой записи с ключом-синонимом может выделяться либо динамически для каждой новой записи, либо для синонима назначается элемент из заранее выделенной области переполнения. После

размещения записи-синонима поле указателя из записи основной таблицы переносится в поле указателя синонима, а на его место в записи основной таблицы записывается указатель на только что размещенный синоним.

Хотя в таблицах с цепочками переполнений и увеличивается размер каждой записи и несколько усложняется обработка за счет обработки указателей, сужение области поиска дает весьма значительный выигрыш в эффективности.

В программной иллюстрации примера 3.21 используется статическая область переполнения, элементы которой динамически распределяются по цепочкам. Роль указателя играют индексы в области переполнения. Специальное значение индекса EMPTY представляет пустой указатель.

При объеме основной области - 15 и области переполнения - 6 включение в таблицу следующей последовательности чисел:

58 0 75 82 96 38 88 52 66 62 78 4 15 79

привело к такому содержимому основной таблицы и области переполнения (каждый элемент представлен парой <число>:<указатель>,E- пустое значение):

0:20 E:E 62:E 78:E 4:21 E:E 96:19 82:18 38:E E:E E:E
E:E E:E 58:17 E:E 75:E 88:E 52:E 66:E 15:16 79:E

Это содержимое таблицы с цепочками переполнения наглядно представлено на рис. 3.18.

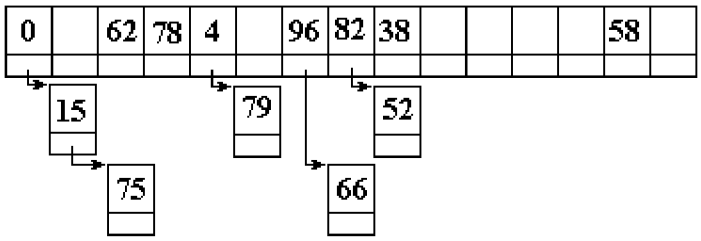


Рис. 3.18. Цепочки переполнения

```
{==== Программный пример 3.21 ====}
{ Хешированная таблица с цепочками переполнений }
Unit HashTbl;
Interface
Procedure Init;
Function Insert(key : integer) : boolean;
Function Fetch(key : integer) : integer;
Implementation
const N=...;    { число записей в таблице }
const NPP=...;  { размер области переполнения }
type rec = record   { запись таблицы }
   key : integer;  { ключ }
   next : integer; { указатель на синоним }
   end;
type Seq = array[1..N+NPP] of rec;  { тип таблицы -
```

```
        основная область и область переполнения }
 var tabl : Seq;                    { таблица }
{ Хеширование - деление по модулю }
 Function Hash(key : integer) : integer;
   begin
    Hash:= key mod N+1; end;
{ Инициализация таблицы - заполнение пустыми записями }
 Procedure Init;
   var i : integer;
   begin
     for i:=1 to N+NPP do
     begin
       tabl[i].key:=EMPTY; tabl[i].next:=EMPTY;
     end;
 end;
{ Добавление элемента в таблицу }
 Function Insert(key : integer) : boolean;
 Var addr1, addr2 : integer;  {адреса- основной, переполнение}
 begin
   addr1:=Hash(key);                  { вычисление адреса }
   Insert:=true;
   if tabl[addr1].key=EMPTY then
          { эл-т в основной области свободен - запись в него }
     tabl[addr1].key:=key
   else                    { эл-т в основной области занят }
            { поиск свободного места в таблице переполнения }
   begin   addr2:=N+1;
     while (tabl[addr2].key<>EMPTY) and (addr2<N+NPP) do
       addr2:=addr2+1;
     if tabl[addr2].key<>EMPTY then Insert:=false { нет места }
     else                  { запись в область переполнения и
                           коррекция указателей в цепочке }
     begin   tabl[addr2].key:=key;
           tabl[addr2].next:=tabl[addr1].next;
           tabl[addr1].next:=addr2;
       end;
     end;
 end;
 Function Fetch(key : integer) : integer;  { Выборка из таблицы }
   Var addr : integer;
   begin
   addr:=Hash(key);
   if tabl[addr].key=key then          { найден на своем месте }
     Fetch:=addr
   else if tabl[addr].key=EMPTY then    { нет в таблице }
```

```
    Fetch:=EMPTY
  else                    { поиск в таблице переполнения }
    begin addr:=tabl[addr].next;
    while (addr<>EMPTY) and (tabl[addr].key<>key) do
      addr:=tabl[addr].next;
    Fetch:=addr;          { адрес в обл.переполнения или EMPTY }
 end;  end;
END.
```

При любом методе построения хешированных таблиц возникает проблема удаления элемента из основной области. При удалении удаляемая запись должна прежде всего быть найдена в таблице. Если запись найдена вторичным хешированием (открытая таблица) или в области переполнения (таблица с общей областью переполнения), то удаляемую запись достаточно пометить как пустую. Если запись найдена в цепочке (таблица с цепочками переполнений), то необходимо также скорректировать указатель предыдущего элемента в цепочке. Если же удаляемая запись находится на своем "законном" месте, то, пометив ее как пустую, мы тем самым сделаем невозможным поиск ее синонимом, возможно, имеющихся в таблице.

Одним из способов решения этой проблемы может быть пометка записи специальным признаком "удаленная". Этот способ часто применяется в таблицах с повторным хешированием и с общей областью переполнений, но он не обеспечивает ни освобождения памяти, ни ускорения поиска при уменьшении числа элементов в таблице. Другой способ – найти любой синоним удаляемой записи и перенести его на "законное" место. Этот способ легко реализуется в таблицах с цепочками, но требует значительных затрат в таблицах с другой структурой, так как требует поиска во всей открытой таблицы или во всей области переполнения с вычислением функции хеширования для каждого проверяемого элемента.

4 ПОЛУСТАТИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ

4.1 Характерные особенности полустатических структур

Полустатические структуры данных характеризуются такими признаками:

- имеют переменную длину и простые процедуры ее изменения;
- изменение длины структуры происходит в определенных пределах, не превышая какого-то максимального (предельного) значения.

Если полустатическую структуру рассматривать на логическом уровне, то о ней можно сказать, что это последовательность данных, связанная отношениями линейного списка. Доступ к элементу может осуществляться по его порядковому номеру.

Физическое представление полустатических структур данных в памяти – это обычно последовательность слотов в памяти, где каждый следующий элемент расположен в памяти в следующем слоте (т.е. вектор). Физическое представление может иметь, также, вид однонаправленного связного списка (цепочки), где каждый следующий элемент адресуется указателем находящимся в текущем элементе. В последнем случае ограничения на длину структуры гораздо менее строгие.

4.2 Стеки

4.2.1 Логическая структура стека

Стек – такой последовательный список с переменной длиной, включение и исключение элементов из которого выполняются только с одной стороны списка, называемого вершиной стека. Применяются и другие названия стека – магазин, очередь, функционирующая по принципу LIFO (Last In – First Out – "последним пришле – первым исключается"). Примеры стека: винтовочный патронный магазин, тупиковый железнодорожный разъезд для сортировки вагонов.

Основные операции над стеком:

- включение нового элемента (английское название push - заталкивать),
- исключение элемента из стека (англ. pop - выскакивать).

Полезными могут быть также вспомогательные операции:

- определение текущего числа элементов в стеке;
- очистка стека;
- неразрушающее чтение элемента из вершины стека, которое может быть реализовано, как комбинация основных операций:

x:=pop(stack); push(stack,x).

Некоторые авторы рассматривают также операции включения/исключения элементов для середины стека, однако структура, для которой возможны такие операции, не соответствует стеку по определению. Для наглядности рассмотрим небольшой пример, демонстрирующий

принцип включения элементов в стек и исключения элементов из стека. На рис. 4.1 изображены состояния стека:

а) пустого;

б-г) после последовательного включения в него элементов 'A', 'B', 'C';

д, е) после последовательного удаления из стека элементов 'C' и 'B';

ж) после включения в стек элемента 'D'.

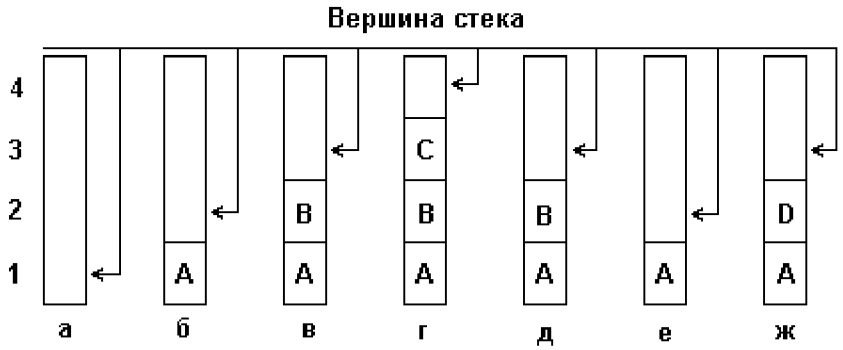


Рис. 4.1. Включение и исключение элементов из стека.

Как видно из рис. 4.1, стек можно представить, например, в виде стопки книг (элементов), лежащей на столе. Присвоим каждой книге свое название, например A, B, C, D… Тогда в момент времени, когда на столе книг нет, про стек аналогично можно сказать, что он пуст, т.е. не содержит ни одного элемента. Если же мы начнем последовательно класть книги одну на другую, то получим стопку книг (допустим, из n книг), или получим стек, в котором содержится n элементов, причем вершиной его будет являться элемент n+1.

Удаление элементов из стека осуществляется аналогичным образом т.е. удаляется последовательно по одному элементу, начиная с вершины, или по одной книге из стопки.

4.2.2 Машинное представление стека и реализация операций

При представлении стека в статической памяти для него выделяется память, как для вектора. В дескрипторе этого вектора кроме обычных для вектора параметров должен находиться также указатель стека – адрес вершины стека. Указатель стека может указывать либо на первый свободный элемент стека, либо на последний записанный в стек элемент. (Какой из этих двух вариантов выбрать, все равно, важно в последствии строго придерживаться его при обработке стека.) В дальнейшем мы будем всегда считать, что указатель стека адресует первый свободный элемент и стек растет в сторону увеличения адресов.

При занесении элемента в стек элемент записывается на место, определяемое указателем стека, затем указатель модифицируется таким образом, чтобы он указывал на следующий свободный элемент (если указатель указывает на последний записанный элемент, то сначала модифицируется указатель, а затем производится запись элемента).

Модификация указателя состоит в прибавлении к нему или в вычитании из него единицы (помните, что наш стек растет в сторону увеличения адресов.

Операция исключения элемента состоит в модификации указателя стека (в направлении, обратном модификации при включении) и выборке значения, на которое указывает указатель стека. После выборки слот, в котором размещался выбранный элемент, считается свободным.

Операция очистки стека сводится к записи в указатель стека начального значения – адреса начала выделенной области памяти.

Определение размера стека сводится к вычислению разности указателей: указателя стека и адреса начала области.

Программный модуль, представленный в примере 4.1, иллюстрирует операции над стеком, расширяющимся в сторону уменьшения адресов. Указатель стека всегда указывает на первый свободный элемент.

В примерах 4.1 и 4.3 предполагается, что в модуле будут уточнены определения предельного размера структуры и типа данных для элемента структуры:

```
const SIZE = ...;
type data = ...;
```

```
{==== Программный пример 4.1 ====}
{ Стек }
unit Stack;
Interface
const SIZE=...;         { предельный размер стека }
type data = ...;{ эл-ты могут иметь любой тип }
Procedure StackInit;
Procedure StackClr;
Function StackPush(a : data) : boolean;
Function StackPop(Var a : data) : boolean;
Function StackSize : integer;
Implementation
Var StA : array[1..SIZE] of data;   { Стек - данные }
{ Указатель на вершину стека, работает на префиксное вычитание }
   top : integer;
Procedure StackInit; {** инициализация - на начало }
begin top:=SIZE; end;       {** очистка = инициализация }
Procedure StackClr;
  begin top:=SIZE; end;
                { ** занесение элемента в стек }
Function StackPush(a: data) : boolean;
begin
  if top=0 then StackPush:=false
  else begin { занесение, затем - увеличение указателя }
    StA[top]:=a; top:=top-1; StackPush:=true;
  end;  end;              { StackPush }
```

```
                { ** выборка элемента из стека }
Function StackPop(var a: data) : boolean;
begin
  if top=SIZE then StackPop:=false
  else begin { указатель увеличивается, затем - выборка }
    top:=top+1; a:=StA[top]; StackPop:=true;
 end;  end;                          { StackPop }
Function StackSize : integer;       {** определение размера }
  begin StackSize:=SIZE-top; end;
END.
```

4.2.3 Стеки в вычислительных системах

Стек является чрезвычайно удобной структурой данных для многих задач вычислительной техники. Наиболее типичной из таких задач является обеспечение вложенных вызовов процедур.

Предположим, имеется процедура A, которая вызывает процедуру B, а та, в свою очередь, - процедуру C. Когда выполнение процедуры A дойдет до вызова B, процедура A приостанавливается и управление передается на входную точку процедуры B. Когда B доходит до вызова C, приостанавливается B и управление передается процедуре C. Когда заканчивается выполнение процедуры C, управление должно быть возвращено в B, причем в точку, следующую за вызовом C. При завершении B управление должно возвращаться в A, в точку, следующую за вызовом B. Правильную последовательность возвратов легко обеспечить, если при каждом вызове процедуры записывать адрес возврата в стек. Так, когда процедура A вызывает процедуру B, в стек заносится адрес возврата в A; когда B вызывает C, в стек заносится адрес возврата в B. Когда C заканчивается, адрес возврата выбирается из вершины стека – а это адрес возврата в B. Когда заканчивается B, в вершине стека находится адрес возврата в A, и возврат из B произойдет в процедуру A.

В микропроцессорах семейства Intel, как и в большинстве современных процессорных архитектур, поддерживается аппаратный стек. Аппаратный стек расположен в ОЗУ, указатель стека содержится в паре специальных регистров — SS:SP, доступных для программиста. Расширяется аппаратный стек в сторону уменьшения адресов, указатель его адресует первый свободный элемент. Выполнение команд CALL и INT, а также аппаратных прерываний включает в себя запись в аппаратный стек адреса возврата. Выполнение команд RET и IRET включает в себя выборку из аппаратного стека адреса возврата и передачу управления по этому адресу. Пара команд — PUSH и POP — обеспечивает использование аппаратного стека для программного решения других задач.

Системы программирования для блочно-ориентированных языков (PASCAL, C и др.) используют стек для размещения в нем локальных переменных процедур и иных программных блоков. При каждой активизации

процедуры память для ее локальных переменных выделяется в стеке; при завершении процедуры эта память освобождается.

Поскольку при вызовах процедур всегда строго соблюдается вложенность, то в вершине стека всегда находится память, содержащая локальные переменные активной в данный момент процедуры. Этот прием делает возможной легкую реализацию рекурсивных процедур. Когда процедура вызывает сама себя, то для всех ее локальных переменных выделяется новая память в стеке, и вложенный вызов работает с собственным представлением локальных переменных. Когда вложенный вызов завершается, занимаемая его переменными область памяти в стеке освобождается, и актуальным становится представление локальных переменных предыдущего уровня. За счет этого в языках PASCAL и C любые процедуры/функции могут вызывать сами себя. В языке PL/1, где по умолчанию приняты другие способы размещения локальных переменных, рекурсивная процедура должна быть определена с описателем RECURSIVE - только тогда ее локальные переменные будут размещаться в стеке.

Рекурсия использует стек в скрытом от программиста виде, но все рекурсивные процедуры могут быть реализованы и без рекурсии, но с явным использованием стека. В программном примере 3.17 была приведена реализация быстрой сортировки Хоара в рекурсивной процедуре. Программный пример 4.2 показывает, как будет выглядеть реализация того же алгоритма но с использованием программного стека.

```
{==== Программный пример 4.2 ====}
{ Быстрая сортировка Хоара (стек) }
Procedure Sort(a : Seq);    { см. раздел 3.8 }
 type board=record         { границы обрабатываемого участка }
  i0, j0 : integer;    end;
 Var i0, j0, i, j, x : integer;
  flag_j : boolean;
  stack  : array[1..N] of board;  { стек }
  stp    : integer; { указатель стека работает на увеличение }
begin              { в стек заносятся общие границы }
 stp:=1; stack[i].i0:=1; stack[i].j0:=N;
 while stp>0 do                { выбрать границы из стека }
 begin  i0:=stack[stp].i0; j0:=stack[stp].j0; stp:=stp-1;
  i:=i0; j:=j0; flag_j:=false;{проход перестановок от i0 до j0}
   while i<j do                     { пока не встретятся i и j }
    begin  if a[i]>a[j] then            { перестановка }
       begin x:=a[i]; a[i]:=a[j]; a[j]:=x; flag_j:= not flag_j;
    end;
    if flag_j then Dec(j) else Inc(i);
 end;
 if i-1>i0 then {занесение в стек границ левого участка}
```

```
 begin  stp:=stp+1; stack[stp].i0:=i0; stack[stp].j0:=i-1;
end;
if j0>i+1 then {занесение в стек границ правого участка}
 begin  stp:=stp+1; stack[stp].i0:=i+1; stack[stp].j0:=j0;
 end;  end;
```

Один проход сортировки Хоара разбивает исходное множество на два множества. Границы полученных множеств запоминаются в стеке. Затем из стека выбираются границы, находящиеся в вершине, и обрабатывается множество, определяемое этими границами. В процессе его обработки в стек может быть записана новая пара границ и т.д. При начальных установках в стек заносятся границы исходного множества. Сортировка заканчивается с опустошением стека.

4.3 Очереди FIFO

4.3.1 Логическая структура очереди

Очередью FIFO (First In – First Out — "первым пришле — первым исключается") называется такой последовательный список переменной длины, в котором включение элементов выполняется только с одной стороны списка (эту сторону часто называют концом или хвостом очереди), а исключение — с другой стороны (называемой началом или головой очереди). Очереди к прилавкам и к кассам являются типичным бытовым примером очереди FIFO.

Основные операции над очередью — те же, что и над стеком — включение, исключение, определение размера, очистка, неразрушающее чтение.

4.3.2 Машинное представление очереди FIFO и реализация операций

При представлении очереди вектором в статической памяти в дополнение к обычным для дескриптора вектора параметрам в нем должны находиться два указателя: на начало очереди (на первый элемент в очереди) и на ее конец (первый свободный элемент в очереди). При включении элемента в очередь элемент записывается по адресу, определяемому указателем на конец, после чего этот указатель увеличивается на единицу. При исключении элемента из очереди выбирается элемент, адресуемый указателем на начало, после чего этот указатель уменьшается на единицу.

Очевидно, что со временем указатель на конец при очередном включении элемента достигнет верхней границы той области памяти, которая выделена для очереди. Однако, если операции включения чередовались с операциями исключения элементов, то в начальной части отведенной под очередь памяти имеется свободное место. Для того, чтобы места, занимаемые исключенными элементами, могли быть повторно использованы, очередь замыкается в кольцо: указатели (на начало и на конец), достигнув конца выделенной области памяти, переключаются на ее начало. Такая организация очереди в памяти называется кольцевой очередью. Возможны, конечно, и другие варианты организации: например, всякий раз, когда указатель конца

достигнет верхней границы памяти, сдвигать все непустые элементы очереди к началу области памяти, но как этот, так и другие варианты требуют перемещения в памяти элементов очереди и менее эффективны, чем кольцевая очередь.

В исходном состоянии указатели на начало и на конец указывают на начало области памяти. Равенство этих двух указателей (при любом их значении) является признаком пустой очереди. Если в процессе работы с кольцевой очередью число операций включения превышает число операций исключения, то может возникнуть ситуация, в которой указатель конца "догонит" указатель начала. Это ситуация заполненной очереди, но если в этой ситуации указатели сравняются, эта ситуация будет неотличима от ситуации пустой очереди. Для различения этих двух ситуаций к кольцевой очереди предъявляется требование, чтобы между указателем конца и указателем начала оставался "зазор" из свободных элементов. Когда этот "зазор" сокращается до одного элемента, очередь считается заполненной и дальнейшие попытки записи в нее блокируются. Очистка очереди сводится к записи одного и того же (не обязательно начального) значения в оба указателя. Определение размера состоит в вычислении разности указателей с учетом кольцевой природы очереди. Программный пример 4.3 иллюстрирует организацию очереди и операции на ней.

```
{==== Программный пример 4.3 ====}
unit Queue;                    { Очередь FIFO - кольцевая }
Interface
const SIZE=...;                { предельный размер очереди }
type data = ...;          { эл-ты могут иметь любой тип }
Procesure QInit;
Procedure Qclr;
Function QWrite(a: data) : boolean;
Function QRead(var a: data) : boolean;
Function Qsize : integer;
Implementation                 { Очередь на кольце  }
var QueueA : array[1..SIZE] of data;     { данные очереди }
  top, bottom : integer;                 { начало  и конец  }
Procedure QInit;               {** инициализация - начало=конец=1 }
  begin top:=1; bottom:=1; end;
Procedure Qclr;                {**очистка - начало=конец }
  begin top:=bottom; end;
Function QWrite(a : data) : boolean;     {**  запись в конец  }
  begin
  if bottom mod SIZE+1=top then { очередь полна } QWrite:=false
  else begin
{ запись, модификация указ.конца с переходом по кольцу }
    Queue[bottom]:=a; bottom:=bottom mod SIZE+1; QWrite:=true;
  end;   end;   { QWrite }
```

```
Function QRead(var a: data) : boolean; {** выборка из начала }
begin
  if top=bottom then QRead:=false  else
{ запись, модификация указ.начала с переходом по кольцу }
  begin  a:=Queue[top]; top:=top mod SIZE + 1; QRead:=true;
 end;   end;              { QRead }
Function QSize : integer;            {** определение размера }
begin
  if top<=bottom then QSize:=bottom-top
  else QSize:=bottom+SIZE-top;
end;              { QSize }
END.
```

4.3.3 Очереди с приоритетами

В реальных задачах иногда возникает необходимость в формировании очередей, отличных от FIFO или LIFO. Порядок выборки элементов из таких очередей определяется приоритетами элементов. Приоритет в общем случае может быть представлен числовым значением, которое вычисляется либо на основании значений каких-либо полей элемента, либо на основании внешних факторов. Так, и FIFO, и LIFO-очереди могут трактоваться как приоритетные очереди, в которых приоритет элемента зависит от времени его включения в очередь. При выборке элемента всякий раз выбирается элемент с наибольшим приоритетом.

Очереди с приоритетами могут быть реализованы на линейных списковых структурах — в смежном или связном представлении. Возможны очереди с приоритетным включеним — в которых последовательность элементов очереди все время поддерживается упорядоченной, т.е. каждый новый элемент включается на то место в последовательности, которое определяется его приоритетом, а при исключении всегда выбирается элемент из начала. Возможны и очереди с приоритетным исключением — новый элемент включается всегда в конец очереди, а при исключении в очереди ищется (этот поиск может быть только линейным) элемент с максимальным приоритетом и после выборки удаляется из последовательности. И в том, и в другом варианте требуется поиск, а если очередь размещается в статической памяти — еще и перемещение элементов.

Наиболее удобной формой для организации больших очередей с приоритетами является сортировка элементов по убыванию приоритетов частично упорядоченным деревом, рассмотренная нами в п.3.9.2.

4.3.4 Очереди в вычислительных системах

Идеальным примером кольцевой очереди в вычислительной системы является буфер клавиатуры в Базовой Системе Ввода-Вывода ПЭВМ IBM PC. Буфер клавиатуры занимает последовательность байтов памяти по адресам от 40:1E до 40:2D включительно. По адресам 40:1A и 40:1C располагаются указатели на начало и конец очереди соответственно. При

нажатии на любую клавишу генерируется прерывание 9. Обработчик этого прерывания читает код нажатой клавиши и помещает его в буфер клавиатуры — в конец очереди. Коды нажатых клавиш могут накапливаться в буфере клавиатуры, прежде чем они будут прочитаны программой. Программа при вводе данных с клавиатуры обращается к прерыванию 16H. Обработчик этого прерывания выбирает код клавиши из буфера — из начала очереди — и передает в программу.

Очередь является одним из ключевых понятий в многозадачных операционных системах (Windows NT, Unix, OS/2, EC и др.). Ресурсы вычислительной системы (процессор, оперативная память, внешние устройства и т.п.) используются всеми задачами, которые одновременно выполняются в среде такой операционной системы. Поскольку многие виды ресурсов реально не допускают одновременного их использования разными задачами, такие ресурсы предоставляются задачам поочередно. Таким образом, задачи, претендующие на использование того или иного ресурса, выстраиваются в очередь к этому ресурсу. Эти очереди обычно приоритетные, однако, довольно часто применяются и FIFO-очереди, так как это единственная логическая организация очереди, которая гарантированно не допускает постоянного вытеснения задачи более приоритетными. LIFO-очереди обычно используются операционными системами для учета свободных ресурсов.

Также в современных операционных системах одним из средств взаимодействия между параллельно выполняемыми задачами являются очереди сообщений, называемые также почтовыми ящиками. Каждая задача имеет свою очередь - почтовый ящик, и все сообщения, отправляемые ей от других задач, попадают в эту очередь. Задача-владелец очереди выбирает из нее сообщения, причем может управлять порядком выборки - FIFO, LIFO или по приоритету.

4.4 Деки

4.4.1 Логическая структура дека

Дек (от англ. deq – double ended queue,т.е очередь с двумя концами) — особый вид очереди в виде последовательного списка, в котором как включение, так и исключение элементов может осуществляться с любого из двух концов списка. Частный случай дека — дек с ограниченным входом и дек с ограниченным выходом.

Логическая и физическая структуры дека аналогичны логической и физической структуре кольцевой FIFO-очереди. Однако, применительно к деку целесообразно говорить не о начале и конце, а о левом и правом конце.

Над деком определены следующие операции:

- включение элемента справа;
- включение элемента слева;
- исключение элемента справа;
- исключение элемента слева;
- определение размера;

- очистка.

На рис. 4.2 в качестве примера показана последовательность состояний дека при включении и удалении пяти элементов. На каждом этапе стрелка указывает с какого конца дека (левого или правого) осуществляется включение или исключение элемента. Элементы соответственно обозначены буквами A, B, C, D, E.

Физическая структура дека в статической памяти идентична структуре кольцевой очереди.

Разработать программный пример, иллюстрирующий организацию дека и операции над ним, не сложно по образцу примеров 4.1, 4.3. В этом модуле должны быть реализованы процедуры и функции:

Function DeqWrRight(a: data): boolean; - включение элемента справа;

Function DeqWrLeft(a: data): boolean; - включение элемента слева;

Function DeqRdRight(var a: data): boolean; - исключение элемента справа;

Function DeqRdLeft(var a:data) : boolean; - исключение элемента слева;

Procedure DeqClr; - очистка;

Function DeqSize : integer; - определение размера.

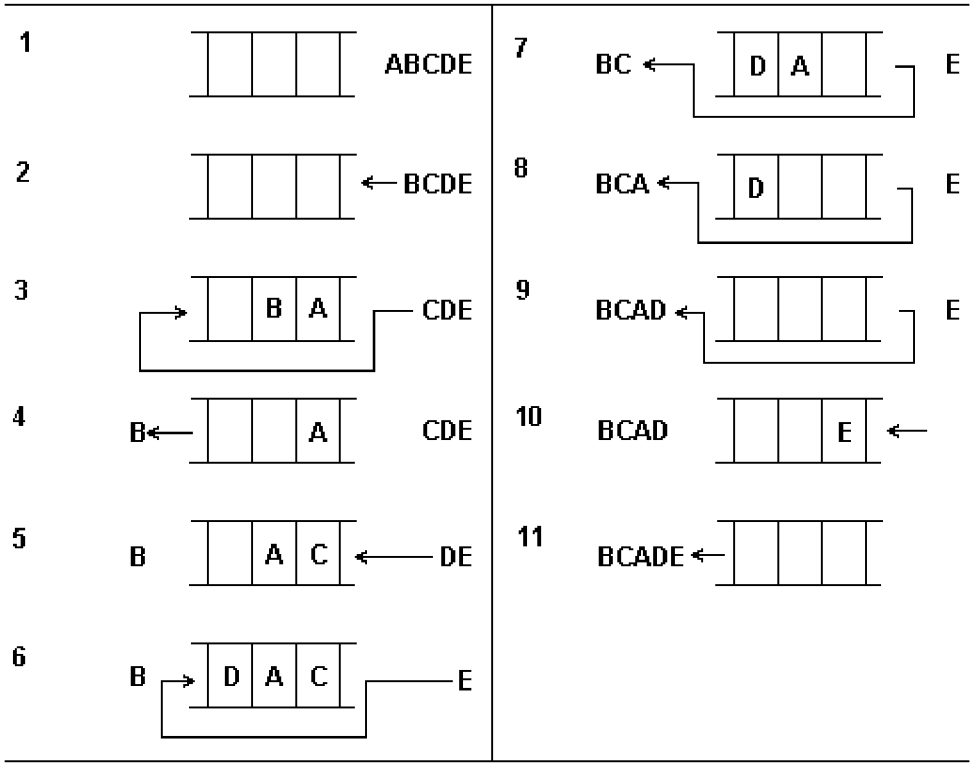


Рис. 4.2. Состояния дека в процессе изменения.

4.4.2 Деки в вычислительных системах

Задачи, требующие структуры дека, встречаются в вычислительной технике и программировании гораздо реже, чем задачи, реализуемые на структуре стека или очереди. Как правило, вся организация дека выполняется программистом без каких-либо специальных средств системной поддержки.

Однако, в качестве примера такой системной поддержки рассмотрим организацию буфера ввода в языке REXX. В обычном режиме буфер ввода

связан с клавиатурой и работает как FIFO-очередь. Однако, в REXX имеется возможность назначить в качестве буфера ввода программный буфер и направить в него вывод программ и системных утилит. В распоряжении программиста имеются операции QUEUE - запись строки в конец буфера и PULL - выборка строки из начала буфера. Дополнительная операция PUSH - запись строки в начало буфера - превращает буфер в дек с ограниченным выходом. Такая структура буфера ввода позволяет программировать на REXX весьма гибкую конвейерную обработку с направлением выхода одной программы на вход другой и модификацией перенаправляемого потока.

4.5 Строки

4.5.1 Логическая структура строки

Строка — это линейно упорядоченная последовательность символов, принадлежащих конечному множеству символов, называемому алфавитом. Строки обладают следующими важными свойствами:

- их длина, как правило, переменна, хотя алфавит фиксирован;
- обычно обращение к символам строки идет с какого-нибудь одного конца последовательности, т.е важна упорядоченность этой последовательности, а не ее индексация; в связи с этим свойством строки часто называют также цепочками;
- чаще всего целью доступа к строке является на отдельный ее элемент (хотя это тоже не исключается), а некоторая цепочка символов в строке.

Символы, входящие в строку, могут принадлежать любому алфавиту. Так, в языке PL/1 наряду с типом данных "символьная строка" — CHAR(n) — существует тип данных "битовая строка" — BIT(n). Битовые строки, составляются из 1-битовых символов, принадлежащих алфавиту: {0, 1}. Все строковые операции с равным успехом применимы как к символьным, так и к битовым строкам.

В зависимости от особенности задачи, свойств применяемого алфавита и представляемого им языка и свойств носителей информации могут применяться и другие способы кодирования символов. В современных вычислительных системах, однако, повсеместно принята кодировка всего множества символов на разрядной сетке фиксированного размера (1 байт).

Хотя строки рассматриваются в главе, посвященной полустатическим структурам данных, в тех или иных конкретных задачах изменчивость строк может варьироваться от полного ее отсутствия до практически неограниченных возможностей изменения. Ориентация на ту или иную степень изменчивости строк определяет и физическое представление их в памяти и особенности выполнения операций над ними. В большинстве языков программирования (C, PASCASL, PL/1 и др.) строки представляются именно как полустатические структуры.

В зависимости от ориентации языка программирования средства работы со строками занимают в языке более или менее значительное место. Рассмотрим три примера возможностей работы со строками.

Язык С является языком системного программирования, типы данных, с которыми работает язык С, максимально приближены к тем типам, с которыми работают машинные команды. Поскольку машинные команды не работают со строками, нет такого типа данных и в языке С. Строки в С представляются в виде массивов символов. Операции над строками могут быть выполнены как операции обработки массивов или же при помощи библиотечных (но не встроенных!) функций строковой обработки.

В языках универсального назначения обычно строковый тип является базовым в языке: STRING в PASCAL, CHAR(n) в PL/1. (В PASCAL длина строки, объявленной таким образом, может меняться от 0 до n, в PL/1 чтобы длина строки могла меняться, она должна быть объявлена с описателем VARING.) Основные операции над строками реализованы как простые операции или встроенные функции. Возможны также библиотеки, обеспечивающие расширенный набор строковых операций.

Язык REXX ориентирован прежде всего на обработку текстовой информации. Поэтому в REXX нет средств описания типов данных: все данные представляются в виде символьных строк. Операции над данными, не свойственные символьным строкам, либо выполняются специальными функциями, либо приводят к прозрачному для программиста преобразованию типов. Так, например, интерпретатор REXX, встретив оператор, содержащий арифметическое выражение, сам переводит его операнды в числовой тип, вычисляет выражение и преобразует результат в символьную строку. Целый ряд строковых операций является простыми операциями языка, а встроенных функций обработки строк в REXX несколько десятков.

4.5.2 Операции над строками

Базовыми операциями над строками являются:

- определение длины строки;
- присваивание строк;
- конкатенация (сцепление) строк;
- выделение подстроки;
- поиск вхождения.

Операция определения длины строки имеет вид функции, возвращаемое значение которой — целое число — текущее число символов в строке.

Операция присваивания имеет тот же смысл, что и для других типов данных.

Операция сравнения строк имеет тот же смысл, что и для других типов данных. Сравнение строк производится по следующим правилам: сравниваются первые символы двух строк. Если символы не равны, то строка, содержащая символ, место которого в алфавите ближе к началу, считается меньшей. Если символы равны, сравниваются вторые, третьи и т.д. символы. При достижении конца одной из строк, строка меньшей длины

считается меньшей. При равенстве длин строк и попарном равенстве всех символов в них строки считаются равными.

Результатом операции сцепления двух строк является строка, длина которой равна суммарной длине строк-операндов, а значение соответствует значению первого операнда, за которым непосредственно следует значение второго операнда. Операция сцепления дает результат, длина которого в общем случае больше длин операндов. Как и во всех операциях над строками, которые могут увеличивать длину строки (присваивание, сцепление, сложные операции), возможен случай, когда длина результата окажется большей, чем отведенный для него объем памяти. Естественно, эта проблема возникает только в тех языках, где длина строки ограничивается. Возможны три варианта решения этой проблемы, определяемые правилами языка или режимами компиляции:

- никак не контролировать такое превышение; возникновение такой ситуации неминуемо приводит к трудно локализуемой ошибке при выполнении программы;
- завершать программу аварийно с локализацией и диагностикой ошибки;
- ограничивать длину результата в соответствии с объемом отведенной памяти.

Операция выделения подстроки выделяет из исходной строки последовательность символов, начиная с заданной позиции n, с заданной длиной l. В языке PASCAL соответствующая функция называется COPY. При реализации операции выделения подстроки в языке программирования и в пользовательской процедуре обязательно должно быть определено правило получения результата для случая, когда начальная позиция n задана такой, что оставшаяся за ней часть исходной строки имеет длину, меньшую заданной длины l, или даже n превышает длину исходной строки. Возможные варианты такого правила:

- аварийное завершение программы с диагностикой ошибки;
- формирование результата меньшей длины, чем задано, возможно даже — пустой строки.

Операция поиска вхождения находит место первого вхождения подстроки-эталона в исходную строку. Результатом операции может быть номер позиции в исходной строке, с которой начинается вхождение эталона или указатель на начало вхождения. В случае отсутствия вхождения результатом операции должно быть некоторое специальное значение, например, нулевой номер позиции или пустой указатель.

На основе базовых операций могут быть реализованы и любые другие, даже сложные операции над строками. Например, операция удаления из строки символов с номерами от n1 до n2, включительно, может быть реализована как последовательность следующих шагов:

- выделение из исходной строки подстроки, начиная с позиции 1, длиной (n1-1) символов;

- выделение из исходной строки подстроки, начиная с позиции (n2+1), длиной, равной длине исходной строки минус n2;
- сцепление подстрок, полученных на предыдущих шагах.

Впрочем, в целях повышения эффективности некоторые вторичные операции также могут быть реализованы как базове — по собственным алгоритмам, с непосредственным доступом к физической структуре строки.

4.5.3 Представление строк в памяти

Представление строк в памяти зависит от того, насколько изменчивыми являются строки в каждой конкретной задаче, и средства такого представления варьируются от абсолютно статического до динамического. Универсальные языки программирования в основном обеспечивают работу со строками переменной длины, но максимальная длина строки должна быть указана при ее создании. Если программиста не устраивают возможности или эффективность тех средств работы со строками, которые предоставляет ему язык программирования, то он может либо определить свой тип данных "строка" и использовать для его представления средства динамической работы с памятью, либо сменить язык программирования на специально ориентированный на обработку текста (CNOBOL, REXX), в которых представление строк базируется на динамическом управлении памятью.

ВЕКТОРНОЕ ПРЕДСТАВЛЕНИЕ СТРОК. Представление строк в виде векторов, принятое в большинстве универсальных языков программирования, позволяет работать со строками, размещенными в статической памяти. Кроме того, векторное представление позволяет легко обращаться к отдельным символам строки как к элементам вектора - по индексу.

Самым простым способом является представление строки в виде вектора постоянной длинны. При этом в памяти отводится фиксированное количество байт, в которые записываются символы строки. Если строка меньше отводимого под нее вектора, то лишние места заполняются пробелами, а если строка выходит за пределы вектора, то лишние (обычно справа строки) символы должны быть отброшены.

На рис.4.3 приведена схема, на которой показано представление двух строк: 'ABCD' и 'PQRSTUVW' в виде вектора постоянной длины на шесть символов.

Рис. 4.3. Представление строк векторами постоянной длины

ПРЕДСТАВЛЕНИЕ СТРОК ВЕКТОРОМ ПЕРЕМЕННОЙ ДЛИНЫ С ПРИЗНАКОМ КОНЦА. Этот и все последующие за ним методы учитывают переменную длину строк. Признак конца - это особый символ, принадлежащий алфавиту (таким образом, полезный алфавит оказывается

меньше на один символ), и занимает то же количество разрядов, что и все остальные символы. Издержки памяти при этом способе составляют 1 символ на строку. Такое представление строки показано на рис.4.4. Специальный символ-маркер конца строки обозначен здесь 'eos'. В языке С, например, в качестве маркера конца строки используется символ с кодом 0.

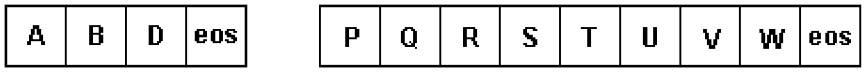


Рис. 4.4. Представление строк переменной длины с признаком конца

ПРЕДСТАВЛЕНИЕ СТРОК ВЕКТОРОМ ПЕРЕМЕННОЙ ДЛИНЫ СО СЧЕТЧИКОМ. Счетчик символов - это целое число, и для него отводится достаточное количество битов, чтобы их с избытком хватало для представления длины самой длинной строки, какую только можно представить в данной машине. Обычно для счетчика отводят от 8 до 16 битов. Тогда при таком представлении издержки памяти в расчете на одну строку составляют 1-2 символа. При использовании счетчика символов возможен произвольный доступ к символам в пределах строки, поскольку можно легко проверить, что обращение не выходит за пределы строки. Счетчик размещается в таком месте, где он может быть легко доступен - в начале строки или в дескрипторе строки. Максимально возможная длина строки, таким образом, ограничена разрядностью счетчика. В PASCAL, например, строка представляется в виде массива символов, индексация в котором начинается с 0; однобайтный счетчик числа символов в строке является нулевым элементом этого массива. Такое представление строк показано на рис.4.5. И счетчик символов, и признак конца в предыдущем случае могут быть доступны для программиста как элементы вектора.

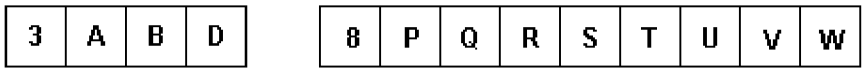


Рис. 4.5. Представление строк переменной длины со счетчиком

В двух предыдущих вариантах обеспечивалось максимально эффективное расходование памяти (1-2 "лишних" символа на строку), но изменчивость строки обеспечивалась крайне неэффективно. Поскольку вектор - статическая структура, каждое изменение длины строки требует создания нового вектора, пересылки в него неизменяемой части строки и уничтожения старого вектора. Это сводит на нет все преимущества работы со статической памятью. Поэтому наиболее популярным способом представления строк в памяти являются вектора с управляемой длиной.

ВЕКТОР С УПРАВЛЯЕМОЙ ДЛИНОЙ. Память под вектор с управляемой длиной отводится при создании строки и ее размер и размещение остаются неизменными все время существования строки. В дескрипторе такого вектора-строки может отсутствовать начальный индекс, так как он может быть зафиксирован раз навсегда установленными соглашениями, но появляется поле текущей длины строки. Размер строки,

таким образом, может изменяться от 0 до значения максимального индекса вектора. "Лишняя" часть отводимой памяти может быть заполнена любыми кодами - она не принимается во внимание при оперировании со строкой. Поле конечного индекса может быть использовано для контроля превышения длиной строки объема отведенной памяти. Представление строк в виде вектора с управляемой длиной (при максимальной длине 10) показано на рис.4.6.

Хотя такое представление строк не обеспечивает экономии памяти, проектировщики систем программирования, как видно, считают это приемлемой платой за возможность работать с изменчивыми строками в статической памяти.

Рис.4.6. Представление строк вектором с управляемой длиной

В программном примере 4.4 приведен модуль, реализующий представление строк вектором с управляемой длиной и некоторые операции над такими строками. Для уменьшения объема в примере в секции Implementation определены не все процедуры/функции. Предоставляем читателю самостоятельно разработать прочие объявленные в секции Interface подпрограммы. Дескриптор строки описывается типом _strdescr, который в точности повторяет структуру, показанную на рис. 4.6. Функция NewStr выделяет две области памяти: для дескриптора строки и для области данных строки. Адрес дескриптора строки, возвращаемый функцией NewStr - тип varstr - является той переменной, значение которой указывается пользователем модуля для идентификации конкретной строки при всех последующих операциях с нею. Область данных, указатель на которую заносится в дескриптор строки, типа _dat_area, описана как массив символов максимально возможного объема 64 Кбайт. Однако, объем памяти, выделяемый под область данных функцией NewStr, как правило, меньший - он задается параметром функции. Хотя индексы в массиве символов строки теоретически могут изменяться от 1 до 65535, значение индекса в каждой

конкретной строке при ее обработке ограничивается полем maxlen дескриптора данной строки. Все процедуры/функции обработки строк работают с символами строки как с элементами вектора, обращаясь к ним по индексу. Адрес вектора процедуры получают из дескриптора строки. Внимание, в процедуре CopyStr длина результата ограничивается максимальной длиной целевой строки.

```
{==== Программный пример 4.4 ====}
{ Представление строк вектором с управляемой длиной }
Unit Vstr;
 Interface
 type _dat_area = array[1..65535] of char;
 type _strdescr = record              { дескриптор строки }
     maxlen, curlen : word; { максимальная и текущая длины }
     strdata : ^_dat_area;            { указатель на данные строки }
     end;
 type varstr = ^_strdescr;   { тип - СТРОКА ПЕРЕМЕННОЙ ДЛИНЫ }
 Function NewStr(len : word) : varstr;
 Procedure DispStr(s : varstr);
 Function LenStr(s : varstr) : word;
 Procedure CopyStr(s1, s2 : varstr);
 Function CompStr(s1, s2 : varstr) : integer;
 Function PosStr(s1, s2 : varstr) : word;
 Procedure ConcatStr(var s1: varstr; s2 : varstr);
 Procedure SubStr(var s1 : varstr; n, l : word);
Implementation
{ Создание строки;  len - максимальная длина строки;
  ф-ция возвращает указатель на дескриптор строки }
Function NewStr(len : word) : varstr;
 var    addr : varstr;
        daddr : pointer;
  begin
   New(addr);          { выделение памяти для дескриптора }
   Getmem(daddr,len);       { выделение памяти для данных }
   { занесение в дескриптор начальных значений }
   addr^.strdata:=daddr; addr^.maxlen:=len; addr^.curlen:=0;
   Newstr:=addr;
end; { Function NewStr }
Procedure DispStr(s : varstr);              { Уничтожение строки }
  begin
  FreeMem(s^.strdata,s^.maxlen);         { уничтожение данных }
  Dispose(s);                        { уничтожение дескриптора }
  end; { Procedure DispStr }
{ Определение длины строки, длина выбирается из дескриптора }
Function LenStr(s : varstr) : word;
```

```
  begin    LenStr:=s^.curlen;    end;        { Function LenStr }
Procedure CopyStr(s1, s2 : varstr);        { Присваивание строк s1:=s2}
var i, len : word;
  begin    { длина строки-результата м.б. ограничена ее макс. длиной }
  if s1^.maxlen<s2^.curlen then len:=s1^.maxlen
  else n:=s2^.curlen;
  { перезапись данных и установка длины результата }
  for i:=1 to n do s1^.strdata^[i]:=s2^.strdata^[i];
  s1^.curlen:=len;
  end;                    { Procedure CopyStr }
{ Сравнение строк - возвращает: 0, если s1=s2; 1 - если s1>s2;
-1 - если s1<s2 }
Function CompStr(s1, s2 : varstr) : integer;
 var i : integer;
 begin   i:=1;              { индекс текущего символа }
 { цикл, пока не будет достигнут конец одной из строк }
 while (i<=s1^.curlen) and (i<=s2^.curlen) do
   { если i-ые символы не равны, функция заканчивается }
 begin  if s1^.strdata^[i]>s2^.strdata^[i] then
   begin   CompStr:=1; Exit;        end;
   if s1^.strdata^[i]<s2^.strdata^[i] then
   begin   CompStr:=-1; Exit;       end;
   i:=i+1;                        { переход к следующему символу }
 end;
 { если выполнение дошло до этой точки, то найден конец одной из
   строк, и все сравненные до сих пор символы были равны;
   строка меньшей длины считается меньшей }
 if s1^.curlen<s2^.curlen then CompStr:=-1
 else if s1^.curlen>s2^.curlen then CompStr:=1
 else CompStr:=0;
 end; { Function CompStr }
. . .
END.
```

СИМВОЛЬНО-СВЯЗНОЕ ПРЕДСТАВЛЕНИЕ СТРОК. Списковое представление строк в памяти обеспечивает гибкость в выполнении разнообразных операций над строками (в частности, операций включения и исключения отдельных символов и целых цепочек) и использование системных средств управления памятью при выделении необходимого объема памяти для строки. Однако, при этом возникают дополнительные расходы памяти. Другим недостатком спискового представления строки является то, что логически соседние элементы строки не являются физически соседними в памяти. Это усложняет доступ к группам элементов строки по сравнению с доступом в векторном представлении строки.

ОДНОНАПРАВЛЕННЫЙ ЛИНЕЙНЫЙ СПИСОК. Каждый символ строки представляется в виде элемента связного списка; элемент содержит код символа и указатель на следующий элемент, как показано на рис. 4.7. Одностороннее сцепление предоставляет доступ только в одном направлении вдоль строки. На каждый символ строки необходим один указатель, который обычно занимает 2-4 байта.

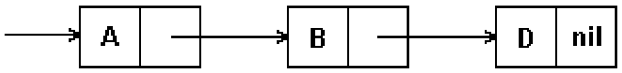


Рис. 4.7. Представление строки однонаправленным связным списком

ДВУНАПРАВЛЕННЫЙ ЛИНЕЙНЫЙ СПИСОК. В каждый элемент списка добавляется также указатель на предыдущий элемент, как показано на рис. 4.8.

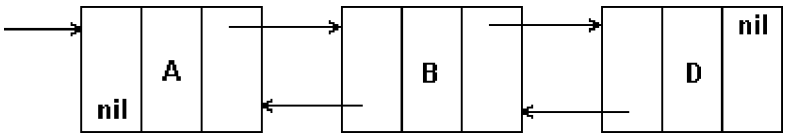


Рис. 4.8. Представление строки двунаправленным связным списком

Двустороннее сцепление допускает двустороннее движение вдоль списка, что может значительно повысить эффективность выполнения некоторых строковых операций. При этом на каждый символ строки необходимо два указателя , т.е. 4-8 байт.

БЛОЧНО-СВЯЗНОЕ ПРЕДСТАВЛЕНИЕ СТРОК. Такое представление позволяет в большинстве операций избежать затрат, связанных с управлением динамической памятью, но в то же время обеспечивает достаточно эффективное использование памяти при работе со строками переменной длины.

МНОГОСИМВОЛЬНЫЕ ЗВЕНЬЯ ФИКСИРОВАННОЙ ДЛИНЫ. Многосимвольные группы (звенья) организуются в список, так что каждый элемент списка, кроме последнего, содержит группу элементов строки и указатель следующего элемента списка. Поле указателя последнего элемента списка хранит признак конца - пустой указатель. В процессе обработки строки из любой ее позиции могут быть исключены или в любом месте вставлены элементы, в результате чего звенья могут содержать меньшее число элементов, чем было первоначально. По этой причине необходим специальный символ, который означал бы отсутствие элемента в соответствующей позиции строки. Обозначим такой символ 'emp', он не должен входить в множество символов, из которых организуется строка. Пример многосимвольных звеньев фиксированной длинны по 4 символа в звене показан на рис. 4.9.

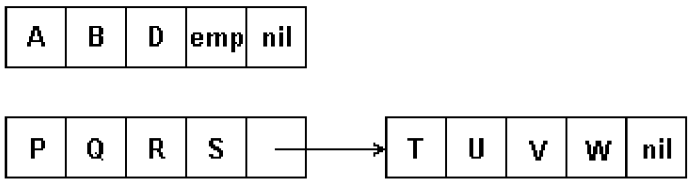


Рис. 4.9. Представление строки многосимвольными звеньями постоянной длины

Такое представление обеспечивает более эффективное использование памяти, чем символьно-связное. Операции вставки/удаления в ряде случаев могут сводиться к вставке/удалению целых блоков. Однако, при удалении одиночных символов в блоках могут накапливаться пустые символы emp, что может привести даже к худшему использованию памяти, чем в символьно-связном представлении.

МНОГОСИМВОЛЬНЫЕ ЗВЕНЬЯ ПЕРЕМЕННОЙ ДЛИНЫ. Переменная длинна блока дает возможность избавиться от пустых символов и тем самым экономить память для строки. Однако появляется потребность в специальном символе - признаке указателя, на рис.4.10 он обозначен символом 'ptr'.

С увеличением длины групп символов, хранящихся в блоках, эффективность использования памяти повышается. Однако негативной характеристикой рассматриваемого метода является усложнение операций по резервированию памяти для элементов списка и возврату освободившихся элементов в общий список доступной памяти.

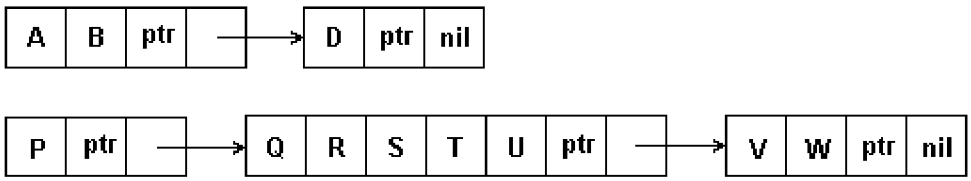


Рис.4.10. Представление строки многосимвольными звеньями переменной длины

Такой метод спискового представления строк особенно удобен в задачах редактирования текста, когда большая часть операций приходится на изменение, вставку и удаление целых слов. Поэтому в этих задачах целесообразно список организовать так, чтобы каждый его элемент содержал одно слово текста. Символы пробела между словами в памяти могут не представляться.

МНОГОСИМВОЛЬНЫЕ ЗВЕНЬЯ С УПРАВЛЯЕМОЙ ДЛИНОЙ. Память выделяется блоками фиксированной длины. В каждом блоке помимо символов строки и указателя на следующий блок содержатся номера первого и последнего символов в блоке. При обработке строки в каждом блоке обрабатываются только символы, расположенные между этими номерами. Признак пустого символа не используется: при удалении символа из строки оставшиеся в блоке символы уплотняются и корректируются граничные номера. Вставка символа может быть выполнена за счет имеющегося в блоке

свободного места, а при отсутствии такового - выделением нового блока. Хотя операции вставки/удаления требуют пересылки символов, диапазон пересылок ограничивается одним блоком. При каждой операции изменения может быть проанализирована заполненность соседних блоков и два полупустых соседних блока могут быть переформированы в один блок. Для определения конца строки может использоваться как пустой указатель в последнем блоке, так и указатель на последний блок в дескрипторе строки. Последнее может быть весьма полезным при выполнении некоторых операций, например, сцепления. В дескрипторе может храниться также и длина строки: считывать ее из дескриптора удобнее, чем подсчитывать ее перебором всех блоков строки.

Пример представления строки в виде звеньев с управляемой длиной на 18 символов показан на рис. 4.11. В программном примере 4.5 приведен модуль, реализующий представление строк звеньями с управляемой длиной. Даже с первого взгляда видно, что он значительно сложнее, чем пример 4.4. Это объясняется тем, что здесь вынуждены обрабатывать как связные (списки блоков), так и векторные (массив символов в каждом блоке) структуры.

Поэтому при последовательной обработке символов строки процедура должна сохранять как адрес текущего блока, так и номер текущего символа в блоке. Для этих целей во всех процедурах/функциях используются переменные cp и bi соответственно. (Процедуры и функции, обрабатывающие две строки - cp1, bi1, cp2, bi2.) Дескриптор строки - тип _strdescr - и блок - тип _block - в точности повторяют структуру, показанную на рис. 4.11. Функция NewStr выделяет память только для дескриптора строки и возвращает адрес дескриптора - тип varstr - он служит идентификатором строки при последующих операциях с нею. Память для хранения данных строки выделяется только по мере необходимости. Во всех процедурах/функциях приняты такие правила работы с памятью:

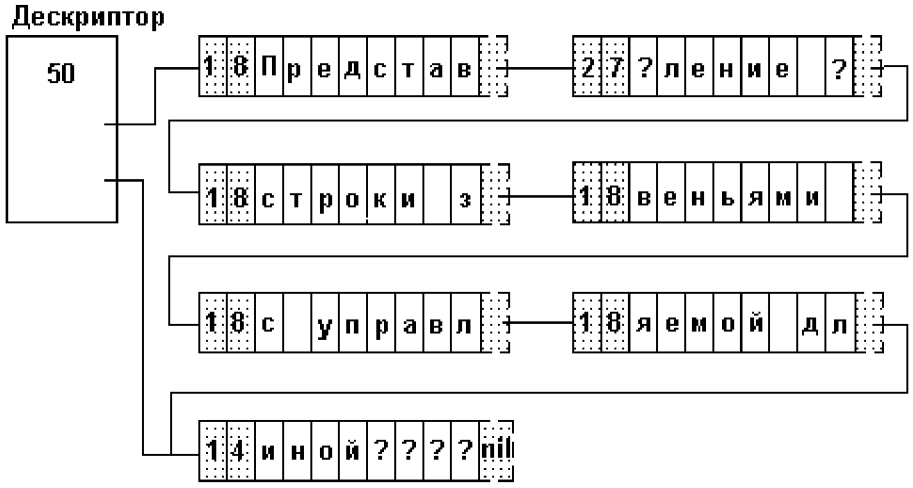


Рис.4.11. Представление строки звеньями управляемой длины

- если выходной строке уже выделены блоки, то используются эти уже выделенные блоки;

- если блоки, выделенные выходной строке, исчерпаны, то по мере необходимости выделяются новые блоки;
- если результирующее значение выходной строки не использует все выделенные строке блоки, лишние блоки освобождаются.

Для освобождения блоков определена специальная внутренняя функция FreeBlock, освобождающая весь список блоков, голова которого задается ее параметром.

Обратите внимание на то, что ни в каких процедурах не контролируется максимальный объем строки результата - он может быть сколь угодно большим, а поле длины в дескрипторе строки имеет тип longint.

```
{==== Программный пример 4.5 ====}
{ Представление строки звеньями управляемой длины }
Unit Vstr;
 Interface
 const BLKSIZE = 8;                { число символов в блоке }
 type  _bptr = ^_block;            { указатель на блок }
      _block = record          { блок }
         i1, i2 : byte;              { номера 1-го и последнего символов }
         strdata : array [1..BLKSIZE] of char;          { символы }
         next : _bptr;               { указатель на следующий блок }
       end;
 type  _strdescr = record          { дескриптор строки }
         len : longint;           { длина строки }
         first, last : _bptr;        { указ.на 1-й и последний блоки }
      end;
type varstr = ^_strdescr;   { тип - СТРОКА ПЕРЕМЕННОЙ ДЛИНЫ }
Function NewStr : varstr;
Procedure DispStr(s : varstr);
Function LenStr(s : varstr) : longint;
Procedure CopyStr(s1, s2 : varstr);
Function CompStr(s1, s2 : varstr) : integer;
Function PosStr(s1, s2 : varstr) : word;
Procedure ConcatStr(var s1: varstr; s2 : varstr);
Procedure SubStr(var s : varstr; n, l : word);
Implementation
Function NewBlock :_bptr; {Внутр. функция-выделение нового блока}
 var n : _bptr;
    i : integer;
  begin
  New(n);                                  { выделение памяти }
  n^.next:=nil; n^.i1:=0; n^.i2:=0;{ начальные значения }
  NewBlock:=n;
end;              { NewBlock }
{*** Внутр.функция - освобождение цепочки блока, начиная с с }
```

```
Function FreeBlock(c : _bptr) : _bptr;
 var x : _bptr;
  begin          { движение по цепочке с освобождением памяти }
  while c<>nil do begin x:=c; c:=c^.next; Dispose(x); end;
  FreeBlock:=nil;                    { всегда возвращает nil }
end; { FreeBlock }
Function NewStr : varstr;   {** Создание строки }
 var addr : varstr;
  begin
   New(addr);                  { выделение памяти для дескриптора }
                { занесение в дескриптор начальных значений }
   addr^.len:=0; addr^.first:=nil; addr^.last:=nil;
   Newstr:=addr;
end; { Function NewStr }
Procedure DispStr(s : varstr);      {** Уничтожение строки }
  begin
  s^.first:=FreeBlock(s^.first);      { уничтожение блоков }
  Dispose(s);                  { уничтожение дескриптора }
end; { Procedure DispStr }
{** Определение длины строки, длина выбирается из дескриптора }
Function LenStr(s : varstr) : longint;
  begin
  LenStr:=s^.len;
  end; { Function LenStr }
        {** Присваивание строк s1:=s2 }
Procedure CopyStr(s1, s2 : varstr);
 var bi1, bi2 : word;  { индексы символов в блоках для s1 и s2 }
    cp1, cp2 : _bptr;  { адреса текущих блоков для s1 и s2 }
    pp : _bptr;          { адрес предыдущего блока для s1 }
  begin
  cp1:=s1^.first; pp:=nil; cp2:=s2^.first;
  if s2^.len=0 then begin
{ если s2 пустая, освобождается вся s1 }
   s1^.first:=FreeBlock(s1^.first); s1^.last:=nil;
   end
  else begin
   while cp2<>nil do begin{ перебор блоков s2 }
    if cp1=nil then begin            { если в s1 больше нет блоков }
{ выделяется новый блок для s1 }
      cp1:=NewBlock;
      if s1^.first=nil then s1^.first:=cp1
      else if pp<>nil then pp^.next:=cp1;
      end;
     cp1^:=cp2^;                { копирование блока }
      { к следующему блоку }
```

```
      pp:=cp1; cp1:=cp1^.next; cp2:=cp2^.next;
      end;{ while }
    s1^.last:=pp;        { последний блок }
{ если в s1 остались лишние блоки - освободить их }
    pp^.next:=FreeBlock(pp^.next);
    end;  { else }
  s1^.len:=s2^.len;
end;              { Procedure CopyStr }
{** Сравнение строк - возвращает:
         0, если s1=s2; 1 - если s1>s2; -1 - если s1<s2 }
Function CompStr(s1, s2 : varstr) : integer;
 var bi1, bi2 : word;
    cp1, cp2 : _bptr;
  begin
  cp1:=s1^.first; cp2:=s2^.first;
  bi1:=cp1^.i1; bi2:=cp2^.i1;
  { цикл, пока не будет достигнут конец одной из строк }
  while (cp1<>nil) and (cp2<>nil) do begin
{ если соответств. символы не равны, ф-ция заканчивается }
    if cp1^.strdata[bi1]>cp2^.strdata[bi2] then begin
      CompStr:=1; Exit;
      end;
    if cp1^.strdata[bi1]<cp2^.strdata[bi2] then begin
      CompStr:=-1; Exit;
      end;
    bi1:=bi1+1;{ к следующему символу в s1 }
    if bi1>cp1^.i2 then begin cp1:=cp1^.next; bi1:=cp1^.i1; end;

    bi2:=bi2+1;{ к следующему символу в s2 }
    if bi2>cp2^.i2 then begin cp2:=cp2^.next; bi2:=cp2^.i1; end;
    end;
        { мы дошли до конца одной из строк,
строка меньшей длины считается меньшей }
  if s1^.len<s2^.len then CompStr:=-1
  else if s1^.len>s2^.len then CompStr:=1
  else CompStr:=0;
end;              { Function CompStr }
   .  .  .
END.
```

Чтобы не перегружать программный пример, в него не включены средства повышения эффективности работы с памятью. Такие средства включаются в операции по выбору программиста. Обратите внимание, например, что в процедуре, связанной с копированием данных (CopyStr) у нас копируются сразу целые блоки. Если в блоке исходной строки были

неиспользуемые места, то они будут и в блоке результирующей строки. Посимвольное копирование позволило бы устранить избыток памяти в строке-результате. Оптимизация памяти, занимаемой данными строки, может производится как слиянием соседних полупустых блоков, так и полным уплотнением данных. В дескриптор строки может быть введено поле - количество блоков в строке. Зная общее количество блоков и длину строки можно при выполнении некоторых операций оценивать потери памяти и выполнять уплотнение, если эти потери превосходят какой-то установленный процент.

5 ДИНАМИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ. СВЯЗНЫЕ СПИСКИ

5.1 Связное представление данных в памяти

Динамические структуры по определению характеризуются отсутствием физической смежности элементов структуры в памяти, непостоянством и непредсказуемостью размера (числа элементов) структуры в процессе ее обработки. В этом разделе рассмотрены особенности динамических структур, определяемые их первым характерным свойством. Особенности, связанные со вторым свойством, рассматриваются в последнем разделе данной главы.

Поскольку элементы динамической структуры располагаются по непредсказуемым адресам памяти, адрес элемента такой структуры не может быть вычислен из адреса начального или предыдущего элемента. Для установления связи между элементами динамической структуры используются указатели, через которые устанавливаются явные связи между элементами. Такое представление данных в памяти называется связным. Элемент динамической структуры состоит из двух полей:

- информационного поля или поля данных, в котором содержатся те данные, ради которых и создается структура; в общем случае информационное поле само является интегрированной структурой - вектором, массивом, записью и т.п.;
- поля связок, в котором содержатся один или несколько указателей, связывающий данный элемент с другими элементами структуры;

Когда связное представление данных используется для решения прикладной задачи, для конечного пользователя "видимым" делается только содержимое информационного поля, а поле связок используется только программистом-разработчиком.

Достоинства связного представления данных - в возможности обеспечения значительной изменчивости структур;

- размер структуры ограничивается только доступным объемом машинной памяти;
- при изменении логической последовательности элементов структуры требуется не перемещение данных в памяти, а только коррекция указателей.

Вместе с тем связное представление не лишено и недостатков, основные из которых:

- работа с указателями требует, как правило, более высокой квалификации от программиста;
- на поля связок расходуется дополнительная память;
- доступ к элементам связной структуры может быть менее эффективным по времени.

Последний недостаток является наиболее серьезным и именно им ограничивается применимость связного представления данных. Если в смежном представлении данных для вычисления адреса любого элемента нам

во всех случаях достаточно было номера элемента и информации, содержащейся в дескрипторе структуры, то для связного представления адрес элемента не может быть вычислен из исходных данных. Дескриптор связной структуры содержит один или несколько указателей, позволяющих войти в структуру, далее поиск требуемого элемента выполняется следованием по цепочке указателей от элемента к элементу. Поэтому связное представление практически никогда не применяется в задачах, где логическая структура данных имеет вид вектора или массива - с доступом по номеру элемента, но часто применяется в задачах, где логическая структура требует другой исходной информации доступа (таблицы, списки, деревья и т.д.).

5.2 Связные линейные списки

Списком называется упорядоченное множество, состоящее из переменного числа элементов, к которым применимы операции включения, исключения. Список, отражающий отношения соседства между элементами, называется линейным. Логические списки мы уже рассматривали в главе 4, но там речь шла о полустатических структурах данных и на размер списка накладывались ограничения. Если ограничения на длину списка не допускаются, то список представляется в памяти в виде связной структуры. Линейные связные списки являются простейшими динамическими структурами данных.

Графически связи в списках удобно изображать с помощью стрелок. Если компонента не связана ни с какой другой, то в поле указателя записывают значение, не указывающее ни на какой элемент. Такая ссылка обозначается специальным именем — nil.

5.2.1 Машинное представление связных линейных списков

На рис. 5.1 приведена структура односвязного списка. На нем поле INF - информационное поле, данные, NEXT - указатель на следующий элемент списка. Каждый список должен иметь особый элемент, называемый указателем начала списка или головой списка, который обычно по формату отличен от остальных элементов. В поле указателя последнего элемента списка находится специальный признак nil, свидетельствующий о конце списка.

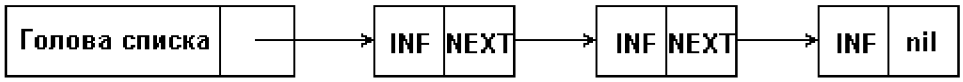


Рис.5.1. Структура односвязного списка

Однако, обработка односвязного списка не всегда удобна, так как отсутствует возможность продвижения в противоположную сторону. Такую возможность обеспечивает двухсвязный список, каждый элемент которого содержит два указателя: на следующий и предыдущий элементы списка. Структура линейного двухсвязного списка приведена на рис. 5.2, где поле NEXT — указатель на следующий элемент, поле PREV — указатель на

предыдущий элемент. В крайних элементах соответствующие указатели должны содержать nil, как и показано на рис. 5.2.

Для удобства обработки списка добавляют еще один особый элемент — указатель конца списка. Наличие двух указателей в каждом элементе усложняет список и приводит к дополнительным затратам памяти, но в то же время обеспечивает более эффективное выполнение некоторых операций над списком.

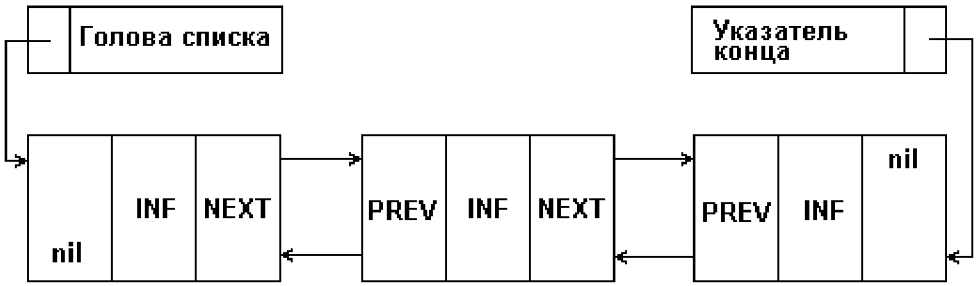


Рис.5.2. Структура двухсвязного списка

Разновидностью рассмотренных видов линейных списков является кольцевой список, который может быть организован на основе как односвязного, так и двухсвязного списков. При этом в односвязном списке указатель последнего элемента должен указывать на первый элемент; в двухсвязном списке в первом и последнем элементах соответствующие указатели переопределяются, как показано на рис.5.3.

При работе с такими списками несколько упрощаются некоторые процедуры, выполняемые над списком. Однако, при просмотре такого списка следует принять некоторых мер предосторожности, чтобы не попасть в бесконечный цикл.

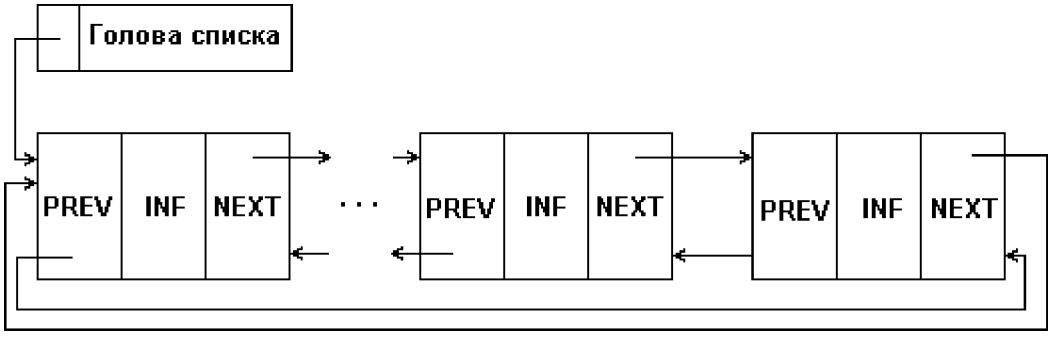


Рис.5.3. Структура кольцевого двухсвязного списка

В памяти список представляет собой совокупность дескриптора и одинаковых по размеру и формату записей, размещенных произвольно в некоторой области памяти и связанных друг с другом в линейно упорядоченную цепочку с помощью указателей. Запись содержит информационные поля и поля указателей на соседние элементы списка, причем некоторыми полями информационной части могут быть указатели на блоки памяти с дополнительной информацией, относящейся к элементу списка. Дескриптор списка реализуется в виде особой записи и содержит

такую информацию о списке, как адрес начала списка, код структуры, имя списка, текущее число элементов в списке, описание элемента и т.д., и т.п. Дескриптор может находиться в той же области памяти, в которой располагаются элементы списка, или для него выделяется какое-нибудь другое место.

5.2.2 Реализация операций над связными линейными списками

Ниже рассматриваются некоторые простые операции над линейными списками. Выполнение операций иллюстрируется в общем случае рисунками со схемами изменения связей и программными примерами.

На всех рисунках сплошными линиями показаны связи, имевшиеся до выполнения и сохранившиеся после выполнения операции. Пунктиром показаны связи, установленные при выполнении операции. Значком 'x' отмечены связи, разорванные при выполнении операции. Во всех операциях чрезвычайно важна последовательность коррекции указателей, которая обеспечивает корректное изменение списка, не затрагивающее другие элементы. При неправильном порядке коррекции легко потерять часть списка. Поэтому на рисунках рядом с устанавливаемыми связями в скобках показаны шаги, на которых эти связи устанавливаются.

В программных примерах подразумеваются определенными следующие типы данных:

- любая структура информационной части списка:
  type data = ...;
- элемент односвязного списка (sll - single linked list):

```
type
    sllptr = ^slltype; { указатель в односвязном списке }
    slltype = record { элемент односвязного списка }
        inf : data;    { информационная часть }
        next : sllptr; { указатель на следующий элемент }
    end;
```

- элемент двухсвязного списка (dll - double linked list):

```
type
    dllptr = ^dlltype;     { указатель в двухсвязном списке }
    dlltype = record { элемент односвязного списка }
        inf : data;    { информационная часть }
        next : sllptr; { указатель на следующий элемент (вперед) }
        prev : sllptr; { указатель на предыдущий элемент (назад) }
    end;
```

Перебор элементов списка. Эта операция, возможно, чаще других выполняется над линейными списками. При ее выполнении осуществляется последовательный доступ к элементам списка — ко всем до конца списка или до нахождения искомого элемента.

Алгоритм перебора для односвязного списка представляется программным примером 5.1.

```
{==== Программный пример 5.1 ====}
{ Перебор 1-связного списка }
Procedure LookSll(head : sllptr);
{ head - указатель на начало списка }
var cur : sllptr;  { адрес текущего элемента }
begin
        cur:=head; { 1-й элемент списка назначается текущим }
        while cur<>nil do begin   < обработка c^.inf >
        { обрабатывается информационная часть того эл-та, на который
         указывает cur. Обработка может состоять в:
                - печати содержимого инф.части;
                - модификации полей инф.части;
                - сравнения полей инф.части с образцом при поиске по
           ключу;
                - подсчете итераций цикла при поиске по номеру;
                - и т.д., и т.п. }
       cur:=cur^.next; { из текущего эл-та выбирается указатель на след.
          эл-т и для следующей итерации следующий эл-т становится
          текущим; если текущий эл-т был последний, то его поле next
          содержит пустой указатель и, т. обр. в cur запишется nil, что
          приведет к выходу из цикла при проверке условия while }
 end;  end;
```

В двухсвязном списке возможен перебор как в прямом направлении (он выглядит точно так же, как и перебор в односвязном списке), так и в обратном. В последнем случае параметром процедуры должен быть tail - указатель на конец списка, и переход к следующему элементу должен осуществляться по указателю назад:

```
cur:=cur^.prev;
```

В кольцевом списке окончание перебора должно происходить не по признаку последнего элемента - такой признак отсутствует, а по достижению элемента, с которого начался перебор. Алгоритмы перебора для двусвязного и кольцевого списка мы оставляем читателю на самостоятельную разработку.

Вставка элемента в список. Вставка элемента в середину односвязного списка показана на рис.5.4 и в примере 5.2.

```
{==== Программный пример 5.2 ====}
{ Вставка элемента в середину 1-связного списка }
Procedure InsertSll(prev : sllptr; inf : data);
{ prev - адрес предыдущего эл-та; inf - данные нового эл-та }
var cur : sllptr;  { адрес нового эл-та }
begin
        { выделение памяти для нового эл-та и запись в его инф.часть }
        New(cur); cur^.inf:=inf;
        cur^.next:=prev^.next; { эл-т, следовавший за предыдущим теперь
```

будет следовать за новым }
prev^.next:=cur; { новый эл-т следует за предыдущим }
end;

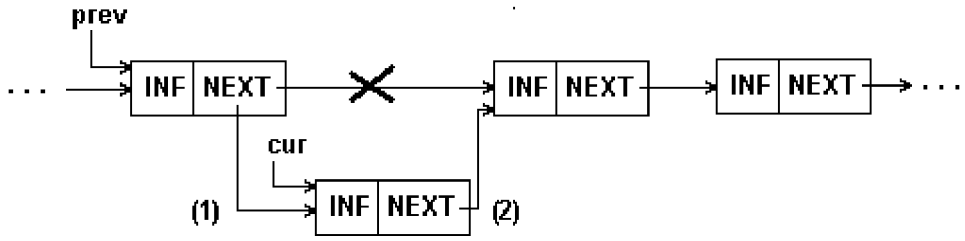


Рис.5.4. Вставка элемента в середину 1-связного списка

Рисунок 5.5 представляет вставку в двухсвязный список.

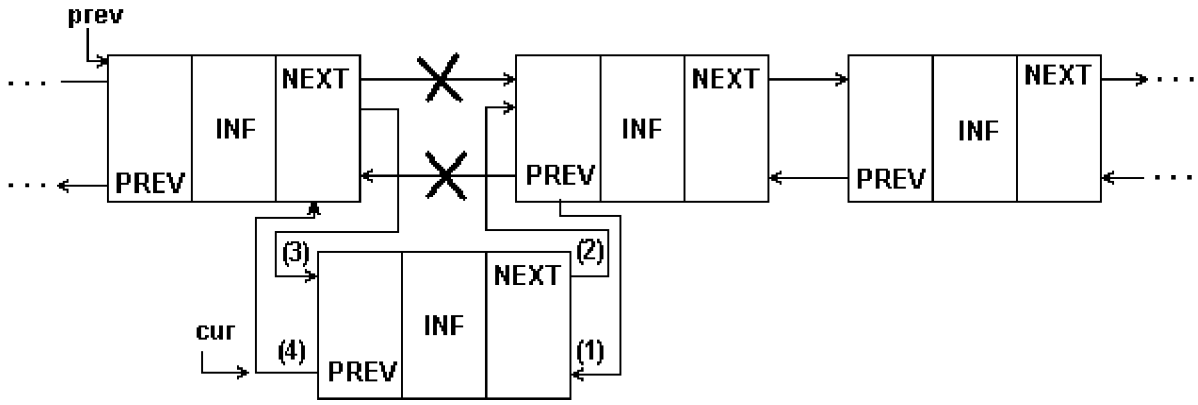


Рис.5.5. Вставка элемента в середину 2-связного списка

Приведенные примеры обеспечивают вставку в середину списка, но не могут быть применены для вставки в начало списка. При последней должен модифицироваться указатель на начало списка, как показано на рис.5.6.

Программный пример 5.3 представляет процедуру, выполняющую вставку элемента в любое место односвязного списка.

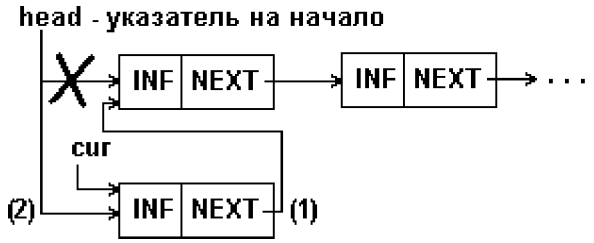


Рис.5.6. Вставка элемента в начало 1-связного списка

```
{==== Программный пример 5.3 ====}
{ Вставка элемента в любое место 1-связного списка }
Procedure InsertSll
var head : sllptr; { указатель на начало списка, может измениться в
        процедуре, если head=nil - список пустой }
prev : sllptr; { указатель на эл-т, после к-рого делается, если prev-nil -
```

```
        вставка перед 1-ым эл-том }
inf : data { - данные нового эл-та }
var cur : sllptr;  { адрес нового эл-та }
begin
 { выделение памяти для нового эл-та и запись в его инф.часть }
 New(cur); cur^.inf:=inf;
 if prev<>nil then begin { если есть предыдущий эл-т - вставка в середину
                  списка, см. прим.5.2 }
        cur^.next:=prev^.next;  prev^.next:=cur;
 end
 else begin { вставка в начало списка }
        cur^.next:=head; { новый эл-т указывает на бывш. 1-й эл-т списка;
             если head=nil, то нов. эл-т будет и последним эл-том списка }
        head:=cur; { новый эл-т становится 1-ым в списке, указатель на
             начало теперь указывает на него }
end; end;
```

Удаление элемента из списка. Удаление элемента из односвязного списка показано на рис.5.7.

Очевидно, что процедуру удаления легко выполнить, если известен адрес элемента, предшествующего удаляемому (prev на рис.5.7.а). Мы, однако, на рис. 5.7 и в примере 5.4 приводим процедуру для случая, когда удаляемый элемент задается своим адресом (del на рис.5.7). Процедура обеспечивает удаления как из середины, так и из начала списка.

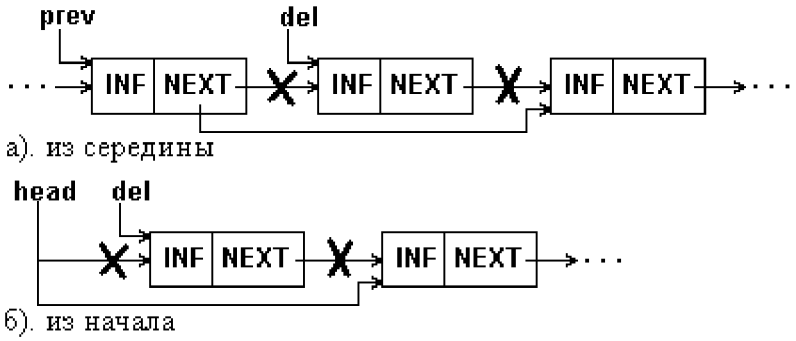


Рис.5.7. Удаление элемента из 1-связного списка

```
{==== Программный пример 5.4 ====}
{ Удаление элемента из любого места 1-связного списка }
Procedure DeleteSll(
var head : sllptr; { указатель на начало списка, может измениться в
          процедуре }
   del : sllptr   { указатель на эл-т, к-рый удаляется }  );
   prev : sllptr;  { адрес предыдущего эл-та }
begin
 if head=nil then begin { попытка удаления из пустого списка
        расценивается как ошибка (в последующих примерах этот случай
```

```
            учитываться на будет) }
    Writeln('Ошибка!'); Halt;
    end;
    if del=head then { если удаляемый эл-т - 1-й в списке, то следующий за
                       ним становится первым }
    head:=del^.next
    else begin { удаление из середины списка }
    { приходится искать эл-т, предшествующий удаляемому; поиск
      производится перебором списка с самого его начала, пока не будет
      найден эл-т, поле next к-рого совпадает с адресом удаляемого
      элемента }
    prev:=head^.next;
    while (prev^.next<>del) and (prev^.next<>nil) do
    prev:=prev^.next;
    if prev^.next=nil then begin { это случай, когда перебран весь список, но
       эл-т не найден, он отсутствует в списке; расценивается как
      ошибка (в последующих примерах этот случай учитываться не будет) }
    Writeln('Ошибка!'); Halt;
    end;
    prev^.next:=del^.next; { предыдущий эл-т теперь указывает на следующий
                             за удаляемым }
       end;
      { элемент исключен из списка, теперь можно освободить занимаемую
        им память }
      Dispose(del);
    end;
```

Удаление элемента из двухсвязного списка требует коррекции большего числа указателей, как показано на рис.5.8.

Процедура удаления элемента из двухсвязного списка окажется даже проще, чем для односвязного, так как в ней не нужен поиск предыдущего элемента, он выбирается по указателю назад.

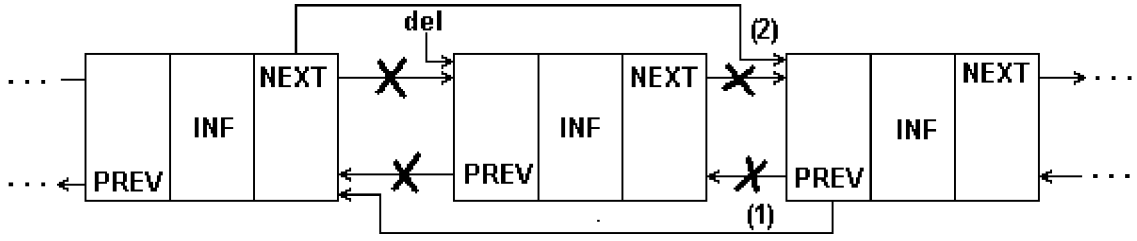


Рис.5.8. Удаление элемента из 2-связного списка

Перестановка элементов списка. Изменчивость динамических структур данных предполагает не только изменения размера структуры, но и изменения связей между элементами. Для связных структур изменение связей не требует пересылки данных в памяти, а только изменения

указателей в элементах связной структуры. В качестве примера приведена перестановка двух соседних элементов списка. В алгоритме перестановки в односвязном списке (рис.5.9, пример 5.5) исходили из того, что известен адрес элемента, предшествующего паре, в которой производится перестановка. В приведенном алгоритме также не учитывается случай перестановки первого и второго элементов.

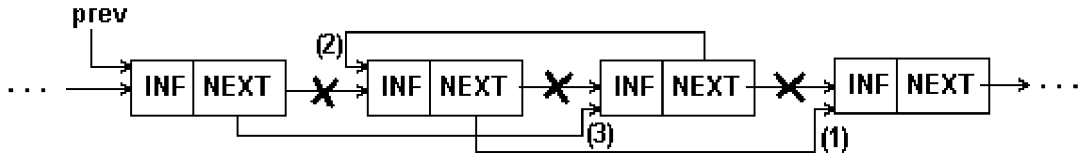


Рис.5.9. Перестановка соседних элементов 1-связного списка

```
{==== Программный пример 5.5 ====}
{ Перестановка двух соседних элементов в 1-связном списке }
Procedure ExchangeSll(
    prev : sllptr   { указатель на эл-т, предшествующий
                переставляемой паре }  );
 var p1, p2 : sllptr;  { указатели на эл-ты пары }
  begin
  p1:=prev^.next;     { указатель на 1-й эл-т пары }
  p2:=p1^.next;       { указатель на 2-й эл-т пары }
  p1^.next:=p2^.next; { 1-й элемент пары теперь указывает на
    следующий за парой }
  p2^.next:=p1;       { 1-й эл-т пары теперь следует за 2-ым }
  prev^.next:=p2;     { 2-й эл-т пары теперь становится 1-ым }
end;
```

В процедуре перестановки для двухсвязного списка (рис.5.10.) нетрудно учесть и перестановку в начале/конце списка.

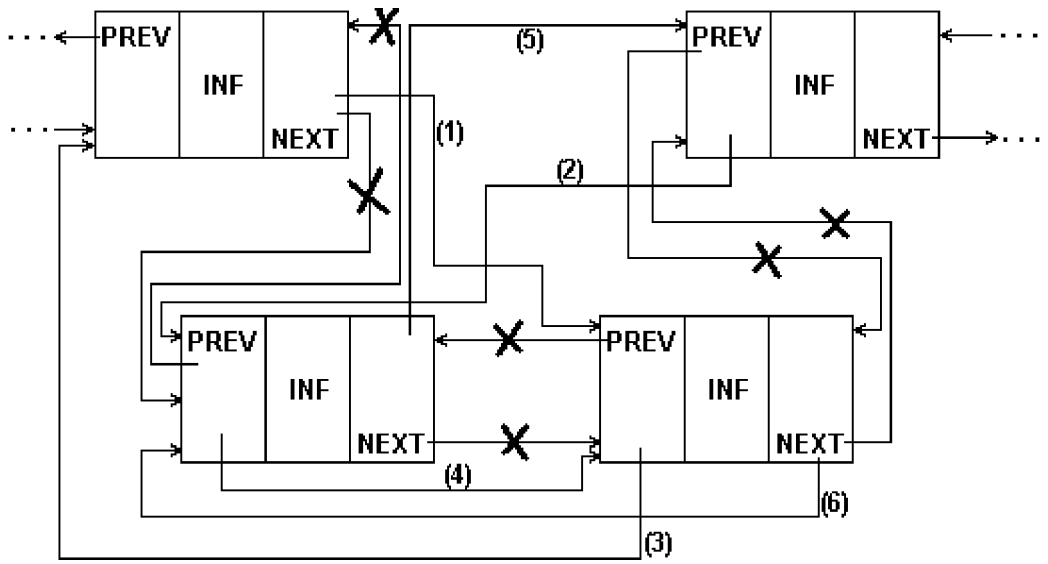


Рис.5.10. Перестановка соседних элементов 2-связного списка

Копирование части списка. При копировании старый список сохраняется в памяти и создается новый список. Информационные поля элементов нового списка содержат те же данные, что и в элементах старого списка, но поля связок в новом списке совершенно другие, поскольку элементы нового списка расположены по другим адресам в памяти. Существенно, что операция копирования предполагает дублирование данных в памяти. Если после создания копии будут изменены данные в исходном списке, то изменение не будет отражено в копии и наоборот. Копирование для односвязного списка показано в программном примере 5.6.

```
{==== Программный пример 5.6 ====}
{ Копирование части 1-связного списка. head - указатель на начало
   копируемой части; num - число эл-тов.
   Ф-ция возвращает указатель на список-копию }
Function CopySll ( head : sllptr; num : integer) : sllptr;
 var cur, head2, cur2, prev2 : sllptr;
 begin
  if head=nil then { исходный список пуст - копия пуста }
    CopySll:=nil
  else begin
    cur:=head; prev2:=nil;
    { перебор исходного списка до конца или по счетчику num }
    while (num>0) and (cur<>nil) do begin
{ выделение памяти для эл-та выходного списка и запись в него
  информационной части }
     New(cur2); cur2^.inf:=cur^.inf;
{ если 1-й эл-т выходного списка - запоминается указатель на
начало, иначе - записывается указатель в предыдущий элемент }
     if prev2<>nil then prev2^.next:=cur2 else head2:=cur2;
     prev2:=cur2;  { текущий эл-т становится предыдущим }
     cur:=cur^.next;  { продвижение по исходному списку }
     num:=num-1;   { подсчет эл-тов }
     end;
   cur2^.next:=nil; { пустой указатель - в последний эл-т
                      выходного списка }
   CopySll:=head2;  { вернуть указатель на начало вых.списка }
end;   end;
```

Слияние двух списков. Операция слияния заключается в формировании из двух списков одного — она аналогична операции сцепления строк. В случае односвязного списка, показанном в примере 5.7, слияние выполняется очень просто. Последний элемент первого списка содержит пустой указатель на следующий элемент, этот указатель служит признаком конца списка. Вместо этого пустого указатель в последний элемент первого списка заносится указатель на начало второго списка. Таким образом, второй список становится продолжением первого.

```
{==== Программный пример 5.7 ====}
{ Слияние двух списков. head1 и head2 - указатели на начала
  списков. На результирующий список указывает head1 }
Procedure Unite (var head1, head2 : sllptr);
 var cur : sllptr;
 begin         { если 2-й список пустой - нечего делать }
  if head2<>nil then begin
    { если 1-й список пустой, выходным списком будет 2-й }
  if head1=nil then head1:=head2
  else     { перебор 1-го списка до последнего его эл-та }
   begin  cur:=head1;
    while cur^.next<>nil do cur:=cur^.next;
    { последний эл-т 1-го списка указывает на начало 2-го }
    cur^.next:=head2;
   end;   head2:=nil; { 2-й список аннулируется }
 end; end;
```

5.2.3. Применение линейных списков

Линейные списки находят широкое применение в приложениях, где непредсказуемы требования на размер памяти, необходимой для хранения данных; большое число сложных операций над данными, особенно включений и исключений. На базе линейных списков могут строится стеки, очереди и деки. Представление очереди с помощью линейного списка позволяет достаточно просто обеспечить любые желаемые дисциплины обслуживания очереди. Особенно это удобно, когда число элементов в очереди трудно предсказуемо.

В программном примере 5.8 показана организация стека на односвязном линейном списке. Это пример функционально аналогичен примеру 4.1 с той существенной разницей, что размер стека здесь практически неограничен.

Стек представляется как линейный список, в котором включение элементов всегда производятся в начала списка, а исключение - также из начала. Для представления его нам достаточно иметь один указатель - top, который всегда указывает на последний записанный в стек элемент. В исходном состоянии (при пустом стеке) указатель top — пустой. Процедуры StackPush и StackPop сводятся к включению и исключению элемента в начало списка. Обратите внимание, что при включении элемента для него выделяется память, а при исключении — освобождается. Перед включением элемента проверяется доступный объем памяти, и если он не позволяет выделить память для нового элемента, стек считается заполненным. При очистке стека последовательно просматривается весь список и уничтожаются его элементы.

При списковом представлении стека оказывается непросто определить размер стека. Эта операция могла бы потребовать перебора всего списка с подсчета числа элементов. Чтобы избежать последовательного перебора

всего списка мы ввели дополнительную переменную stsize, которая отражает текущее число элементов в стеке и корректируется при каждом включении/исключении.

```
{==== Программный пример 5.8 ====}
{ Стек на 1-связном линейном списке }
unit Stack;
Interface
type data = ...; { эл-ты могут иметь любой тип }
Procedure StackInit;
Procedure StackClr;
Function StackPush(a : data) : boolean;
Function StackPop(Var a : data) : boolean;
Function StackSize : integer;
Implementation
type stptr = ^stit;  { указатель на эл-т списка }
     stit = record   { элемент списка }
       inf : data;   { данные }
       next: stptr;  { указатель на следующий эл-т }
       end;
Var top : stptr; { указатель на вершину стека }
   stsize : longint;  { размер стека }
{** инициализация - список пустой }
Procedure StackInit;
  begin   top:=nil; stsize:=0;  end; { StackInit }
{** очистка - освобождение всей памяти }
Procedure StackClr;
 var x : stptr;
  begin   { перебор эл-тов до конца списка и их уничтожение }
  while top<>nil do
    begin  x:=top; top:=top^.next; Dispose(x);  end;
  stsize:=0;
end; { StackClr }
Function StackPush(a: data) : boolean;    { занесение в стек }
 var x : stptr;
  begin      { если нет больше свободной памяти - отказ }
  if MaxAvail<SizeOf(stit) then StackPush:=false
  else   { выделение памяти для эл-та и заполнение инф.части }
    begin  New(x); x^.inf:=a;
             { новый эл-т помещается в голову списка }
      x^.next:=top; top:=x;
      stsize:=stsize+1; { коррекция размера }
      StackPush:=true;
    end;
end; { StackPush }
```

```
Function StackPop(var a: data) : boolean;   { выборка из стека }
 var x : stptr;
  begin
  { список пуст - стек пуст }
  if top=nil then StackPop:=false
  else begin
    a:=top^.inf; { выборка информации из 1-го эл-та списка }
    { 1-й эл-т исключается из списка, освобождается память }
    x:=top; top:=top^.next; Dispose(x);
    stsize:=stsize-1; { коррекция размера }
    StackPop:=true;
end;  end; { StackPop }
Function StackSize : integer;    { определение размера стека }
  begin   StackSize:=stsize;   end; { StackSize }
END.
```

Программный пример для организация на односвязном линейном списке очереди FIFO разработайте самостоятельно. Для линейного списка, представляющего очередь, необходимо будет сохранять: top — указатель на первый элемент списка, и bottom — на последний элемент.

Линейные связные списки иногда используются также для представления таблиц — в тех случаях, когда размер таблицы может существенно изменяться в процессе ее существования. Однако, то обстоятельство, что доступ к элементам связного линейного списка может быть только последовательным, не позволяет применить к такой таблице эффективный двоичный поиск, что существенно ограничивает их применимость. Поскольку упорядоченность такой таблицы не может помочь в организации поиска, задачи сортировки таблиц, представленных линейными связными списками, возникают значительно реже, чем для таблиц в векторном представлении. Однако, в некоторых случаях для таблицы, хотя и не требуется частое выполнение поиска, но задача генерации отчетов требует расположения записей таблицы в некотором порядке. Для упорядочения записей такой таблицы применимы любые алгоритмы из раздела 3.9.

Некоторые алгоритмы, возможно, потребуют каких-либо усложнений структуры, например, быструю сортировку Хоара целесообразно проводить только на двухсвязном списке, в цифровой сортировке удобно создавать промежуточные списки для цифровых групп и т.д. Мы приведем два простейших примера сортировки односвязного линейного списка. В обоих случаях мы предполагаем, что определены типы данных:

```
type lptr = ^item; { указатель на элемент списка }
    item = record    { элемент списка }
      key : integer; { ключ }
      inf : data;    { данные }
      next: lptr;    { указатель на элемент списка }
```

```
end;
```

В обоих случаях сортировка ведется по возрастанию ключей. В обоих случаях параметром функции сортировки является указатель на начало не отсортированного списка, функция возвращает указатель на начало отсортированного списка. Прежний, несортированный список перестает существовать.

Пример 5.9 демонстрирует сортировку выборкой. Указатель newh является указателем на начало выходного списка, исходно — пустого. Во входном списке ищется максимальный элемент. Найденный элемент исключается из входного списка и включается в начало выходного списка. Работа алгоритма заканчивается, когда входной список станет пустым. Обратим внимание на несколько особенностей алгоритма. Во-первых, во входном списке ищется всякий раз не минимальный, а максимальный элемент. Поскольку элемент включается в начало выходного списка (а не в конец выходного множества, как было в программном примере 3.7), элементы с большими ключами оттесняются к концу выходного списка и последний, таким образом, оказывается отсортированным по возрастанию ключей. Во-вторых, при поиске во входном списке сохраняется не только адрес найденного элемента в списке, но и адрес предшествующего ему в списке элемента — это впоследствии облегчает исключение элемента из списка (вспомните пример 5.4). В-третьих, обратите внимание на то, что у нас не возникает никаких проблем с пропуском во входном списке тех элементов, которые уже выбраны — они просто исключены из входной структуры данных.

```
{==== Программный пример 5.9 ====}
{ Сортировка выборкой на 1-связном списке }
Function Sort(head : lptr) : lptr;
 var newh, max, prev, pmax, cur : lptr;
  begin       newh:=nil;       { выходной список - пустой }
  while head<>nil do { цикл, пока не опустеет входной список }
    begin   max:=head; prev:=head; { нач.максимум - 1-й эл-т }
    cur:=head^.next;     { поиск максимума во входном списке }
    while cur<>nil do begin
      if cur^.key>max^.key then begin
{ запоминается адрес максимума и адрес предыдущего эл-та }
        max:=cur; pmax:=prev;
    end;    prev:=cur; cur:=cur^.next; { движение по списку }
     end;       { исключение максимума из входного списка }
    if max=head then head:=head^.next
    else pmax^.next:=max^.next;
    { вставка в начало выходного списка }
    max^.next:=newh; newh:=max;
  end;  Sort:=newh;
 end;
```

В программном примере 5.10 — иллюстрации сортировки вставками — из входного списка выбирается (и исключается) первый элемент и вставляется в выходной список "на свое место" в соответствии со значениями ключей. Сортировка включением на векторной структуре в примере 3.11 требовала большого числа перемещений элементов в памяти. Обратите внимание на то, что в двух последних примерах пересылок данных не происходит, все записи таблиц остаются на своих местах в памяти, меняются только связи между ними — указатели.

```
{==== Программный пример 5.10 ====}
{ Сортировка вставками на 1-связном списке }
type data = integer;
Function Sort(head : lptr) : lptr;
 var newh, cur, sel : lptr;
  begin
  newh:=nil;  { выходной список - пустой }
  while head<>nil do begin { цикл, пока не опустеет
                        входной список }
    sel:=head;  { эл-т, который переносится в выходной список }
    head:=head^.next;        { продвижение во входном списке }
    if (newh=nil) or (sel^.key<newh^.key) then begin
{выходной список пустой или элемент меньше 1-го-вставка в начало}
    sel^.next:=newh; newh:=sel;   end
    else begin              { вставка в середину или в конец }
     cur:=newh;
      { до конца выходного списка или пока ключ следующего
       эл-та не будет больше вставляемого }
     while (cur^.next<>nil) and (cur^.next^.key<sel^.key) do
      cur:=cur^.next;
     { вставка в выходной список после эл-та cur }
     sel^.next:=cur^.next;    cur^.next:=sel;
    end;   end;   Sort:=newh;
 end;
```

5.3 Мультисписки

В программных системах, обрабатывающих объекты сложной структуры, могут решаться разные подзадачи, каждая из которых требует, возможно, обработки не всего множества объектов, а лишь какого-то его подмножества. Так, например, в автоматизированной системе учета лиц, пострадавших вследствие аварии на ЧАЭС, каждая запись об одном пострадавшем содержит более 50 полей в своей информационной части. Решаемые же автоматизированной системой задачи могут потребовать выборки, например:

- участников ликвидации аварии;

- переселенцев из зараженной зоны;
- лиц, состоящих на квартирном учете;
- лиц с заболеваниями щитовидной железы ;
- и т.д., и т.п.

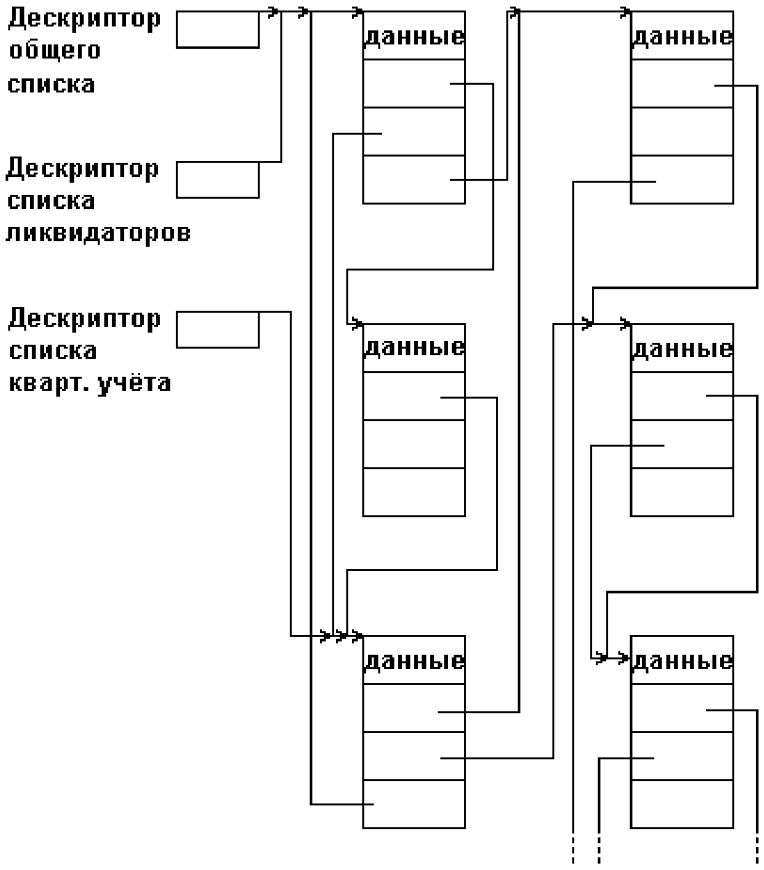


Рис.5.11. Пример мультисписка

Для того, чтобы при выборке каждого подмножества не выполнять полный просмотр с отсеиванием записей, к требуемому подмножеству не относящихся, в каждую запись включаются дополнительные поля ссылок, каждое из которых связывает в линейный список элементы соответствующего подмножества. В результате получается многосвязный список или мультисписок, каждый элемент которого может входить одновременно в несколько односвязных списков. Пример такого мультисписка для названной нами автоматизированной системы показан на рис.5.11.

К достоинствам мультисписков помимо экономии памяти (при множестве списков информационная часть существует в единственном экземпляре) следует отнести также целостность даннях — в том смысле, что все подзадачи работают с одной и той же версией информационной части и изменения в данных, сделанные одной подзадачей немедленно становятся доступными для другой подзадачи.

Каждая подзадача работает со своим подмножеством, как с линейным списком, используя для этого определенное поле связок.

Специфика мультисписка проявляется только в операции исключения элемента из списка. Исключение элемента из какого-либо одного списка еще не означает необходимости удаления элемента из памяти, так как элемент может оставаться в составе других списков. Память должна освобождаться только в том случае, когда элемент уже не входит ни в один из частных списков мультисписка. Обычно задача удаления упрощается тем, что один из частных списков является главным — в него обязательно входят все имеющиеся элементы. Тогда исключение элемента из любого неглавного списка состоит только в переопределении указателей, но не в освобождении памяти. Исключение же из главного списка требует не только освобождения памяти, но и переопределения указателей как в главном списке, так и во всех неглавных списках, в которые удаляемый элемент входил.

5.4 Нелинейные разветвленные списки

5.4.1 Основные понятия

Нелинейным разветвленным списком является список, элементами которого могут быть тоже списки. В разделе 5.2 были рассмотрены двухсвязные линейные списки. Если один из указателей каждого элемента списка задает порядок обратный к порядку, устанавливаемому другим указателем, то такой двусвязный список будет линейным. Если же один из указателей задает порядок произвольного вида, не являющийся обратным по отношению к порядку, устанавливаемому другим указателем, то такой список будет нелинейным.

В обработке нелинейный список определяется как любая последовательность атомов и списков (подсписков), где в качестве атома берется любой объект, который при обработке отличается от списка тем, что он структурно неделим.

Если мы заключим списки в круглые скобки, а элементы списков разделим запятыми, то в качестве списков можно рассматривать такие последовательности:

(a,(b,c,d),e,(f,g))
( )
((a))

Первый список содержит четыре элемента: атом a, список (b,c,d) (содержащий в свою очередь атомы b,c,d), атом e и список (f,g), элементами которого являются атомы f и g. Второй список не содержит элементов, тем не менее нулевой список, в соответствии с нашим определением является действительным списком. Третий список состоит из одного элемента: списка (a), который, в свою очередь, содержит атом a. Другой способ представления, часто используемый для иллюстрации списков, — графические схемы, аналогичен способу представления, применяемому при изображении линейных списков. Каждый элемент списка обозначается прямоугольником; стрелки или указатели показывают, являются ли прямоугольники элементами одного и того же списка или элементами подсписка. Пример такого представления дан на рис.5.12.

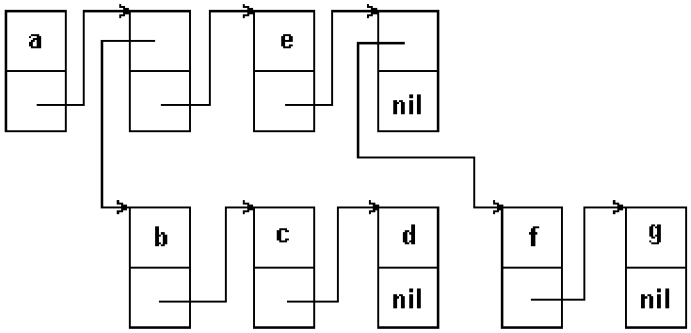


Рис.5.12. Схематичное представление разветвленного списка

Разветвленные списки описываются тремя характеристиками: порядком, глубиной и длиной.

Порядок. Над элементами списка задано транзитивное отношение, определяемое последовательностью, в которой элементы появляются внутри списка. В списке (x,y,z) атом x предшествует y, а y предшествует z. При этом подразумевается, что x предшествует z.

Данный список не эквивалентен списку (y,z,x). При представлении списков графическими схемами порядок определяется горизонтальными стрелками. Горизонтальные стрелки истолковываются следующим образом: элемент из которого исходит стрелка, предшествует элементу, на который она указывает.

Глубина. Это максимальный уровень, приписываемый элементам внутри списка или внутри любого подсписка в списке. Уровень элемента предписывается вложенностью подсписков внутри списка, т.е. числом пар круглых скобок, окаймляющих элемент. В списке, изображенном на рис.5.12), элементы a и e находятся на уровне 1, в то время как оставшиеся элементы - b, c, d, f и g имеют уровень 2. Глубина входного списка равна 2. При представлении списков схемами концепции глубины и уровня облегчаются для понимания, если каждому атомарному или списковому узлу приписать некоторое число l. Значение l для элемента x, обозначаемое как l(x), является числом вертикальных стрелок, которое необходимо пройти для того, чтобы достичь данный элемент из первого элемента списка. На рис.5.12 l(a)=0, l(b)=1 и т.д. Глубина списка является максимальным значением уровня среди уровней всех атомов списка.

Длина - число элементов уровня 1 в списке. Например, длина списка на рис.5.12 равна 3.

Типичный пример применения разветвленного списка — представление последнего алгебраического выражения в виде списка. Алгебраическое выражение можно представить в виде последовательности элементарных двухместных операций вида:

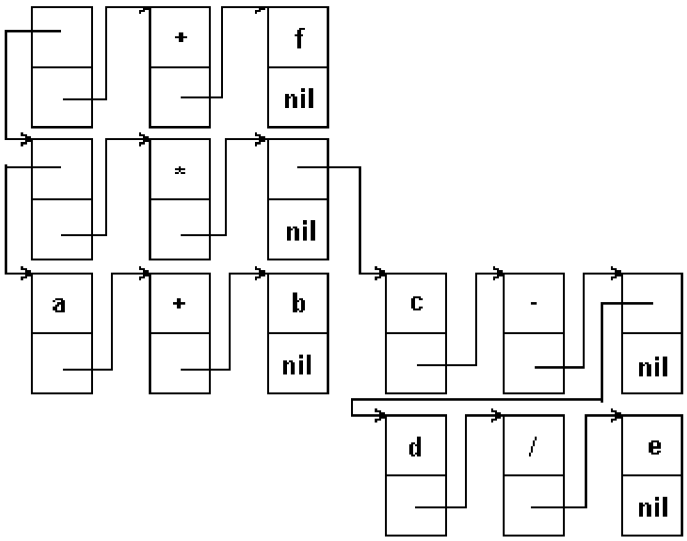


Рис.5.13. Схема списка, представляющего алгебраическое выражение

Выражение:

(a+b)*(c-(d/e))+f

будет вычисляться в следующем порядке:

a+b
d/e
c-(d/e)
(a+b)*(c-d/e)
(a+b)*(c-d/e)+f

При представлении выражения в виде разветвленного списка каждая тройка "операнд -знак-операнд" представляется в виде списка, причем, в качестве операндов могут выступать как атомы - переменные или константы, так и подсписки такого же вида. Скобочное представление нашего выражения будет иметь вид:

(((a,+,b),*,(c,-,(d,/,e)),+,f)

Глубина этого списка равна 4, длина - 3.

5.4.2 Представление списковых структур в памяти.

В соответствии со схематичным изображением разветвленных списков типичная структура элемента такого списка в памяти должна быть такой, как показано на рис.5.14.

data - данные атома	down - указатель на подсписок того же уровня	next - указатель на следующий элемент

Рис.5.14. Структура элемента разветвленного списка

Элементы списка могут быть двух видов: атомы — содержащие данные и узлы — содержащие указатели на подсписки. В атомах не используется поле down элемента списка, а в узлах — поле data. Поэтому

логичным является совмещение этих двух полей в одно, как показано на рис.5.15.

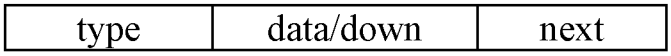


Рис.5.15. Структура элемента разветвленного списка

Поле type содержат признак атом/узел, оно может быть 1-битовым. Такой формат элемента удобен для списков, атомарная информация которых занимает небольшой объем памяти. В этом случае теряется незначительный объем памяти в элементах списка, для которых не требуется поля data. В более общем случае для атомарной информации необходим относительно большой объем памяти. Наиболее распространенный в данной ситуации формат структуры узла представленный на рис.5.16.

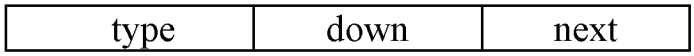
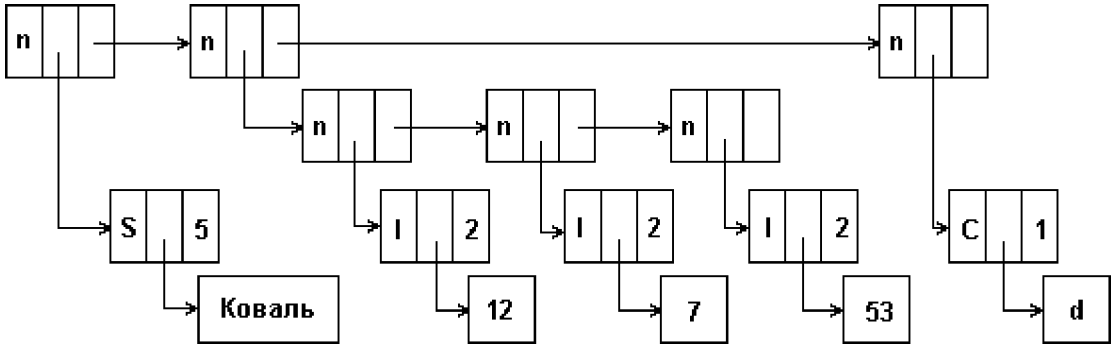


Рис. 5.16. Структура элемента разветвленного списка

В этом случае указатель down указывает на данные или на подсписок. Поскольку списки могут составляться из данных различных типов, целесообразно адресовать указателем down не непосредственно данные, а их дескриптор, в котором может быть описан тип данных, их длина и т.п. Само описание того, является ли адресуемый указателем данных объект атомом или узлом также может находиться в этом дескрипторе. Удобно сделать размер дескриптора данных таким же, как и элемента списка. В этом случае размер поля type может быть расширен, например, до 1 байта и это поле может индицировать не только атом/подсписок, но и тип атомарных данных, поле next в дескрипторе данных может использоваться для представления еще какой-то описательной информации, например, размера атома. На рис.5.17 показано представление элементами такого формата списка: (КОВАЛЬ,(12,7,53),d). Первая (верхняя) строка на рисунке представляет элементы списка, вторая - элементы подсписка, третья - дескрипторы данных, четвертая - сами данные. В поле type каждого элемента мы использовали коды: n - узел, S - атом, тип STRING, I - атом, тип INTEGER, C - атом, тип CHAR.



5.17. Пример представления списка элементами одного формата

5.4.3 Операции обработки списков

Базовыми операциями при обработке списков являются операции (функции): car, cdr, cons и atom.

Операция car в качестве аргумента получает список (указатель на начало списка). Ее возвращаемым значением является первый элемент этого списка (указатель на первый элемент). Например:

- если X - список (2,6,4,7), то car(X) - атом 2;
- если X - список ((1,2),6), то car(X) - список (1,2);
- если X - атом то car(X) не имеет смысла и в действительности не определено.

Операция cdr в качестве аргумента также получает список. Ее возвращаемым значением является остаток списка - указатель на список после удаления из него первого элемента. Например:

- если X - (2,6,4), то cdr(X) - (6,4);
- если X - ((1,2),6,5), то cdr(X) - (6,5);
- если список X содержит один элемент, то cdr(X) равно nil.

Операция cons имеет два аргумента: указатель на элемент списка и указатель на список. Операция включает аргумент-элемент в начало аргумента-списка и возвращает указатель на получившийся список. Например:

- если X - 2, а Y - (6,4,7), то cons(X,Y) - (2,6,4,7);
- если X - (1,2), Y - (6,4,7), то cons(X,Y) - ((1,2),6,4,7).

Операция atom выполняет проверку типа элемента списка. Она должна возвращать логическое значение: true - если ее аргумент является атомом или false - если ее аргумент является подсписком.

В программном примере 5.11 приведена реализация описанных операций как функций языка PASCAL. Структура элемента списка, обрабатываемого функциями этого модуля определена в нем как тип litem и полностью соответствует рис.5.16. Помимо описанных операций в модуле определены также функции выделения памяти для дескриптора данных - NewAtom и для элемента списка - NewNode. Реализация операций настолько проста, что не требует дополнительных пояснений.

```
{==== Программный пример 5.11 ====}
{ Элементарные операции для работы со списками }
Unit ListWork;
Interface
type lpt = ^litem; { указатель на элемент списка }
  litem = record
     typeflg : char; { Char(0) - узел, иначе - код типа }
     down : pointer; { указатель на данные или на подсписок }
     next: lpt;  { указатель на текущем уровне }
     end;
```

```
Function NewAtom(d: pointer; t : char) : lpt;
Function NewNode(d: lpt) : lpt;
Function Atom(l : lpt) : boolean;
Function Cdr(l : lpt) : lpt;
Function Car(l : lpt) : lpt;
Function Cons(l1, l : lpt) : lpt;
Function Append(l1,l : lpt) : lpt;
Implementation
{*** создание дескриптора для атома }
Function NewAtom(d: pointer; t : char) : lpt;
 var l : lpt;
  begin   New(l);
   l^.typeflg:=t; { тип данных атома }
   l^.down:=d;    { указатель на данные }
   l^.next:=nil;   NewAtom:=l;
  end;
{*** создание элемента списка для подсписка }
Function NewNode(d: lpt) : lpt;
 var l : lpt;
   begin
   New(l);
   l^.typeflg:=Chr(0); { признак подсписка }
   l^.down:=d;    { указатель на начало подсписка }
   l^.next:=nil;
   NewNode:=l;
   end;
{*** проверка элемента списка: true - атом, false - подсписок }
Function Atom(l : lpt) : boolean;
  begin   { проверка поля типа }
  if l^.typeflg=Chr(0) then Atom:=false
  else Atom:=true;
  end;
Function Car(l : lpt) : lpt;  {выборка 1-го элемента из списка }
  begin  Car:=l^.down; { выборка - указатель вниз }     end;
Function Cdr(l : lpt) : lpt;{исключение 1-го элемента из списка}
  begin   Cdr:=l^.next; { выборка - указатель вправо }   end;
{*** добавление элемента в начало списка }
Function Cons(l1,l : lpt) : lpt;
 var l2 : lpt;
  begin  l2:=NewNode(l1); { элемент списка для добавляемого }
  l2^.next:=l;           { в начало списка }
  Cons:=l2;              { возвращается новое начало списка }
  end;
{*** добавление элемента в конец списка }
Function Append(l1,l : lpt) : lpt;
```

```
 var l2, l3 : lpt;
  begin
  l2:=NewNode(l1);   { элемент списка для добавляемого }
  { если список пустой - он будет состоять из одного эл-та }
  if l=nil then Append:=l2
  else begin   { выход на последний эл-т списка }
    l3:=l; while l3^.next<>nil do l3:=l3^.next;
    l3^.next:=l2;    { подключение нового эл-та к последнему }
    Append:=l;       { функция возвращает тот же указатель }
    end;  end;
END.
```

В примере 5.11 в модуль базовых операций включена функция Append - добавления элемента в конец списка. На самом деле эта операция не является базовой, она может быть реализована с использованием описанных базовых операций, без обращения к внутренней структуре элемента списка, хотя, конечно, такая реализация будет менее быстродействующей. В программном примере 5.12 приведена реализация нескольких простых функций обработки списков, которые могут быть полезными при решении широкого спектра задач. В функциях этого модуля, однако, не используется внутренняя структура элемента списка.

```
{==== Программный пример 5.12 ====}
{ Вторичные функции обработки списков }
Unit ListW1;
Interface
uses listwork;
Function Append(x, l : lpt) : lpt;
Function ListRev(l, q : lpt) : lpt;
Function FlatList(l, q : lpt) : lpt;
Function InsList(x, l : lpt; m : integer) : lpt;
Function DelList(l : lpt; m : integer) : lpt;
Function ExchngList(l : lpt; m : integer) : lpt;
Implementation
{*** добавление в конец списка l нового элемента x }
Function Append(x, l : lpt) : lpt;
  begin
  { если список пустой - добавить x в начало пустого списка }
  if l=nil then Append:=cons(x,l)
  { если список непустой
    - взять тот же список без 1-го эл-та - cdr(l);
    - добавить в его конец эл-т x;
    - добавить в начало 1-й эл-т списка }
  else Append:=cons(car(l),Append(x,cdr(l)));
end; { Function Append }
```

```
{*** Реверс списка l; список q - результирующий, при первом
 вызове он должен быть пустым }
Function ListRev(l, q : lpt) : lpt;
  begin
  { если входной список исчерпан, вернуть выходной список }
  if l=nil then ListRev:=q
  { иначе: - добавить 1-й эл-т вх.списка в начало вых.списка,
    - реверсировать, имея вх. список без 1-го эл-та,
      а вых.список - с добавленным эл-том }
  else ListRev:=ListRev(cdr(l),cons(car(l),q));
end; { Function ListRev }
{*** Превращение разветвленного списка l в линейный; список q -
 результирующий, при первом вызове он должен быть пустым }
Function FlatList(l, q : lpt) : lpt;
  begin
  { если входной список исчерпан, вернуть выходной список }
  if l=nil then FlatList:=q
  else
   { если 1-й эл-т вх. списка - атом, то
    - сделать "плоской" часть вх. списка без 1-го эл-та;
    - добавить в ее начало 1-й эл-т }
   if atom(car(l)) then
     FlatList:=cons(car(l),FlatList(cdr(l),q))
   { если 1-й эл-т вх. списка - подсписок, то
    - сделать "плоской" часть вх.списка без 1-го эл-та;
    - сделать "плоским" подсписок 1-го эл-та }
   else FlatList:=FlatList(car(l),FlatList(cdr(l),q));
end; { Function FlatList }
{*** вставка в список l элемента x на место с номером m
 ( здесь и далее нумерация эл-тов в списке начинается с 0 ) }
Function InsList(x, l : lpt; m : integer) : lpt;
  begin
  { если m=0, эл-т вставляется в начало списка }
  if m=0 then InsList:=cons(x,l)
  { если список пустой, он и остается пустым }
  else if l=nil then InsList:=nil
  { - вставить эл-т x на место m-1 в список без 1-го эл-та;
    - в начало полученного списка вставить 1-й эл-т }
  else InsList:=cons(car(l),InsList(x,cdr(l),m-1));
end; { Function InsList }
{*** удаление из списка l на месте с номером m }
Function DelList(l : lpt; m : integer) : lpt;
  begin
  { если список пустой, он и остается пустым }
  if l=nil then DelList:=nil
```

```
  { если m=0, эл-т удаляется из начала списка }
  else if m=0 then DelList:=cdr(l)
  { - удалить эл-т x на месте m-1 в список без 1-го эл-та;
    - в начало полученного списка вставить 1-й эл-т }
  else DelList:=cons(car(l),DelList(cdr(l),m-1));
end; { Function DelList }
{*** перестановка в списке l эл-тов местах с номерами m и m+1 }
Function ExchngList(l : lpt; m : integer) : lpt;
  begin          { если список пустой, он и остается пустым }
  if l=nil then ExchngList:=nil
  else   if m=0 then
{если m=0, а следующего эл-та нет, список остается без изменений}
      if cdr(l)=nil then ExchngList:=l
{ если m=0 ( обмен 0-го и 1-го эл-тов):
- берется список без двух 1-ых эл-тов - cdr(cdr(l));
- в его начало добавляется 0-й эл-т;
- в начало полученного списка добавляется 1-й эл-т - car(cdr(l))}
    else ExchngList:= cons(car(cdr(l)),cons(car(l),cdr(cdr(l))))
   else ExchngList:=cons(car(l),ExchngList(cdr(l),m-1));
end; { Function ExchngList }
END.
```

Поскольку в функциях этого примера широко используются вложенные вызовы, в том числе и рекурсивные, в нижеследующих разборах описание каждого следующего вложенного вызова сдвигается вправо.

Функция Append добавляет элемент x в конец списка l. Рассмотрим ее выполнение на примере вызова: Append(4,(1,2,3)).

Поскольку аргумент-список не пустой, выполняется ветвь else. Она содержит оператор:

```
Append:=cons(car(l),Append(x,cdr(l)));
```

Важно точно представить себе последовательность действий по выполнению этого оператора:

- car(l) = 1
- cdr(l) = (2,3);
- Append(4,(2,3))) - при этом рекурсивном вызове выполнение вновь пойдет по ветви else, в которой:
  - car(l) = 2;
  - cdr(l) = (3);
  - Append(4,(3))) - выполнение вновь пойдет по ветви else, в которой:
    - car(l) = 3;
    - cdr(l) = nil;
    - Append(4,nil) - в этом вызове список-аргумент пустой, поэтому выполнится Append:=cons(4,nil) и вызов вернет список: (4);
    - cons(car(l),Append(x,cdr(l))) - значения аргументов функции

- cons для этого уровня вызовов: cons(3,(4))=(3,4);
- на этом уровне Append возвращает список (3,4);

- cons(car(l),Append(x,cdr(l))) - на этом уровне: cons(2,(3,4))=(2,3,4);
- на этом уровне Append возвращает список (2,3,4);

- cons(car(l),Append(x,cdr(l))) - на этом уровне: cons(1,(2,3,4)) = (1,2,3,4);
- на этом уровне Append возвращает список (1,2,3,4).

Функция ListRev выполняет инвертирование списка - изменения порядка следования его элементов на противоположный. При обращении к функции ее второй аргумент должен иметь значение nil. Пример: ListRev(1,(2,3),4),nil).

Входной список не пустой, поэтому выполнение идет по ветви else, где:

ListRev:=ListRev(cdr(l),cons(car(l),q));

Последовательность действий:

* cdr(l) = ((2,3),4);
* car(l) = 1;
* cons(car(l),q) = (1) - список q при этом - пустой;
  * рекурсивный вызов ListRev(((2,3),4), (1)):
  * cdr(l) = (4);
  * car(l) = (2,3);
  * cons(car(l),q) = ((2,3),1) - список q - (1);
  * рекурсивный вызов ListRev((4), ((2,3),1)):
    * cdr(l) = nil;
    * car(l) = 4;
    * cons(car(l),q) = (4,(2,3),1);
    * рекурсивный вызов ListRev(nil, (4,(2,3),1)):
      * поскольку исходный список пустой, вызов возвращает список: (4,(2,3),1);
    * вызов возвращает список: (4,(2,3),1);
  * вызов возвращает список: (4,(2,3),1);
* вызов возвращает список: (4,(2,3),1).

В программном примере 5.13 применение ветвящихся списков показано для решения более прикладной задачи. Представленная здесь программа - калькулятор, она вычисляет значение введенного арифметического выражения, составляющими которого могут быть целые числа, знаки четырех арифметических операций и круглые скобки.

Для упрощения примера мы ввели следующие ограничения:

- вся арифметика - целочисленная;
- программа не проверяет правильность исходной записи;
- в выражении не допускается унарный минус.

```
{==== Программный пример 5.13 ====}
{ Калькулятор. Вычисление арифметических выражений }
```

```
program Calc;
Uses ListWork;
type cptr = ^char;
     iptr = ^ integer;
const { цифровые символы }
     digits : set of char = ['0'..'9'];
     { знаки операций с высоким приоритетом }
     prty : set of char = ['*','/'];
var s : string;   { исходная строка }
    n : integer;  { номер текущего символа в исх. строке }
{*** Представление исходной строки в списочной форме }
Function Creat_Lst : lpt;
 var lll : lpt;   { указатель на начало текущего списка }
     s1 : char;   { текущий символ строки }
     st : string; { накопитель строки-операнда }
   {* Создание атома для Integer }
   Procedure NewInt;
   var ip : iptr; cc : integer;
     begin
     if Length(st)>0 then begin
       { если в st накоплено цифровое представление числа, оно
        переводится в тип integer, для него создается атом и
        записывается в конец списка }
       New(ip); Val(st,ip^,cc);
       lll:=Append(NewAtom(ip,'I'),lll);
       st:='';  { накопитель строки сбрасывается }
      end;  end; { Procedure NewInt }
   Procedure NewChar;      { Создание атома для Char }
   var cp : cptr;
     begin  { выделяется память для 1 символа, в  ней сохраняется
значение s1,для него создается атом, записывается в конец списка}
     New(cp); cp^:=s1;
     lll:=Append(NewAtom(cp,'C'),lll);
   end; { Procedure NewChar }
   begin  { Function Creat_Lst }
   { исходный список пустой, накопитель строки - пустой }
   lll:=nil;  st:='';
   while n<=length(s) do begin  { цикл до конца исходной строки }
     s1:=s[n]; n:=n+1;
     case s1 of
       '(' : { начало скобочного подвыражения: для него создается
          новый список - Creat_Lst, который оформляется как под-
          список - NewNode и добавляется в конец текущего
          списка - Append }
         lll:=Append(NewNode(Creat_Lst),lll);
```

```
    ')' : { конец скобочного выражения - последнее число в
      скобках добавляется в конец текущего списка и текущий
      список сформирован - конец функции }
      begin
        NewInt; Creat_Lst:=lll; Exit;
        end;
    else  {begin} { цифра или знак операции }
      if s1 in Digits then { цифры накапливаются в st }
        st:=st+s1
      else begin { знак операции }
        NewInt;  { созд. атом для ранее накопленного числа }
        NewChar; { созд. атом для знака }
     end;  {    end;}   end;  { case }   end;  { while }
  NewInt; { созд. атом для ранее накопленного числа }
  Creat_Lst:=lll;
end; { Function Creat_Lst }
{*** Выделение в подсписки высокоприоритетных операций }
Function FormPrty(l : lpt) : lpt;
 var op1, op, op2 : lpt; { 1-й операнд, знак, 2-й операнд }
    l2,l3 : lpt;
    cp: ^char;
  begin
  l2:=nil; { выходной список пустой }
  { выделение 1-го операнда }
  op1:=car(l);  l:=cdr(l);
  { если 1-й операнд - подсписок - обработка подсписка }
  if not atom(op1) then op1:=FormPrty(op1);
  while l<>nil do begin  { до опустошения исходного списка }
    { выделение знака операции }
    op:=car(l);  l:=cdr(l);
    { выделение 2-го операнда }
    op2:=car(l); l:=cdr(l);
    { если 2-й операнд - подсписок - обработка подсписка }
    if not atom(op2) then op2:=FormPrty(op2);
    if cptr(op^.down)^ in prty then
      { если знак операции приоритетный, то создается подспи-
       сок: 1-й операнд, знак, 2-й операнд, этот подсписок
       далее является 1-ым операндом }
      op1:=cons(op1,cons(op,cons(op2,nil)))
    else begin { если знак неприоритетный, 1-й операнд и знак
       записываются в выходной список, 2-й операнд далее
       является 1-ым операндом }
      l2:=Append(op,Append(op1,l2));
      op1:=op2;
   end;  end;
```

```
    FormPrty:=Append(op1,l2); { последний операнд добавляется в
      выходной список }
  end; { Function FormPrty }
  {*** Вычисление выражения }
  Function Eval(l : lpt) : integer;
  var op1, op, op2 : lpt;
   begin
    { выделение 1-го операнда }
    op1:=car(l);  l:=cdr(l);
    { если 1-й операнд - подсписок - вычислить его выражение }
    if not atom(op1) then iptr(op1^.down)^:=Eval(op1);
    while l<>nil do begin
      { выделение знака операции }
      op:=car(l);  l:=cdr(l);
      { выделение 2-го операнда }
      op2:=car(l); l:=cdr(l);
      { если 2-й операнд - подсписок - вычислить его выражение }
      if not atom(op2) then iptr(op2^.down)^:=Eval(op2);
      { выполнение операции, результат - в op1 }
      case cptr(op^.down)^ of
        '+' : iptr(op1^.down)^:=iptr(op1^.down)^+iptr(op2^.down)^;
        '-' : iptr(op1^.down)^:=iptr(op1^.down)^-iptr(op2^.down)^;
        '*' : iptr(op1^.down)^:=iptr(op1^.down)^*iptr(op2^.down)^;
        '/' : iptr(op1^.down)^:=iptr(op1^.down)^ div iptr(op2^.down)^;
        end;
      end;
     Eval:=iptr(op1^.down)^; { возврат последнего результата }
  end; { Function Eval }
  {*** Главная программа }
  var l : lpt;
    begin
    write('>'); readln(s);  { ввод исходной строки }
    { формирование списка }
    n:=1; l:=Creat_Lst;
    { выделение приоритетных операций }
    l:=FormPrty(l);
    { вычисление и печать результата }
    writeln(s,'=',Eval(l));
  END.
```

Выполнение программы состоит во вводе строки, представляющей исходное выражение и последовательных обращений к трем функциям: Creat_Lst, FormPrty и Eval.

Функция Creat_Lst преобразует исходную строку в список. В функции поэлементно анализируются символы строки. Различаемые

символы: левая круглая скобка, правая скобка, знаки операций и цифры. Цифровые символы накапливаются в промежуточной строке. Когда встречается символ-разделитель - правая скобка или знак операции накопленная строка преобразуется в число, для него создается атом с типом 'I' и включается в конец списка. Для знака операции создается атом с типом 'C' и тоже включается в конец списка. Левая скобка приводит к рекурсивному вызову Creat_Lst. Этот вызов формирует список для подвыражения в скобках, формирование списка заканчивается при появлении правой скобки. Для сформированного таким образом списка создается узел, и он включается в основной список как подсписок. Так, например, для исходной строки:

5+12/2-6*(11-7)+4

функцией Creat_Lst будет сформирован такой список:

(5,+,12,/,2,-,6,*,(11,-,7),+,4)

Следующая функция - FormPrty - выделяет в отдельные подсписки операции умножения и деления, имеющие более высокий приоритет, и их операнды. Функция просматривает список и выделяет в нем последовательные тройки элементов "операнд -знак-операнд". Если один из операндов является подсписком, то он обрабатывается функцией FormPrty. Если знак является одним из приоритетных знаков, то из тройки формируется подсписок, который становится первым операндом для следующей тройки. Если знак не приоритетный, то второй операнд тройки становится первым для следующей тройки. Список нашего примера после обработки его функцией FormPrty превратится в:

(5,+,(12,/,2),-,(6,*,(11,-,7)),+,4)

Наконец, функция Eval выполняет вычисления. Она во многом похожа на функцию FormPrty: в ней также выделяются тройки "операнд1-0знак-операнд". Если один или оба операнда являются подсписками, то сначала вычисляются эти подсписки и заменяются на атомы - результаты вычисления. Если оба операнда - атомы, то над ними выполняется арифметика, задаваемая знаком операции. Поскольку в первую очередь вычисляются подсписки, то подвыражения, обозначенные скобками в исходной строке, и операции умножения и деления выполняются в первую очередь. Для нашего примера порядок вычислений будет таков:

12 / 2 = 6; 5 + 6 = 11; 11 - 7 = 4; 6 * 4 = 24;
24 + 4 = 28; 11 - 28 = -17

5.5 Язык программирования LISP

LISP является наиболее развитым и распространенным языком обработки списков. "Идеология" и терминология этого языка в значительной степени повлияла на общепринятые подходы к обработке списков и использовалась и нами в предыдущем изложении. Все данные в LISP представляются в виде списков, структура элемента списка соответствует рис.5.15. LISP обеспечивает базовые функции обработки списков - car, cdr,

cons, atom. Также многие вторичные функции реализованы в языке как базовые - для повышения их эффективности. Помимо чисто списковых операций в языке обеспечиваются операции для выполнения арифметических, логических операций, отношения, присваивания, ввода-вывода и т.д. Операция cond обеспечивает ветвление.

Сама LISP-программа представляется как список, записанный в скобочной форме. Элементами простого программного списка является имя операции/функции и ее параметры. Параметрами могут быть в свою очередь обращения к функциям, которые образуют подсписки. Как правило, вся программа на LISP представляет собой единственное обращение к функции с множеством вложенных обращений - рекурсивных или к другим функциям. Поэтому программирование на языке LISP часто называют "функциональным программированием". Функции, приведенные нами в примере 5.11 являются переложением на язык PASCAL их LISP-реализаций.

Системы программирования LISP строятся и как компиляторы, и как интерпретаторы. Однако, независимо от подхода к построению системы программирования, она обязательно включает в себя "сборку мусора" (см. раздел 5.7). Обратите внимание на то, что в примере 5.11, представляя PASCAL-реализацию операций языка LISP, мы в некоторых функциях выделяли память, но нигде ее не освобождали. Система программирования LISP автоматически следит за использованием памяти и обеспечивает ее освобождение.

Другие языки обработки списков, например IPL-V, COMMIT в большей мере ориентированы на решение прикладных задач, а не на обработку абстрактных списков, хотя использование списковых структур заложено в основы в их реализации.

5.6 Управление динамически выделяемой памятью

Динамические структуры по определению характеризуется непостоянством и непредсказуемостью размера. Поэтому память под отдельные элементы таких структур выделяется в момент, когда они "начинают существовать" в процессе выполнения программы, а не во время трансляции. Когда в элементе структуры больше нет необходимости, занимаемая им память освобождается.

В современных вычислительных средах большая часть вопросов, связанных с управлением памятью решается операционными системами или системами программирования. Для программиста прикладных задач динамическое управление памятью либо вообще прозрачно, либо осуществляется через достаточно простой и удобный интерфейс стандартных процедур/функций. Однако, перед системным программистом вопросы управления памятью встают гораздо чаще. Во-первых, эти вопросы в полном объеме должны быть решены при проектировании операционных систем и систем программирования, во-вторых, некоторые сложные приложения могут сами распределять память в пределах

выделенного им ресурса, наконец в-третьих, знание того, как в данной вычислительной среде распределяется память, позволит программисту построить более эффективное программное изделие даже при использовании интерфейса стандартных процедур.

В общем случае при распределении памяти должны быть решены следующие вопросы:

- способ учета свободной памяти;
- дисциплины выделения памяти по запросу;
- обеспечение утилизации освобожденной памяти.

В распоряжении программы обычно имеется адресное пространство, которое может рассматриваться как последовательность ячеек памяти с адресами, линейно возрастающими от 0 до N. Какие-то части этого адресного пространства обычно заняты системными программами и данными, какие-то - кодами и статическими данными самой программы, оставшаяся часть доступна для динамического распределения. Обычно доступная для распределения память представляет собой непрерывный участок пространства с адресными границами от n1 до n2. В управлении памятью при каждом запросе на память необходимо решать, по каким адресам внутри доступного участка будет располагаться выделяемая память.

В некоторых системах программирования выделение памяти автоматизировано полностью: система не только сама определяет адрес выделяемой области памяти, но и определяет момент, когда память должна выделяться. Так, например, выделяется память под элементы списков в языке LISP, под символьные строки в языках SNOBOL и REXX. В других системах программирования - к ним относится большинство универсальных процедурных языков программирования - моменты выделения и освобождения памяти определяются программистом.

Программист должен выдать запрос на выделение/освобождение памяти при помощи стандартной процедуры/функции - ALLOCATE/FREE в PL/1, malloc/free в C, New/Dispose в PASCAL и т.п. Система сама определяет размещение выделяемого блока и функция выделения памяти возвращает его адрес. Наконец, в уже названных выше задачах системного программирования программист зачастую должен определить также и адрес выделяемой области.

Память всегда выделяется блоками - т.е. обязательно непрерывными последовательностями смежных ячеек. Блоки могут быть фиксированной или переменной длины. Фиксированный размер блока гораздо удобнее для управления: в этом случае вся доступная для распределения память разбивается на "кадры", размер каждого из которых равен размеру блока, и любой свободный кадр годится для удовлетворения любого запроса. К сожалению, лишь ограниченный круг реальных задач может быть сведен к блокам фиксированной длинны.

Одной из проблем, которые должны приниматься во внимание при управлении памятью является проблема фрагментации (дробления) памяти. Она заключается в возникновении "дыр" - участков памяти, которые не

могут быть использованы. Различаются дыры внутренние и внешние. Внутренняя дыра - неиспользуемая часть выделенного блока, она возникает, если размер выделенного блока больше запрошенного. Внутренние дыры характерны для выделения памяти блоками фиксированной длины. Внешняя дыра - свободный блок, который в принципе мог бы быть выделен, но размер его слишком мал для удовлетворения запроса. Внешние дыры характерны для выделения блоками переменной длины. Управление памятью должно быть построено таким образом, чтобы минимизировать суммарный объем дыр.

Система управления памятью должна прежде всего "знать", какие ячейки имеющейся в ее распоряжении памяти свободны, а какие - заняты. Методы учета свободной памяти основываются либо на принципе битовой карты, либо на принципе списков свободных блоков.

В методах битовой карты создается "карта" памяти - массив бит, в котором каждый однобитовый элемент соответствует единице доступной памяти и отражает ее состояние: 0 - свободна, 1 - занята. Если считать единицей распределения единицу адресации - байт, то сама карта памяти будет занимать 1/8 часть всей памяти, что делает ее слишком дорогостоящей. Поэтому при применении методов битовой карты обычно единицу распределения делают более крупной, например, 16 байт. Карта, таким образом, отражает состояние

каждого 16-байтного кадра. Карта может рассматриваться как строка бит, тогда поиск участка памяти для выделения выполняется как поиск в этой строке подстроки нулей требуемой длины.

В другой группе методов участки свободной памяти объединяются в связные списки. В системе имеется переменная, в которой хранится адрес первого свободного участка. В начале первого свободного участка записывается его размер и адрес следующего свободного участка. В простейшем случае список свободных блоков никак не упорядочивается. Поиск выполняется перебором списка.

Дисциплины выделения памяти решают вопрос: какой из свободных участков должен быть выделен по запросу. Выбор дисциплины распределения не зависит от способа учета свободной памяти. Две основные дисциплины сводятся к принципам "самый подходящий" и "первый подходящий". По дисциплине "самый подходящий" выделяется тот свободный участок, размер которого равен запрошенному или превышает его на минимальную величину. По дисциплине "первый подходящий" выделяется первый же найденный свободный участок, размер которого не меньше запрошенного. При применении любой дисциплины, если размер выбранного для выделения участка превышает запрос, выделяется запрошенный объем памяти, а остаток образует свободный блок меньшего размера. В некоторых системах вводится ограничение на минимальный размер свободного блока: если размер остатка меньше некоторого граничного значения, то весь свободный блок выделяется по запросу без остатка. Практически во всех случаях дисциплина "первый подходящий"

эффективнее дисциплины "самый подходящий". Это объясняется во-первых, тем, что при поиске первого подходящего не требуется просмотр всего списка или карты до конца, во-вторых, тем, что при выборе всякий раз "самого подходящего" остается больше свободных блоков маленького размера - внешних дыр.

Когда в динамической структуре данных или в отдельном ее элементе нет больше необходимости, занимаемая ею память должна быть утилизована, т.е. освобождена и сделана доступной для нового распределения. В тех системах, где память запрашивается программистом явным образом, она и освобождена должна быть явным образом. Даже в некоторых системах, где память выделяется автоматически, она освобождается явным образом (например, операция DROP в языке REXX). В таких системах, конечно, задача утилизации решается просто. При представлении памяти на битовой карте достаточно просто сбросить в 0 биты, соответствующие освобожденным кадрам.

При учете свободной памяти списками блоков освобожденный участок должен быть включен в список, но одного этого недостаточно. Следует еще позаботиться о том, чтобы при образовании в памяти двух смежных свободных блоков они слились в один свободный блок суммарного размера. Задача слияния смежных блоков значительно упрощается при упорядочении списка свободных блоков по адресам памяти - тогда смежные блоки обязательно будут соседними элементами этого списка.

Задача утилизации значительно усложняется в системах, где нет явного освобождения памяти: тогда на систему ложится задача определения того, какие динамические структуры или их элементы уже не нужны программисту. Один из методов решения этой задачи предполагает, что система не приступает к освобождению памяти до тех пор, пока свободной памяти совсем не останется. Затем все зарезервированные блоки проверяются и освобождаются те из них, которые больше не используются. Такой метод называется "сборкой мусора". Программа, сборки мусора вызывается тогда, когда нет возможности удовлетворить некоторый частный запрос на память, или когда размер доступной области памяти стал меньше некоторой заранее определенной границы. Алгоритм сборки мусора обычно бывает двухэтапным. На первом этапе осуществляется маркировка (пометка) всех блоков, на которые указывает хотя бы один указатель. На втором этапе все неотмеченные блоки возвращаются в свободный список, а метки стираются. Важно, чтобы в момент включения сборщика мусора все указатели были установлены на те блоки, на которые они должны указывать. Если необходимо в некоторых алгоритмах применять методы с временным рассогласованием указателей, необходимо временно отключить сборщик мусора - пока имеется такое рассогласование. Один из самых серьезных недостатков метода сборки мусора состоит в том, что расходы на него увеличиваются по мере уменьшения размеров свободной области памяти.

Другой метод - освобождать любой блок, как только он перестает использоваться. Он обычно реализуется посредством счетчиков ссылок - счетчиков, в которых записывается, сколько указателей на данный блок имеется в данный момент времени. Когда значение счетчика становится равным 0, соответствующий блок оказывается недоступным и, следовательно, не используемым. Блок возвращается в свободный список. Такой метод предотвращает накопление мусора, не требует большого числа оперативных проверок во время обработки данных. Однако и у этого метода есть определенные недостатки. Во-первых, если зарезервированные блоки образуют циклическую структуру, то счетчик ссылок каждого из них не равен 0, когда все связи, идущие извне блоков в циклическую структуру, будут уничтожены. Это приводит к появлению мусора. Существуют различные возможности устранить этот недостаток: запретить циклические и рекурсивные структуры; отмечать циклические структуры флажками, и обрабатывать их особым образом; потребовать, чтобы любая циклическая структура всегда имела головной блок, счетчик циклов которого учитывал бы только ссылки от элементов, расположенных вне цикла, и чтобы доступ ко всем блокам этой структуры осуществлялся только через него. Во-вторых, требуются лишние затраты времен и памяти на ведение счетчиков ссылок.

В некоторых случаях может быть полезен метод восстановления ранее зарезервированной памяти, называемый уплотнением. Уплотнение осуществляется путем физического передвижения блоков данных с целью сбора всех свободных блоков в один большой блок. Преимущество этого метода в том, что после его применения выделение памяти по запросам упрощается. Единственная серьезная проблема, возникающая при использовании метода - переопределение указателей. Механизм уплотнения использует несколько просмотров памяти.

Сначала определяются новые адреса всех используемых блоков, которые были отмечены в предыдущем проходе, а затем во время следующего просмотра памяти все указатели, связанные с отмеченными блоками, переопределяются. После этого отмеченные блоки переставляются. Механизма освобождения памяти в методе восстановления совсем нет. Вместо него используется механизм маркировки, который отмечает блоки, используемые в данный момент. Затем, вместо того, чтобы освобождать каждый не отмеченный блок путем введения в действие механизма освобождения памяти, помещающего этот блок в свободный список, используется уплотнитель, который собирает неотмеченные блоки в один большой блок в одном конце области памяти. Недостаток метода в том, что из-за трех просмотров памяти велики затраты времени. Однако повышенная скорость резервирования в определенных условиях может компенсировать этот недостаток.

Практическая эффективность методов зависит от многих параметров, таких как частота запросов, статистическое распределение размеров запрашиваемых блоков, способ использования системы - групповая

обработка или стратегия обслуживания при управлении вычислительным центром.

6 НЕЛИНЕЙНЫЕ СТРУКТУРЫ ДАННЫХ

6.1 Графы

6.1.1 Логическая структура, определения

Граф — это сложная нелинейная многосвязная динамическая структура, отображающая свойства и связи сложного объекта.

Многосвязная структура обладает следующими свойствами:

1). на каждый элемент (узел, вершину) может быть произвольное количество ссылок;

2). каждый элемент может иметь связь с любым количеством других элементов;

3) каждая связка (ребро, дуга) может иметь направление и вес.

В узлах графа содержится информация об элементах объекта. Связи между узлами задаются ребрами графа. Ребра графа могут иметь направленность, показываемую стрелками, тогда они называются ориентированными, ребра без стрелок - неориентированные.

Граф, все связи которого ориентированные, называется ориентированным графом или орграфом; граф со всеми неориентированными связями - неориентированным графом; граф со связями обоих типов - смешанным графом. Обозначение связей: неориентированных - (A,B), ориентированных - <A,B>. Примеры изображений графов даны на рис.6.1. Скобочное представление графов рис.6.1: а).((A,B),(B,A)) и б).(<A,B>,<B,A>).

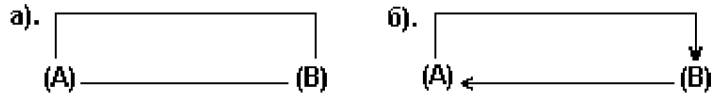


Рис.6.1. Граф неориентированный (а) и ориентированный (б).

Для ориентированного графа число ребер, входящих в узел, называется полустепенью захода узла, выходящих из узла - полустепенью исхода. Количество входящих и выходящих ребер может быть любым, в том числе и нулевым. Граф без ребер является нуль-графом.

Если ребрам графа соответствуют некоторые значения, то граф и ребра называются взвешенными. Мультиграфом называется граф, имеющий параллельные (соединяющие одни и те же вершины) ребра, в противном случае граф называется простым.

Путь в графе - это последовательность узлов, связанных ребрами; элементарным называется путь, в котором все ребра различны, простым называется путь, в котором все вершины различны. Путь от узла к самому себе называется циклом, а граф, содержащий такие пути - циклическим.

Два узла графа смежны, если существует путь от одного из них до другого. Узел называется инцидентным к ребру, если он является его вершиной, т.е. ребро направлено к этому узлу.

Логически структура-граф может быть представлена матрицей смежности или матрицей инцидентности.

Матрицей смежности для n узлов называется квадратная матрица adj порядка n. Элемент матрицы a(i,j) равен 1, если узел j смежен с узлом i (есть путь <i,j>), и 0 -в противном случае (рис.6.2).

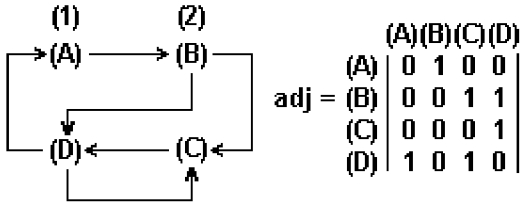


Рис.6.2. Граф и его матрица смежности

Если граф неориентирован, то a(i,j)=a(j,i), т.е. матрица симметрична относительно главной диагонали.

Матрицы смежности используются при построении матриц путей, дающих представление о графе по длине пути: путь длиной в 1 - смежный участок - <A,B>, путь длиной 2 - (<A,B>,<B,C>), ... в n смежных участков: где n - максимальная длина, равная числу узлов графа. На рис.6.3 даны путевые матрицы пути adj2, adj3, adj4 для графа рис.6.2.

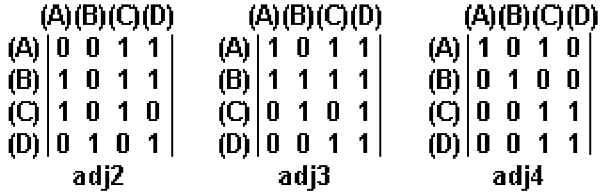


Рис.6.3. Матрицы путей

Матрицы инцидентности используются только для орграфов. В каждой строке содержится упорядоченная последовательность имен узлов, с которыми данный узел связан ориетрированными (исходящими) ребрами. На рис.6.4 показана матрица инцидентности для графа рис. 6.2.

узлы \ номера связей	1	2
A	B	-
B	C	D
C	D	-
D	A	C

Рис.6.4. Матрица инцидентности

6.1.2 Машинное представление орграфов

Существуют два основных метода представления графов в памяти ЭВМ: матричный, т.е. массивами, и связными нелинейными списками. Выбор метода представления зависит от природы данных и операций, выполняемых над ними. Если задача требует большого числа включений и

исключений узлов, то целесообразно представлять граф связными списками; в противном случае можно применить и матричное представление.

МАТРИЧНОЕ ПРЕДСТАВЛЕНИЕ ОРГРАФОВ. При использовании матриц смежности их элементы представляются в памяти ЭВМ элементами массива. При этом, для простого графа матрица состоит из нулей и единиц, для мультиграфа - из нулей и целых чисел, указывающих кратность соответствующих ребер, для взвешенного графа - из нулей и вещественных чисел, задающих вес каждого ребра.

Например, для простого ориентированного графа, изображенного на рис.6.2 массив определяется как:

mas:array[1..4,1..4]=((0,1,0,0),(0,0,1,1),(0,0,0,1),(1,0,1,0))

Матрицы смежности применяются, когда в графе много связей и матрица хорошо заполнена.

СВЯЗНОЕ ПРЕДСТАВЛЕНИЕ ОРГРАФОВ. Орграф представляется связным нелинейным списком, если он часто изменяется или если полустепени захода и исхода его узлов велики. Рассмотрим два варианты представления орграфов связными нелинейными списковыми структурами.

В первом варианте два типа элементов - атомарный и узел связи (см. раздел 5.5). На рис.6.5 показана схема такого представления для графа рис.6.2. Скобочная запись связей этого графа:

(<A,B>, <B,C>, <C,D>, <B,D>, <D,C>)

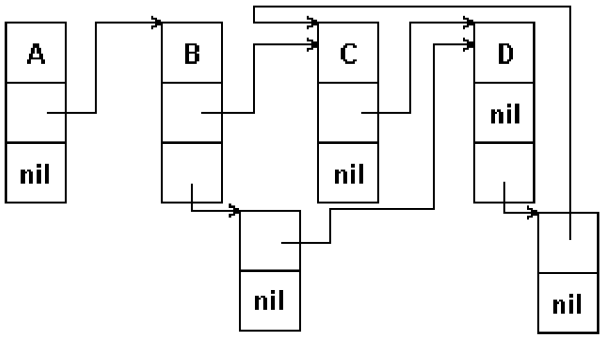


Рис.6.5. Машинное представление графа элементами двух типов

Более рационально представлять граф элементами одного формата, двойными: атом-указатель и указатель-указатель или тройными: указатель - data/down - указатель (см.раздел 5.5). На рис.6.6 тот же граф представлен элементами одного формата.

Многосвязная структура - граф - находит широкое применение при организации банков данных, управлении базами данных, в системах программного иммитационного моделирования сложных комплексов, в системах исскуственного интеллекта, в задачах планирования и в других сферах. Алгоритмы обработки нелинейных разветвленных списков, к которым могут быть отнесены и графы, даны в разделе 5.5.

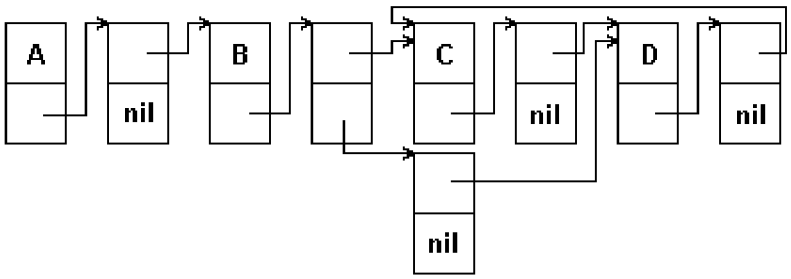
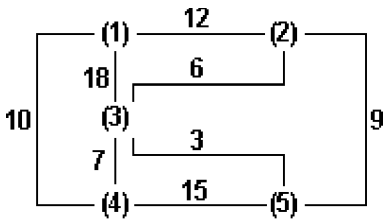


Рис.6.6. Машинное представление графа однотипными элементами

В качестве примера приведем программу, находящую кратчайший путь между двумя указанными вершинами связного конечного графа.

Пусть дана часть карты дорожной сети и нужно найти наилучший маршрут от города 1 до города 5. Такая задача выглядит достаточно простой, но "наилучший" маршрут могут определять многие факторы. Например: (1) расстояние в километрах; (2) время прохождения маршрута с учетом ограничений скорости; (3) ожидаемая продолжительность поездки с учетом дорожных условий и плотности движения; (4) задержки, вызванные проездом через города или объездом городов; (5) число городов, которое необходимо посетить, например, в целях доставки грузов. Задачи о кратчайших путях относятся к фундаментальным задачам комбинаторной оптимизации.

Среди десятков алгоритмов для отыскания кратчайшего пути один из лучших принадлежит Дейкстре. Алгоритм Дейкстры, определяющий кратчайшее расстояние от данной вершины до конечной, легче пояснить на примере. Рассмотрим граф, изображенный на рис.6.7, задающий связь между городами на карте дорог. Представим граф матрицей смежности A, в которой: A(i,j)-длина ребра между узлами i и j. Используя полученную матрицу и матрицы, отражающие другие факторы, можно определить кратчайший путь.



A =

	(1)	(2)	(3)	(4)	(5)	
	0	12	18	10	-	(1)
	12	0	6	-	9	(2)
	18	6	0	7	3	(3)
	10	-	7	0	15	(4)
	-	9	3	15	0	(5)

"-" - ребро отсутствует

Рис.6.7. Часть дорожной карты, представленная в виде взвешенного графа и его матрицы смежности

```
{========== Программный пример 6.1 ==============}
{ Алгоритм Дейкстры }
Program ShortWay;
Const n=5;   max=10000;
Var a: Array [1..n,1..n] of Integer;
   v0,w,edges: Integer;
   from,tu,length: Array [1..n] of Integer;
```

```
Procedure adjinit;
{ Эта процедура задает веса ребер графа  посредством  определения
его матрицы смежности A размером N x N }
Var i,j: Integer;
Begin
{ "Обнуление" матрицы (вершины не связаны) }
 For i:=1 to n do
  For j:=1 to n do a[i,j]:=max;
 For i:=1 to n do a[i,i]:=0;
{ Задание длин ребер, соединяющих смежные узлы графа }
 a[1,2]:=12;  a[1,3]:=18; a[1,4]:=10;
 a[2,1]:=12;  a[2,3]:=6;   a[2,5]:=9;
 a[3,1]:=18;  a[3,2]:=6;   a[3,4]:=7;  a[3,5]:=3;
 a[4,1]:=10;  a[4,3]:=7;   a[4,5]:=15;
 a[5,2]:=9;    a[5,3]:=3;   a[5,4]:=15;
End;
Procedure printmat;
{ Эта  процедура  выводит  на  экран  дисплея матрицу смежности A
взвешенного графа }
Var i,j: Integer;
Begin  writeln;
 writeln('Матрица смежности взвешенного графа (',n,'x',n,'):');
 writeln;
 For i:=1 to n do
 Begin  write ('¦');
   For j:=1 to n do
    If a[i,j]=max Then write('  ----')  Else write(a[i,j]:6);
   writeln(' ¦')
  End; writeln;
 writeln (' ("----" - ребро отсутствует)')
End;
```

Procedure dijkst;

{ Эта процедура определяет кратчайшее расстояние от начальной вершины V0 до конечной вершины W в связном графе с неотрицательными весами с помощью алгоритма, принадлежащего Дейкстре. Результатом работы этой процедуры является дерево кратчайших путей с корнем V0.

---- Входные и выходные переменные ---

A(I,J)	длина ребра, соединяющего вершины I и J. Если ребро отсутствует, то A(I,J) = 10000 (произвольному большому числу).
V0	начальная вершина.
W	конечная вершина.
N	вершины в графе пронумерованы 1,...,N.
FROM(I)	содержит I-е ребро в дереве кратчайших путей от вершины
TU(I)	FROM(I) к вершине TU(I)

```
LENGTH(I)   длины LENGTH(I).
   EDGES    число ребер в дереве кратчайших путей на данный момент.
--- Внутренние переменные ---
DIST(I)     кратчайшее расстояние от UNDET(I) до частичного дерева
            кратчайших путей.
NEXT        очередная вершина, добавляемая к дереву кратчайших путей.
NUMUN       число неопределенных вершин.
UNDET(I)    список неопределенных вершин.
VERTEX(I)   вершины частичного дерева кратчайших путей, лежащие на
            кратчайшем пути от UNDET(I) до V0. 1
}
Label stpoint;
Var dist,undet,vertex: array[1..n] of Integer;
   next,numun,i,j,k,l,jk: Integer;
Begin
 edges:=0; next:=v0; numun:=n-1;
 For i:=1 to n do
  Begin  undet[i]:=i; dist[i]:=a[v0,i]; vertex[i]:=v0 End;
 undet[v0]:=n; dist[v0]:=dist[n];
 goto stpoint;
 Repeat
{ Исключение вновь определенной вершины из списка неопределенных}
  dist[k]:=dist[numun]; undet[k]:=undet[numun];
  vertex[k]:=vertex[numun];
{ Остались ли неопределеные вершины ? }
  dec(numun);
{ Обновление кратчайш. расстояния до всех неопределенных вершин }
  For i:=1 to numun do
  Begin   j:=undet[i]; jk:=l+a[next,j];
     If dist[i]>jk Then Begin  vertex[i]:=next; dist[i]:=jk  End
  End;
stpoint:{Запоминание кратчайшего расст.до неопределенной вершины}
  k:=1; l:=dist[1];
  For i:=1 to numun do
   If dist[i]<l Then Begin  l:=dist[i]; k:=i  End;
{ Добавление ребра к дереву кратчайших путей }
  inc(edges);  from[edges]:=vertex[k]; tu[edges]:=undet[k];
  length[edges]:=l; next:=undet[k]
 Until next = w        { Достигли ли мы w }
End;
Procedure showway;
{ Эта процедура выводит на экран  дисплея  кратчайшее расстояние
между вершинами V0 и W взвешенного графа, определенное
процедурой dijkst }
Var i: Integer;
```

```
Begin
 writeln;  writeln('Кратчайшее расстояние между');
 writeln('узлами ',v0,' и ',w,' равно ',length[edges])
End;
{ Основная программа }
Begin
 adjinit;  printmat;   v0:=1;w:=5;
 dijkst;   showway;    readln
End.
```

6.2 Деревья

6.2.1 Основные определения

Дерево - это граф, который характеризуется следующими свойствами:

1. Существует единственный элемент (узел или вершина), на который не ссылается никакой другой элемент - и который называется КОРНЕМ (рис. 6.8 - A,G,M - корни).
2. Начиная с корня и следуя по определенной цепочке указателей, содержащихся в элементах, можно осуществить доступ к любому элементу структуры.
3. На каждый элемент, кроме корня, имеется единственная ссылка, т.е. каждый элемент адресуется единственным указателем.

Название "дерево" проистекает из логической эквивалентности древовидной структуры абстрактному дереву в теории графов. Линия связи между парой узлов дерева называется обычно ВЕТВЬЮ. Те узлы, которые не ссылаются ни на какие другие узлы дерева, называются ЛИСТЬЯМИ (или терминальными вершинами) (рис. 6.8- b,k,l,h - листья). Узел, не являющийся листом или корнем, считается промежуточным или узлом ветвления (нетерминальной или внутренней вершиной).

Для ориентированного графа число ребер, исходящих из некоторой начальной вершины V, называется ПОЛУСТЕПЕНЬЮ ИСХОДА этой вершины. Число ребер, для которых вершина V является конечной, называется ПОЛУСТЕПЕНЬЮ ЗАХОДА вершины V, а сумма полустепеней исхода и захода вершины V называется ПОЛНОЙ СТЕПЕНЬЮ этой вершины.

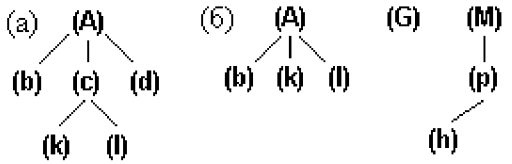


Рис. 6.8. (а) дерево; (б). лес

Ниже будет представлен важный класс орграфов - ориентированные деревья - и соответствующая им терминология. Деревья нужны для описания любой структуры с иерархией. Традиционные примеры таких

структур: генеалогические деревья, десятичная классификация книг в библиотеках, иерархия должностей в организации, алгебраическое выражение, включающее операции, для которых предписаны определенные правила приоритета.

Ориентированное дерево - это такой ациклический орграф (ориентированный граф), у которого одна вершина, называемая корнем, имеет полустепень захода, равную 0, а остальные - полустепени захода, равные 1. Ориентированное дерево должно иметь по крайней мере одну вершину. Изолированная вершина также представляет собой ориентированное дерево.

Вершина ориентированного дерева, полустепень исхода которой равна нулю, называется КОНЦЕВОЙ (ВИСЯЧЕЙ) вершиной или ЛИСТОМ; все остальные вершины дерева называют вершинами ветвления. Длина пути от корня до некоторой вершины называется УРОВНЕМ (НОМЕРОМ ЯРУСА) этой вершины. Уровень корня ориентированного дерева равен нулю, а уровень любой другой вершины равен расстоянию (т.е. модулю разности номеров уровней вершин) между этой вершиной и корнем. Ориентированное дерево является ациклическим графом, все пути в нем элементарны.

Во многих приложениях относительный порядок следования вершин на каждом отдельном ярусе имеет определенное значение. При представлении дерева в ЭВМ такой порядок вводится автоматически, даже если он сам по себе произволен. Порядок следования вершин на некотором ярусе можно легко ввести, помечая одну вершину как первую, другую - как вторую и т.д. Вместо упорядочивания вершин можно задавать порядок на ребрах. Если в ориентированном дереве на каждом ярусе задан порядок следования вершин, то такое дерево называется УПОРЯДОЧЕННЫМ ДЕРЕВОМ.

Введем еще некоторые понятия, связанные с деревьями. Рассмотрим дерево, показаное на рис.6.8(а)

Узел с называется ПРЕДКОМ (или ОТЦОМ), а узлы k и l называются НАСЛЕДНИКАМИ (или СЫНОВЬЯМИ) их соответственно между собой называют БРАТЬЯМИ. Причем левый сын является старшим сыном, а правый - младшим.

Число поддеревьев данной вершины называется СТЕПЕНЬЮ этой вершины. (В данном примере с имеет 2 поддерева, следовательно СТЕПЕНЬ вершины с равна 2).

Если из дерева убрать корень и ребра, соединяющие корень с вершинами первого яруса, то получится некоторое множество несвязанных деревьев. Множество несвязанных деревьев называется ЛЕСОМ (рис. 6.8(b)).

6.2.2 Логическое представление и изображение деревьев.

Имеется ряд способов графического изображения деревьев. Первый способ заключается в использовании для изображения поддеревьев известного метода диаграмм Венна, второй - метода вкладывающихся друг в

друга скобок, третий способ - это способ, применяемый при составлении оглавлений книг. Последний способ, базирующийся на формате с нумерацией уровней, сходен с методами, используемыми в языках программирования. При применении этого формата каждой вершине приписывается числовой номер, который должен быть меньше номеров, приписанных корневым вершинам присоединенных к ней поддеревьев. Отметим, что корневые вершины всех поддереьев данной вершины должны иметь один и тот же номер.

МЕТОД ВЛОЖЕННЫХ СКОБОК
(V0(V1(V2(V5)(V6))(V3)(V4))(V7(V8)(V9(V10))))

Все эти представления демонстрируют одну и ту же структуру и поэтому эквивалентны. С помощью графа можно наглядно представить разветвляющиеся связи, которые по понятным причинам привели к общеупотребительному термину "дерево".

6.2.3 Бинарные деревья.

Существуют m-арные деревья, т.е. такие деревья у которых полустепень исхода каждой вершины меньше или равна m (где m может быть равно 0,1,2,3 и т.д.). Если полустепень исхода каждой вершины в точности равна либо m, либо нулю, то такое дерево называется ПОЛНЫМ m-АРНЫМ ДЕРЕВОМ.

При m=2 такие деревья называются соответственно БИНАРНЫМИ, или ПОЛНЫМИ БИНАРНЫМИ.

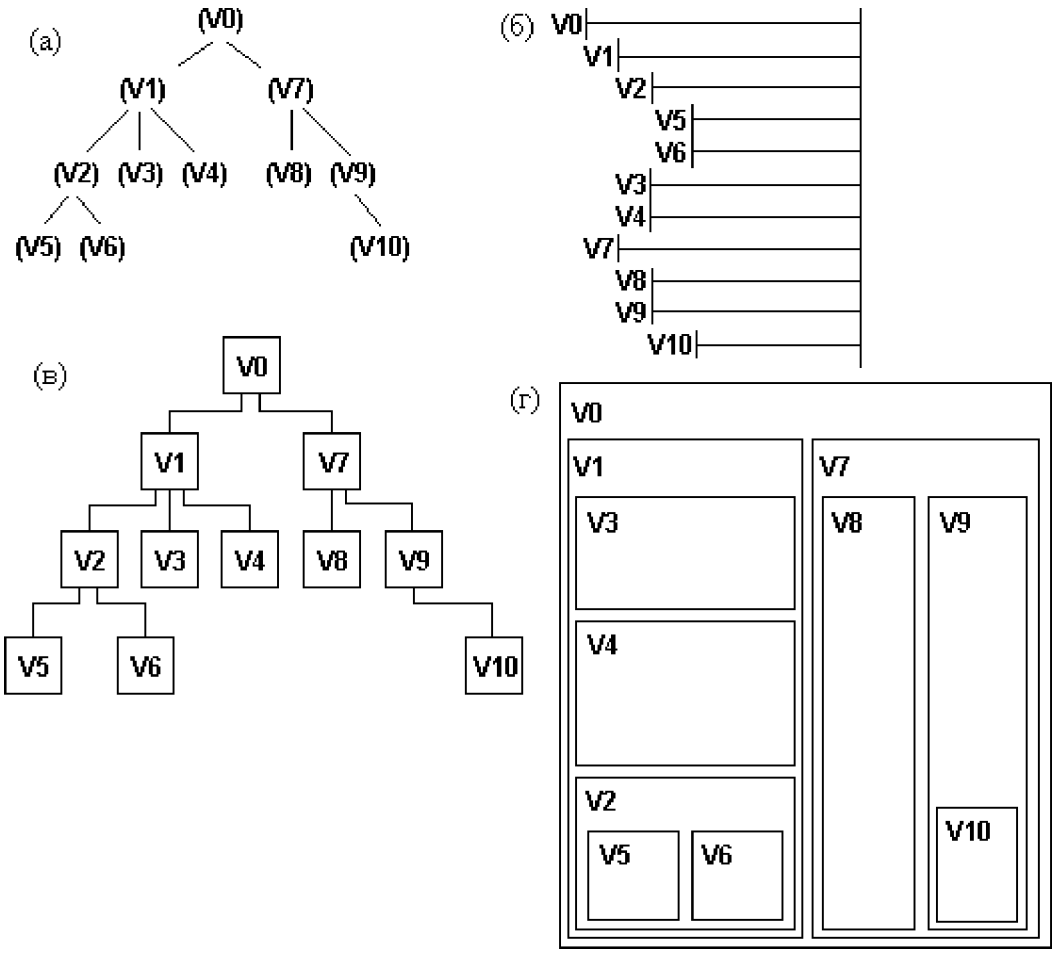


Рис.6.9. Представление дерева : а)- исходное дерево,
б)- оглавление книг, в)- граф, г)- диаграмма Венна

На рисунке 6.10(а) изображено бинарное дерево, 6.10(б)- полное бинарное дерево, а на 6.10(в) показаны все четыре возможных расположения сыновей некоторой вершины бинарного дерева.

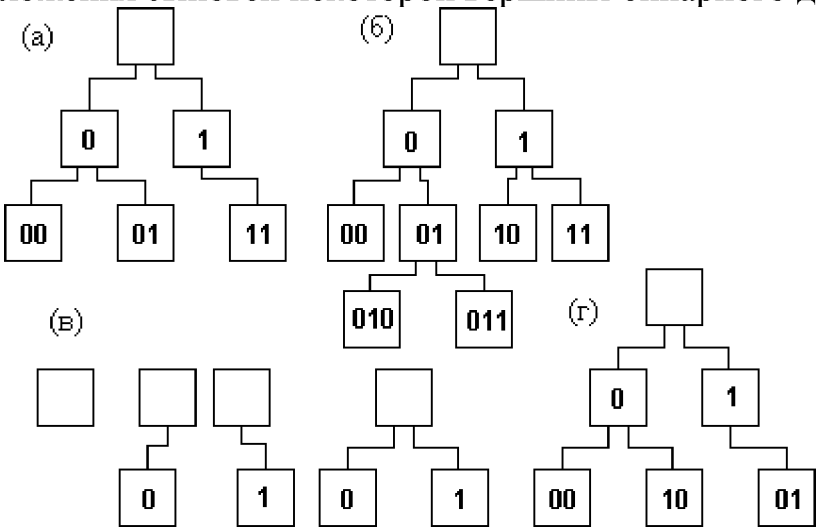


Рис. 6.10. Изображения бинарных деревьев

Бинарные деревья, изображенные на рис.6.10(а) и 6.10(г), представляют собой разные позиционные деревья, хотя они не являются разными упорядоченными деревьями.

В позиционном бинарном дереве каждая вершина представлена единственным образом посредством строки символов над алфавитом {0,1}, при этом корень характеризуется пустой строкой. Любой сын вершины "u" характеризуется строкой, префикс (начальная часть) которой является строкой, характеризующей "u".

Примером бинарного дерева является фамильное дерево с отцом и матерью человека в качестве его потомков. Еще один пример - это арифметическое выражение с двухместными операциями, где каждая операция представляет собой ветвящийся узел с операндами в качестве поддеревьев.

Представить m-арное дерево в памяти ЭВМ сложно, т.к. каждый элемент дерева должен содержать столько указателей, сколько ребер выходит из узла (при m-3,4.5.6... соответствует 3,4,5,6... указателей). Это приведет к повышенному расходу памяти ЭВМ, разнообразию исходных элементов и усложнит алгоритмы обработки дерева. Поэтому m-арные деревья, лес необходимо привести к бинарным для экономии памяти и упрощению алгоритмов. Все узлы бинарного дерева представляются в памяти ЭВМ однотипными элементами с двумя указателями (см.разд. 6,2,5), кроме того, операции над двоичными деревьями выполняются просто и эффективно.

6.2.4 Представление любого дерева, леса бинарными деревьями.

Дерево и лес любого вида можно преобразовать единственным образом в эквивалентное бинарное дерево.

Правило построения бинарного дерева из любого дерева:

1. В каждом узле оставить только ветвь к старшему сыну (вертикальное соединение);
2. Соединить горизонтальными ребрами всех братьев одного отца;
3. Таким образом перестроить дерево по правилу: левый сын - вершина, расположенная под данной; правый сын - вершина, расположенная справа от данной (т.е. на ярусе с ней).
4. Развернуть дерево таким образом, чтобы все вертикальные ветви отображали левых сыновей, а горизонтальные - правых.

В результате преобразования любого дерева в бинарное получается дерево в виде левого поддерева, подвешенного к корневой вершине.

В процессе преобразования правый указатель каждого узла бинарного дерева будет указывать на соседа по уровню. Если такового нет, то правый указатель NIL. Левый указатель будет указывать на вершину следующего уровня. Если таковой нет, то указатель устанавливается на NIL.

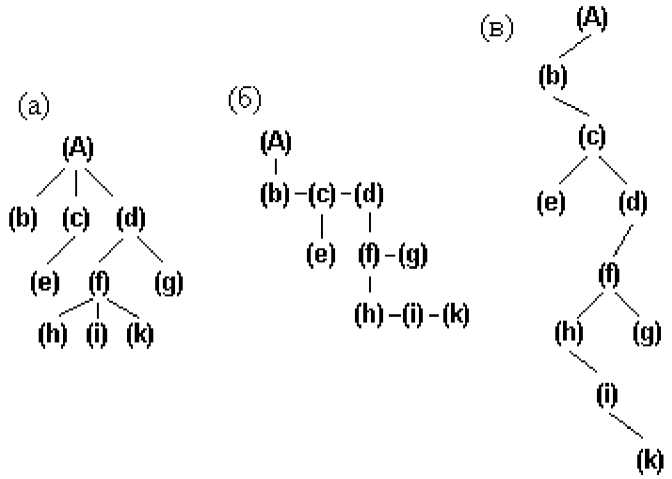


Рис.6.11. а). исходное дерево; б). промежуточный результат перестройки дерева; в). представление дерева в виде бинарного

Описанный выше метод представления произвольных упорядоченных деревьев посредством бинарных деревьев можно обобщить на представление произвольного упорядоченного леса.

Правило построения бинарного дерева из леса: корни всех поддеревьев леса соединить горизонтальными связями. В полученном дереве узлы в данном примере будут располагаться на трех уровнях. Далее перестраивать по ранее рассмотренному плану: в начале поддерево с корнем А, затем В и затем Н. В результате преобразования упорядоченного леса в бинарное дерево получается полное бинарное дерево с левым и правым поддеревом.

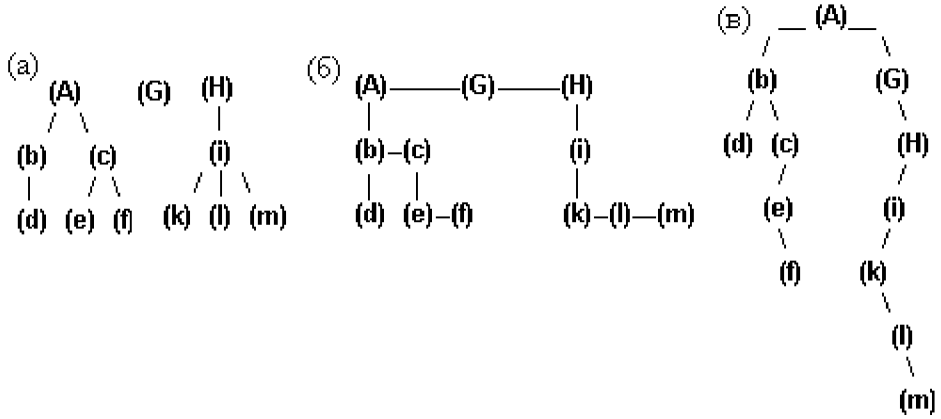


Рис.6.12. а). упорядоченный лес; б) промежуточный результат перестройки леса; в). представление леса в виде двоичного дерева

В результате преобразования упорядоченного леса в бинарное дерево получается полное бинарное дерево с левым и правым поддеревом.

6.2.5 Машинное представление деревьев в памяти ЭВМ.

Деревья можно представлять с помощью связных списков и массивов (или последовательных списков).

Чаще всего используется связное представление деревьев, т.к. оно очень сильно напоминает логическое. Связное хранение состоит в том, что задается связь от отца к сыновьям. В бинарном дереве имеется два указателя, поэтому удобно узел представить в виде структуры:

LPTR	DATA	RPTR

где LPTR - указатель на левое поддерево,
RPTR - указатель на правое поддерево,
DATA - содержит информацию, связанную с вершиной.

На рис. 6.13 показано бинарное дерево и его машинное представление.

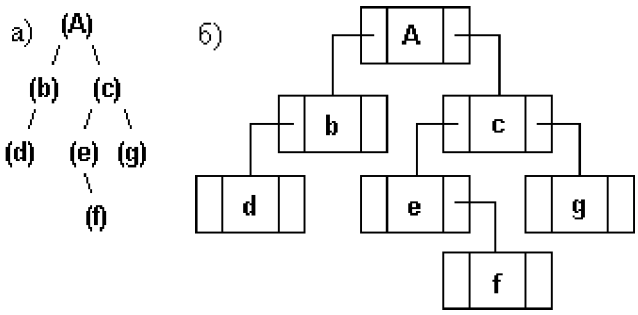


Рис. 6.13. а). логическое представление дерева
б). машинное связное представление дерева

Последовательное представление деревьев удобно и эффективно в случае, если древовидная структура в течение времени своего существования не подвергается значительным изменениям, за счет включения вершин, удаления вершин и т.д.

Выбор метода последовательного представления деревьев определяется также набором тех операций, которые должны быть выполнены над древовидными структурами. (Пример статистической древовидной структуры - пирамидальный метод сортировки). Простейший метод представления дерева в виде последовательной структуры заключается во введении вектора FATHER, задающего отбор для всех его вершин. Этот метод можно использовать также для представления леса. Недостаток метода - он не отображает упорядочения вершин дерева. Если на рис.6.14 поменять местами вершины 9 и 10, последовательное представление останется тем же.

Последовательное представление дерева, логическая диаграмма которого дана на рис. 6.14 , задается следующим образом:

i	1	2	3	4	5	6	7	8	9	10
FATHER [i]	0	1	1	1	2	3		7	4	4,

где ветви определяются как {(FATHER[i],i)}, i = 2,3,...,10.

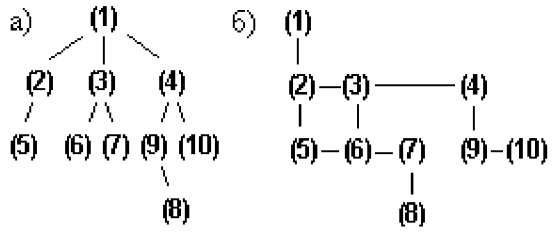


Рис. 6.14. Диаграммы дерева: а) исходное;
б) перестройка в бинарное

Вершины 2,3,4 являются сыновьями вершины 1, вершина 5 - сыном вершины 2, вершины 6,7 - сыновьями вершины 3, вершина 8 имеет отца вершина 7 и вершины 9 и 10 - сыновья вершины 4.

Числовые поля данных используются здесь, чтобы упростить представление дерева. Корневая вершина не имеет отца, поэтому вместо отца для нее задается нулевое значение.

Общее правило: если T обозначает индекс корневой вершины дерева, то FATHER[T] = 0.

Другой метод последовательного представления деревьев заключается в использовании физической смежности элементов машинной памяти вместо одного из полей LPTR или RPTR, например, способ опускания полей, т.е. чтобы вершины появлялись в нисходящем порядке. Дерево (рис.6.14(б)), можно описать как:

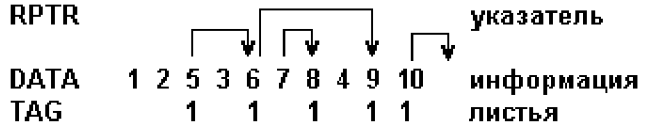


Рис. 6.15. Последовательное представление дерева методом опускания полей

где RPTR,DATA и TAG представляют векторы. В данном методе указатель LPTR не требуется, т.к. если бы он не был пуст, то указывал бы на вершину, стоящую непосредственно справа от данной. Вектор TAG - бинарный вектор, в котором единицы отмечают концевые вершины исходного дерева. При таком представлении имеются значительные потери памяти, т.к. свыше половины указателей RPTR оказываются пустыми. Эти пустые места можно использовать путем установки указателя RPTR каждой данной вершины на вершину, которая следует непосредственно за поддеревом, расположенном под ней. В таком представлении поле RPTR переименовывается в RANGE:

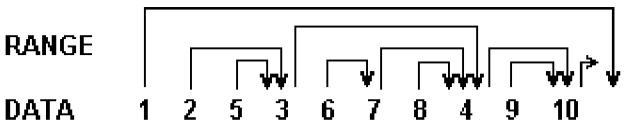


Рис. 6.16. Последовательное представление дерева с размещением вершин в возрастающем порядке

В этом случае поле TAG не требуется поскольку концевой узел определяется условием RANGE(P) = P + 1.

Третий метод состоит в представлении дерева общего вида на основе его восходящего обхода. Такое представление состоит из двух векторов: один вектор описывает все вершины дерева в восходящей последовательности, а второй - задает полустепени исхода этих вершин (см. рис.6.17). Восходящий метод представления удобен для вычисления функцией, заданных на определенных вершинах дерева (например, использование таких функций для генерации объектного кода по обратной польской записи некоторого выражения).

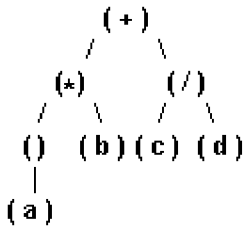


Рис. 6.17. Последовательное представление дерева на основе восходящего обхода

В заключении приведем два важных понятия.

Подобие бинарных деревьев - два дерева подобны, если они имеют одинаковую структуру (форму).

Эквивалентные бинарные деревья - два дерева эквивалентные, если они подобны, и если соответствующие вершины у них содержат одинаковую информацию.

6.2.6 Основные операции над деревьями.

Над деревьями определены следующие основные операции, для которых приведены реализующие их программы.

1) Поиск узла с заданным ключом (Find).
2) Добавление нового узла (Dob).
3) Удаление узла (поддерева) (Udal).
4) Обход дерева в определенном порядке:
   - Нисходящий обход (процедура Preorder , рекурсивная процедура r_Preoder);
   - Смешанный обход (процедура Inorder, рекурсивная процедура r_Inorder);
   - Восходящий обход (процедура Postorder, рекурсивная процедура r_Postorder).

Приведенные ниже программы процедур и функций могут быть непосредственно использованы при решении индивидуальных задач. Кроме выше указанных процедур приведены следующие процедуры и функции:

- процедура включения в стек при нисходящем обходе (Push_st);

- функция извлечения из стека при нисходящем обходе (Pop_st);
- процедура включения в стек при восходящем и смешанном обходе

(S_Push);

- функция извлечения из стека при восходящем и смешанном обходе (S_Pop).

Для прошитых деревьев:

* функция нахождения сына данного узла (Inson);
* функция нахождения отца данного узла (Inp);
* процедура включения в дерево узла слева от данного (leftIn);

ПОИСК ЗАПИСИ В ДЕРЕВЕ (Find). Нужная вершина в дереве ищется по ключу. Поиск в бинарном дереве осуществляется следующим образом.

Пусть построено некоторое дерево и требуется найти звено с ключом X. Сначала сравниваем с X ключ, находящийся в корне дерева. В случае равенства поиск закончен и нужно возвратить указатель на корень в качестве результата поиска. В противном случае переходим к рассмотрению вершины, которая находится слева внизу, если ключ X меньше только что рассмотренного, или справа внизу, если ключ X больше только что рассмотренного. Сравниваем ключ X с ключом, содержащимся в этой вершине, и т.д. Процесс завершается в одном из двух случаев:

1) найдена вершина, содержащая ключ, равный ключу X;
2) в дереве отсутствует вершина, к которой нужно перейти для выполнения очередного шага поиска.

В первом случае возвращается указатель на найденную вершину. Во втором - указатель на звено, где остановился поиск, (что удобно для построения дерева). Реализация функции Find приведена в программном примере 6.2.

```
{===  Программный пример 6.2.  Поиск звена по ключу  === }
Function Find(k:KeyType;d:TreePtr;var rez:TreePtr):bollean;
{ где k - ключ, d - корень дерева, rez - результат }
Var
   p,q: TreePtr;
     b: boolean;
Begin
  b:=false;   p:=d;                    { ключ не найден }
  if d <> NIL  then
   repeat q: =p; if p^.key = k  then  b:=true  { ключ найден }
        else begin                      { указатель на отца }
             if k < p^.key then p:=p^.left { поиск влево }
                           else p:=p^.right { поиск вправо}
          end;
        until b or (p=NIL);
  Find:=b; rez:=q;
```

```
End; { Find }
```

ДОБАВЛЕНИЕ НОВОГО УЗЛА (Dop). Для включения записи в дерево прежде всего нужно найти в дереве ту вершину, к которой можно "подвести" (присоединить) новую вершину, соответствующую включаемой записи. При этом упорядоченность ключей должна сохраняться.

Алгоритм поиска нужной вершины, вообще говоря, тот же самый, что и при поиске вершины с заданным ключом. Эта вершина будет найдена в тот момент, когда в качестве очередного указателя, определяющего ветвь дерева, в которой надо продолжить поиск, окажется указатель NIL (случай 2 функции Find). Тогда процедура вставки записывается так, как в программном примере 6.3.

```
{=== Программный пример 6.3. Добавление звена ===}
Procedure Dob (k:KeyType; var d:TreePtr; zap:data);
{ k - ключ, d - узел дерева, zap - запись }
Var
  r,s: TreePtr;
    t: DataPtr;
Begin
   if not Find(k,d,r) then
    begin        (* Занесение в новое звено текста записи *)
      new(t); t^:=zap; new(s);  s^.key:=k;
      s^.ssil:=t; s^.left:=NIL; s^.right:=NIL;
       if d = NIL   then d:=s      (* Вставка нового звена *)
       else if k < r^.key  then r^.left:=s
            else r^.right:=s;
    end;    End; { Dop }
```

ОБХОД ДЕРЕВА. Во многих задачах, связанных с деревьями, требуется осуществить систематический просмотр всех его узлов в определенном порядке. Такой просмотр называется прохождением или обходом дерева.

Бинарное дерево можно обходить тремя основными способами: нисходящим, смешанным и восходящим (возможны также обратный нисходящий, обратный смешанный и обратный восходящий обходы). Принятые выше названия методов обхода связаны с временем обработки корневой вершины: До того как обработаны оба ее поддерева (Preorder), после того как обработано левое поддерево, но до того как обработано правое (Inorder), после того как обработаны оба поддерева (Postorder). Используемые в переводе названия методов отражают направление обхода в дереве: от корневой вершины вниз к листьям - нисходящий обход; от листьев вверх к корню - восходящий обход, и смешанный обход - от самого левого листа дерева через корень к самому правому листу.

Схематично алгоритм обхода двоичного дерева в соответствии с нисходящим способом может выглядеть следующим образом:

1. В качестве очередной вершины взять корень дерева. Перейти к пункту 2.
2. Произвести обработку очередной вершины в соответствии с требованиями задачи. Перейти к пункту 3.
3. а).Если очередная вершина имеет обе ветви, то в качестве новой вершины выбрать ту вершину, на которую ссылается левая ветвь, а вершину, на которую ссылается правая ветвь, занести в стек; перейти к пункту 2;

б) если очередная вершина является конечной, то выбрать в качестве новой очередной вершины вершину из стека, если он не пуст, и перейти к пункту 2; если же стек пуст, то это означает, что обход всего дерева окончен, перейти к пункту 4;

в) если очередная вершина имеет только одну ветвь, то в качестве очередной вершины выбрать ту вершину, на которую эта ветвь указывает, перейти к пункту 2.

4. Конец алгоритма.

Для примера рассмотрим возможные варианты обхода дерева (рис.6.18).

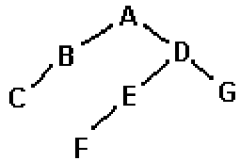


Рис.6.18. Схема дерева

При обходе дерева представленного на рис.6.18 этими тремя методами мы получим следующие последовательности:

ABCDEFG (нисходящий);
CBAFEDG (смешанный);
CBFEGDA (восходящий).

НИСХОДЯЩИЙ ОБХОД (Preorder, r_Preorder).В соответствии с алгоритмом, приведенным выше, текст процедуры имеет вид, представленный в программном примере 6.4. В этом и последующих примерах мы предполагаем, что существует общееопределение типа стека, как:

```
Type    Stack=^Zveno;
        Zveno = record
           next: Stack;
              el: pointer;
            end;
```

{=== Программный пример 6.4. Нисходящий обход ===}

```
Procedure Preorder (t: TreePtr);
Var    st: stack;
    p: TreePtr;
(*------------ Процедура занесения в стек указателя ------*)
Procedure Push_st (var st:stack; p:pointer);
Var     q: stack;
begin   new(q);  q^.el:=p;   q^.next:=st;    st:=q;  end;
(*----------- Функция извлечения из стека указателя ------*)
Function Pop_st (var st: stack):pointer;
Var   e: stack;
begin  Pop_st:=st^.el;  e:=st;  st:=st^.next;  dispose(e);  end;
Begin
    if t = NIL  then
     begin    writeln('Дерево пусто');  exit;   end
       else begin   st:=nil;   Push_st(St,t);   end;
    while st <> nil do    { контроль заполнения стека }
      begin   p:=Pop_st(st);
            while p <> nil do
               begin    {  Обработка данных звена   }
                 if p^.right <> nil  then Push_st(st,p^.right);
                 p:=p^.left;
               end; end;
End; { Preorder }
```

Трассировка нисходящего обхода приведена в табл.6.1:

Таблица 6.1

@ узла в стеке	@ указателя	узел	обработка узла	выходная строка
	p: = t	A	A	A
←D	RPA	D		
	LPA	B	B	AB
	RPB = nil			
	LPB	C	C	ABC
	RPC = nil			
	LPC = nil			
→	D	D	D	ABCD
←G	RPD	G		
	LPD	E	E	ABCDE
	RPE = nil			
	LPE	F	F	ABCDEF
	RPF = nil			
	LPF = nil			
→	G	G	G	ABCDEFG
	RPG = nil			

	LPG = nil			

РЕКУРСИВНЫЙ НИСХОДЯЩИЙ ОБХОД. Алгоритм существенно упрощается при использовании рекурсии. Так, нисходящий обход можно описать следующим образом:

1). Обработка корневой вершины;

2). Нисходящий обход левого поддерева;

3). Нисходящий обход правого поддерева.

Алгоритм рекурсивного нисходящего обхода реализован в программном примере 6.5.

```
{=== Программный пример 6.5.  Рекурсивный нисходящий обход  ===}
  Procedure r_Preorder (t: TreePtr);
  begin
    if t = nil   then begin  writeln('Дерево пусто'); exit;  end;
  (*------------------- Обработка данных звена --------------*)
            ..............................
    if t^.left <> nil   then r_Preorder(t^.left);
    if t^.right <> nil  then r_Preorder(t^.right);
  End; { r_Preorder }
```

СМЕШАННЫЙ ОБХОД (Inorder, r_Inorder). Смешанный обход можно описать следующим образом:

1) Спуститься по левой ветви с запоминанием вершин в стеке;

2) Если стек пуст то перейти к п.5;

3) Выбрать вершину из стека и обработать данные вершины;

4) Если вершина имеет правого сына, то перейти к нему; перейти к п.1.

5) Конец алгоритма.

В программном примере 6.6. реализован алгоритм смешанного обхода дерева.

```
{===  Программный пример 6.6.  Процедура смешанного обхода  ===}
  Procedure Inorder (t: TreePtr);
   label 1;
  Var   st: stack;
      p: TreePtr;
   (*---------- Процедура занесения в стек указателя ---------*)
   Procedure Push_st (var st:stack; p:pointer);
   Var      q: stack;
   begin   new(q);  q^.el:=p;  q^.next:=st;  st:=g;   end;
   (*------------ Функция извлечения из стека указателя ------*)
   Function Pop_st (var st: stack):pointer;
   Var  e: stack;
   begin  Pop_st:=st^.el;  e:=st;   st:=st^.next;  dispose(e); end;
   Begin
```

```
  if t = NIL  then begin  writeln('Дерево пусто'); exit; end
    else begin  p:=t;  st:=nil;  end;
1: while p^.left <> nil do
      begin   (* Спуск по левой ветви и заполнение очереди *)
        Push_st(st,p);   p:=p^.left;  end;
  if st = NIL then exit;{ контроль заполнения стека }
  p:=Pop_st(st);{выборка очередного элемента на обработку}
(*--------------- Обработка данных звена --------------*)
  ..............................
  if p^.right <> nil
    then begin  p:=p^.right;     { переход к правой ветви }
         goto 1;   end;
End; { Inorder }
```

Трассировка смешанного обхода приведена в табл. 6.2:

Рекурсивный смешанный обход описывается следующим образом:

1) Смешанный обход левого поддерева;
2) Обработка корневой вершины;
3) Смешанный обход правого поддерева.

Таблица 6.2

@ узла в стеке	@ указателя	узел	обработка узла	выходная строка
~~A~~	pt:=@A	A		
~~B~~	LPA	B		
~~C~~	LPB	C		
	LPC = nil			
	C	C	C	C
	RPC = nil			
	B	B	B	CB
	RPB = nil			
	A	A	A	CBA
~~D~~	RPA	D		
~~E~~	LPD	E		
~~F~~	LPE	F		
	LPF = nil			
	F	F	F	CBAF
	RPF = nil			
	E	E	E	CBAFE
	RPE = nil			
	D	D	D	CBAFED
~~G~~	RPD	G		
	LPG = nil			
	G	G	G	CBAFEDG

	RPG = nil		стек пуст	конец алгоритма

Текст программы рекурсивной процедуры (r_Inorder) демонстрируется в программном примере 6.7.

```
{=== Программный пример 6.7.  Рекурсивный смешанный обход === }
  Procedure r_Inorder(t: TreePtr);
 begin
   if t = nil then
   begin  writeln('Дерево пусто');  exit;   end;
   if t^.left <> nil   then R_inorder (t^.left);
  (*--------------- Обработка данных звена --------------*)
   ..............................
   if t^.right <> nil  then R_inorder(t^.right);
  End;
```

ВОСХОДЯЩИЙ ОБХОД (Postorder, r_Postorder). Трудность заключается в том, что в отличие от Preorder в этом алгоритме каждая вершина запоминается в стеке дважды: первый раз - когда обходится левое поддерево, и второй раз - когда обходится правое поддерево. Таким образом, в алгоритме необходимо различать два вида стековых записей: 1-й означает, что в данный момент обходится левое поддерево; 2-й - что обходится правое, поэтому в стеке запоминается указатель на узел и признак (код-1 и код-2 соответственно). Алгоритм восходящего обхода можно представить следующим образом:

1) Спуститься по левой ветви с запоминанием вершины в стеке как 1-й вид стековых записей;
2) Если стек пуст, то перейти к п.5;
3) Выбрать вершину из стека, если это первый вид стековых записей, то возвратить его в стек как 2-й вид стековых запи сей; перейти к правому сыну; перейти к п.1, иначе перейти к п.4;
4) Обработать данные вершины и перейти к п.2;
5) Конец алгоритма.

Текст программы процедуры восходящего обхода (Postorder) представлен в программном примере 6.8.

```
{=== Программный пример 6.8.  Восходящий обход  ====}
 Procedure Postorder (t: TreePtr);
 label 2;
 Var    p: TreePtr;
  top: point;          { стековый тип }
 Sign: byte;           { sign=1 - первый вид стековых записей}
                    { sign=2 - второй вид стековых записей}
 Begin     (*------------ Инициализация -----------------*)
   if t = nil  then
```

```
    begin  writeln('Дерево пусто');   exit;   end
  else begin  p:=t;  top:=nil;  end;  {инициализация стека}
  (*------- Запоминание адресов вдоль левой ветви -------*)
  2: while p <> nil do
    begin s_Push(top,1,p);  { заносится указатель 1-го вида}
        p:=p^.left;  end;
  (*-- Подъем вверх по дереву с обработкой правых ветвей ----*)
    while top <> nil do
    begin p:=s_Pop(top,sign); if sign = 1  then
      begin  s_Push(top,0,p); { заносится указатель 2-го вида  }
           p:=p^.right;   goto 2;  end
            else    (*---- Обработка данных звена ---------*)
      .....................................
      end;    End; { Postorder }
```

РЕКУРСИВНЫЙ СМЕШАННЫЙ ОБХОД описывается следующим образом:

1). Восходящий обход левого поддерева;

2). Восходящий обход правого поддерева;

3). Обработка корневой вершины.

Текст программы процедуры рекурсивного обхода (r_Postorder) демонстрируется в програмном примере 6.9.

```
{===  Программный  пример 6.9. Рекурсивный нисходящий обход ====}
  Procedure r_Postorder (t: TreePtr);
   Begin
     if t = nil then begin  writeln('Дерево пусто'); exit; end;
     if t^.left <> nil  then r_Postorder (t^.left);
     if t^.right <> nil  then r_Postorder (t^.right);
   (*-------------- Обработка данных звена --------------*)
     .....................................
   End; { r_Postorder }
```

Если в рассмотренных выше процедурах поменять местами поля left и right, то получим процедуры обратного нисходящего, обратного смешанного и обратного восходящего обходов.

ПРОЦЕДУРЫ ОБХОДА ДЕРЕВА, ИСПОЛЬЗУЮЩИЕ СТЕК. Процедура включения элемента в стек при нисходящем и смешанном обходе (Push_st) приведена в программном примере 6.10.

```
{ ===  Програмнный  пример 6.10   ====}
 Procedure Push_st (var st: stack; p: pointer);
 Var   q: stack;
 begin  new(q);  q^.el:=p;   q^.next:=st;  st:=q;   end;
```

Функция извлечения элемента из стека при нисходящем и смешанном обходе (Pop_st) приведена в программном примере 6.11.

```
{ === Програмнный пример 6.11 ====}
Function Pop_st (var st: stack):pointer;
Var e: stack;
begin
   Pop_st:=st^.el;
   e:=st;      { запоминаем указатель на текущую вершину }
   st:=st^.next;{сдвигаем указатель стека на следующий элемент}
   dispose(e); { возврат памяти в кучу }
end;
```

При восходящем обходе может быть предложен следующий тип стека:

```
point=^st;
  st = record
        next: point;
          l: integer;
        add: pointer;
       end;
```

Процедура включения элемента в стек при восходящем обходе (S_Push) приведена в программном примере 6.12.

```
{ === Програмнный пример 6.12 ====}
Procedure S_Push (var st: point; Har: integer; add: pointer);
Var   q: point;
begin
  new(q);            { выделяем место для элемента }
  q^.l:=Har;         { заносим характеристику }
  q^.add:=add;       { заносим указатель }
  q^.next:=st;       { модифицируем стек }
 st:=q;
end;
```

Функция извлечения элемента из стека при восходящем обходе (S_Pop) демонстрируется в программном примере 6.13.

```
{ === Програмнный пример 6.13 ====}
Function S_Pop (var st: point; var l: integer):pointer;
Var
 e,p: point;
begin
 l:=st^.l;
```

```
  S_Pop:=st^.add;
  e:=st;          { запоминаем указатель на текущую вершину}
  st:=st^.next; {сдвигаем указатель стека на след. элемент  }
  dispose(e);              { уничтожаем выбранный элемент }
end;
```

ПРОШИВКА БИНАРНЫХ ДЕРЕВЬЕВ. Под прошивкой дерева понимается замена по определенному правилу пустых указателей на сыновей указателями на последующие узлы, соответствующие обходу.

Рассматривая бинарное дерево, легко обнаружить, что в нем имеется много указателей типа NIL. Действительно в дереве с N вершинами имеется (N+1) указателей типа NIL. Это незанятое пространство можно использовать для изменения представления деревьев. Пустые указатели заменяются указателями - "нитями", которые адресуют вершины дерева, и расположенные выше. При этом дерево прошивается с учетом определенного способа обхода. Например, если в поле left некоторой вершины P стоит NIL, то его можно заменить на адрес, указывающий на предшественника P, в соответствии с тем порядком обхода, для которого прошивается дерево. Аналогично, если поле right пусто, то указывается преемник P в соответствии с порядком обхода.

Поскольку после прошивания дерева поля left и right могут характеризовать либо структурные связи, либо "нити", возникает необходимость различать их, для этого вводятся в описание структуры дерева характеристики левого и правого указателей (FALSE и TRUE).

Таким образом, прошитые деревья быстрее обходятся и не требуют для этого дополнительной памяти (стек), однако требуют дополнительной памяти для хранения флагов нитей, а также усложнены операции включения и удаления элементов дерева.

Прошитое бинарное дерево на Паскале можно описать следующим образом:

```
type   TreePtr:=^S_Tree;
       S_Tree = record
           key: KeyType;     { ключ }
           left,right: TreePtr;     { левый и правый сыновья }
           lf,rf: boolean;     { характеристики связей }
           end;
```

где lf и rf - указывают, является ли связь указателем на элемент или нитью. Если lf или rf равно FALSE, то связка является нитью.

До создания дерева головная вершина имеет следующий вид:

Рис. 6.19. Головная вершина

Здесь пунктирная стрелка определяет связь по нити. Дерево подшивается к левой ветви.

В программном примере 6.14 приведена функция (Inson) для определения сына (преемника) данной вершины.

```
{ === Программный пример 6.14 ====}
(*------ Функция, находящая преемника данной вершины X ----*)
(*-------- в соответствии со смешанным обходом ------------*)
Funtion Inson (x: TreePtr):TreePtr;
begin
   Inson:=x^.right;
   if not (x^.rf) then exit;              { Ветвь левая ?}
   while Inson^.lf do                  { связь не нить }
       Inson:=Inson^.left;      { перейти по левой ветви }
end; { Inson }
```

В программном примере 6.15 приведена функция (Int) для определения отца (предка) данной вершины.

```
{ === Програмнный пример 6.15 ====}
(*---------- Функция, выдающая предшественника узла ------*)
(*------- в соответствии со смешанным обходом ------------*)
Function Inp (x:TreePtr):TreePtr;
begin
   Inp:=x^.left;
   if not (x^.lf) then exit;
   while Inp^.rf do   Inp:=Inp^.right;   { связка не нить }
end;
```

В программном примере 6.16 приведена функция, реализующая алгоритм включения записи в прошитое дерево (leftIn). Этот алгоритм вставляет вершину P в качестве левого поддерева заданной вершины X в случае, если вершина X не имеет левого поддерева. В противном случае новая вершина вставляется между вершиной X и вершиной X^.left. При этой операции поддерживается правильная структура прошивки дерева, соответствующая смешанному обходу.

```
{ === Програмнный пример 6.16 ====}
(*- Вставка p слева от x или между x и его левой вершиной -*)
Procedure LeftIn (x,p: TreePtr);
Var
  q: TreePtr;
begin
(*--------------- Установка указателя ------------------*)
  p^.left:=x^.left;   p^.lf:=x^.lf;   x^.left:=p;
  x^.lf:=TRUE;   p^.right:=x;       p^.rf:=FALSE;
  if p^.lf then
```

```
(*-------- Переустановка связи с предшественником --------*)
  begin  q:=TreePtr(Inp(p));   q^.right:=p;   q^.rf:=FALSE;
  end;   end; { LeftIn }
```

Для примера рассмотрим прошивку дерева, приведенного на рис.6.13. при нисходящем обходе. Машинное представление дерева при нисходящем обходе с прошивкой приведено на рис.6.20.

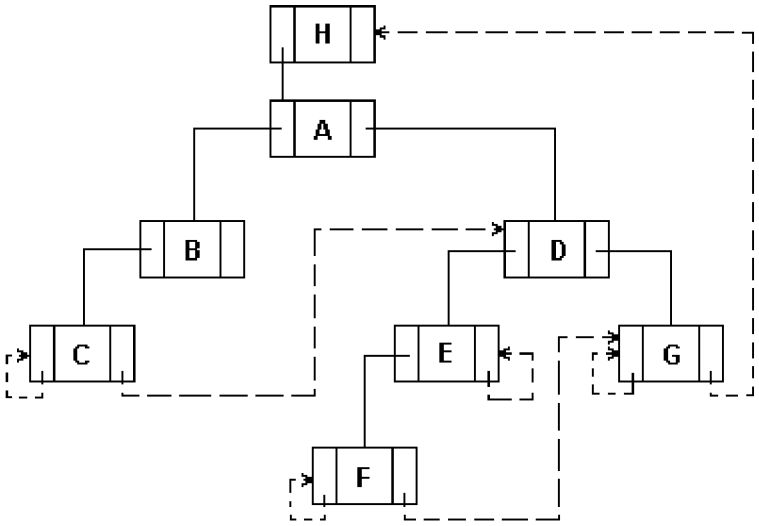


Рис. 6.20. Машинное связное представление исходного дерева, представленного на рис.6.13 при нисходящем обходе с прошивкой

Трассировка нисходящего обхода с прошивкой приведена в табл.6.3.

Рассмотрим на примере того же дерева прошивку при смешанном обходе. Машинное представление дерева при смешанном обходе с прошивкой приведено на рис.6.26.

Таблица 6.3

@ указателя	узел	обработка узла	выходная строка
PT:=H	H		
LPH	A	A	A
LPA	B	B	AB
LPB	C	C	ABC
-LPC			ABC
-RPC	D	D	ABCD
LPD	E	E	ABCDE
LPE	F	F	ABCDEF
-LPF			ABCDEF
-RPF	G	G	ABCDEFG

-LPG			ABCDEFG
-RPG	H		ABCDEFG конец алгоритма

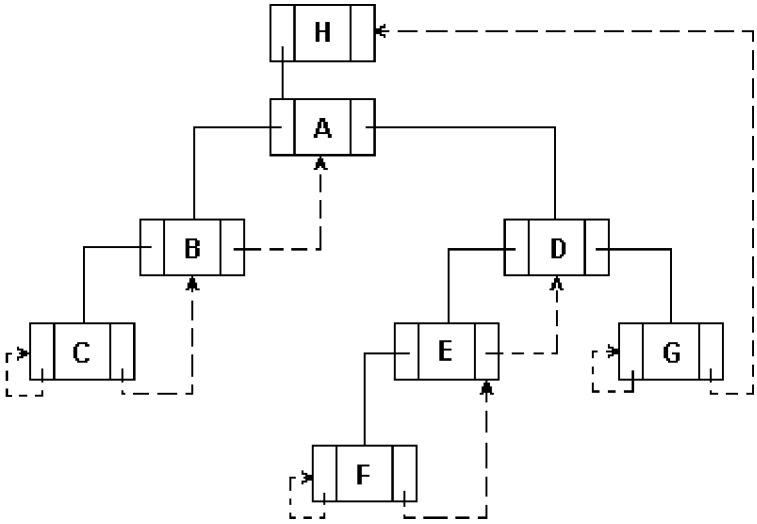


Рис. 6.21. Машинное связное представление дерева при смешанном обходе с прошивкой

Трассировка смешанного обхода с прошивкой приведена в табл.6.4.

Таблица 6.4.

@ указателя	узел	обработка узла	выходная строка
P:=PT	H		
LPH	A		
LPA	B		
LPB	C		
-LPC	C	C	C
-RPC	B	B	CB
-RPB	A	A	CBA
RPA	D		CBA
LPD	E		CBA
LPE	F		CBA
-LPF	F	F	CBAF
-RPF	E	E	CBAFE
-RPE	D	D	CBAFED
RPD	G		CBAFED
-LPG	G		CBAFED
-RPG	H		CBAFED конец алгоритма

6.3 Приложения деревьев

Деревья имеют широкое применение при реализации трансляторов таблиц решений, при работе с арифметическими выражениями, при создании и ведении таблиц символов, где их используют для отображения структуры предложений, в системах связи для экономичного кодирования сообщений и во многих других случаях. Рассмотрим некоторые из них.

6.3.1 Деревья Хаффмена (деревья минимального кодирования)

Пусть требуется закодировать длинное сообщение в виде строки битов: A B A C C D A кодом, минимизирующим длину закодированного сообщения.

1) назначим коды:

символ	код
A	010
B	100
C	000
D	111

каждый символ тремя битами

получим строку:

010 100 010 000 000 111 010

A B A C C D A

7*3=21 всего 21 бит - неэффективно

2) Сделаем иначе: предположим, что каждому символу назначен 2-битовый код:

символ	код
A	00
B	01
C	10
D	11

каждый символ двумя битами

получим строку:

00 01 00 10 10 11 00

A B A C C D A

Тогда кодировка требует лишь 14 бит.

3) Выберем коды, которые минимизируют длину сообщения по частоте вхождений символа в строку: много вхождений - код короткий, мало вхождений - код длинный. A -3 раза, C -2 раза, B -1 раз, D -1 раз, то есть, можно:

1. использовать коды переменной длины.
2. код одного символа не должен совпадать с кодом другого (декодирование идет слева направо).

символ	код	
A	0	Если A имеет код 0 т.к часто встречается, то B, C, D -
B	10	начинаются с 1, если 2-й бит=0,то это C, следующий может
C	110	быть 0 или 1: 1 - D, 0 - B; то-есть: B и D различаются по
D	111	последнему биту, A -по первому, C - по второму, B и D - по третьему

Таким образом, если известны частоты появления символов в сообщении, то метод реализует оптимальную схему кодирования.

Частота появления группы символов равна сумме частот появления каждого символа.

Сообщение ABACCDA кодируется как 0110010101110 и требует лишь 13 бит.

В очень длинных сообщениях, которые содержат символы, встречающиеся очень редко - экономия существенна.

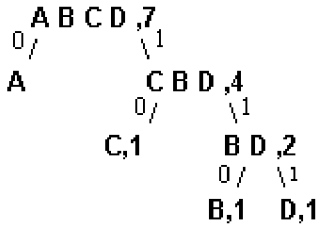


Рис.6.22 Дерево Хаффмена

Обычно коды создаются на основе частоты во всем множестве сообщений, а не только в одном.

6.3.2 Деревья при работе с арифметическими выражениями

Операция объединения двух символов в один использует структуру бинарного дерева. Каждый узел содержит символ и частоту вхождения. Код любого символа может быть определен просмотром дерева снизу вверх, начиная с листа. Каждый раз при прохождении узла приписываем слева к коду 0, если поднимаемся по левой ветви и 1, если поднимаемся по правой ветви. Как только дерево построено код любого символа алфавита может быть определен просмотром дерева снизу вверх, начиная с места, представляющего этот символ. Начальное значение кода пустая строка. Каждый раз, когда мы поднимаемся по левой ветви, к коду слева приписывается ноль, если справа - 1. Часть info узла дерева содержит частоту появления символа представляемого этим узлом. Дерево Хаффмена строго бинарное. Если в алфавите п символов, то дерево Хаффмена может быть представлено массивом узлов размером 2п-1. Поскольку размер памяти, требуемой под дерево известен, она может быть выделена заранее.

МАНИПУЛИРОВАНИЕ АРИФМЕТИЧЕСКИМ ВЫРАЖЕНИЕМИ.

Дано выражение: a*(-b)+c/d

Операции выполняются над выражениями, представленными в виде бинарных деревьев. Такие выражения можно символьно складывать, перемножать, вычитать, дифференцировать, интегрировать, сравнивать на эквивалентность и т.д. Т.е. получаются символьные выражения, которые можно закодировать в виде таблиц:

(-) - операция унарного минуса;

(^) - операция возведения в степень;

(+) - операция сложения;

(*) - операция умножения;

(/) - операция деления.

(E) - указательная переменная, адресующая корень дерева, каждая вершина которого состоит из левого указателя (LPTR), правого указателя (RPTR) и информационного поля TYPE.

LPTR	TYPE	RPTR

Для неконцевой вершины поле TYPE задает арифметическую операцию, связанную с этой вершиной. Значения поля TYPE вершин +,-,*, /, (-) и равны 1, 2, 3, 4, 5, 6 соответственно.

Рис.6.23 Представление выражения в виде дерева

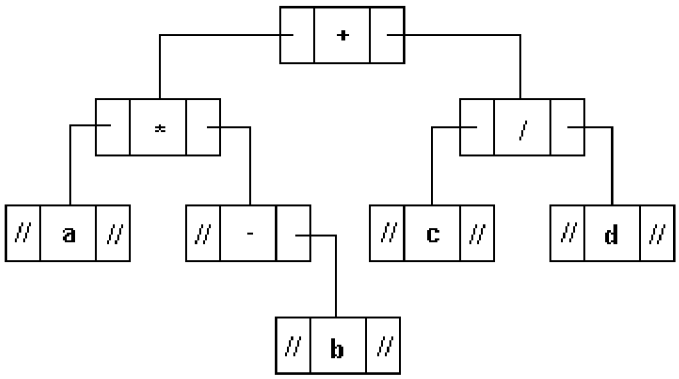


Рис. 6.24 Представление выражения в виде бинарного дерева.

Для концевых вершин поле символа в TYPE имеет значение 0, что означает константу или переменную. В этом случае правый указатель вершины задает адрес таблицы символов, который соответствует данной константе или переменной. В дереве указывается тип оператора, а не он сам, что позволяет упростить обработку таких деревьев.

Процедура вычислений:

Создается семимерный массив меток и его элементам задаются требуемые значения.Оператор генерирует метку исходя из значения поля корневой вершины. И передается управление управление оператору, помеченного меткой. Если данная вершина концевая, то в качестве значения выдается значение переменной или константы, обозначенной этой вершиной. Эта операция выполняется путем использования правого указателя данной вершины для ссылки на нужную запись в таблице символов. Для неконцевой вершины инициализируются рекурсивные вычисления ее поддеревьев, характеризующих операнды текущего оператора. Этот процесс продолжается до тех пор, пока не будут достигнуты листья дерева. Для полученных

листьев значения выбираются из соответствующих записей таблицы символов.

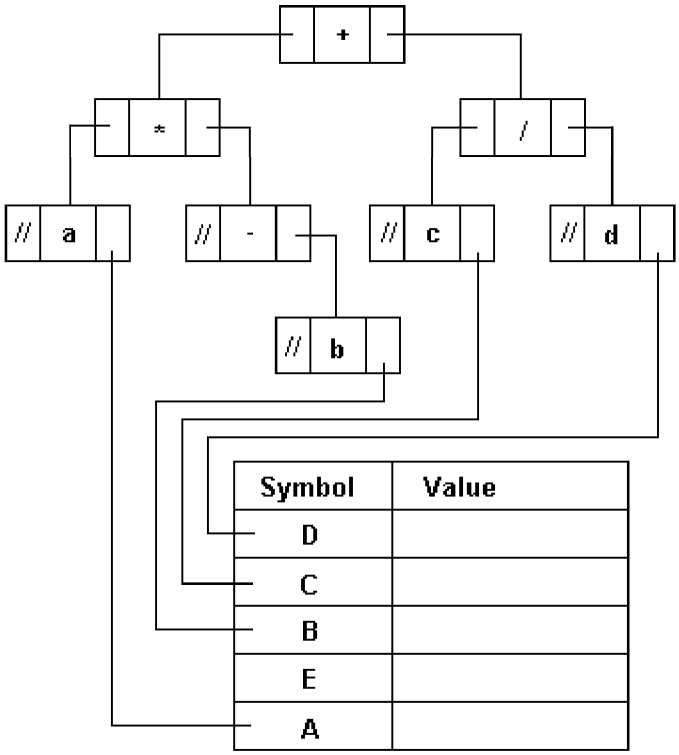


Рис.2.25 Таблица символов

Ниже приводится программа, вычисляющая арифметическое выражение.

6.3.3 Формирование таблиц символов.

В качестве примера приложения бинарных деревьев сформулируем алгоритм ведения древовидно-структурированной таблицы символов.

Основной критерий, которому должна удовлетворять программа ведения таблицы символов, состоит в максимальной эффективности поиска в этой таблицы. Это требование возникает на этапе компиляции, когда осуществляется много ссылок на записи таблицы символов. Необходимо, чтобы над таблицей символов можно было бы проверить две операции - включение записей в таблицу и поиск их в ней.

Причем, каждая из этих операций содержит операцию просмотра таблицы.

Древовидное представление таблицы выбирают по двум причинам:

1. Записи символов по мере их возникновения равномерно распределяются в соответствии с лексикографическим порядком, то при хранении записей в дереве в таком же порядке табличный просмотр становится почти эквивалентен двоичному просмотру.
2. В древовидной структуре легко поддерживать лексикографический порядок, т.к. при включении в нее новой записи необходимо изменить лишь несколько указателей.

Для простоты предположим, что при ведении таблицы символов используется достаточно развитая система записей, допускающая символьные строки переменной длины.

Кроме того, предположим, что подпрограмма ведения таблицысимволов используется при создании дерева данного блока программы. Это предположение ведет к тому, что попытка повторного включения записи вызывает ошибку. В глобальном контексте повторные записи допустимы, если они соответствую разным уровням блочной структуры программы.

В некотором смысле таблица символов представляет собой множество деревьев - по одному для каждого уровня блочной структуры программы.

Вершины бинарного дерева таблицы символов имеют формат:

LPTR	SYMBOLS	INFO	RPTR

SYMBOLS - поле символьной строки, задающей идентификатор или имя переменной (для обеспечения фиксированной длины описания вершин здесь можно хранить не саму строку, а лишь ее описатель);

INFO - некоторое множество полей, содержащих дополнительно информацию об этом идентификаторе, например его тип данных.

Новая вершина создается путем исполнения оператора P при этом ее адрес запоминается в переменной P.

Еще предполагается, что перед любым исполнением программы ведения таблицы символов на некотором чистом уровне блочной структуры уже имеется соответствующая головная вершина дерева, в поле SYMBOLS в которое занесено значение, лексикографически большее, чем любой допустимый идентификатор. Эта вершина адресуется указателем HEAD[n], где n означает номер уровня блочной структуры. Т.е. предполагается, что при входе в блок осуществляется обращение к основной переменной, управляющей созданием головных вершин деревьев.

Операции включения записи в таблицу и операция поиска в таблице содержат значительное количество одинаковых действий (например, просмотр), поэтому рассмотрим только алгоритм TABLE, а различать включение или поиск по переменной FLAG. Если FLAG - истинно - то включение глобальной переменной, если - ложно - поиск.

DATA - содержит имя идентификатора и дополнительную информацию для него.

Если включение новой записи было выполнено успешно, то FLAG сохраняет свое первоначальное значение противоположное начальному, что указывает на ошибку, означающую, что искомый идентификатор уже присутствует в таблице данного уровня и выполняемый алгоритм завершается. Если FLAG = ложь, то надо выполнить операцию поиска записи. В этом случае переменная NAME содержит имя идентификатора, который необходимо найти, а значение переменной. При успешном поиске переменная DATA устанавливается на поле INFO соответствующее записи

таблицы символов. FLAG сохраняет свое значение и осуществляет возврат к вызванной программе. При неудаче операции поиска, FLAG меняет свое значение и выходит из алгоритма. В этом случае основная программа должна осуществлять поиск записи в таблице, более низких уровней. Деревья с головными вершинами HEAD[n-1], HEAD[n-2] b т.д.

АЛГОТИТМ TABLE. На вход подается глобальная переменная n, идентифицирующая номер уровня текущего блока и глобальная переменная FLAG, задающая требуемую операцию. Описываемый алгоритм выполняет эту операцию над древовидной структурированной таблицей символов, локальной относительно блока уровня u. Параметры DATA и NAME используются для передачи данных между алгоритмом и от того больше или меньше значение NAME кода исследуемой записи таблицы, осуществляется установка указателя на левый или правый потолок данной вершины и возврат к шагу 2 для дальнейших сравнений. Поскольку дерево упорядочено таким образом, что код каждой вершины левого (правого) поддерева лексикографических меньше (больше), чем код корневой вершины, то попытка спуска по пустому дереву означает, что требуемая запись в таблице отсутствует; при этом определяется место, где данная запись расположена.

В этом случае, если требовалось найти запись, то выдается сообщение об ошибке, в противном случае создается новая вершина, в нее заносится нужная информация и она включается в уже существующую древовидную структуру слева или справа от исследуемой вершины.

ОПИСАНИЕ ПРОГРАММЫ:

Последовательность решения задачи:

1) Ввод выражения;

2) Построение бинарного дерева из данного выражения;

3) Вычисление математического выражения;

4) Вывод дерева на экран;

5) Вывод результата на экран.

Процедуры программы:

Процедура Tree - преобразует математическое выражение в бинарное дерево. Процедура работает с помощью рекурсивного нисходящего обхода. Имеет подпроцедуру UnderTree.

Подпроцедура UnderTree - специальная процедура. Создает поддеревья исходя из приоритета математической операции. Имеет подпроцедуру Skob. Подпроцедура Skob - учитывает скобки в математическом выражении. Процедура Calc - вычисляет математическое выражение. Процедура использует рекурсивный нисходящий обход. Процедура Symb - определяет в дереве где переменная или константа, и где знак операции. Эта процедура нужна для вычисления математического выражения. Процедура использует рекурсивный нисходящий обход. Процедура OutTree - выводит дерево на экран. Процедура использует рекурсивный нисходящий обход.

```
{===== Программный пример 6.17 ====== }
Program MathExpr;       { Эта программа вычисляет }
{математические  выражения : *, /, +, -, ^.            }
Uses CRT;
    Type tr=^rec;           {Тип дерево}
      rec=record
            pole:string;   {Информационное поле }
             sym:boolean;  {Флаг символа       }
              zn:real;     {Значение переменной }
            rend:boolean;  {Вспомогательный флаг}
             l,r:tr;       {Указатели на потомка}
          end;
    Var root,el : tr;         {вершина и узлы дерева}
         st : string;     {вспомогательная переменная}
        i,j : byte;       { -------"-------}
        x,y : integer;    { координаты для вывода дерева}
         g : byte;        {вспомогательная переменная}
        yn : char;        { -------"-------}
      code : integer;     { для procedure VAL }
    {Процедура Tree }
{Преобразование арифметического выражения  в бинарное дерево }
{ Процедура использует рекурсивный нисходящий обход }
    Procedure Tree(p:tr);
      Procedure undertree(c:char);       { создает поддеревья}
        Procedure Skob;     {процедура для учитывания скобок}
        begin   i:=i+1;
         repeat
           If p^.pole[i]='(' then Skob;   i:=i+1;
         until p^.pole[i]=')';
        end; {End of Skob}
      begin
        for i:=1 to Length(p^.pole) do
          begin  if p^.pole[i]='('
               then begin  g:=i;   Skob;
          if (p^.pole[i+1]<>'+')   and (p^.pole[i+1]<>'-')
           and (p^.pole[i+1]<>'*') and (p^.pole[i+1]<>'/')
           and (p^.pole[g-1]<>'*') and (p^.pole[g-1]<>'/')
           and (p^.pole[g-1]<>'-') and (p^.pole[g-1]<>'+')
           and (p^.pole[i+1]<>'^') and (p^.pole[g-1]<>'^')
           then begin delete(p^.pole,i,1);
                 delete(p^.pole,1,1);  i:=0;
                 end;  end;
          if p^.pole[i]=c then
            begin New(p^.l); p^.l^.l:=nil;
            p^.l^.r:=nil;
```

```
            p^.l^.pole:=copy(p^.pole,1,i-1);
            p^.l^.zn:=0; p^.l^.sym:=false;
            New(p^.r); p^.r^.l:=nil;
            p^.r^.r:=nil;
            p^.r^.pole:=copy(p^.pole,i+1,ord(p^.pole[0]));
            p^.r^.zn:=0; p^.r^.sym:=false;
            i:=ord(p^.pole[0]); p^.pole:=c;
            end;  end;
      end; {end of UnderTree}
   begin
     if p<>nil then
       {Строятся поддеревья в зависимости от приоритета}
       {арифметической операции                    }
      begin   UnderTree('+');   UnderTree('-');
            UnderTree('*');   Undertree('/');
            Undertree('^');   Tree(p^.l);  Tree(p^.r);
           end;
   end; {End of Tree}
   { Вычисление значения арифметического выражения     }
   { Процедура использует рекурсивный нисходящий обход}
   Procedure Calc(p:tr);
   begin
    if p<> nil then begin
          if p^.l^.sym and p^.r^.sym then begin
            case p^.pole[1] of
             '+' : begin   p^.zn:=p^.l^.zn+p^.r^.zn; p^.sym:=true;   end;
             '-' : begin   p^.zn:=p^.l^.zn-p^.r^.zn; p^.sym:=true;  end;
             '*' : begin  p^.zn:=p^.l^.zn*p^.r^.zn;  p^.sym:=true;   end;
             '/' : begin  p^.zn:=p^.l^.zn/p^.r^.zn; p^.sym:=true; end;
             '^' : begin   p^.zn:=EXP(p^.r^.zn*LN(p^.l^.zn));
               p^.sym:=true;  end;
            end; {end of case}      end;
        Calc(p^.l);   Calc(p^.r);
        end;
   end; {end of calc}
{ Процедура определяет где в дереве переменная или значение,  }
{ и где знак операции. Использует рекурсивный нисходящий обход}
Procedure Symb(p:tr);
begin
   if p<> nil then begin
          if p^.pole[1] in ['a'..'z']
            then begin  p^.sym:=true; Write(p^.pole,'= ');
                Readln(p^.zn);  end;
          if p^.pole[1] in ['0'..'9']  then begin  p^.sym:=true;
           VAL(p^.pole,p^.zn,code);   end;
```

```
          Symb(p^.l);  Symb(p^.r); end;
    end; {End of Symb}
    { Процедура выводит на экран полученное дерево     }
    { Процедура использует рекурсивный нисходящий обход}
    Procedure OutTree(pr:tr;f:byte);
    begin
      y:=y+2;
      if pr<>nil then begin
        If f=1 then begin x:=x-5; end;
        if f=2 then begin x:=x+9; end;
        GotoXY(X,Y);
        {Если f=0, то выводится корневая вершина}
        if f=0 then Writeln('[',pr^.pole,']');
        {Если f=1, то - левое поддерево}
        if f=1 then Writeln('[',pr^.pole,']/');
        {Если f=2, то - правое поддерево}
        if f=2 then Writeln('\[',pr^.pole,']');
        OutTree(pr^.l,1);   OutTree(pr^.r,2);
      end;    y:=y-2;
    end; {End of OutTree}
    begin          {Главная программа}
     repeat
      Window(1,1,80,25);  x:=22; y:=0;
      TextBackGround(7); TextColor(Blue);  ClrScr;
  {Ввод выражения, которое надо вычислить}
Writeln('Введите ваше выражение:');
  GotoXY(40,4);    Write('Используйте следующие операции:');
  GotoXY(50,5);    Write(' + , - , * , / , ^ ');
  GotoXY(40,7);    Write('Программа применяет деревья для');
  GotoXY(40,8);    Write('вычисления арифметического  вы-');
  GotoXY(40,9);    Write('ражения.');
  GotoXY(1,2);     Readln(st);
  {root    Создается корневая вершина}
  New(el); el^.l:=nil; el^.r:=nil; El^.pole:=st;
  el^.zn:=0; el^.sym:=false; el^.rend:=false; root:=el;
  {end of root}
  Tree(root);        {Создается дерево}
  {Ввод значений переменных}
  Writeln('Введите значения:'); Symb(root); Window(30,1,80,25);
  TextBackGround(Blue); TextColor(White); ClrScr;
  WriteLn('Дерево выглядит так:');      {Вывод дерева на экран}
  OutTree(root,0);
    repeat
      if root^.l^.sym and root^.r^.sym
        then begin   Calc(root); root^.rend:=true; end
```

```
        else calc(root);
    until root^.rend;
    Window(1,23,29,25);   TextBackGround(Red);
    TextColor(Green);     ClrScr;
    Writeln('Результат =',root^.zn:2:3);     {Вывод результата }
    Write('Еще?(y/n)');   readln(yn);
      until yn='n';
end.
```

Результат работы программы представлен на рис 6.26.

```
Введите ваше выражение:    Дерево выглядит так:
a+d*(e-3)^2                          [+]
Введите значения:                    / \
a= 2                           [a]/     \[*]
d= 3                                   / \
e= 5                               [d]/     \[^]
Результат выражения=14.000                   / \
                                         [-]/     \[2]
                                           / \
                                      [e]/     \[3]
```

Рис. 6.26. Результат работы программы

6.4 СБАЛАНСИРОВАННЫЕ ДЕРЕВЬЯ

ОПРЕДЕЛЕНИЯ. Одной из наиболее часто встречающихся задач является поиск необходимых данных. Существуют различные методы, отличающиеся друг от друга временем поиска, сложностью алгоритмов, размерами требуемой памяти. Обычно стремятся всячески сократить время, затрачиваемое на поиск необходимого элемента. Одним из самых быстрых методов является поиск по упорядоченному бинарному дереву. При удачной структуре дерева время поиска элементов не превышает в среднем log N. Но при неудачной структуре время поиска может значительно возрасти, достигая N\2. (N - число элементов дерева).

Одним из методов, улучшающих время поиска в бинарном дереве является создание сбалансированных деревьев обладающих минимальным временем поиска.

Одно из определений сбалансированности было дано Адельсоном-Вельским и Ландисом:

Дерево является СБАЛАНСИРОВАННЫМ тогда и только тогда, когда для каждого узла высота его двух поддеревьев различается не более, чем на 1.

Поэтому деревья, удовлетворяющие этому условию, часто называют "АВЛ-деревьями" (по фамилиям их изобретателей).

Операции выполняемые над сбалансированным деревом: поиск, вставка, удаление элемента.

Обратимся к задаче поддержания структуры дерева таким образом, чтобы за время, не превышающее (log N), могла быть выполнена каждая из следующих операций:

1) вставить новый элемент;

2) удалить заданный элемент;

3) поиск заданного элемента.

С тем чтобы предупредить появление несбалансированного дерева, вводится для каждого узла (вершины) дерева показатель сбалансированности, который не может принимать одно из трех значений, левое - (L), правое - (R), сбалансированное - (B), в соответствии со следующими определениями:

левое - узел левоперевешивающий, если самый длинный путь по ее левому поддереву на единицу больше самого длинного пути по ее правому поддереву;

сбалансированное - узел называется сбалансированный, если равны наиболее длинные пути по обеим ее поддеревьям;

правое - узел правоперевешивающий, если самый длинный путь по ее правому поддереву на единицу больше самого длинного пути по ее левому поддереву;

В сбалансированном дереве каждый узел должен находится в одном из этих трех состояний. Если в дереве существует узел, для которого это условие несправедливо, такое дерево называется несбалансированным.

ОПЕРАЦИЯ ВСТАВКИ ВЕРШИНЫ В СБАЛАНСИРОВАННОЕ ДЕРЕВО. Предполагается, что новая вершина вставляется на уровне листьев, или терминальных вершин (как левое или правое поддерево). При такой вставке показатели сбалансированности могут изменится только у тех вершин, которые лежат на пути между корнем дерева и вновь вставляемым листом.

Алгоритм включения и балансировки полностью определяется способом хранения информации о сбалансированности дерева. Определение типа узла имеет вид:

```
TYPE ref=^node;          { указатель              }
  node=record
        key:integer;  { ключ узла              }
   left,right:ref;     { указатели на ветви     }
        bal:-1..+1;   { флаг сбалансированности }
      end;
```

Процесс включения узла состоит из последовательности таких трех этапов:

1. Следовать по пути поиска, (по ключу), пока не будет найден ключ или окажется,что ключа нет в дереве.
2. Включить новый узел и определить новый показатель сбалансированности.
3. Пройти обратно по пути поиска и проверить показатель сбалансированности у каждого узла.

На каждом шаге должна передаваться информация о том, увеличилась ли высота поддерева (в которое произведено включение). Поэтому можно ввести в список параметров переменную h, означающую "высота поддерева увеличилась".

Необходимые операции балансировки полностью заключаются в обмене значениями ссылок. Фактически ссылки обмениваются значениями по кругу, что приводит к однократному или двукратному "повороту" двух или трех узлов.

ПРИНЦИП РАБОТЫ АЛГОРИТМА. Рассмотрим бинарное дерево представленное на рис. 6.27 (а), которое состоит только из двух узлов. Включение ключа 7 дает вначале несбалансированное дерево (т.е. линейный список). Его балансировка требует однократного правого (RR) поворота, давая в результате идеально сбалансированное дерево (б). Последующее включение узлов 2 и 1 дает несбалансированное поддерево с корнем 4. Это поддерево балансируется однократным левым (LL) поворотом (г). Далее включение ключа 3 сразу нарушает критерий сбалансированности в корневом узле 5. Сбалансированность теперь восстанавливается с помощью более сложного двукратного поворота налево и направо (LR); результатом является дерево (д). Теперь при следующем включении потерять сбалансированность может лишь узел 5. Действительно, включение узла 6 должно привести к четвертому виду балансировки: двукратному повороту направо и налево (RL). Окончательное дерево показано на рис.6.27 (а -е).

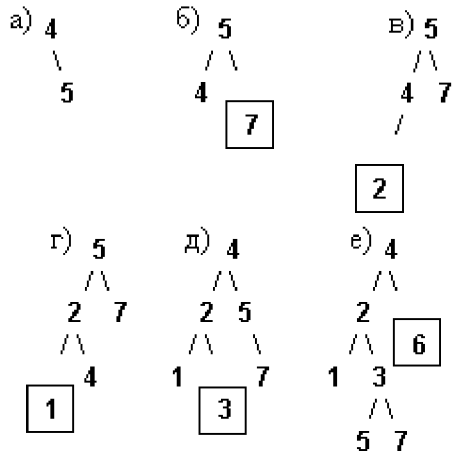


Рис. 6.27. Последовательное включение узлов в сбалансированное дерево.

АЛГОРИТМ Insert_&_Balanse включения узла в сбалансированное дерево.

Дано: сбалансированное бинарное дерево с корнем ROOT.

Поля: LPTR, RPTR, KEY (ключ), BAL (показатель сбалансированности), DATA (информация).

Заданы переменные: ключ - x, информация - INF.

Алгоритм вставляет в дерево новый элемент, сохраняя сбалансированность дерева. Если элемент уже присутствует в дереве, то выводится соответствующее сообщение.

Переменная h используется как флаг, указывающий на то, что было произведено включение элемента. P - текущий указатель при перемещении по дереву, p1 и p2 - вспомогательные указатели. Count - счетчик вставленных элементов.

```
Начало Insert_&_Balanse:
1. Поиск места для вставки:
Если x < KEY(p)
то: если p=nil
    то: ВСТАВИТЬ_ЭЛЕМЕНТ и перейти к п. 3;
    иначе:  p=LPTR(p) и перейти к п. 1;
     повторный вызов Insert_&_Balanse;
Если x > KEY(p)
то: если p=nil
   то: ВСТАВИТЬ_ЭЛЕМЕНТ и перейти к п. 5;
   иначе: p=RPTR(p) и перейти к п. 1;
    повторный вызов Insert_&_Balanse;
2. Совпадение:
  Напечатать "Элемент уже вставлен" и конец.
3. Изменение показателей сбалансированности:
  (производилась вставка в левое поддерево)
если BAL(p)=1 то:
  BAL(p)=0; h=false;   { перевеш.-> сбалансир.}
  перейти к п. 7
если BAL(p)=0 то
  BAL(p)=-1;          { перевеш.-> критическ.}
  перейти к п. 7
4. Балансировка при возрастании левого поддерева:
если p=ROOT то ROOT=LPTR(p);  { перенос корневой
                                          вершины }
 p1=LPTR();           { вводим дополнительный указатель }
если BAL(p1)=-1
то:
  { однокр. LL-поворот }
 LPTR(p)=RPTR(p1); RPTR(p1)=p;
 BAL(p)=0; p=p1;
 перейти к п. 7
иначе:
 { двукратный LR-поворот }
 если p1=ROOT
 то ROOT=RPTR(p1); { перенос корневой вершины }
 p2:=RPTR(p1); RPTR(p1)=LPTR(p2);
 LPTR(p1)=p1; LPTR(p)=RPTR(p2);
```

RPTR(p2)=p;
(изменение показателей сбалансированности)
если BAL(p2)=-1 то BAL(p)=1 иначе BAL(p)=0;
если BAL(p2)=1 то BAL(p1)=-1 иначе BAL(p1)=0;
p=p2;
BAL(p)=0; h=false;
перейти к п. 7;
5. Изменение показателей сбалансированности:
(производилась вставка в правое поддерево)
если BAL(p)=-1 то:
BAL(p)=0; h=false; { перевеш.-> сбалансир.}
перейти к п. 7
если BAL(p)=0 то
BAL(p)=1; { перевеш.-> критическ.}
перейти к п. 7
6. Балансировка при возрастании правого поддерева:
если p=ROOT то ROOT=RPTR(p); { перенос корневой
вершины }
p1=RPTR(p); { вводим дополнительный указатель }
если BAL(p1)=1
то:
{ однокр. RR-поворот }
RPTR(p)=LPTR(p1); LPTR(p1)=p;
BAL(p)=0; p=p1;
перейти к п. 7
иначе:
{ двукратн. LR-поворот }
если p1=ROOT
то ROOT=LPTR(p1); { перенос корневой вершины }
p2:=LPTR(p1); LPTR(p1)=RPTR(p2);
RPTR(p1)=p1; RPTR(p)=LPTR(p2);
LPTR(p2)=p;
(изменение показателей сбалансированности)
если BAL(p2)=1 то BAL(p)=-1 иначе BAL(p)=0;
если BAL(p2)=-1 то BAL(p1)=1 иначе BAL(p1)=0;
p=p2;
BAL(p)=0; h=false;
7. Выход.

(Т.к. процедура осуществляет рекурсивный вызов самой себя в п.3, то здесь производится возврат в место предыдущего вызова. Последний выход осуществляется в вызывающую программу).

Конец Insert_&_Balanse.

8. Алгоритм процедуры ВСТАВИТЬ_ЭЛЕМЕНТ:

Начало:

LPTR(p)=nil; RPT(p)=nil; BAL=0; { обнуление указателей }

```
DATA=INF; KEY=x;              { занесение информации }
 h=true;          { установка флага вставки элемента }
  если count=0     { это первый элемент ?          }
  то ROOT=p;
count=count+1;
  Конец.
```

Описание работы:

П.1 - осуществляется поиск места для вставки элемента. Производится последовательный рекурсивный вызов процедурой самой себя. При нахождении места для вставки к дереву добавляется новыйэлемент с помощью процедуры ВСТАВИТЬ_ЭЛЕМЕНТ.

П.2 - Если такой элемент уже существует в дереве, то выводится сообщение об этом и выполнение процедуры завершается.

П.3 (П.5) - производит изменение показателей сбалансированности после включения нового элемента в левое (правое для П.5)поддерево.

П.4 (П.6) - производит балансировку дерева путем перестановки указателей - т.е. LL- и LR-повороты (RR- и RL-повороты в П.6)

П.7 - с помощью рекурсивных вызовов в стеке запоминается весь путь до места создания новой вершины. В П.7 производится обратный обход дерева, корректировка всех изменившихся показателей сбалансированности (в П. 3 и 5) и при необходимости балансировка. Это позволяет производить правильную балансировку, даже если критическая вершина находится далеко то места вставки.

ТЕКСТ ПРОЦЕДУРЫ Insert_&_Balanse. Процедура выполняет действия по вставка элемента в бинарное дерево с последующей балансировкой в соответствии с приведенным выше алгоритмом.

```
{=====Программный пример 6.18=========}
Procedure Insert_&_Balanse (x:integer; var p,root:ref; var h:boolean);
{ x=KEY, p=p, root=ROOT, h=h }
 var   p1,p2:ref;   {h=false}
Begin
 if p=nil
  then  Create(x,p,h)   {слова нет в дереве,включить его}
  else if x=p^.key then
      begin  gotoXY(35,3);  write('Ключ найден!');
           readln; exit; end;
  if  x < p^.key then
   begin   Insert_&_Balanse(x,p^.left,root,h);
    if h then          {выросла левая ветвь}
    case p^.bal of
     1: begin p^.bal:=0; h:=false; end;
     0: p^.bal:=-1;
    -1: begin            {балансировка}
        if p=root then  root:=p^.left;
```

```
            p1:=p^.left;        {смена указателя на вершину}
            if p1^.bal=-1   then
             begin      {однократный LL-поворот}
               p^.left:=p1^.right; p1^.right:=p;
               p^.bal:=0; p:=p1;
             end
             else begin    {2-кратный LR-поворот}
               if p1=root then  root:=p1^.right; p2:=p1^.right;
               p1^.right:=p2^.left; p2^.left:=p1;
               p^.left:=p2^.right; p2^.right:=p;
                if p2^.bal=-1 then p^.bal:=+1 else p^.bal:=0;
                if p2^.bal=+1 then p1^.bal:=-1 else p1^.bal:=0;
               p:=p2;
              end;   p^.bal:=0; h:=false;
          end;      end;{case}
       end  { h then}
     else if x > p^.key then begin
      Insert_&_Balanse(x,p^.right,root,h);
        if h then          {выросла правая ветвь}
         case p^.bal of
          -1: begin p^.bal:=0; h:=false; end;
           0: p^.bal:=+1;
           1: begin              {балансировка}
                if p=root then root:=p^.right;
                p1:=p^.right;    {смена указателя на вершину}
                if p1^.BAL=+1  then
      begin             {однократный RR-поворот}
       p^.right:=p1^.left; p1^.left:=p; p^.BAL:=0; p:=p1; end
      else begin         {2-кратный  RL-поворот}
          if p1=root then  root:=p1^.left;
          p2:=p1^.left; p1^.left:=p2^.right; p2^.right:=p1;
          p^.right:=p2^.left; p2^.left:=p;
          if p2^.BAL=+1 then p^.BAL:=-1 else p^.BAL:=0;
          if p2^.BAL=-1 then p1^.BAL:=+1 else p1^.BAL:=0;
          p:=p2; end;
          p^.BAL:=0; h:=false;
         end; { begin 3 }
         end;{ case }
       end;  {then }
End {Search};
```

ТЕКСТ ПРОЦЕДУРЫ ДОБАВЛЕНИЯ ЭЛЕМЕНТА. Процедура создает новый элемент, заполняет его информационные поля и обнуляет указатели. При создании первого элемента он автоматически становится корнем дерева.

```
Procedure Create (x:integer; var p:ref; var h:boolean);
                { создание нового элемента }
Begin
  NEW(p); h:=true; with p^ do
     begin key:=x; left:=nil; right:=nil; BAL:=0; end;
  if count=0 then root:=p; count:=count+1; End;
```

ОПЕРАЦИЯ УДАЛЕНИЯ ИЗ СБАЛАНСИРОВАННОГО ДЕРЕВА. Удаление элемента из сбалансированного дерева является еще более сложной операцией чем включение, так как может удаляться не только какой-либо из листьев, но и любой узел (в том числе и корень). Поэтому при удалении необходимо правильно изменить структуру связей между элементами, а затем произвести балансировку полученного дерева.

В результате удаления какого-либо узла могут возникнуть ситуации аналогичные тем, что возникают при добавлении элемента:

1. Вершина была лево- или правоперевешивающей, а теперь стала сбалансированной.
2. Вершина была сбалансированной, а стала лево- или правоперевешивающей.
3. Вершина была перевешивающей, а вставка новой вершины в перевешивающее поддерево создала несбалансированное поддерево привела к появлению критической вершины. Необходимо провести балансировку.

В общем процесс удаления элемента состоит из следующих этапов:

1. Следовать по дереву, пока не будет найден удаляемый элемент.
2. Удалить найденный элемент, не разрушив структуры связей между элементами.
3. Произвести балансировку полученного дерева и откорректировать показатели сбалансированности.

На каждом шаге должна передаваться информация о том, уменьшилась ли высота поддерева (из которого произведено удаление).

Поэтому мы введем в список параметров переменную h, означающую "высота поддерева уменьшилась".

Простыми случаями являются удаление терминальных узлов и узлов с одним потомком. Если же узел, который надо удалить имеет два поддерева, мы будем заменять его самым правым узлом левого поддерева.

Для балансировки дерева после удаления используем две (симметричные) операции балансировки: Balance_R используется, когда уменьшается высота правого поддерева, а Balance_L - левого поддерева. Процедуры балансировки используют те же способы (LL-, LR-, RL- и RR-повороты), что и в процедуре вставки элемента.

ПРИМЕР УДАЛЕНИЯ РАЗЛИЧНЫХ УЗЛОВ ИЗ СБАЛАНСИРОВАННОГО ДЕРЕВА.

Узел, который необходимо удалить, обозначен двойной рамкой. Если задано сбалансированное дерево (рис.6.28. а), то последовательное удаление узлов с ключами 4,8,6,5,2,1 и 7 дает деревья (рис.6.28 б...з).

Удаление ключа 4 само по себе просто, т.к. он представляет собой терминальный узел. Однако при этом появляется несбалансированность в узле 3. Его балансировка требует однократного левого поворота налево. Балансировка вновь становится необходимой после удаления узла 6. На этот раз правое поддерево корня балансируется однократным поворотом направо.

Удаление узла 2, хотя само по себе просто, так как он имеет только одного потомка, вызывает сложный двукратный поворот направо и налево.

И четвертый случай: двукратный поворот налево и направо вызывается удалением узла 7, который прежде заменяется самым правымэлементом левого поддерева, т.е. узлом с ключом 3.

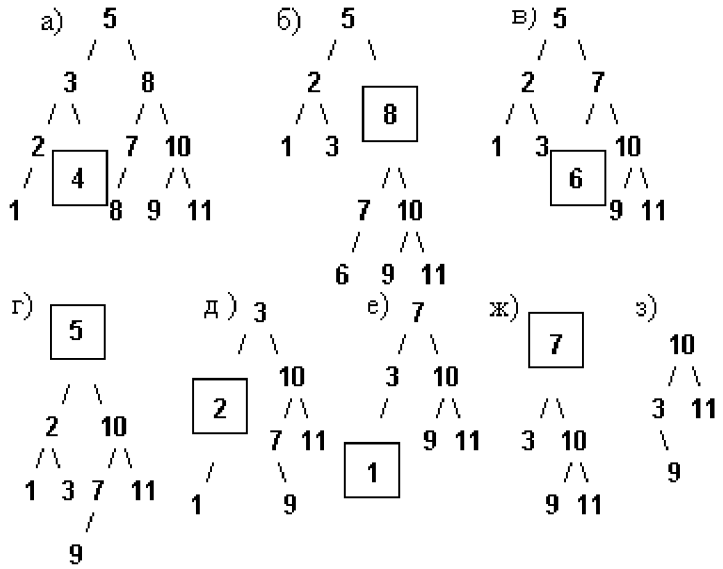


Рис.6.28. Удаление узлов из сбалансированого дерева.

Удаление элемента из сбалансированного дерева удобнее разбить на 4 отдельных процедуры:

1. Delete - осуществляет рекурсивный поиск по дереву удаляемого элемента, вызывает процедуры удаления и балансировки.
2. Del - осуществляет собственно удаление элемента и вызов при необходимости процедуры балансировки.
3. Balance_L и Balance_R - производят балансировку и коррекцию показателей сбалансированности после удаления элемента из левого (правого) поддерева.

АЛГОРИТМ ПРОЦЕДУРЫ Delete. Дано: сбалансированное бинарное дерево с корнем ROOT. Поля: LPTR, RPTR, KEY (ключ), BAL (показатель сбалансированности), DATA (информация).

Задан: ключ - x, информация - INF.

Алгоритм находит удаляемый элемент и вызывает процедуры удаления и последующей балансировки бинарного дерева. Если элемент отсутствует в дереве, выдается соответствующее сообщение.

Переменная h используется как флаг, указывающий на то, что было произведено удаление элемента. P - текущий указатель при перемещении по дереву, q - вспомогательный указатель.

Начало Delete
1. Поиск удаляемого элемента
Если x < KEY(p)
то: p=LPTR(p);
Вызов Delete;
если h=true то Вызов Balance_L;
перейти к п.5
Если x > KEY(p)
то: p=RPTR(p);
Вызов Delete;
если h=true то вызов Balance_R;
перейти к п.5
Если p=nil
то: напечатать "Ключа в дереве нет!";
конец;
2. Удаление элемента : (если все предыдущие условия не выполнены, то указатель указывает на элемент, подлежащий удалению).
Удаляется элемент с одним поддеревом.
q=p; { вводим вспомогательный указатель }
если RPTR(q)=nil
то: p=LPTR(q);
h=true;
перейти к п.4;
если LPTR(q)=nil
то: p=RPTR(q);
h=true;
перейти к п.4;
3. Удаление элемента с двумя поддеревьями:
q=LPTR(q);
если h=true то: вызов Balance_L;
перейти к п.4
4. Напечатать "Узел удален из дерева".
5. Выход.
Конец Delete;

ОПИСАНИЕ РАБОТЫ АЛГОРИТМА:

П.1 - осуществляет поиск удаляемого элемента с помощью рекурсивных вызовов процедуры Delete (т.е. - самой себя). При этом в стеке сохраняется весь путь поиска. Если было произведено удаление элемента, то производится вызов соответствующей процедуры балансировки. Если элемент с заданным ключом не найден, то выводится соответствующее сообщение.

П.2 - производится удаление элемента с одной ветвью простым переносом указателя. Устанавливается флаг удаления элемента.

П.3 - производится вызов процедуры Del, производящей удаление элемента с двумя поддеревьями.

П.5 - т.к. эта процедура рекурсивная, то производится возврат в место предыдущего вызова, либо в главную программу.

АЛГОРИТМ ПРОЦЕДУРЫ Del. Дано: указатель - r на элемент дерева с двумя поддеревьями.

Алгоритм производит удаление этого элемента, с сохранением связей с нижележащими элементами, и вызов процедуры балансировки.

Используется вспомогательный указатель q, описанный в процедуре Delete.

```
Начало Del.
 1. Поиск последнего элемента в правом поддереве
Если RPTR(r) <> nil  { элемент не найден }
то: r=RPTR(r);
   вызов процедуры Del;
   если h=true то вызов процедуры Balance_R;
   перейти к п.2;
   иначе: KEY(q)=KEY(r); r=RPTR(r); h=true;
 2. Выход.
Конец Del;
```

ОПИСАНИЕ РАБОТЫ:

П.1 - производится рекурсивный поиск самого правого элемента в поддереве. Если элемент найден, то он ставится на место удаленного элемента, устанавливается флаг удаления, и осуществляется выход. Если установлен флаг удаления элемента, то вызывается процедура балансировки.

П.5 - т.к. эта процедура рекурсивная, то производится возврат в место предыдущего вызова, либо в вызывающую процедуру (Delete).

АЛГОРИТМ ПРОЦЕДУРЫ Balance_L. Дано: бинарное дерево, в левом поддереве которого было произведено удаление элемента.

Задан: указатель p на место удаленного элемента.

Алгоритм производит балансировку бинарного дерева и корректировку показателей сбалансированности.

P1 и P2 - вспомогательные указатели, b1 и b2 - вспомогательные показатели сбалансированности.

```
Начало Balance_L:
   1. Корректировка показателей сбалансированности:
Если BAL(p)=-1
  то: BAL(p)=0;          { перевеш. -> сбалансир. }
     конец
Если . BAL(p)=0
  то: BAL(p)=1; h=false;   { сбалансир. -> перевеш. }
     конец
p1=RPTR(p); b1=BAL(p1);    { BAL(p)=1 - критическая. }
```

2. Однократный RR-поворот :

Если b1 >= 0

то:

Если p=ROOT то ROOT=RPTR(p); { перенос корня }

RPTR(p)=LPTR(p1); LPTR(P1)=p;

{ корректировка показателей сбалансированности }

если b1=0

то: BAL(p)=1; BAL(p1)=-1; h=false;

иначе: BAL(p)=0; BAL(p1)=0;

p=p1;

конец

2. Двукратный RL-поворот :

если b1 < 0

если p1=ROOT то ROOT=RPTR(p); { перенос корня }

p2=LPTR(p1); b2=BAL(p2);

LPTR(p1)=RPTR(p2); RPTR(p2)=p1;

RPTR(p)=LPTR(p2); LPTR(p2)=p;

{ корректировка показателей сбалансированности }

если b2=1 то BAL(p)=-1 иначе BAL(p)=0;

если b2=-1 то BAL(p1)=1 иначе BAL(p1)=0;

p=p2; BAL(p2)=0;

конец

Конец Balance_L;

ОПИСАНИЕ РАБОТЫ АЛГОРИТМА:

П.1 - если вершина не является критической, то производится изменение показателей сбалансированности. Если вершина критическая - создаются вспомогательные указатели.

П.2 и 3 - производят балансировку дерева однократным RR(п.2) и двукратным RL- (п.3) поворотами и изменение показателей сбалансированности.

АЛГОРИТМ ПРОЦЕДУРЫ Balance_R. Дано: бинарное дерево, в левом поддереве которого было произведено удаление элемента.

Алгоритм, входные данные и вспомогательные переменные аналогичны алгоритму Balance_L, изменены на противоположные только условия выбора и направления указателей.

Начало Balance_R:

1. Корректировка показателей сбалансированности:

Если BAL(p)=1

то: BAL(p)=0; { перевеш. -> сбалансированная. }

конец

Если BAL(p)=0

то: BAL(p)=-1; h=false; { сбалансир. -> перевешивающая. }

конец

p1=LPTR(p); b1=BAL(p1); { BAL(p)=1 - критическая. }

2. Однократный LL-поворот :

```
если b1 <= 0
то:
  если . p=ROOT то ROOT=LPTR(p);       { перенос корня }
  LPTR(p)=RPTR(p1); RPTR(P1)=p;
   { корректировка показателей сбалансированности }
  если b1=0
     то: BAL(p)=-1; BAL(p1)=1; h=false;
   иначе: BAL(p)=0; BAL(p1)=0;
  p=p1;
  конец
 3. Двукратный RL-поворот :
если b1 > 0
 если p1=ROOT то ROOT=LPTR(p);      { перенос корня }
 p2=RPTR(p1); b2=BAL(p2);
 RPTR(p1)=LPTR(p2); LPTR(p2)=p1;
 LPTR(p)=RPTR(p2);  RPTR(p2)=p;
  { корректировка показателей сбалансированности }
 если b2=-1 то BAL(p)=1 иначе BAL(p)=0;
 если b2=1 то BAL(p1)=-1 иначе BAL(p1)=0;
  p=p2; BAL(p2)=0;
конец
Конец Balance_R;
```

Метод работы аналогичен алгоритму Balance_L.

ТЕКСТЫ ПРОЦЕДУР Delete 1, 0 Del 1, 0Balance_L и Balance_R.

Так как процедуры Del, Balance_L и Balance_R используются только процедурой Delete, то их можно выполнить вложенными в Delete.

```
{=====Программный пример 6.20 ========}
Procedure Delete (x:integer; var p,root:ref; var h:boolean);
  var q:ref;      {h:false}
procedure Balance_L ( var p:ref; var h:boolean);
   {уменьшается высота левого поддерева}
 var  p1,p2:ref;
      b1,b2:-1..+1;
begin          { h-true, левая ветвь стала короче }
  case p^.BAL of
  -1: p^.BAL:=0;
   0: begin p^.BAL:=+1; h:=false; end;
   1: begin {балансировка}
       p1:=p^.right;  b1:=p1^.BAL;
    if b1 >= 0 then  begin  { однократный RR-поворот }
       if p=root then  root:=p^.right; p^.right:=p1^.left;
         p1^.left:=p;
          if b1 = 0 then  begin
            p^.BAL:=+1; p1^.BAL:=-1; h:=false; end
```

```
            else begin  p^.BAL:=0; p1^.BAL:=0; end;
           p:=p1;
          end
        else begin { 2-кратный RL-поворот }
           if p1=root then  root:=p1^.right; p2:=p1^.left;
           b2:=p2^.BAL; p1^.left:=p2^.right; p2^.right:=p1;
           p^.right:=p2^.left; p2^.left:=p;
            if b2=+1 then p^.BAL:=-1 else p^.BAL:=0;
            if b2=-1 then p1^.BAL:=+1 else p1^.BAL:=0;
           p:=p2; p2^.BAL:=0; end;
      end; { begin 3 }
  end;  { case }
end; {Balance_L}
procedure Balance_R (var p:ref; var h:boolean);
           { уменьшается высота правого поддерева }
  var  p1,p2:ref;
      b1,b2:-1..+1;
begin         { h-true, правая ветвь стала короче }
  case p^.BAL of
    1: p^.BAL:=0;
    0: begin p^.BAL:=-1; h:=false; end;
   -1: begin  { балансировка }
        p1:=p^.left;  b1:=p1^.BAL;
        if b1 <= 0   then begin  { однократный LL-поворот }
           if p=root then  root:=p^.left;
           p^.left:=p1^.right; p1^.right:=p;
           if b1 = 0
            then begin p^.BAL:=-1; p1^.BAL:=+1; h:=false;  end
            else begin  p^.BAL:=0; p1^.BAL:=0; end;
           p:=p1;
          end
        else begin { 2-кратный LR-поворот }
           if p1=root then  root:=p1^.left;
           p2:=p1^.right; b2:=p2^.BAL;
           p1^.right:=p2^.left; p2^.left:=p1;
           p^.left:=p2^.right;  p2^.right:=p;
            if b2=-1 then p^.BAL:=+1  else p^.BAL:=0;
            if b2=+1 then p1^.BAL:=-1 else p1^.BAL:=0;
           p:=p2;  p2^.BAL:=0;
         end;   end;    end;
end; {Balance_R}
Procedure Del (var r:ref; var h:boolean);
begin   { h-false }
  if r^.right <> nil   then
   begin  Del(r^.right,h);
```

```
      if h then Balance_R(r,h);
    end  else begin    q^.key:=r^.key; r:=r^.left; _  .h:=true; end;
end;{Del}
Begin          {Delete}
 if p=nil
  then begin TextColor(white);  GotoXY(a,b+2);
write ('Ключа в дереве нет'); h:=false;  end
  else    if x < p^.key
    then begin Delete(x,p^.left,root,h);
     if h then Balance_L(p,h);   end
    else   if x > p^.key  then
       begin  Delete(x,p^.right,root,h);
       if h then Balance_R(p,h);   end
     else begin  { удаление p^ }
        q:=p; if q^.right=nil
         then begin  p:=q^.left; h:=true;  end
         else  if q^.left=nil  then
             begin  p:=q^.right; h:=true; end
           else begin Del(q^.left,h);
            if h then Balance_L(p,h);
           end;
        GotoXY(a,b);
        writeln(' Узел с ключом ',x,' удален из дерева.');
end;
End{Delete};
```

ПОИСК ЭЛЕМЕНТА. Поиск элемента в сбалансированном дереве уже применялся в операциях вставки и удаления элементов. Поэтому необходимо отдельно рассмотреть эту операцию.

Пусть дано некоторое бинарное дерево, в котором каждый левый элемент меньше вышележащего, а правый - больше.

Для нахождения элемента с заданным ключом начинаем поиск с корневого элемента, сравнивая его ключ с искомым. Если искомый ключ меньше, то продолжаем поиск по левому поддереву (так как его элемент меньше текущего), а если ключ больше - то по правому (его элемент больше). Сравнивая аналогичным образом искомый ключ с ключом текущего элемента мы будем последовательно спускаться по дереву до тех пор, пока ключи искомого и текущего элемента не совпадут - элемент найден. Если мы дошли до уровня листьев (ниже элементов уже нет), а элемент не найден, значит он отсутствует в дереве.

Этот алгоритм пригоден для поиска в любых бинарных деревьях, но при работе со сбалансированными деревьями время поиска элемента минимально.

АЛГОРИТМ Search. Дано: ключ - X.

Алгоритм производит рекурсивный обход сбалансированного дерева и находит элемент с заданным ключом, либо сообщает об отсутствии такого элемента.

1. Поиск элемента:
Если x < key(p)
то: если p=nil
то: напечатать "Элемент отсутствует" и конец.
иначе: p=LPTR(p) и вызвать процедуру Search;
Если x > key(p)
то: если p=nil
то: напечатать "Элемент отсутствует" и конец.
иначе: p=RPTR(p) и вызвать процедуру Search;
2. Совпадение:
Напечатать "Элемент найден";
Произвести операции обработки элемента и конец.

ТЕКСТ ПРОЦЕДУРЫ Search.

```
{=====Программный пример 6.21 ===========}
Procedure Search (x:integer; var p:ref);
begin
  if x > p^.key then
    if p=nil  then writeln('Элемент на найден')
            else Search(x,p^.right);
  if x < p^.key then
    if p=nil  then writeln('Элемент на найден')
            else Search(x,p^.left);
  writeln('элемент найден');
  { Здесь - процедура обработки элемента }
end;
```

Так как операция поиска применяется в процедуре вставки элемента в сбалансированное дерево, то нет особого смысла выделять эту операцию в отдельную процедуру. Проще предусмотреть, что при отсутствии элемента производится его включение, а если элемент уже есть - то производятся необходимые действия над элементом.

На первый взгляд работа со сбалансированными бинарными деревьями требует лишних затрат времени на дополнительные операции по поддержанию необходимой структуры дерева и усложнение алгоритмов.

Но на самом деле затраты на балансировку легко компенсируются минимальным временем поиска элементов при вставке, удалении и обработке элементов, по сравнению с другими методами хранения. В то же время четкая структура расположения элементов не усложняет, а наоборот - упрощает алгоритмы работы с такими деревьями.

ОПИСАНИЕ ПРОГРАММЫ РАБОТЫ СО СБАЛАНСИРОВАННЫМИ ДЕРЕВЬЯМИ.

1. Процедура NOTE.

В процессе работы пользователя с программой MAVERIC выводит подсказку в нижней части экрана. Подсказка содержит коды клавиш,определяющих режим работы программы.

2. Процедура CREATE.

Создает новый узел дерева,в том числе и корень; записывает ключ дерева и обнуляет указатели узла на его ветви. Включает счетчик узлов и определяет корень дерева,путем установки на него указателя ROOT. Указатель ROOT устанавливается только в случае,если счетчик узлов дерева равен 0.

3. Процедура SEARCH.

Входным элементом для процедуры SEARCH является определяемый пользователем ключ для поиска или создания нового узла. Новый ключ сравнивается с ключом предыдущего узла. Если узла с таким ключом нет в дереве,то вызывается процедура CREATE.

В зависимости от того, больше или меньше ключ нового узла ключа узла предыдущего выбирается вид включения нового узла в дерево - справа или слева. На каждом этапе работы процедуры проверяется флаг "h" определяющий,увеличилась ли высота поддерева; а также проверяется поле "p^.bal" определяющее способ балансировки.

Процедура SEARCH является рекурсивной процедурой,т.е. она вызывает сама себя. При первом проходе процедура SEARCH обращается к корню дерева,затем проходит по всему дереву,последовательно вызывая ветви корня, затем ветви ветвей и так далее. В случае,если необходима балансировка, процедура SEARCH производит так называемые "повороты" ветвей дерева путем переопределения указателей. Если балансировка затрагивает корень дерева, процедура переопределяет корень,меняя указатель ROOT, а затем производит балансировку.

4. Процедура DELETE.

Процедура DELETE удаляет ключ из дерева и,если необходимо, производит балансировку. Входным параметром является определяемый пользователем ключ. Процедура DELETE имеет три подпроцедуры: balance_R, balance_L и Del. Подпроцедуры balance_R и balance_L являются симметричными и выполняют балансировку при уменьшении высоты правого или левого поддеревьев соответственно.

Если узла с заданным пользователем ключом нет в дереве, то выводится соответствующее сообщение. Если данный ключ меньше ключа предыдущего узла, то происходит рекурсивный вызов процедуры Delete и обход дерева по левой ветви. Если возникает необходимость балансировки, то вызывается подпроцедура balance_L. Если заданный пользователем ключ больше ключа предыдущего узла, то производится обход дерева по правой ветви и в случае необходимости балансировки вызывается подпроцедура balance_R.

Если подпроцедуры балансировки затрагивают корень дерева, то меняется указатель на корень дерева - ROOT. Эта операция заложена в обоих подпроцедурах balance_R и balance_L.

При обнаружении узла с заданным пользователем ключом подпроцедура Del производит операцию удаления данного узла.

5. Процедура OUTTREE.

Рекурсивная процедура OutTree выводит изображение дерева намонитор. Входными параметрами является указатель на корень дерева ROOT и переменная F определяющая,является ли текущий узел корнем или правой или левой ветвью.

После каждой операции над деревом процедура OutTree выводит изображение дерева заново,предварительно очистив экран.

6. Основная программа.

Программа Maveric работает в текстовом режиме,для чего в начале инициализируется модуль CRT. Основная программа выводит заставку и ожидает нажатия одной из определенных в программе клавиш.

При помощи процедуры Note внизу экрана выводится подсказка со списком определенных клавиш и соответствующих им операций.При нажатии клавиши B вызывается процедура Create,при нажатии клавиши S вызывается процедура Search, при нажатии D - процедура Delete. Программа работает в диалоговом режиме.

Режим работы с пользователем прекращается при нажатии клавиши ESC

.

```
{======Программный  пример 6.22 ====== }
Program Maveric;
USES CRT;
label L1,L2,L3,L4;
TYPE ref=^node;          { указатель на узел }
   node=record
         key:integer; { ключ узла }
         left,right:ref;    { указатели на ветви }
         bal:-1..+1;  { флаг сбалансированности }
         end;
VAR  root,            { указатель на корень дерева }
     p:ref;        { новое дерево }
     x:integer;    { ключ узла }
     h:boolean;    { true-высота поддерева увеличилась }
     n:char;       { клавиша подсказки }
     Ta,Tb,          { координаты начала вывода дерева }
     a,b:integer;    { координаты вывода подсказки }
    count:byte;      { счетчик узлов дерева }
 Procedure Note;        { процедура вывода подсказки }
  Begin
    TextBackground (white);  GotoXY(5,25);  textcolor(black);
```

```
      write('B-новое дерево     S-поиск по ключу   ');
      write (' D-удаление по ключу    Esc-выход');
End;
Procedure Create (x:integer; var p:ref; var h:boolean);
                    { создание нового дерева }
 Begin
      NEW(p);    h:=true;
      with p^ do
          begin    key:=x;
                 left:=nil; right:=nil;  bal:=0;
          end;
      if count=0 then root:=p;
        count:=count+1;
 End;
Procedure Search(x:integer; var p,root:ref; var h:boolean);
 var    p1,p2:ref;    {h=false}
 Begin
  if p=nil then  Create(x,p,h) {слова нет в дереве,включить его}
    else  if  x < p^.key  then
  begin  Search(x,p^.left,root,h);
        if h then          {выросла левая ветвь}
        case p^.bal of
        1: begin p^.bal:=0; h:=false; end;
        0: p^.bal:=-1;
       -1: begin           {балансировка}
           if p=root then  root:=p^.left;
                    {смена указателя на вершину}
             p1:=p^.left;
           if p1^.bal=-1 then
           begin      {однократный LL-поворот}
              p^.left:=p1^.right; p1^.right:=p;
              p^.bal:=0; p:=p1;
           end  else
           begin      {2-х кратный LR-поворот}
             if p1=root then  root:=p1^.right;
             p2:=p1^.right;
             p1^.right:=p2^.left;   p2^.left:=p1;
             p^.left:=p2^.right;    p2^.right:=p;
             if p2^.bal=-1 then p^.bal:=+1 else p^.bal:=0;
             if p2^.bal=+1 then p1^.bal:=-1 else p1^.bal:=0;
               p:=p2;
           end;    p^.bal:=0; h:=false;
       end;   end;
  end  else  if x > p^.key  then
      begin  Search(x,p^.right,root,h);
```

```
          if h then          {выросла правая ветвь}
          case p^.bal of
         -1: begin p^.bal:=0; h:=false; end;
          0: p^.bal:=+1;
          1: begin           {балансировка}
                if p=root then root:=p^.right;
                        {смена указателя на вершину}
                  p1:=p^.right;
                if p1^.bal=+1 then
                begin      {однократный RR-поворот}
                   p^.right:=p1^.left; p1^.left:=p;
                   p^.bal:=0; p:=p1;    end
    else  begin      {2-х кратный  RL-поворот}
       if p1=root then  root:=p1^.left;
       p2:=p1^.left;   p1^.left:=p2^.right;   p2^.right:=p1;
       p^.right:=p2^.left;    p2^.left:=p;
       if p2^.bal=+1 then p^.bal:=-1 else p^.bal:=0;
       if p2^.bal=-1 then p1^.bal:=+1 else p1^.bal:=0;
       p:=p2;  end;
       p^.bal:=0; h:=false;
  end;   end;  end;
  End {Search};
Procedure  Delete (x:integer; var p,root:ref; var h:boolean);
 var q:ref;      {h:false}
    procedure balance_L ( var p:ref; var h:boolean);
     {уменьшается высота левого поддерева}
     var  p1,p2:ref;
        b1,b2:-1..+1;
     begin    {h-true,левая ветвь стала короче}
        case p^.bal of
        -1: p^.bal:=0;
         0: begin p^.bal:=+1; h:=false; end;
         1: begin {балансировка}
           p1:=p^.right;  b1:=p1^.bal;
           if b1 >= 0 then
             begin  {однократный RR-поворот}
              if p=root then  root:=p^.left;
                p^.right:=p1^.left;  p1^.left:=p;
              if b1 = 0 then
                begin p^.bal:=+1; p1^.bal:=-1; h:=false;
                end   else
                     begin p^.bal:=0; p1^.bal:=0;  end;
                   p:=p1;
                end  else
            begin {2-х кратный RL-поворот}
```

```
              if p1=root then  root:=p1^.left;
              p2:=p1^.left;  b2:=p2^.bal;
              p1^.left:=p2^.right; p2^.right:=p1;
              p^.right:=p2^.left; p2^.left:=p;
            if b2=+1 then p^.bal:=-1 else p^.bal:=0;
            if b2=-1 then p1^.bal:=+1 else p1^.bal:=0;
              p:=p2;  p2^.bal:=0;
     end;  end;   end;
 end; {balance_L}
procedure balance_R (var p:ref; var h:boolean);
 {уменьшается высота правого поддерева}
 var  p1,p2:ref;
  b1,b2:-1..+1;
  begin    {h-true,правая ветвь стала короче}
     case p^.bal of
      1: p^.bal:=0;
      0: begin p^.bal:=-1; h:=false; end;
     -1: begin {балансировка}
            p1:=p^.left;  b1:=p1^.bal;
          if b1 <= 0 then
          begin  {однократный LL-поворот}
           if p=root then  root:=p^.right;
             p^.left:=p1^.right;  p1^.right:=p;
           if b1 = 0 then
             begin p^.bal:=-1; p1^.bal:=+1; h:=false;
             end   else
             begin p^.bal:=0; p1^.bal:=0;
             end;
             p:=p1;
       end  else  begin {2-х кратный LR-поворот}
              if p1=root then  root:=p1^.right;
              p2:=p1^.right;  b2:=p2^.bal;
              p1^.right:=p2^.left; p2^.left:=p1;
              p^.left:=p2^.right; p2^.right:=p;
            if b2=-1 then p^.bal:=+1 else p^.bal:=0;
            if b2=+1 then p1^.bal:=-1 else p1^.bal:=0;
            p:=p2;  p2^.bal:=0;
          end;  end;   end;
 end; {balance_R}
Procedure Del (var r:ref; var h:boolean);
 begin {h-false}
    if r^.right <> nil then
    begin  Del(r^.right,h); if h then balance_R(r,h);
    end    else
         begin q^.key:=r^.key;
```

```
                    r:=r^.left;    h:=true;    end;
      end;{Del}
  Begin       {Delete}
     if p=nil then
     begin TextColor(white);  GotoXY(a,b+2);
          write ('Ключа в дереве нет'); h:=false;
    end   else  if x < p^.key then
    begin Delete(x,p^.left,root,h); if h then balance_L(p,h);
          end   else  if x > p^.key then
    begin Delete(x,p^.right,root,h); if h then balance_R(p,h);
          end   else  begin {удаление p^} q:=p;
                      if q^.right=nil then
                        begin p:=q^.left; h:=true;
                        end   else
                      if q^.left=nil then
                        begin p:=q^.right; h:=true;
                        end   else
                      begin  Del(q^.left,h);
                           if h then balance_L(p,h);
                      end;
                      {dispose(q);}
               GotoXY(a,b);
             writeln('Узел с ключом ',x,' удален из дерева.');
             end;
  End{Delete};
  Procedure OutTree(pr:ref;f:byte);
   Begin
       Tb:=Tb+2;
       If f=1 then  Ta:=Ta-2;
       if f=2 then  Ta:=Ta+8;
       if pr<>nil  then   begin   GotoXY(TA,TB);
           case f of
                    0: Writeln('[',pr^.key,']');
                    1: Writeln('[',pr^.key,']/');
                    2: Writeln('\[',pr^.key,']');
           end;
           OutTree(pr^.left,1);    OutTree(pr^.right,2);
         end;
    Tb:=Tb-2;   Ta:=Ta-2;
   End;       {OutTree}
BEGIN         {main program}
L4:
     count:=0;  a:=25;    b:=5;
     TextBackground(black); ClrScr;
     TextBackground (red);  gotoxy(a,b);
```

```
    textcolor(white);  writeln(' WELCOME TO THE LAND ');
    gotoxy(a,b+1);     WRITE(' OF BALANCED TREES ');
    while n <> #27 do
    begin  note;  n:=readkey;
       case n  of
              #66: goto L1; {'B'}
              #68: goto L3; {'D'}
              #83: goto L2; {'S'}
              #98: begin    {'b'}
       L1:     clrscr;  TextBackground (green);  gotoxy(a,b);
             writeln ('Введите ключ для нового дерева');
             gotoxy(a+32,b);   read(x);   Create(x,p,h);
              end;
              #115: begin   {'s'}
       L2:          ClrScr;
            TextBackground (blue);   gotoxy(a,b);
            TextColor(white);
            writeln('Введите ключ для поиска и включения');
            gotoxy(a+40,b);  read(x);
            Search(x,p,root,h);   Ta:=26;   Tb:=10;
            OutTree(root,0);   end;
               #100: begin   {'d'}
       L3:   ClrScr;   TextBackground (yellow);
            gotoxy(a,b);   TextColor(black);
            writeln('Введите ключ для удаления узла');
            gotoxy(a+32,b);  read(x);
            Delete(x,p,root,h);
             Ta:=26;   Tb:=10;   OutTree(root,0);
      end;    end;    end;
    Dispose(p);  ClrScr;     TextBackground (red);
    GotoXY(a,b); TextColor(white);
    writeln('Are you sure?  Yes/No');
    GotoXY (a+23,b);   n:=readkey;
    if (n=#78) or (n=#110) then goto L4;
END.  {main program}
```

ЛИТЕРАТУРА

1. Агафонов В.Н. Типы и абстракция данных в языках программирования. - М.: Мир, 1982. С. 265-327.
2. Ахо А., Холкрофт Дж., Ульман Дж. Построение и анализ вычислительных алгортимов. - М .М ир, 1979. - 536 с.
3. Баррон Д. Введение в языки программирования. - М.: Мир, 1980. - 190 с.
4. Брукс Ф.П. Как проектируются и создаются программные комплексы. - М.: Наука, 1979
5. Виноградов М.М. Модели данных и отображения моделей данных : алгебраический подход // Теория и приложения систем баз данных,- М.: ЦЭМИ АП СССР, 1984. - С. 26-40.
6. Вирт Н. Систематическое программирование. Введение - М.: Мир, 1977. - 184 с.
7. Вирт Н. Алгоритмы + структуры данных = программы. - М.: Мир, 1985. - 406 с.
8. Вирт Н. Алгоритмы и структуры данных. - М.: Мир, 1989. - 360 с.
9. Гудман С., Хидетниеми С. Введение в разработку и анализ алгоритмов. - М.: Мир, 1981. - 368 с.
10. Данные в языках программирования. Под ред. В. Н. Агафонова. - М.: Мир, 1982. - 328 с.
11. Джордейн Р. Справочник программиста персональных компьютеров типа IBM PC, XT и АП. - М.: Финансы и статистика, 1992. - 544 с.
12. Джонс Ж., Харроу К. Решение задач в системе Турбо-Паскаль. - М.: Финансы и статистика, 1991. - 718 с.
13. Дрибас В.П. Реляционные модели баз данных. - Минск: Изд. БГУ, 1982. - 192 с.
14. Зайцев В.Ф. Кодирование информации в ЕС ЭВМ. - М.: Радио и связь, 1986. - 102 с.
15. Замулин А.В. Система программирования баз данных и знаний. - Новосибирск: Наука, 990. - 32 с.
16. Замулин А.В. Типы данных в языках программирования и базах данных. - Новосибирск: Наука, 1987. - 150 с.
17. Керниган Б., Ритчи Д. Язык программирования Си. - М.: Финансы и статистика, 1992. - 271 с.
18. Кнут Д. Искусство программирования для ЭВМ. т.1. Основные алгоритмы. - М.: Мир, 1976. - 735 с.
19. Кнут Д. Искусство программирования для ЭВМ. т.З. Сортировка и поиск. - М.:Мир, 1976. - 797 с.
20. Костин Е.Е., Шаньгин В.Ф. Организация и обработка структур данных в вычислительных системах. - М.: Высш. школа, 1987. - 242 с.
21. Елеман Д., Смит М. Типы данных // Данные в языках программирования. - М.: Мир, 1982. С.196-213.

22. Ленгсам Й., Огенстайн М., Тененбаум А. Структуры данных для персональных ЭВМ. - М.: Мир, 1989. - 568 с.
23. Макаровский Б.Н. Информационные системы и структуры данных. Учебное пособие вузов. - М.: Статистика, 1980. - 190 с.
24. Морс С.П., Альберт Д.Д. Архитектура микропроцессора 80286. - М.: Радио и связь, 1990. - 300 с.
25. Нагао М., Катаяма Т., Уэмура. Структуры и базы данных. - М.: Мир, 1984.
26. Пратт Т. Языки программирования. Разработка и реализация. - М.: Мир, 1979. - 574 с.
27. Разумов О.С. Организация данных в вычислительных системах. - М.: Статистика, 1978. - 184 с.
28. Трамбле Ж., Соренсон П. Введение в структуры данных. - М.: Машиностроение, 1982. - 784 с.
29. Трофимова И.П. Системы обработки и хранения информации. - М.: Высш. школа, 1989. - 191 с.
30. Уоркли Дж. Архитектура и программное обеспечение микро ЭВМ. Том 1. Структуры данных. - М.: Мир, 1984.
31. Флорес И. Структуры и управление данными. - М.: Радио и связь, 1982. - 118 с.

32. Фостер Дж. Обработка списков. - М.: Мир, 1974. - 72 с.
33. Холл П. Вычислительные структуры (Введение в нечисленное программирование). - М.: Мир, 1978. - 214 с.
34. Хоор К. О структурной организации данных // Структурное программирование, - М.: Мир, 1975. С. 98-197.

Учебное издание

Еленев Валерий Дмитриевич,
Гоголев Михаил Юрьевич

АЛГОРИТМИЧЕСКИЕ ЯЗЫКИ И ТЕХНОЛОГИИ ПРОГРАММИРОВАНИЯ НА ЯЗЫКАХ ВЫСОКОГО УРОВНЯ

Курс лекций

В авторской редакции

Подписано в печать 12.07.2010. Формат 60x84 1/16.
Бумага офсетная. Печать офсетная. Печ. л. 14,0.
Тираж 50 экз. Заказ .

Самарский государственный
аэрокосмический университет.
443086, Самара, Московское шоссе, 34.

Изд-во Самарского государственного
аэрокосмического университета.
443086, Самара, Московское шоссе, 34.