

# Лекция 1.

## Данные и способы их представления в ЭВМ

### **Машинно-независимые типы данных**

Под типом данных понимается характеристика переменной, определяющая ее размер и способ представления в памяти ЭВМ, а также операции, возможные с данной переменной.

#### Пример.

Операция умножения возможна только с числовыми типами данных, а конкатенация – только со строковыми переменными.

По сути, тип данных ограничивает возможности по работе с переменной, но использование типов позволяет проще работать с областями памяти ЭВМ.

Необходимо отметить, что сами типы данных в том понимании, в котором их принято употреблять, являются лишь абстракцией уровня компилятора. Сам компьютер работает лишь с цифрами в двоичной системе счисления. В связи с этим можно определить машинно-независимые типы данных.

Характеристиками машинно-независимых типов данных являются:

1. независимость данных от конкретного языка программирования;
2. независимость данных от архитектуры конкретной ЭВМ.

Дальнейшее изложение материала проведем по схеме сопоставления машинно-независимых, или абстрактных, типов данных с их реальными аналогами языка Pascal.

Первыми исторически появились числовые данные.

### **Числовые типы данных**

В зависимости от поставленной задачи программисту могут потребоваться либо целочисленные переменные, понятие которых эквивалентно понятию целых чисел в математике, либо вещественные переменные, соответствующие рациональным числам. Это обусловлено тем, что любое представление числа в памяти ЭВМ конечно. После окончания числа подразумевается, что все остальные цифры числа равны нулю. Таким образом, мы получаем рациональное число, имеющее в периоде 0. Разделяют два способа представления вещественных переменных – с фиксированной запятой и с плавающей запятой.

### **Целые типы**

Целые числа в памяти ЭВМ обычно представляются в двоичной системе счисления в виде знаковых или беззнаковых величин. Состояние числа определяется флагом SF центрального процессора (для Intel x86).

В знаковых целых числах в качестве признака знака «минус» используется старший бит числа.

#### Пример.

Рассмотрим однобайтный код:

00001010 – вне зависимости от SF это число 10.

10001010 – если SF равен 0, это число 266, если SF равен 1, это число -10.

Для осуществления операций сложения и вычитания целых чисел вводится понятие дополнительного кода. Дополнительный код отрицательного числа формируется по следующим правилам:

1. записать модуль отрицательного числа в прямом коде, помещая в неиспользуемые старшие биты нули;
2. сформировать обратный код числа, инвертировав его биты;
3. прибавить единицу к обратному коду числа.

#### Пример.

Получить дополнительный код однобайтного целого числа -15.

00001111 – прямой код модуля числа

11110000 – обратный код

11110001 – прибавление единицы и получение дополнительного кода

Для положительных чисел дополнительный код равен прямому.

С получившемся числом в дополнительном коде производят арифметическое сложение, причем при переполнении старший разряд теряется, что позволяет сохранить отношение числа к нулю.

После выполнения операции сложения необходимо восстановить число по правилам:

- если старший разряд числа содержит 0, то оставить число без изменения;
- если старший разряд числа содержит 1, то преобразовать число из дополнительного кода в прямой, выполнив действия:
  1. отнять единицу от числа;
  2. инвертировать биты числа;
  3. записать единицу в старший разряд.

#### Пример.

Сложить однобайтовые целые числа -15 и -14.

Дополнительный код числа -15: 11110001.

Дополнительный код числа -14: 11110010.

После сложения: 11100010.

После перевода: 10011101, что соответствует числу -29.

Таблица 0.1 содержит целые типы языка Pascal. Каждый тип имеет две основные характеристики: размер числа и признак знака. При сравнении типов shortint и byte хорошо видно различие между знаковыми и беззнаковыми типами одинакового размера.

При n-битовом хранении целого числа диапазон значений для беззнаковых чисел равен  $[0..2^n-1]$ , для знаковых чисел –  $[-2^{n-1}..2^{n-1}-1]$ .

**Таблица 0.1**

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

#### **Вещественные типы**

Вещественное число в памяти ЭВМ может быть представлено как:

Знак числа	Порядок	Знак порядка	Мантисса
------------	---------	--------------	----------

Чаще вместо порядка используется характеристика, получаемая прибавлением к порядку такого смещения, чтобы характеристика была всегда положительной. В этом случае формат записи вещественного числа будет выглядеть так:

Знак числа	Характеристика	Мантисса
------------	----------------	----------

Характеристика вещественного числа вычисляется по формуле:

$$X=2^{n-1}+k+p, \quad (1.1)$$

где n – число бит, отводимых на характеристику;

p – порядок числа;

k – поправочный коэффициент фирмы IBM, равный +1 для real и –1 для single, double, extended.

Для увеличения количества значащих цифр и исключения переполнения при умножении мантиссы (M) числа подвергают нормализации. Нормализованная мантисса кроме случая, когда  $M=0$ , лежит в интервале  $R^{-1} \leq M < 1$ , где R-основание системы счисления.

В компьютерах IBM PC нормализованная мантисса принадлежит интервалу  $1 \leq M < 2$  и содержит свой старший бит слева от точки. Для данных типа real, single, double этот бит не хранится и используется для увеличения порядка в формате single или для хранения знака в формате real.

Алгоритм формирования машинного представления вещественного числа в памяти ЭВМ:

1. Представить число в двоичном коде;
2. Нормализовать число;
3. Определить характеристику числа по формуле (1.1);
4. Записать число в отведенное место памяти с учетом условий:
  - для чисел типа real характеристика храниться в младшем байте памяти, для чисел типа single, double, extended – в старшем;
  - 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 real, single, double не храниться (является скрытой).

Таблица 0.2

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

Пример.

Записать машинное представление десятичного числа 15.375 в формате real.

1. представляем число в двоичной системе счисления: 1111.011
2. нормализуем число  $1.111011 \cdot 2^3$ ,  $p=3$ ;
3.  $X = 2^7 + 1 + 3 = 132$ ,  $S=0$ ;
4. в двоичной системе счисления с учетом сдвига порядка и отбрасывания неявной единицы:  $X=10000100$ ,  $M=1110110\dots 0$ ;

мл.байт					ст.байт
10000100	00000000	00000000	00000000	00000000	01110110
X..X	M..M	M..M	M..M	M..M	SM..M

### Битовые типы данных

Ряд задач программирования требует работы с отдельными двоичными разрядами данных. 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 типу данных.

Над битовыми типами возможны три группы операций: операции булевой алгебры, операции сдвигов и операции сравнения.

К операциям булевой алгебры относятся: НЕ(not), ИЛИ(or), И(and), исключающее ИЛИ(xor).

Операции сдвига выполняют смещение двоичного кода на заданное количество разрядов влево или вправо. Возможен арифметический, логический и циклический сдвиг.

В операциях сравнения битовые данные интерпретируются как целые без знака и сравниваются как целые числа.

Пример.

Логические операции над битовыми данными.

x=11010101;

y=10101001.

not x = 00101010,

x or y = 11111101,

x and y = 10000001,

x xor y = 01111100.

В языке Pascal роль битовых типов выполняют беззнаковые целые byte и word, над которыми определены побитовые операции (not, or, and, xor, shr, shl).

### **Символьный тип**

Значением символьного типа являются символы из некоторого predetermined множества. В большинстве современных ЭВМ этим множеством является ASCII (American Standard Code for Information Interchange). Это множество состоит из 256 упорядоченных символов. На каждый символ множества в машине отводится 1 байт.

Пример.

Символ «1» имеет ASCII код 49. Его машинное представление 00110001.

Первые 127 кодов ASCII включают в себя символы латинского алфавита и некоторые специальные символы. Символы национальных алфавитов занимают «свободные места» в таблице кодов. Одна таблица может поддерживать только один национальный алфавит. Поэтому была создана кодировка UNICODE, где каждый символ кодируется двумя байтами. Это обеспечивает более  $2^{16}$  возможных кодовых комбинаций и позволяет хранить в одной таблице кодов все национальные алфавиты.

В языке Pascal реализацией символьного типа является тип char, представляющий собой однобайтный код символа.

Расширением символьного типа являются строки.

### **Строки**

Можно выделить следующие типы строковых данных:

- Строки фиксированной длины;
- Строки с конечным символом.

При хранении строк с фиксированной длиной длина строки хранится в первом байте (тип string языка Pascal). Это накладывает ограничение на длину строки в 255 символов.

При втором способе представления строки, ее длина ограничена специальным символом, символизирующим конец строки (обычно символ с кодом 0). Для работы с этими строками необходимо выделить память. В языке Pascal данные строки реализуются типом pchar.

### **Массивы**

Массивом называется структура данных, которая характеризуется:

1. фиксированным набором элементов одного типа;
2. каждый элемент обладает уникальным набором значений индексов;
3. количество индексов определяет размерность массива: два индекса – двумерный массив, три индекса – трехмерный массив, один индекс – одномерный массив (вектор);

4. обращение к элементу массива выполняется по имени массива и значениям индексов для данного элемента.

Массивы хранятся в непрерывной области памяти. Принцип распределения элементов определяется языком программирования. В языке Pascal элементы распределены по строкам.

Любое представление структуры массива заключается в отображении массива с компонентами типа T на память, которая представляет собой массив с компонентами типа WORD.

Массив следует отображать так, чтобы вычисление адреса его компоненты было как можно проще и эффективно.

Например, при хранении двумерного массива по последовательно распределенным адресам память выделяется так, чтобы

$$\text{АДРЕС}(A[J,K]) = a_0 + a_1J + a_2K, \quad (1.2)$$

где  $a_0, a_1, a_2$  – константы.

Вообще, k-мерный массив с элементами  $A[I_1, I_2, \dots, I_k]$ , длина которых равна с слов при

$$0 \leq I_1 \leq d_1, \quad 0 \leq I_2 \leq d_2, \dots, 0 \leq I_k \leq d_k$$

может храниться в памяти в виде

$$\begin{aligned} \text{АДРЕС}(A[I_1, I_2, \dots, I_k]) &= \text{АДРЕС}(A[0, 0, \dots, 0]) + c(d_2 + 1) \dots (d_k + 1)I_1 + \dots + c(d_k + 1)I_{k-1} + cI_k = \\ &= \text{АДРЕС}(A[0, 0, \dots, 0]) + \sum_{1 \leq r \leq k} a_r I_r, \end{aligned} \quad (1.3)$$

$$\text{где } a_r = c \prod_{r \leq s \leq k} (d_s + 1) \quad (1.4)$$

Пример.

**Type**

A = **array**[5..10, 2..8] of byte;

## **Записи**

Запись – совокупность полей разного типа данных, обращение к которым возможно по имени поля. Записи являются удобным средством для представления программных моделей реальных объектов предметной области, ибо каждый объект обладает набором свойств, характеризующихся данными различных типов.

Пример.

Запись – совокупность сведений о некотором студенте.

Объект «студент» обладает свойствами:

«личный номер» – характеризуется целым положительным числом;

«ФИО» – характеризуется строкой символов.

«курс» – характеризуется целым положительным числом;

«группа» – характеризуется строкой символов.

При обработке данных составные типы данных, такие как записи, используются в файлах или банках данных. Доступ к полям осуществляется по помощи оператора доступа (в Pascal – точка).

Существует два способа представления записи в памяти ЭВМ:

- в виде последовательности полей, занимающих непрерывную область памяти;
- В виде связного списка с указателями на значения полей записи.

При первом способе достаточно иметь указатель на начало области записи и смещение относительно начала. Этот способ дает экономию памяти, но тратит время на вычисление адресов полей записи.

Во втором случае память тратится на хранение указателей на поля записи, но доступ к значениям полей осуществляется быстрее.

Пример.

**Var**

```

Student : record
    Num:byte;
    Name:string[20];
    Kourse:byte;
    Group:string[5];

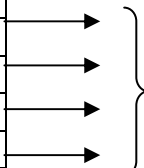
```

**End;**

1)

+0	+1	+22	+23
31	Иванов А.В.	2	BT21

2)

Rec	Student		4	
Byte	1	Num		
String	21	Name		
Byte	1	Kourse		
String	8	Group		

## Таблицы

Таблица – массив, элементами которого являются сложные структуры. Доступ к элементам таблицы производится не по индексу, а по ключу. Ключ – это свойство, идентифицирующее данный элемент во множестве однотипных элементов. Ключ может включаться в элемент таблицы отдельным полем, а может вычисляться по положению элемента.

Доступа к элементу по ключу реализуется процедурами поиска.

Пример.

**Type**

```

TRec=record
    Pole1:integer;
    Pole2:real;
    Pole3:string;
End;
Table=array[1..100] of TRec;

```

Широкое распространение таблицы и записи имеют в теории баз данных.

## Объекты

В целом строение объекта напоминает строение записи. Отличительной чертой объектов является объединение данных и методов их обработки в одну структуру. Поля объекта принято называть свойствами, а процедуры и функции объекта – методами. Характерными чертами объектно-ориентированного программирования являются так называемые:

- инкапсуляция
- наследование
- полиморфизм.

Инкапсуляция это и есть объединение данных и методов их обработки внутри объекта.

Наследование – это способность объекта – потомка иметь свойства и методы, аналогичные свойствам и методам объекта - родителя, без явного описания.

Полиморфизм – способность потомка изменять свойства и методы унаследованные от родителя. В Pascal'е полиморфизм распространяется только на методы.

Пример.

**Type**

```

Tobject=object
    Property1:Byte;

```

```

Property2:string;
Procedure Metod1;
Function Metod2(a:integer; var s:string):Boolean;
End;

Tobject2=object (Tobject)
Property3:integer;
Function Metod3:string;
End;

```

В настоящее время объектно-ориентированный подход применяется во всех сферах применения ЭВМ: базы данных, операционные системы, языки программирования.

### **Указатели**

Указателем называют переменную, содержащую в качестве своего значения адрес ячейки памяти. Указатели могут быть типизированными, т.е. указатель ссылается на блок памяти содержащий значение переменной заранее определенного типа. Нетипизированные указатели позволяют работать с переменными, тип которых заранее не известен.

Основными операциями с указателями являются: присваивание, получение адреса и выборка. Присваивание – бинарная операция, оба операнда которой – указатели. При присваивании происходит копирование адреса, но не самого значения хранящегося по адресу. Если указатели типизированные, то типы операндов должны совпадать. Получение адреса – унарная операция, операнд которой может иметь любой тип. Результатом является типизированный указатель содержащий адрес операнда. Операция выборки – унарная, ее операндом является обязательно типизированный указатель, результат – данные выбранные из памяти по адресу, заданному операндом.

#### Пример.

**Var**

```

A:^integer;
B:^integer;

```

**Begin**

```

New(A); {выделяем память под указатель A}
New(B); {выделяем память под указатель B}
A^:=10; {в ячейке, адресуемой A, хранится 10}
B^:=20; {в ячейке, адресуемой B, хранится 20}
B:=A; {A и B указывают на одну и ту же ячейку 10}
B^:=5; {A и B указывают на одну и ту же ячейку 5}
Dispose(A); {освобождение памяти, выделенной под указатель A}
Dispose(B); {ОШИБКА! Эта ячейка уже не существует. Ячейка со
значением 20 потеряна}

```

**End;**

### **Списки**

Списком называется сложная структура данных элементы которой расположены в памяти не последовательно, а в произвольном месте. Списки строятся на основе записи, одно или несколько полей которой – указатели на следующие элементы списка.

Списки удобны своей динамической структурой – в любой момент времени список занимает столько места в памяти, сколько у него имеется элементов в настоящий момент времени. Тогда как массив сразу занимает заранее отведенное место, и его размер не может быть изменен. А по статистике в среднем реально элементы массива занимают около 50 % памяти отведенной под массив. Что свидетельствует о не рациональном использовании памяти.

Различают следующие типы списков:

- Линейные
- Иерархические
- Ортогональные

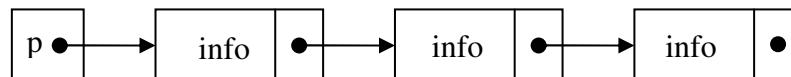
## Линейные списки

Линейной называют структуру при движении по элементам, по которой возможен лишь один единственный путь.

Пример.

Одномерный массив является линейной структурой, а сеть дорог не является линейной.

Односвязанный линейный список имеет вид



Его описание на языке Pascal может быть:

**Type**

```

POneList = ^TOnelist;
TOnelist = record
    Info: string;
    Next: PList;
End;
  
```

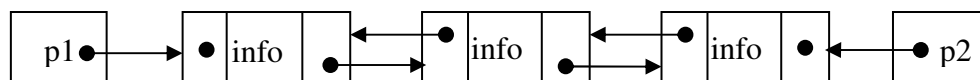
Преимущества такого списка:

- простота добавления элемента в начало списка
- простота удаления элемента из начала списка
- простота реализации поиска элемента

Недостатки:

- сложность работы с концом списка
- невозможность прямой адресации элементов

Двусвязанный линейный список характеризуется тем, что в каждом элементе есть ссылки на предыдущий и последующий элементы.



Его описание на языке Pascal может быть:

**Type**

```

PTwoList = ^TTwoList;
TTwoList = record
    Info: string;
    Prev: PTwoList;
    Next: PTwoList;
End;
  
```

Преимущества:

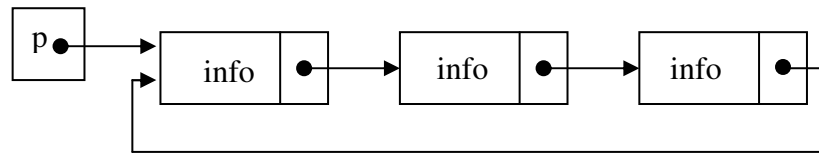
- упрощается работа со списком со второго конца
- по ссылке на любой элемент можно получить весь список

Недостатки:

- хранение «лишней» ссылки внутри поля
- необходимость постоянного согласования ссылки на «конец»
- последовательный доступ к элементам



Признаком кольцевого списка является то, что у последнего элемента ссылка на следующий указывает на начало списка. Тип элементов соответствует типу односвязанного списка.



Преимущества:

- с любого элемента списка можно получить доступ ко всем элементам.
- удобна организация циклических алгоритмов (ОС)

Недостатки:

- сложные операции добавления и удаления элемента
- удаление всего списка требует дополнительных действий

Кольцевой список также может быть основан на двухсвязанной структуре.

## Иерархические списки

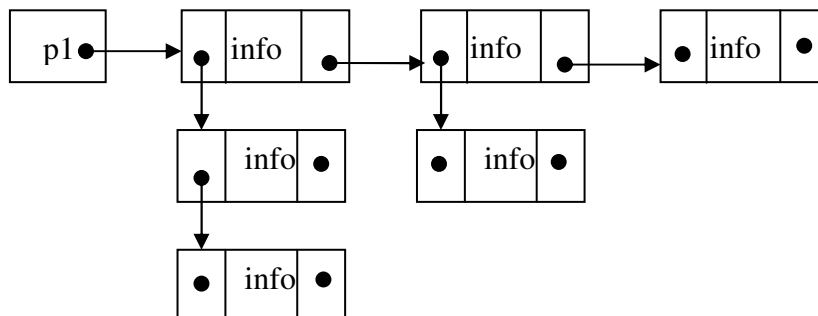
Если между элементами списка существует некая зависимость типа «ведущий-подчиненный» удобнее представлять эти элементы в виде иерархического списка. У каждого элемента списка имеется ссылка на следующий элемент и на «подчиненный» элемент.

**Type**

```

PIList = ^TIList;
TIList = record
    Info: string;
    Next: PIList;
    Sub: PIList;
  end;
  
```

**End;**



Такая структура позволяет организовать любую степень вложенности (размерность). Примером использования иерархических списков может служить система меню в приложениях.

## Ортогональные списки

Рассмотрим пример двумерного ортогонального списка. Для данного случая каждый элемент списка будет иметь две ссылки, и на каждый элемент списка будут ссылаться два элемента (за исключением головных, которые могут быть элементами массивов).

**Type**

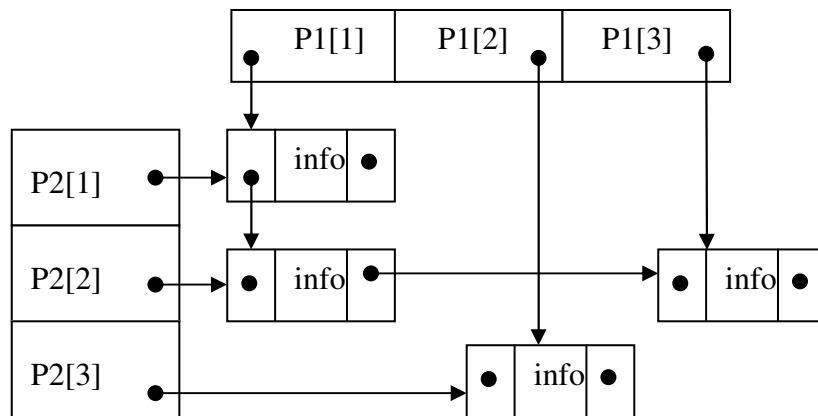
```

PList = ^TList;
TList = record
    Info: string;
    Left: PList;
  end;
  
```

```

Down:PList;
End;
Tarray:array [1..N] of PList;

```



Всвязи с таким представлением добавление и удаление элементов из списка осложняется необходимостью отслеживать связи и целостность списка.

Ортогональные списки используются для представления многомерных массивов, имеющих большое количество нулевых элементов (разряженных матриц).

## Графы

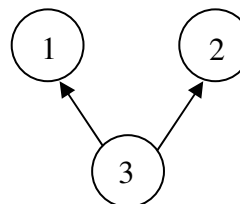
Графом называется структура, состоящая из двух множеств – множества вершин и множества ребер. Причем каждому ребру соответствует пара вершин, одна из которых называется истоком (из нее выходит ребро) другая – стоком (вершина в которую входит ребро).

Различают два вида графов: ориентированные и неориентированные. Ориентированные графы характеризуются

Графы являются математическими моделями транспортных и других сетей. Существует несколько способов представления графов и алгоритмов работы с ними. Наиболее распространенным способом представления графов является матрица смежности – двумерный массив, в котором каждой вершине соответствует один столбец и одна строка, а на пересечении строк и столбцов имеется признак наличия связи между ними.

Пример :

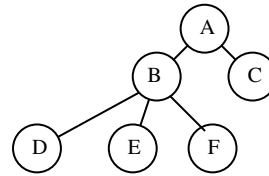
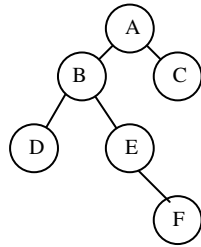
	1	2	3
1			
2			
3	1	1	



## Деревья

Наиболее интересным, с точки зрения практического применения является частный случай графов – деревья. Встречается много определений дерева. Но мы будем пользоваться следующим: дерево - это граф без циклов.

Различают два типа деревьев: бинарные и сильноветвящиеся. Бинарное дерево – дерево, у каждой вершины которого имеется не более двух исходящих ребер (ветвей) и не более одного входящего. Если это условие не выполняется, то дерево называется сильноветвящимся.



Подробнее деревья и их приложения рассмотрим ниже, а сейчас остановимся на представлении бинарных (двоичных) деревьев. Бинарное дерево представляет собой список, у каждого элемента которого существует две ссылки: на левое и правое поддерево.

### Type

```

PTree=^TTree;
TTree=record
    Info:string;
    Lt,Rt:PTree;
End;
```

## Другие типы данных

### Логический тип

Частным случаем битового типа является логический тип, множество значений которого состоит из 0 (false) и 1 (true). В языке Pascal данные логического типа занимают 1 байт, при этом значению false соответствует нулевое значение байта, а значению true – ненулевое.

Пример.

```

Var
    a:boolean;
Begin
    A:=5>7;
End.
```

### Множества

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.

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 байт.

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

$$ByteSize = (max \div 8) - (min \div 8) + 1, \quad (1.5)$$

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

Пример.

```

Var
    S1:set of Byte;
```

Если S1, S2, S3 множества, то над ними 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.

### Перечислимый тип

Перечислимый тип представляет собой упорядоченный тип данных, определяемый программистом, т.е. программист перечисляет все значения, которые может принимать переменная этого типа. Значения типа должны быть уникальными и их количество не может превышать 256.

Пример.

**Type**

```
color=(red,blue,green);
```

Для переменной перечислимого типа выделяется один байт, в который записывается порядковый номер присваиваемого значения. Порядковый номер определяется из описания типа, причем нумерация начинается с 0. Имена из списка перечислимого типа являются константами.

Пример.

**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.

### Типы диапазона

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.

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

Пример.

**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 - 1010111000001000.

# Лекция 2

## Понятие алгоритма. Виды алгоритмов.

### Понятие алгоритма

**Алгоритм** — точное и понятное предписание исполнителю совершить последовательность действий, направленных на решение поставленной задачи.

Название "алгоритм" произошло от латинской формы имени среднеазиатского математика аль-Хорезми — Algorithmi. Алгоритм — одно из основных понятий информатики и математики.

Дадим несколько определений.

**Исполнитель алгоритма** — это некоторая абстрактная или реальная система, способная выполнять действия, предписываемые алгоритмом (пример, процессор ЭВМ).

Исполнителя характеризуют:

- среда;
- элементарные действия;
- система команд;
- отказы.

**Среда** (или обстановка) — это "место обитания" исполнителя. Среда характеризуется начальными условиями задачи и состоянием исполнителя.

**Система команд** — некоторое строго заданное множество команд исполнителя (пример: система команд ЦП ЭВМ). Для каждой команды должны быть заданы условия применимости (в каких состояниях среды может быть выполнена команда) и описаны результаты выполнения команды.

После вызова команды исполнитель совершает соответствующее **элементарное действие**, т.е. осуществляет шаг алгоритма.

**Отказы** — состояние системы, возникающее, если команда вызывается при недопустимом для нее состоянии среды (ошибки, сбои).

Обычно исполнитель ничего не знает о цели алгоритма. Он выполняет все полученные команды, не задавая вопросов "почему" и "зачем".

### Свойства алгоритмов

**Понятность** для исполнителя - т.е. исполнитель алгоритма должен знать, как его выполнять. Иными словами, исполнитель может выполнять только команды системы команд исполнителя.

**Дискретность** - т.е. алгоритм должен представлять процесс решения задачи как последовательное выполнение простых (или ранее определенных) шагов (элементарных действий).

**Определенность** - т.е. каждое правило алгоритма должно быть четким, однозначным и не оставлять места для произвола. Благодаря этому свойству выполнение алгоритма носит механический характер и не требует никаких дополнительных указаний или сведений о решаемой задаче.

Пример.

Существует метод нахождения середины заданного отрезка при помощи циркуля:

1. Прочертить окружности радиуса большего, чем половина отрезка из левого и правого концов отрезка.
2. Провести прямую через точки пересечения этих окружностей.
3. Точка пересечения проведенной прямой и заданного отрезка будет являться серединой отрезка.

Описанный способ не является алгоритмом, т.к. в его первом шаге содержится неопределенность: не задана величина конкретного раствора циркуля. Правильной записью было бы: "прочертить окружности в полтора раза большего, чем половина отрезка, из левого и правого концов отрезка".

**Результативность** Выполнение алгоритма должно привести к получению определённых результатов решения соответствующей задачи, а если задача с каким-то набором исходных данных не имеет решения, то выдаётся соответствующий сигнал.

Пример.

Программа без оператора write не будет отвечать свойству результативности, т.к. о результате программы нельзя будет ничего сказать.

**Массовость.** Это означает, что алгоритм решения задачи разрабатывается в общем виде, т.е. он должен быть применим для некоторого класса задач, различающихся лишь исходными данными. При этом исходные данные могут выбираться из некоторой области, которая называется областью применимости алгоритма.

Пример.

Задача нахождения корней квадратного уравнения может быть решена как:

1. Найти дискриминант квадратного уравнения:  $d=b^2-4ac$ ;

2. Найти корни квадратного уравнения:  $x_{1,2} = \frac{-b \pm \sqrt{d}}{2a}$ ;

Этот алгоритм отвечает свойству массовости, если область применимости алгоритма обозначена, как множество квадратных уравнений, имеющих положительный дискриминант. Однако, для множества всех квадратных уравнений, алгоритм не отвечает свойству массовости, т.к. не содержит проверку на отрицательный дискриминант и следовательно на отсутствие корней.

**Правильность.** В результате выполнения алгоритма должны быть получены правильные результаты решения задач, для которых он составлен.

Пример.

Правильное решение численного метода может быть получено только в случае, если известно, что данный метод сходится при заданных условиях.

**Конечность.** Выполнение алгоритма должно завершиться за конечное число шагов. Примером бесконечного алгоритма может служить элементарное «зацикливание».

## **Формы представления алгоритмов**

Наиболее распространены следующие формы представления алгоритмов:

- словесная (записи на естественном языке);
- графическая (изображения из графических символов);
- псевдокоды (полуформализованные описания алгоритмов на условном алгоритмическом языке, включающие в себя как элементы языка программирования, так и фразы естественного языка, общепринятые математические обозначения и др.);
- программная (тексты на языках программирования).

## **Словесный способ записи алгоритмов**

Словесный способ записи алгоритмов представляет собой описание последовательных этапов обработки данных. Алгоритм задается в произвольном изложении на естественном языке.

### Пример.

Нахождение наибольшего элемента массива.

1. Задать массив чисел.
2. Объявить максимальным элементом массива первый элемент.
3. Сравнить следующий элемент массива с максимальным. Если элемент больше максимального, то объявить его максимальным.
4. Если массив не кончился, то идти к шагу 3, закончить в противном случае.

Словесный способ не имеет широкого распространения по следующим причинам:

- такие описания строго не формализуемы;
- страдают многословностью записей;
- допускают неоднозначность толкования отдельных предписаний.

## Графический способ записи алгоритмов

Графический способ представления алгоритмов является более компактным и наглядным по сравнению со словесным.

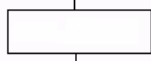


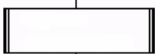

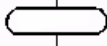
При графическом представлении алгоритм изображается в виде последовательности связанных между собой функциональных блоков, каждый из которых соответствует выполнению одного или нескольких действий.

Такое графическое представление называется схемой алгоритма или блок-схемой.

В блок-схеме каждому типу действий соответствует свой графический символ. Символы соединяются линиями переходов, определяющими очередность выполнения действий.

**Таблица 0.1**

**Множество элементов блок-схем.**

Символ	Графическое обозначение
Процесс	
Условие	
Цикл	
Подпрограмма	
Ввод/вывод	
Пуск/останов	

Представленное множество является избыточно-полным, т.е. любой алгоритм можно описать с помощью представленных выше элементов, и элемент «цикл» выражается через другие элементы множества.

## Псевдокод

Псевдокод представляет собой систему обозначений и правил, предназначенную для единообразной записи алгоритмов. Он занимает промежуточное место между естественным и формальным языками.

С одной стороны, он близок к обычному естественному языку, поэтому алгоритмы могут на нем записываться и читаться как обычный текст. С другой стороны, в псевдокоде используются

некоторые формальные конструкции и математическая символика, что приближает запись алгоритма к общепринятой математической записи.

Пример.

<оператор условия> ::= <если><логическое выражение>  
<то><выражение 1>  
<в противном случае><выражение 2>

Read(a);	<u>если</u>	a>5
<b>If</b> a>5 <b>then</b>	<u>то</u>	
write('Yes')		печатать (YES)
<b>else</b> write('No')	<u>в противном случае</u>	
		печатать (NO)

## Программная запись алгоритма

Программная запись алгоритма – это представление алгоритма на конкретном языке программирования, предназначенное для последующей компиляции и исполнения.

Пример.

```
Var
    A,b:integer;
    C:boolean
Begin
    Read(a,b);
    C:= a>b;
    Writeln(c);
End.
```

## Виды алгоритмов

Алгоритмы по структуре блок-схем делятся на: линейные, разветвляющиеся и циклические. По типу исполнения алгоритмы бывают: итерационные и рекурсивные.

## Линейные алгоритмы

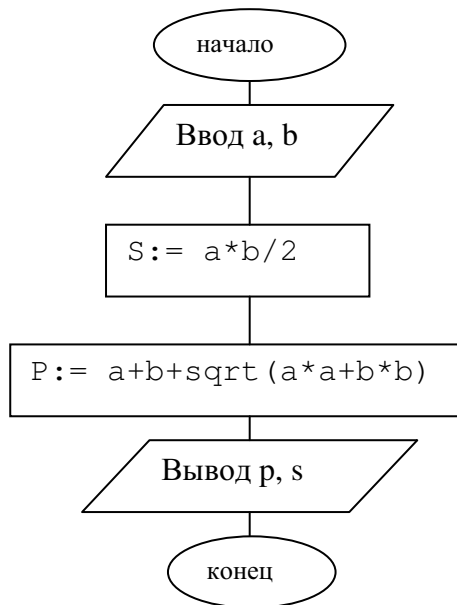
Линейным называется алгоритм, команды которого выполняются последовательно одна за другой в строгом порядке.

Пример.

1. кулинарный рецепт.  
2. задача нахождения площади и периметра  
прямоугольного прямоугольника по его катетам.

```
Var
    a, b, p, s: real;
Begin
    Read(a);
    Read(b);
    S:= a*b/2;
    P:= a+b+sqrt(a*a+b*b);
    Write(s,p);
End.
```





## Разветвляющиеся алгоритмы

Разветвляющимся называется алгоритм, последовательность выполнения команд которого зависит от выполнения или невыполнения некоторого условия.

### Пример.

1. Алгоритм перехода улицы по сигналу светофора
2. Алгоритм решения квадратного уравнения

Найти корни заданного квадратного уравнения или сообщить об их отсутствии.

**Const**

Eps=0.001;

**Var**

a, b, c, D, x1, x2: real;

**Begin**

Read(a);

Read(b);

Read(c);

**If** (abs(a)<eps) **then**

writeln ("Уравнение не квадратное");

**else**

**Begin**

D:= b\*b-4\*a\*c;

**If** (D < 0 ) **then**

Writeln("Решений нет");

**Else**

**Begin**

**If** (D>0) **then**

**Begin**

X1:=(-b+sqrt(D))/(2\*a);

X2:=(-b-sqrt(D))/(2\*a);

Writeln("x1=", x1, "x2=", x2);

**End**

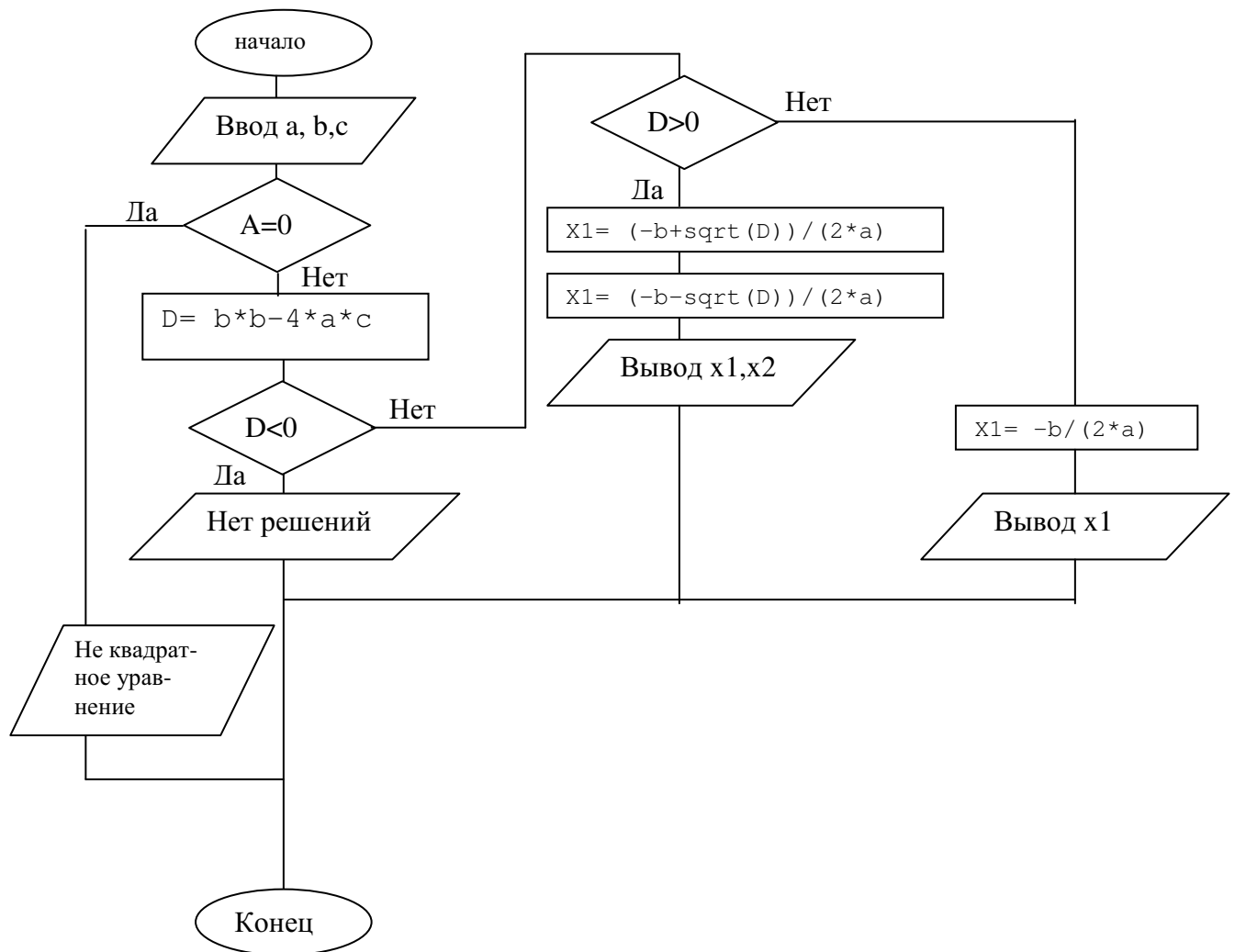
**Else**

**Begin**

X1:=-b/(2\*a);

Writeln("x1=x2=", x1);

End;  
 End;  
 End;  
 End.



## Циклические алгоритмы

Циклическим называется алгоритм, выполнение команд которого повторяется многократно над новыми исходными данными.

Пример.

1. Работа четырехтактного двигателя внутреннего сгорания
2. Составить алгоритм определения суммы квадратов  $n$  членов арифметической прогрессии, первый член которой равен  $a_1$ , а разность равна  $d$ .

**Var**

$a_1, n, d, s, a_i$ : real;  
 $i$ : integer;

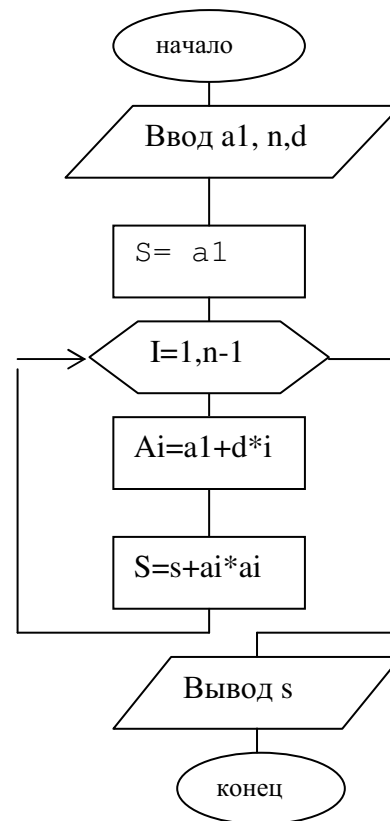
**Begin**

Read( $a_1$ );  
 Read( $n$ );  
 Read( $d$ );  
 $s := a_1 * a_1$ ;  
**For**  $i := 1$  to  $n-1$  **do**

```

Begin
    Ai:= a1+d*i;
    S:=s+ ai*ai;
End;
Writeln(s);
End.

```



## Итерационные алгоритмы

Итерационный алгоритм – это алгоритм, каждый шаг которого использует данные, вычисленные на предыдущем шаге.

Пример.

Вычисление корней уравнения  $f(x)=0$  методом деления отрезка пополам с точностью  $\epsilon$ .

Если корень уравнения находится в пределах отрезка  $[a, b]$ , то условие корректности алгоритма:  $f(a) * f(b) < 0$ .

```

Var
    A, b, f, e, m: real;
Procedure f(x: real);
Begin
    F:= x*x+6*x-15
End;
Begin
    Read(a);
    Read(b);
    Read(e);
    M:= (a+b) / 2;
    While abs(b-a) > e do
    begin
        If (f(a)*f(m) < 0 then b:=m)
        Else a:=m;
        M:= (a+b) / 2;
    End;
    Write(m);
End.

```

## Рекурсивные алгоритмы

Рекурсивный алгоритм – это алгоритм, частично состоящий или определяемый с помощью самого себя.

Пример.

Нахождение значения функции  $n!$

Рекурсивное определение:

1.  $f(0) = 1$ ;

2.  $F(N) = f(N-1) * N$ .

Алгоритм:

**Var**

N:integer;

**Function** f(N:integer):integer;

**Begin**

**If** (n=0) **then** f:= 1

**Else** f:=f (n-1) \*N;

**End;**

**Begin**

    Read(n) ;

    Writeln(f(n)) ;

**End.**

## Оценка сложности алгоритма

Чтобы оценить сложность алгоритма, необходимо руководствоваться следующими данными:

- Процессор совершает элементарные действия, которые представлены набором операций: присваивания, сравнения, арифметических действий и некоторых других.
- Каждый процессор совершает каждую операцию по-своему, т.е. тратит на нее разное время.

На основании этого любой алгоритм можно оценить с точки зрения трудоемкости и эффективности.

## Оценка трудоемкости алгоритма

Трудоемкостью алгоритма будем называть количество элементарных операций определенного типа, необходимых для осуществления алгоритма.

Следует отметить, что трудоемкость носит вероятностный характер, т.к. зависит от состояния среды исполнителя алгоритма (т.е. от начальных условий задачи). В связи с этим, любому алгоритму можно дать оценку трудоемкости в лучшем, худшем и среднестатистическом случае.

Пример.

При работе с массивами начальное состояние среды может быть одним из следующих: массив упорядочен (лучший случай), массив частично упорядочен, массив упорядочен в обратном порядке (худший случай, когда в массиве нельзя найти ни одной пары упорядоченных элементов).

Оценим трудоемкость алгоритма поиска элемента в массиве из 1000 отсортированных элементов.

Алгоритм:

1.  $L=1$ ;

2.  $R=N$ ;

3. Если  $L \geq R$  тогда 9

4.  $m:=(L+R) \div 2$

5. Если  $x > a[m]$  тогда  $L:=m+1$

6. Если  $x < a[m]$  тогда  $R:=m$

7. Если  $x=a[m]$  тогда 10
8. Переход на 3
9. нет элемента , переход 11
10. номер элемента  $m$ , переход 11
11. конец

В лучшем случае применения этого алгоритма мы найдем нужный элемент при первом же сравнении.

В худшем случае для поиска элемента в массиве при таком алгоритме понадобится 11 сравнений (исходя из того, что на каждом шаге набор просматриваемых элементов уменьшается ровно пополам, а  $2^{10} = 1024$ . понадобится одно дополнительное сравнение на последнем шаге алгоритма, для того чтобы убедиться является ли последний оставшийся элемент массива искомым или нет).

Среднестатистическая оценка данного алгоритма будет равна 5 сравнениям (худшая оценка, деленная пополам), т.к. алгоритм является простым и искомое число имеет равномерное распределение по массиву.

## Оценка эффективности алгоритма

Эффективность – это критерий, отображающий скорость работы алгоритма на различных начальных условиях.

Эффективность зависит от трудоемкости алгоритма, а также от специфических особенностей среды (например, типа компьютера).

### Пример.

Говорят, что одна сортировка эффективнее (лучше) другой, если она выполняется быстрее на тех же самых начальных условиях среды.

**Таблица 0.2**

**Время работы различных программ сортировки на массиве 2048 элементов (с).**

Тип сортировки	Массив		
	Упорядоченный	Случайный	В обратном порядке
Прямое включение	0.22	50.74	103.8
Сортировка Шелла	0.8	7.08	12.34
HeapSort	2.32	2.22	2.12

Эффективность различных алгоритмов может быть оценена и по другим критериям, одним из которых может быть точность.

## Лекция 4

### Нелинейные структуры данных.

### Деревья. Бинарные деревья. Деревья поиска. Обходы и прошивка деревьев

#### **Нелинейные структуры данных**

Если движение по элементам структуры невозможно осуществлять линейно от одного к другому, то такая структура будет называться нелинейной. Среди множества всех нелинейных структур данных мы подробно рассмотрим следующие:

- иерархические и ортогональные списки;
- графы;
- В-деревья и бинарные деревья;
- AVL-деревья

#### **Деревья**

Дадим основные определения.

Формально **дерево** (tree) можно определить как конечное множество  $T$  одного или более узлов со следующими свойствами:

- 1) существует один выделенный узел – корень (root) данного дерева  $T$ ;
- 2) остальные узлы распределены среди  $m \geq 0$  непересекающихся множеств  $T_1, T_2, \dots, T_m$ , и каждое из этих множеств, в свою очередь, является деревом.  $T_1, T_2, \dots, T_m$  – называются поддеревьями данного корня.

Из определения следует, что каждый узел дерева является корнем некоторого поддерева данного дерева.

**Степень узла** – это количество поддеревьев узла.

Узел со степенью 0 называется терминальной вершиной или **листом**.

Узел со степенью большей 0 называется **внутренним** узлом дерева.

**Уровень** узла по отношению к дереву  $T$  определяется рекурсивно следующим образом: уровень корня равен 0, а уровень другого узла на единицу больше, чем уровень корня ближайшего поддерева дерева  $T$ , содержащего данный узел.

**Упорядоченное дерево** – дерево, у которого имеет значение относительный порядок поддеревьев.

**Бинарное дерево** (binary tree) – это конечное множество узлов, которое является либо пустым, либо состоит из корня и двух непересекающихся бинарных деревьев, которые называются левым и правым поддеревьями данного корня.

Иными словами, каждый узел бинарного дерева имеет не более двух поддеревьев.

**Лес** (forest) – множество, не содержащее ни одного непересекающегося дерева, или содержащее несколько непересекающихся деревьев.

Между определениями дерева и леса существует связь: если удалить корень из дерева – получим лес. И наоборот, добавив узел в лес, все деревья которого рассматриваются как поддеревья нового узла, получим дерево.

## Способы изображения деревьев

Древовидную структуру можно представить графически несколькими способами, которые могут выглядеть совсем не так, как настоящие деревья в природе.

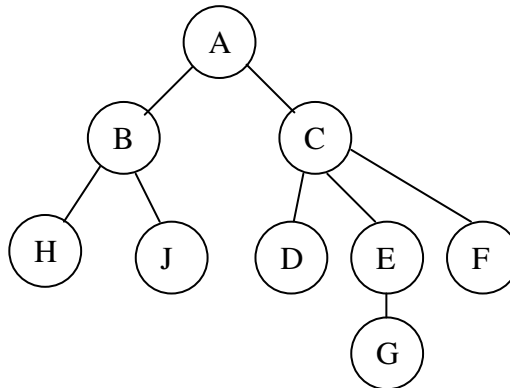


Рис.1. Обычная схема изображения дерева

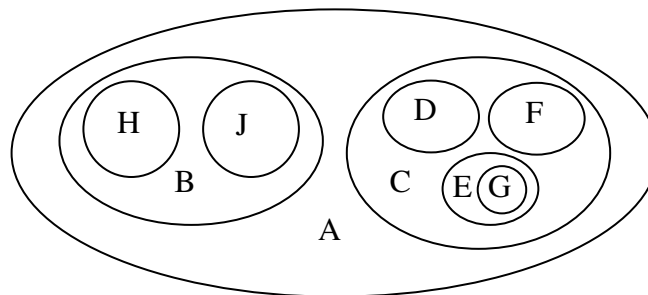


Рис.2. Изображение дерева с помощью вложенных множеств

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

Рис.3. Изображение дерева с помощью вложенных скобок

## Бинарные деревья

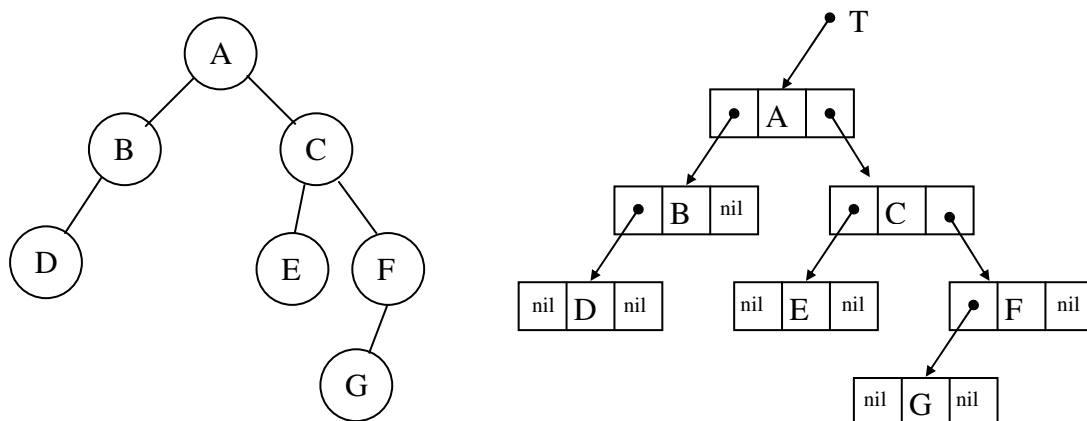
### Представление бинарных деревьев в языках программирования

Определение дерева на языке Pascal может выглядеть следующим образом:

#### Type

```
Ptree: ^Ttree;  
Ttree: record  
    Val: integer; {значение узла}  
    Left:Ptree; {ссылка на левое поддерево}  
    Right:Ptree; {ссылка на правое поддерево}  
End;
```

Для представления леса используется массив указателей на деревья.



**Рис.4. Пример бинарного дерева**

### Обходы бинарных деревьев

В алгоритмах работы с древовидными структурами часто встречается понятие обхода (traversing) дерева. Обход дерева представляет собой посещение каждого узла дерева в точности один раз. Обходы нужны для обработки узлов дерева в определенном порядке.

Существует три принципиально разных способа обхода бинарного дерева: в прямом порядке (preorder), в центрированном порядке (inorder) и в обратном порядке (postorder). Эти три способа определяются рекурсивно. Если бинарное дерево пусто, то для его обхода ничего делать не потребуется, в противном случае обход выполняется в три этапа.

Прямой порядок обхода:

1. Попасть в корень
2. Пройти левое поддерево
3. Пройти правое поддерево

```

Procedure Preorder (tree:PTree);
Begin
  If (tree <> nil)
  Begin
    Write( tree^.val);
    Preorder (tree^.left);
    Preorder (tree^.right);
  End;
End;
  
```

Для дерева на Рис.4. при прямом порядке обхода получим последовательность: ABDCEFG.

Центрированный порядок обхода:

1. Пройти левое поддерево
2. Попасть в корень
3. Пройти правое поддерево

```

Procedure Inorder (tree:PTree);
Begin
  If (tree <> nil)
  Begin
    Inorder (tree^.left);
    Write( tree^.val);
    Inorder (tree^.right);
  End;
End;
  
```



**End;**

Для дерева на Рис.4. при центрированном порядке обхода получим последовательность: DBAECGF.

Обратный порядок обхода:

1. Пройти левое поддерево
2. Пройти правое поддерево
3. Попасть в корень

```
Procedure Postorder(tree:PTree);  
Begin  
    If (tree <> nil)  
    Begin  
        Postorder(tree^.left);  
        Postorder(tree^.right);  
        Write( tree^.val);  
    End;  
End;
```

Для дерева на Рис.4. при обратном порядке обхода получим последовательность: DBEGFCA.

### ***Деревья поиска***

Двоичные деревья часто употребляются для представления множества данных, среди которых идет поиск элементов по уникальному ключу. Если дерево организовано так, что для каждой вершины  $t_i$  справедливо утверждение, что все ключи левого поддерева  $t_i$  меньше ключа  $t_i$ , а все ключи правого поддерева  $t_i$  больше его, то такое дерево будем называть **деревом поиска**.

Пример.

Построение дерева поиска.

```
Procedure AddNode(var tree:Ptree; key:integer);  
Begin  
    If (tree <> nil) then  
    Begin  
        If (key < tree^.val) then AddNode(tree^.left)  
        Else AddNode(tree^.right);  
    End  
    Else  
    Begin  
        New(tree);  
        tree ^.left:=nil;  
        tree ^.right:=nil;  
        tree ^.val:=key;  
    End;  
End;  
  
Procedure MakeTree(var Tree:Ptree; file_name:string);  
Var  
    Key:integer;  
    F:file of integer;  
Begin  
    Assign(f, file_name);
```

```

Reset(f);
While not Eof() do
Begin
    Read(f, key);
    AddNode(Tree, key);
End;
Close(f);
End;

```

## Поиск

Поиск заданного ключа в дереве поиска основан на сравнении заданного ключа с ключом текущей вершины. Описанная ниже функция в случае успешного поиска ключа возвращает указатель на найденную вершину и nil в противном случае.

```

function Poisk(tree:Ptree; key:integer):Ptree;
Begin
    If tree<>nil then
        Begin
            If (key < tree^.val) then Poisk(tree^.left);
            If (key > tree^.val) then Poisk(tree^.right);
            If (key = tree^.val) then Poisk:=tree;
        end
    Else
        Poisk=nil;
    End;

```

## Удаление элемента по ключу

Рассмотрим задачу исключения из упорядоченного дерева вершины с ключом key. Если требуется исключить терминальную вершину или узел с одним потомком, то трудностей не возникает. Если же число потомков узла равно двум, то для сохранения упорядоченности элементов в дереве поступают следующим образом: удаляемый элемент заменяют либо самым правым элементом его левого поддеревя, либо самым левым элементом его правого поддеревя.

В представленной ниже процедуре различают 3 случая:

- Узла с ключом key нет
- Узел с ключом key имеет не более одного потомка
- Узел с ключом key имеет двух потомков

```

procedure DelKey(var Tree:Ptree;key:integer);
var q:Ptree;
    procedure del(var Tree:Ptree);{обработка вершины степени 2}
    begin
        if Tree^.right <> nil then del(tree^.right)
        else
            begin
                q^.val:=tree^.val;
                q:=Tree;
                Tree:=Tree^.left;
            end;
        end;
    begin
        if Tree <> nil then

```

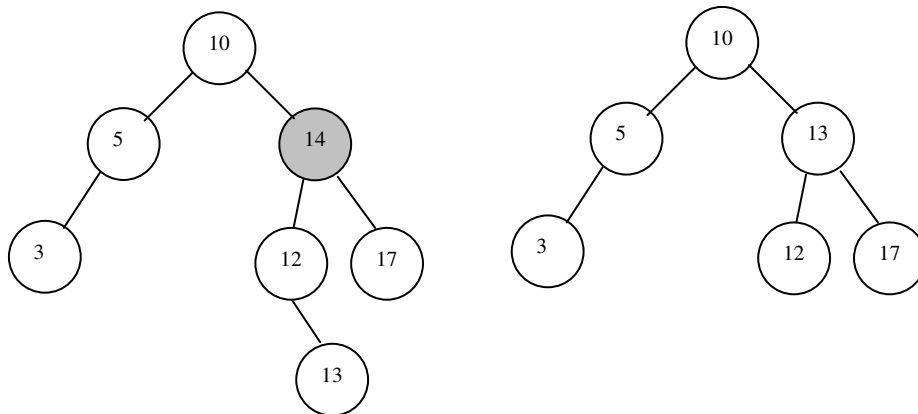
```

if key < Tree^.val then DelKey(Tree^.left,key)
else if key > Tree^.val then DelKey(Tree^.right,key)
else
begin
    q:=Tree;
    if q^.right = nil then Tree:=q^.left
    else if q^.left = nil then Tree:=q^.right
    else Del(q^.left);
    dispose(q);
end;
end;

```

Пример.

Удаление вершины с ключом 14



## Прошитые деревья

Для того, чтобы из любого узла дерева можно было попасть в произвольный узел используются так называемые прошитые деревья.

В прошитых деревьях нулевые ссылки на левое поддерево заменяются «нитевидными» ссылками на узел-предшественник, нулевые ссылки на правое поддерево заменяются «нитевидными» ссылками на узел-последователь согласно обходу дерева.

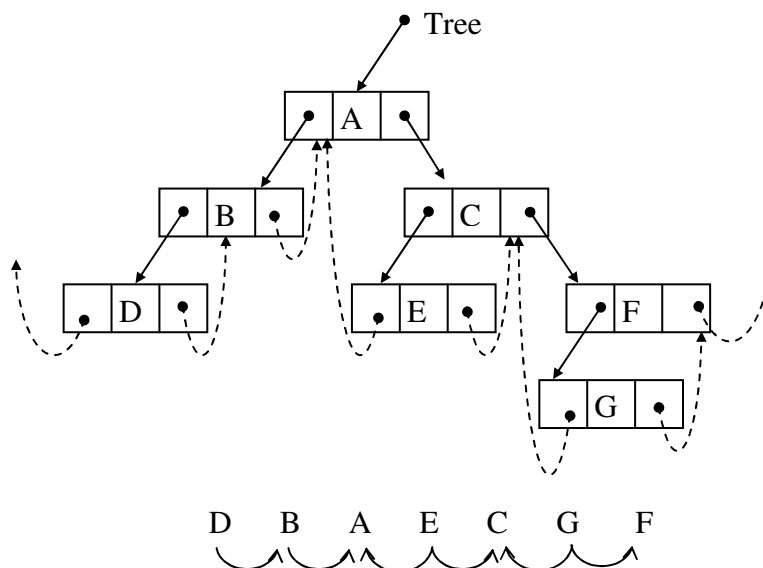


Рис.5. Дерево, прошитое при центрированном обходе



**Рис.6. Дерево, прошитое при обратном обходе**



**Рис.7. Дерево, прошитое при прямом обходе**

Для каждого узла прошитого дерева следует различать обычные и нитевидные ссылки, поэтому определение дерева будет несколько сложнее:

#### **Type**

```
PTree=^TTree;
TTree=record
    val:integer;
    left:PTree;
    right:PTree;
    ltag:boolean; {признак ссылки-нити на левое поддерево}
    rtag:boolean; {признак ссылки-нити на правое поддерево}
end;
```

На практике целесообразно использовать деревья, прошитые при центрированном обходе, т.к. их реализация проще и эффективнее.

#### **Прошивка дерева при центрированном обходе**

Основная сложность в данном методе прошивки кроется в установлении крайней правой и крайней левой нитевидных ссылок. Если мы не ставим перед собой задачу построить циклическую структуру, то эти ссылки остаются нулевыми. В противном случае их можно замкнуть на корень или друг на друга согласно последовательности обхода (Рис.5).

```
Procedure Proshivka (Tree,prev,next:PTree);
Begin
    if (Tree<>nil) then
        begin
            if (Tree^.left<>nil) then Proshivka (Tree^.left,prev,Tree)
            else begin
                Tree^.left:=prev;
                Tree^.ltag:=true; {установка левой ссылки-нити}
            end;
            if (Tree^.right<>nil) then Proshivka (Tree^.right,Tree,next)
            else begin
                Tree^.right:=next;
                Tree^.rtag:=true; {установка правой ссылки-нити}
            end;
        end;
End;
```

## Лекция 5

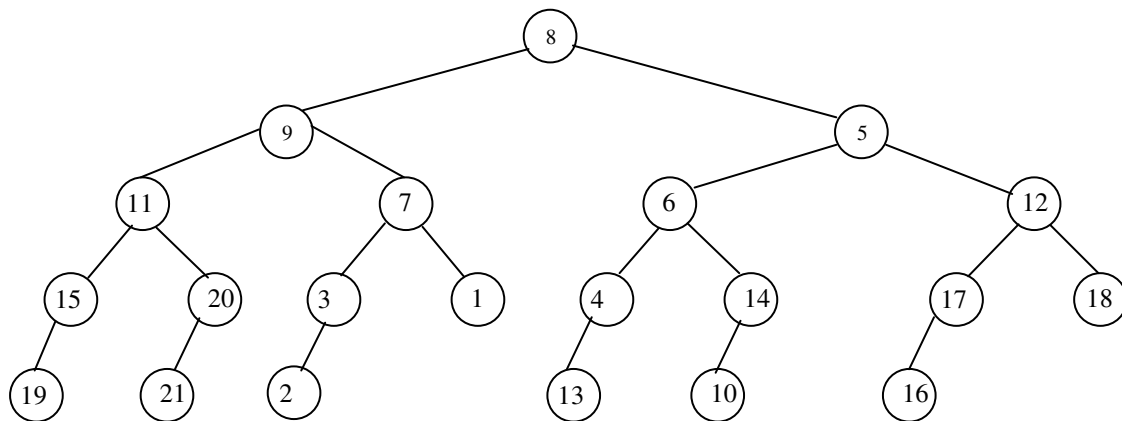
### Сбалансированные деревья. AVL – деревья. Деревья оптимального поиска

#### **Идеально сбалансированные деревья**

Идеально-сбалансированным деревом называется дерево, для каждой вершины которого выполняется условие: число вершин левого и правого поддерева отличается не более чем на 1.

#### Пример.

Построить идеально сбалансированное дерево по заданной последовательности из 21 элемента: 8 9 11 15 19 20 21 7 3 2 1 5 6 4 13 14 10 12 17 16 18



Алгоритм построения идеально сбалансированного дерева с  $n$  вершинами выглядит так:

1. Взять одну вершину в качестве корня
2. Построить тем же способом левое поддерево с  $n_l = n \div 2$  вершинами
3. Построить тем же способом левое поддерево с  $n_r = n - n_l - 1$  вершинами

Представленный алгоритм имеет существенный недостаток – необходимость знать заранее число вершин дерева. Другой способ построения сбалансированного дерева состоит в том, что добавление каждой новой вершины происходит в то поддерево, высота которого меньше. Высота поддерева определяется числом его уровней.

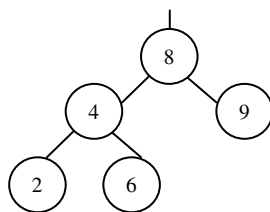
Сложность работы с такого типа деревьями представляет процедура вставки нового узла. Если включение происходит случайно, то восстановление сбалансированности дерева после включения – довольно сложная операция. Для выхода из сложившегося положения прибегают к менее строгому определению сбалансированности.

#### **AVL-деревья**

Одно из определений сбалансированного дерева было предложено Г.М.Адельсоном-Вельским и Е.М.Ландисом:

дерево называется сбалансированным тогда и только тогда, когда высоты двух поддеревьев каждой из его вершин отличаются не более чем на 1.

Такие деревья называют AVL-деревьями по имени их создателей. Заметим, что все идеально сбалансированные деревья являются AVL-деревьями. Далее в лекции, говоря «сбалансированное дерево», будем иметь в виду AVL-дерево.



**Рис.1 АВЛ-дерево**

### **Включение в сбалансированное дерево**

Обозначим корень дерева  $r$ , а левое и правое поддеревья как  $L$  и  $R$  соответственно. Допустим, что включение новой вершины в  $L$  влечет увеличение ее высоты на 1, тогда возможно 3 случая:

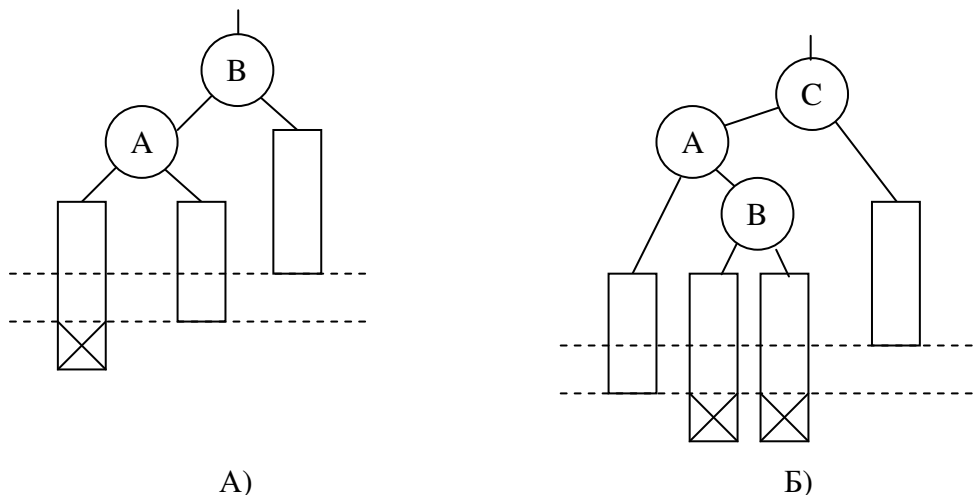
1.  $h_L = h_R$ :  $L$  и  $R$  станут разной высоты, но критерий сбалансированности не будет нарушен;
2.  $h_L < h_R$ :  $L$  и  $R$  станут равной высоты, и сбалансированность улучшится;
3.  $h_L > h_R$ : критерий сбалансированности нарушится и дерево необходимо перестраивать.

Перестройка дерева, восстанавливающая его сбалансированность, называется поворотом.

Поворот должен удовлетворять следующим требованиям:

1. Прохождение трансформированного дерева в центрированном порядке должно быть таким же, как и для первоначального дерева (т.е. трансформированное дерево должно оставаться деревом бинарного поиска)
2. Трансформированное дерево должно быть сбалансированным

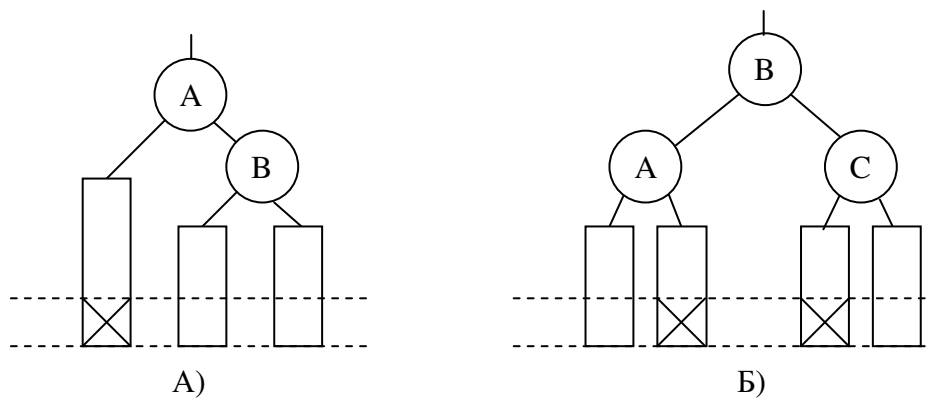
При включении в сбалансированное дерево нового узла возможны только два принципиальных случая (Рис.2). Остальные случаи являются их частными модификациями.



**Рис.2 Несбалансированность, возникшая из-за включения**

Одиный LL поворот происходит тогда, когда родительский узел  $B$  и его левый сын  $A$  начинают перевешивать влево после включения узла в позицию  $X$  (Рис.2а). В результате такого поворота  $A$  замещает своего родителя, который становится его правым сыном. Бывшее правое поддерево узла  $A$  присоединяется к  $B$  в качестве левого поддерева. Это сохраняет упорядоченность. Поворот уравнивает как родителя, так и его левого сына (Рис.3а).

Двойной LR поворот происходит тогда, когда родительский узел  $C$  становится перевешивающим влево, а его левый сын  $A$  - перевешивающим вправо (Рис.2б). В результате поворота узел  $B$ , корень правого поддерева узла  $A$ , замещает узел  $C$ , а его левое и правое поддерева распределяются между узлами  $A$  и  $C$  соответственно (Рис.3б).



**Рис.3 Восстановление сбалансированности**

Одинарный RR – поворот и двойной RL – поворот являются зеркальным отражением рассмотренных выше случаев.

Алгоритм включения в сбалансированное дерево зависит от того, каким образом хранить информацию о сбалансированности дерева. В рассмотренном ниже описании типа показатель сбалансированности вершины дерева интерпретируется как разность между высотой правого и левого поддерева.

```

type
  ptr=^node;
  node=record
    key:integer; {ключ вершины}
    left,right:ptr; {ссылки на левое и правое поддерева}
    bal:-1..1; {показатель сбалансированности вершины}
  end;
{вставка в сбалансированное дерево}

```

Алгоритм включения вершины состоит из трех последовательных частей:

1. Проход по пути поиска, до тех пор, пока не убедимся, что ключа в дереве нет (имеется в виду, что ключи не могут повторяться).
2. Включение новой вершины и определение результирующего показателя сбалансированности.
3. «Отступление» по пути поиска и определение результирующего показателя сбалансированности. Если необходимо – балансировка.

Для представленного ниже алгоритма вводится булевский параметр h, показывающий, что высота поддерева увеличилась.

```

procedure search(x:integer; var p:ptr; var h:boolean);
var p1,p2:ptr;
begin
  if p=nil then {вставка нового узла}
  begin
    new(p);
    h:=true;
    with p^ do begin
      key:=x;
      left:=nil;
      right:=nil;
      bal:=0; end
  end

```

```

else
  if p^.key>x then
  begin
    search(x,p^.left,h);
    if h then {выросла левая ветвь}
    case p^.bal of
      1: begin p^.bal:=0; h:=false end;
      0: begin p^.bal :=-1 end;
      -1: begin {балансировка}
        p1:=p^.left;
        if p1^.bal = -1 then {однократный LL-поворот}
        begin
          p^.left:=p1^.right;
          p1^.right:=p;
          p^.bal:=0;
          p:=p1;
        end
        else {двойной LR-поворот }
        begin
          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 case}
  end{endif p^.key>x }
else
  if p^.key < x then
  begin
    search(x,p^.right,h);
    if h then {выросла правая ветвь}
    case p^.bal of
      -1: begin p^.bal:=0; h:=false end;
      0: begin p^.bal :=1 end;
      1: begin {балансировка}
        p1:=p^.right;
        if p1^.bal = 1 then {однократный RR-поворот}
        begin
          p^.right:=p1^.left;
          p1^.left:=p;
          p^.bal:=0;
          p:=p1;
        end
        else {двойной RL-поворот}

```

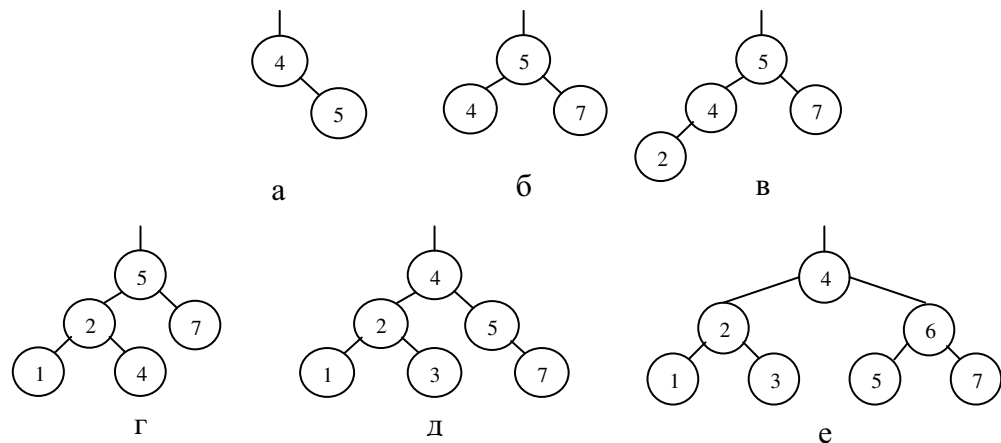


```

begin
  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 case}
end {endif p^.key<x}
else
end;

```

Проиллюстрируем работу алгоритма на примере. В дерево (Рис.4а) последовательно включаются вершины: 7, 2, 4, 3, 6.



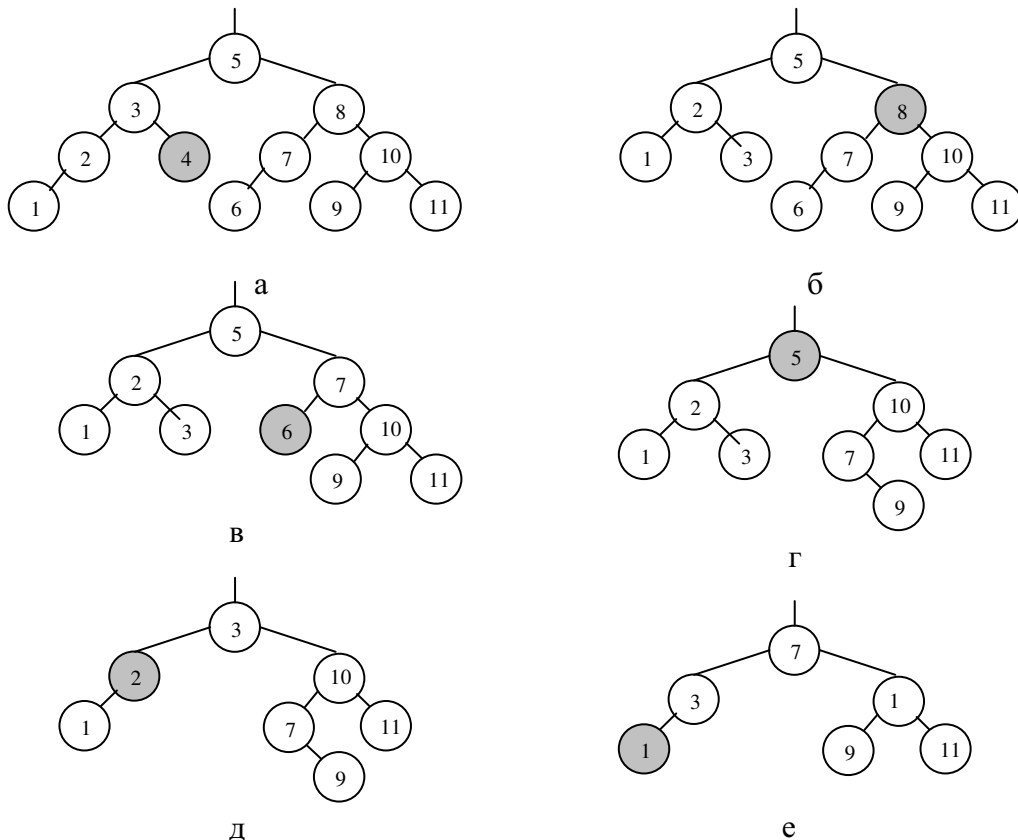
**Рис.4 Включение в сбалансированное дерево**

Включение вершины 7 вначале приводит к несбалансированному дереву, которое приводится к идеально сбалансированному с помощью одинарного RR-поворота (Рис.4б). Последующее включение вершин 2 и 1 приводит к несбалансированной вершине 4, она балансируется с помощью LL-поворота (Рис.4г). Остальные два случая (Рис.4д,е) связаны с LR и RL –поворотами соответственно.

### Исключение из сбалансированного дерева

Алгоритм исключения из сбалансированного дерева повторяет такой же алгоритм для бинарного дерева поиска. Различие состоит в том, что при удалении вершины из сбалансированного дерева следует каждый раз восстанавливать его сбалансированность.

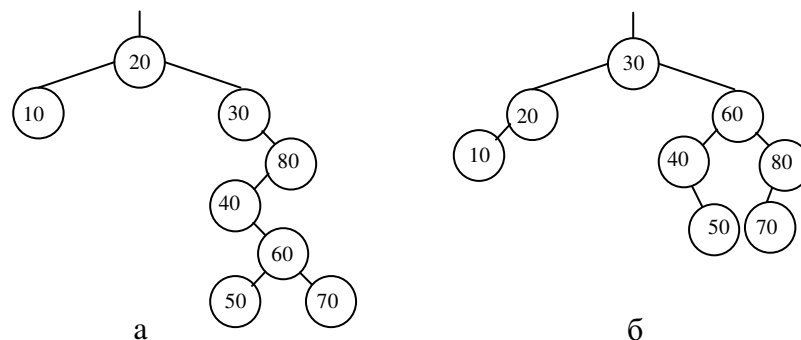
Рассмотрим пример (Рис.5).



**Рис.5** Исклучение из сбалансированного дерева

Из сбалансированного дерева последовательно исключаются вершины с ключами: 4, 8, 6, 5, 2, 1, 7. В результате исключения ключа 4 получается несбалансированная вершина 3 (Рис.5а). Соответствующая операция балансировки требует одинарного LL-поворота (Рис.5б). При исключении вершины 6 (Рис.5в) снова требуется балансировка посредством одинарного RR-поворота (Рис.5г). Исключение вершины 2 приведет к двукратному RL-повороту.

## Применение АВЛ-деревьев



**Рис.6** Представление массива {20, 30, 80, 40, 10, 60, 50, 70} бинарным деревом поиска (а) и АВЛ-деревом (б)

Бинарные деревья поиска предназначены для быстрого доступа к данным. В идеале разумно сбалансированное дерево имеет высоту порядка  $O(\log_2 n)$ . Однако при некотором стечении обстоятельств дерево может оказаться вырожденным. Тогда высота его будет  $O(n)$ , и доступ к данным существенно замедлится. Методы вставки и удаления в классе AVL-деревьев

гарантируют, что все узлы останутся сбалансированными по высоте. На Рис.6 показаны эквивалентные представления массива AVL-деревом и бинарным деревом поиска. Бинарное дерево поиска имеет высоту 5, в то время как высота AVL-дерева равна 3. В общем случае высота сбалансированного дерева не превышает  $O(\log_2 n)$ . Таким образом, AVL-дерево является мощной структурой хранения, обеспечивающей быстрый доступ к данным.

### **Деревья оптимального поиска**

Дадим несколько определений.

**Длиной пути** к вершине  $x$  называется число ветвей или ребер, которое нужно пройти от корня к вершине  $x$ .

**Длиной внутреннего пути**, или длиной пути всего дерева, называется сумма длин путей всех компонентов дерева.

Во всех предыдущих рассуждениях о деревьях поиска подразумевалось, что частота обращения ко всем вершинам дерева одинакова. Однако, встречаются ситуации, когда есть информация о вероятностях обращения к отдельным ключам. Обычно, для таких ситуаций характерно «постоянство» ключей, т.е. в дерево поиска не включаются новые ключи, и не исключаются старые, структура дерева остается неизменной.

#### Пример.

Сканер транслятора, определяющий, не относится ли заданное слово к классу зарезервированных. Статистические измерения на сотнях транслируемых программ могут дать точную информацию о частотах появления отдельных ключей.

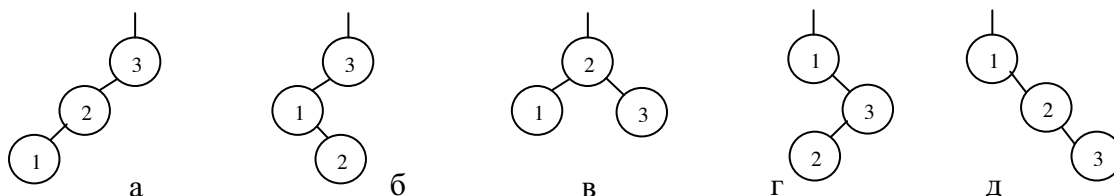
Предположим, что в дереве поиска вероятность обращения к  $i$ -ой вершине  $p_i$  и  $\sum_{i=1}^n p_i = 1$ .

Необходимо организовать дерево поиска так, чтобы общее число шагов поиска, подсчитанное для достаточно большого числа обращений, было минимальным. С этой целью припишем в определении к длине пути к каждой вершине некоторый вес и условимся, что корень находится на уровне 1, тогда внутренняя взвешенная длина  $P$  пути будет представлять собой сумму всех путей от корня к каждой вершине, умноженная на вероятность обращения к этой вершине:

$$P = \sum_{i=1}^n p_i \cdot h_i,$$

где  $h_i$  - уровень вершины  $i$ . Наша цель – минимизировать при заданном распределении вероятностей взвешенную длину пути.

Рассмотрим множество ключей 1, 2, 3 со следующими вероятностями обращения к ним:  $p_1=1/7$ ,  $p_2=2/7$  и  $p_3=4/7$ . Эти три ключа можно расставить в дереве поиска пятью различными способами (Рис.7).



**Рис.7 Дерево поиска с тремя вершинами**

Взвешенные длины путей деревьев на Рис.7 будут составлять:  $11/7$ ,  $14/7$ ,  $12/7$ ,  $15/7$  и  $17/7$ . Таким образом, оптимальным оказывается не идеально сбалансированное дерево (Рис.7в), а вырожденное (Рис.7а).

# Лекция 6

## В-деревья

### Деревья приоритетного поиска

#### В-деревья

До сих пор в поле нашего зрения были только бинарные деревья. Мы говорили, что если двоичное дерево сбалансировано, то количество шагов поиска в нем не превысит величины  $\log_2(N)$ , где  $N$  – число элементов в дереве.

Теперь разобьем исходное дерево на поддеревья таким образом, чтобы каждое содержало фиксированное число вершин. Получившиеся группы элементов будем называть страницами (Рис.1).

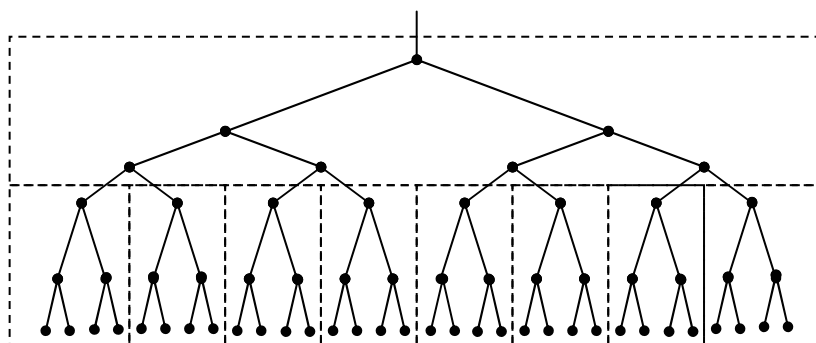


Рис.1 Двоичное дерево разделенное на страницы.

#### Пример.

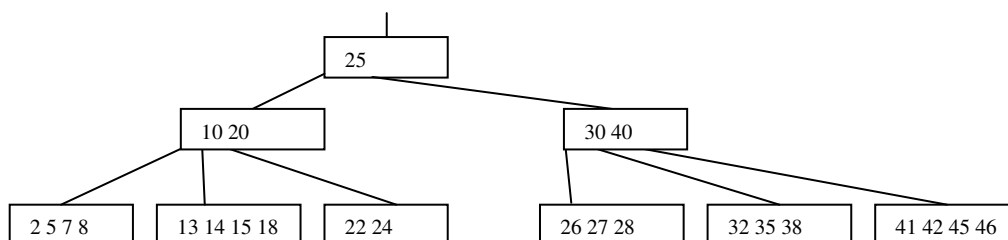
Если использовать для множества данных, состоящих из миллиона элементов, двоичное дерево, то для поиска элемента потребуется в среднем  $\log_2(10^6) \approx 20$  обращений. Если же обращение происходит не к отдельным узлам, а к страницам, содержащим, например, 100 элементов, то потребуется только  $\log_{100}(10^6) = 3$  шагов поиска.

Условимся, что каждая страница может содержать от  $n$  до  $2n$  вершин. Таким образом, для дерева с  $N$  элементами и максимальным размером страницы в  $2n$  вершин в самом худшем случае потребуется  $\log_n N$  обращений к страницам. А именно эти обращения составляют основные затраты при поиске элементов.

Полученная структура называется В-деревом, а  $n$  – порядок В-дерева.

#### Свойства В-деревьев:

1. Каждая страница содержит не более  $2n$  элементов (ключей).
2. Каждая страница, кроме корневой, содержит не менее  $n$  элементов.
3. Каждая страница либо представляет собой лист, т.е. не имеет потомков, либо имеет  $m+1$  потомков, где  $m$  – число ключей на этой странице.
4. Все страницы-листья находятся на одном уровне.



## Рис.2 В-дерево порядка 2

Рассмотрим В-дерево (Рис.2) порядка 2. Все страницы дерева содержат от 2 до 4 элементов. Исключение составляет лишь корневая вершина, в которой может находиться один элемент. Все листья находятся на третьем уровне. Если спроецировать В-дерево на один единственный уровень, включая потомков между ключами их родительской страницы, то ключи идут в возрастающем порядке слева направо.

### Поиск и включение для В-деревьев

Пусть страница В-дерева имеет вид как на Рис.3 и задан некоторый аргумент поиска  $x$ . Если поиск ключа на текущей странице дерева неудачен, то мы попадаем в одну из следующих ситуаций:

1.  $k_i < x < k_{i+1}$  для  $1 \leq i < m$ . Поиск продолжается на странице  $p_i$
2.  $k_m < x$ . Поиск продолжается на странице  $p_m$ .
3.  $x < k_1$ . Поиск продолжается на странице  $p_0$ .

Если указанная ссылка пуста, то в дереве нет элемента с ключом  $x$ .

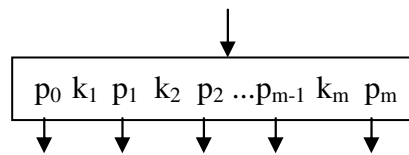


Рис.3 Страница В-дерева с  $m$  ключами

В отличие от бинарных деревьев рост В-деревьев происходит от листьев к корню. Если элемент нужно поместить на страницу с  $m < 2n$  элементами, то процесс включения затрагивает лишь текущую страницу. Включение в полную страницу затрагивает структуру дерева и может привести к появлению новых страниц.

Рассмотрим пример (Рис.4), где в В-дерево порядка 2 включается элемент 22.

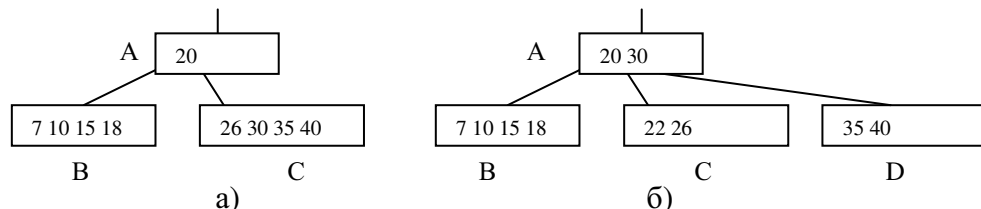


Рис. 4 Включение в В-дерево ключа со значением 22

Включение происходит за 3 шага:

1. Обнаруживается, что ключ 22 отсутствует. Включение в страницу C не возможно, т.к. она уже заполнена (Рис.4а)
2. Страница C разделяется на две страницы (т.е. вводится новая страница D)
3. Ключи – их всего  $2n+1$  – поровну распределяются в C и D, а средний ключ переносится на один уровень вверх на «родительскую страницу» (Рис.4б)

Включение элемента в родительскую страницу может снова привести к переполнению страницы, и разделение будет распространяться. Таким способом оно может дойти до корня, и только в этом случае увеличится высота дерева.

Алгоритм построения В-дерева требует описания следующих типов данных:

## Type

```
Ptr = ^page;
{ключ}
Item = record
    Key:integer; {значение ключа}
    P:ptr;
End;
{страница B-дерева}
Page = record
    M:[0..2n]; {фактическое число элементов на странице}
    P0:ptr;
    E:array [1..2n] of item; {массив ключей страницы}
End;

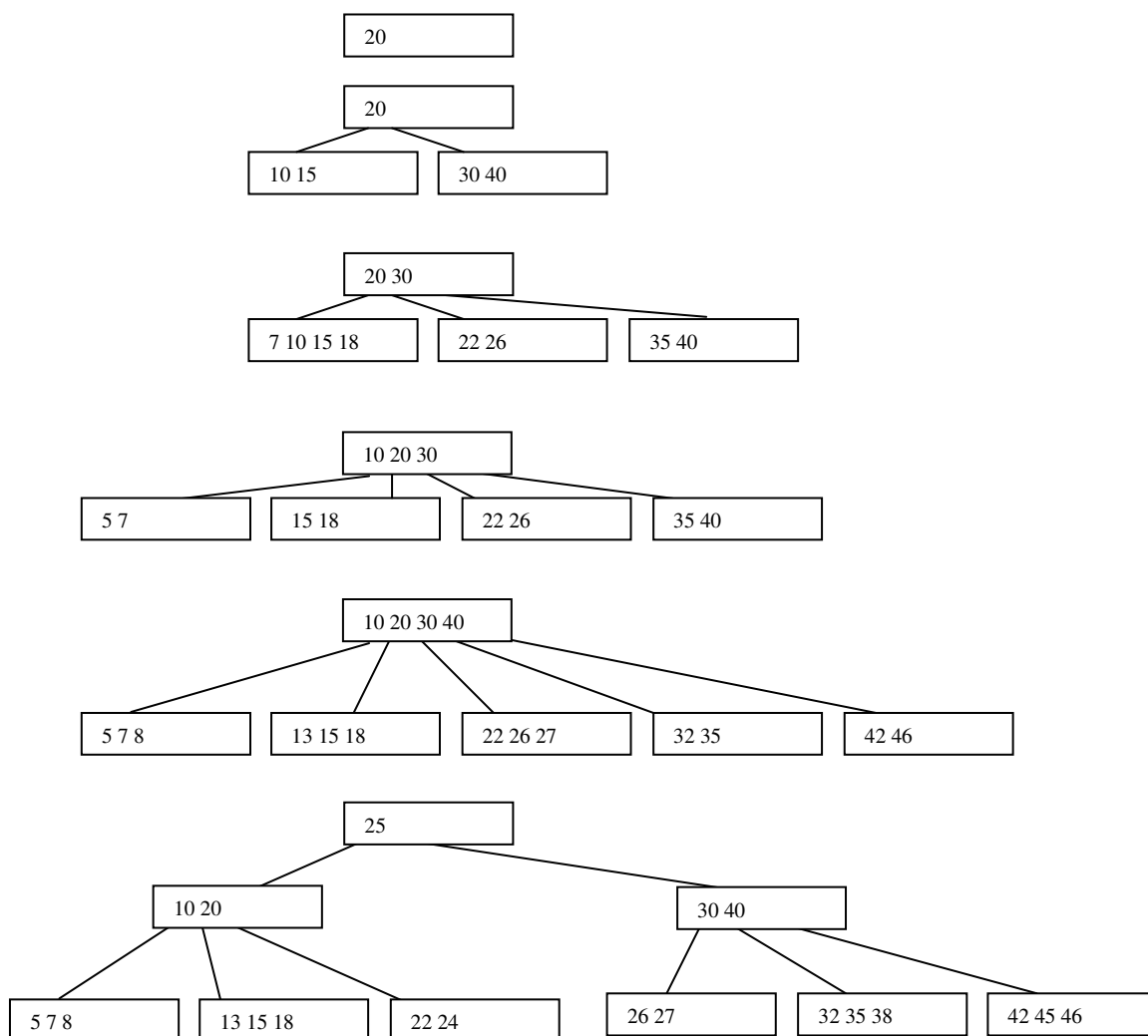
{поиск и включение элемента}
Procedure search (x:integer; a :ptr; var h:boolean; var u:item);
Begin
    If a= nil then {x в дереве нет, включение}
        Элементу u присвоить x, h установить в true, указывая, что
        элемент u в дереве передается вверх
    Else
        With a do
            Поиск x в массиве e
            If найдено then обработка
            Else
                begin
                    Search(x,потомок, h, u);
                    If h then{элемент}
                        Begin
                            If число элементов на странице a < 2n then
                                Включение u на страницу a и установка h
                                равным false
                            else расщепление страницы и передача вверх
                                среднего элемента
                        end;
                    End;
                End;
    End;
```

Булевский параметр h как и в алгоритме построения сбалансированного дерева указывает на рост дерева. Если процедура search в основной программе была вызвана со значением h = true, то требуется разделение корневой страницы, в противном же случае h сигнализирует о том, что некоторый элемент нужно передать вверх по дереву.

## Пример.

Построить B-дерево по следующей последовательности:

20; 40 10 30 15; 35 7 26 18 22; 5; 42 13 46 27 8 32; 38 24 45 25;  
( ; - означает появление новой страницы) Рис.5



**Рис.5 Рост В-дерева**

## Исключение из В-деревьев

При исключении элемента из В-дерева возможно два случая:

1. Исключаемый элемент находится на листовой странице
2. Элемент не находится на листовой странице, и его нужно заменить одним из двух лексикографически смежных элементов.

Исключение ключа в случае 2 похоже на исключение из двоичного дерева. Мы спускаемся вниз вдоль самых правых ссылок до листовой страницы Р, заменяя исключаемый элемент на правый элемент из Р и уменьшая размер Р на единицу.

При уменьшении размера страницы число элементов на ней может стать меньше  $n$ , что нарушит основное условие В-дерева. Эта ситуация потребует корректировки дерева.

Единственный выход – позаимствовать элемент с одной из соседних страниц, например с Q. Обычно элементы Р и Q поровну распределяются на обе страницы. Этот процесс называется балансировкой страниц. Может возникнуть ситуация, когда страница Q содержит минимальное число ключей, тогда общее число элементов на Р и Q будет  $2n-1$  и можно «слить» обе страницы в одну, добавив сюда средний элемент из родительской (для Р и Q) страницы, а затем уничтожив Q.

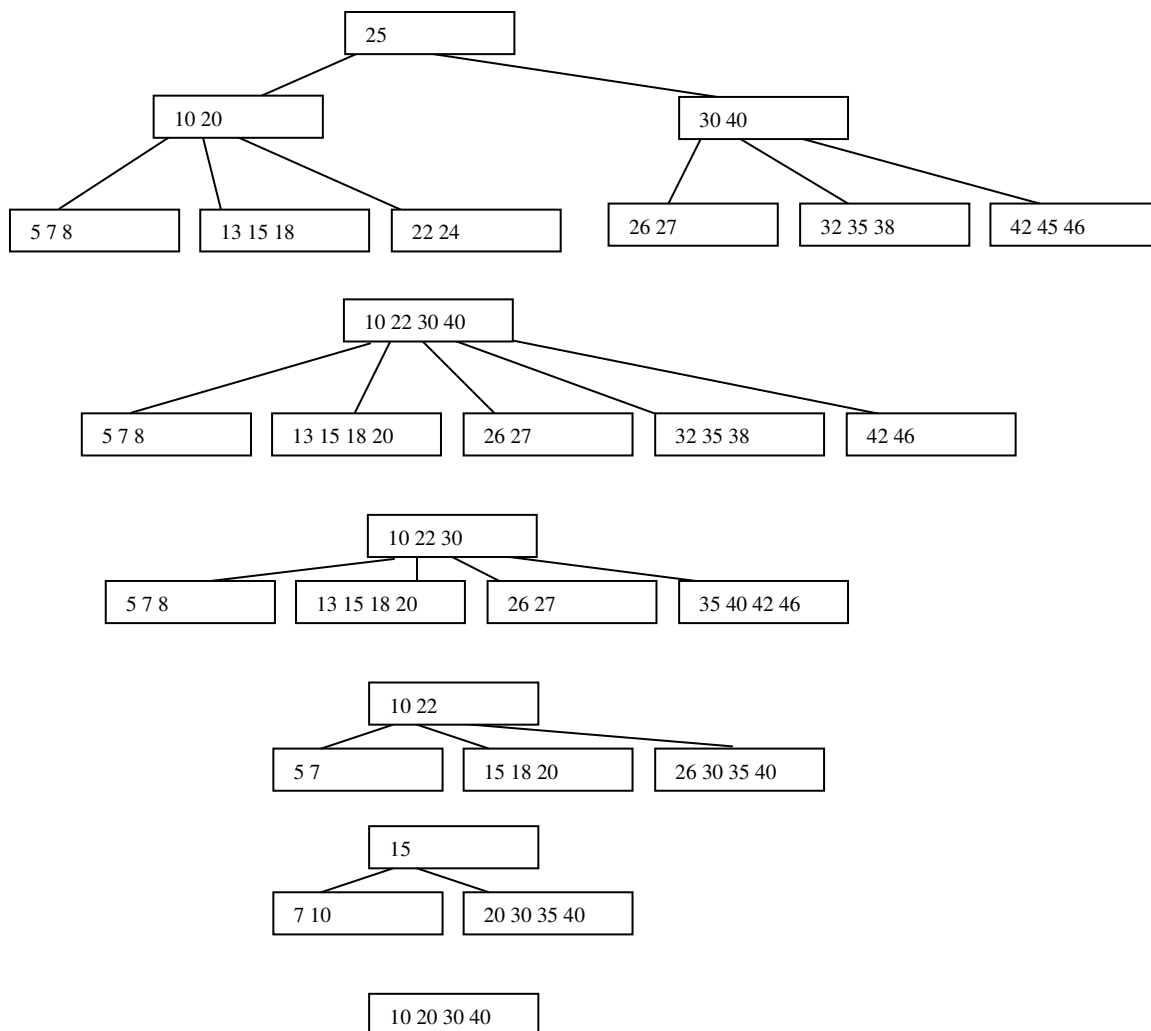
Удаление среднего ключа на родительской странице вновь может привести к аналогичной ситуации, тогда потребуются дополнительные процедуры (балансировки или слияния) на следующем уровне. В экстремальном случае этот процесс может дойти до корня дерева, и, если корень сократится до нулевого размера, то высота дерева уменьшится.

Пример.

Исключить из дерева (Рис.5) ключи:

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

( ; – фиксирует моменты уничтожения страниц) Рис.6



**Рис.6 Деграция В-дерева**



## Деревья приоритетного поиска

Двоичные деревья являются эффективной организацией данных, для которых существует отношение линейной упорядоченности. Рассмотрим тип данных, представляющий собой точки на плоскости, т.е. данные из двумерного пространства  $(x, y)$ .

Используя полученные знания о деревьях поиска, мы можем построить два типа деревьев:

1. бинарное дерево поиска, расположение ключей в котором отвечает правилу:

$$p.\text{left} \neq \text{nil} \rightarrow P.\text{left}.x < p.x$$

$$p.\text{right} \neq \text{nil} \rightarrow P.\text{right}.x > p.x$$

2. пирамиду, называемую также деревом с приоритетом

$$p.\text{left} \neq \text{nil} \rightarrow p.y \leq P.\text{left}.y$$

$$p.\text{right} \neq \text{nil} \rightarrow p.y \leq p.\text{right}.y$$

Объединив эти два правила, мы получим **декартово дерево** или **дерево поиска с приоритетом**:

$$p.\text{left} \neq \text{nil} \rightarrow (p.\text{left}.x < p.x) \& (p.y \leq P.\text{left}.y)$$

$$p.\text{right} \neq \text{nil} \rightarrow (p.x < p.\text{right}.x) \& (p.y \leq p.\text{right}.y)$$

Однако, использование таких деревьев крайне затруднено из-за сложности поддержания структуры дерева. При включении некоторого узла в такое дерево может возникнуть ситуация (Рис.7), требующая больших вычислительных затрат.

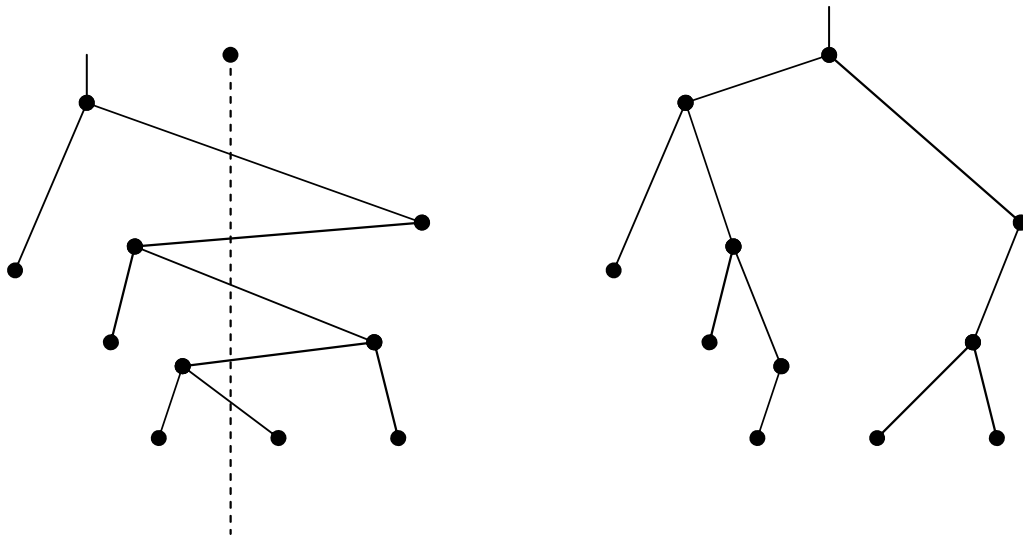


Рис.7 Включение в дерево приоритетного поиска

В связи с этим рассмотрим структуру, предложенную Маккрейтом. Вместо неограниченного пространства поиска будем рассматривать область, ограниченную прямоугольником с двумя открытыми сторонами. Граничные значения координаты  $x$  назовем  $x_{\min}$  и  $x_{\max}$ .

Свяжем с каждой вершиной  $p$  некоторый интервал  $[p.L .. p.R)$ , в котором может находиться значение вершины. Затем постулируем, что левый потомок вершины, если он есть, должен находиться внутри левой половины этого интервала, а правый – внутри правой половины.

Для каждого потомка интервал уменьшается вдвое, и высота дерева ограничена величиной  $\log_2(x_{\max} - x_{\min})$ .

Обобщим введенные условия правилом:

$$p.\text{left} \neq \text{nil} \rightarrow (p.L \leq p.\text{left}.x < p.M) \& (p.y \leq P.\text{left}.y)$$

$$p.\text{right} \neq \text{nil} \rightarrow (p.M \leq p.\text{right}.x < p.R) \& (p.y \leq p.\text{right}.y),$$

$$\text{где } p.M = (p.L + p.R) \text{ div } 2,$$

$$p.\text{left}.L = p.L,$$

$$p.\text{left}.R = p.M,$$

$p.\text{right}.L = p.M,$

$p.\text{right}.R = p.R$  для всех вершин  $p$ ,

для корня  $\text{root}.L = x_{\min}, \text{root}.R = x_{\max}.$

Данная структура называется **корневым деревом поиска с приоритетом** (radix priority search trees).

Число операций, затрачиваемых на поиск, включение и исключение из таких деревьев ограничено логарифмической оценкой.

## Лекция 8

### Введение в объекты

#### **Объекты материального мира**

Все, что нас окружает вокруг – материальные объекты. Они обладают свойствами, которые отличают одни объекты от других, помогают их классифицировать. Кроме того, часть свойств характеризует объект по заранее выделенным признакам, таким как цвет, форма, размер. Они могут быть существенными для идентификации и описания объекта, а могут быть и не нужны.

Компьютерная модель объекта – это набор параметров, с некоторой точностью описывающих заданный объект. Уровень детализации модели (сложность описания объекта) определяет в конечном итоге ее возможности.

#### Пример.

Рассмотрим стол.

Простейшая компьютерная модель стола будет состоять из следующих переменных: длина, ширина, высота, цвет, материал.

Предложенное описание может дать отдаленное представление о столе, как о предмете мебели, но совершенно не отвечает на такие вопросы, как форма стола (круглый, квадратный), вместимость (количество ящиков, внутренний объем) и назначение (письменный, кухонный, журнальный).

Оперировать отдельными переменными, отвечающими за различные свойства объекта неудобно, т.к. велика вероятность путаницы между одинаковыми параметрами различных объектов, кроме того, доступ к этим параметрам разрешен всем желающим, и как следствие контроль над данными невозможен.

Для исключения этих проблем при написании компьютерных моделей разработана специальная технология, позволяющая оперировать объектами, как единым целым. Иными словами, говоря «стол», мы одновременно подразумеваем и все его свойства, и методы описания и изменения этих свойств.

Эта технология носит название объектно-ориентированного подхода.

ООП предоставляет возможность моделирование множества однородных объектов, отличающихся лишь в частности.

#### Пример.

Журнальный стол и письменный стол принадлежат к одному и тому же виду объектов, который можно назвать общим словом «столы». Их роднит набор одних свойств, перечисленных в предыдущем примере, но различают величины, которые могут принимать эти свойства, а также назначение объектов, т.е. функции, которые с ними связаны.

Имея описанный выше объект «стол», добавляя к нему свойства «выдвижной ящик» и «подставка для ног», а также новые функциональные возможности, получаем объект «письменный стол».

Таким образом, мы на примерах рассмотрели основные свойства ООП:

- инкапсуляция
- наследование,
- полиморфизм.

Не смотря на то, что ООП широко распространено в компьютерном мире, в разные языках принята разная терминология ООП. Например, в языке Pascal тип, описывающий

объект, называется Object, в то время как в языке C++ он носит название class, а объектом называется переменная типа class (экземпляр класса). Следует понимать, что в каждом языке ООП реализован по-своему.

## Инкапсуляция

*Инкапсуляция*– это объединение внутри объекта данных и методов их обработки, где данные являются свойствами объекта, а методы – функциями, реализующими доступ к данным и операции над ними.

Понятие инкапсуляции тесно связано с сокрытием информации внутри объекта. Т.к. существуют методы, обеспечивающие доступ к данным внутри объекта, программист имеет возможность проконтролировать корректность передаваемых значений. Если при этом сами данные недоступны из внешней по отношению к объекту программы, тогда можно предусмотреть механизм, при котором данные внутри объекта принимают лишь допустимые значения.

Пример .

Рассмотрим объект автомобиль, который движется равномерно и прямолинейно. Поставим перед моделью объекта задачу определения перемещения автомобиля относительно начальной точки движения. Описание типа будет выглядеть следующим образом:

Type

```
TAuto = object
```

Private:

```
V:real; {скорость автомобиля}
```

```
T:real; {время в пути}
```

**Public:**

```
Procedure SetTime(time:real); {установить время}
                                {движения}
```

```
Function GetTime: real;           {вернуть время движения}
```

```
Procedure SetSpeed(speed:real); {установить скорость}
                                {движения}
```

**Function** GetSpeed: real; {вернуть скорость движения}

```
Procedure Run; {вернуть перемещение}
```

End;

Опустим пока объяснение ключевых слов `private` и `public`. Рассмотрим содержимое нашего объекта. Для описанного объекта мы выделили два основных свойства: скорость и время. Остальные характеристики не представляют для нас интереса в рамках поставленной задачи.

Основное назначение процедур SetTime и SetSpeed в том, чтобы задавать и контролировать значения свойств объекта s и t, вводимых из вне, в противовес прямому обращению к этим полям.

```
Procedure Tavto.SetTime (time:real);
```

## Begin

```
If time < 0 then write("Время не может быть отрицательным")
```

```
ElseT := time;
```

End;

```
Procedure Tavto.SetSpeed (speed:real);
```

## Begin

$$V := \text{speed};$$

**End;**

Функции GetSpeed и GetTime только возвращают значения соответствующих параметров. Функция Run выдает интересующую нас информацию о пройденном автомобилем пути.

```
FunctionTavto.GetTime:real;
```

```
Begin
```

```
    GetTime := T;
```

```
End;
```

```
FunctionTavto.GetSpeed:real;
```

```
Begin
```

```
    GetSpeed := V;
```

```
End;
```

```
ProcedureTavto.Run;
```

```
Begin
```

```
    Write("Перемещение относительно начальной точки: ",  
          abs(v*t));
```

```
End;
```

Так может выглядеть кусок программы, использующей описанный выше тип:

```
var
```

```
    MyAuto : auto;
```

```
Begin
```

```
...
```

```
    MyAuto.SetTime(15.0);
```

```
    MyAuto.SetSpeed(20.0);
```

```
    MyAuto.Run;
```

```
...
```

```
End.
```

Результатом работы программы будет строка:

Перемещение относительно начальной точки : 300.00

В случае неверного ввода значений скорости или времени программа все равно выдаст результат, но его правильность будет сомнительна. Об ошибке сообщит соответствующая процедура write в функциях SetSpeed и SetTime.

### ***Усложнение модели. Наследование.***

Более сложный вид движения – равноускоренное. На основе описанного типа Tauto создадим тип Tautomobile, моделирующий этот вид движения. Для создания типа Tautomobil используем свойство наследования.

*Наследование*– это способность нового объекта - потомка перенимать часть свойств объекта - родителя. В ООП механизм наследования позволяет объекту – потомку иметь данные и методы родительского объекта.

```
TAutomobil = object (TAuto) {Tautomobil – потомок TAuto}
```

```
    Private:
```

```
        A:real; {ускорение}
```

```
    Public:
```

```

procedure SetAcceleration(acc:real);{установить ускорение}
function GetAcceleration: real;      {вернуть ускорение}
Procedure Run;                      {вернуть перемещение}
End;

```

Поля  $v$  и  $t$  явно не описаны в объекте Tautomobile, но присутствуют там, т.к. Они есть у его родителя. Это же касается методов SetTime, GetTime, SetSpeed и GetSpeed. Поле  $a$  – ускорение является атрибутом только объекта Tautomobil, как и новые функции SetAcceleration и GetAcceleration.

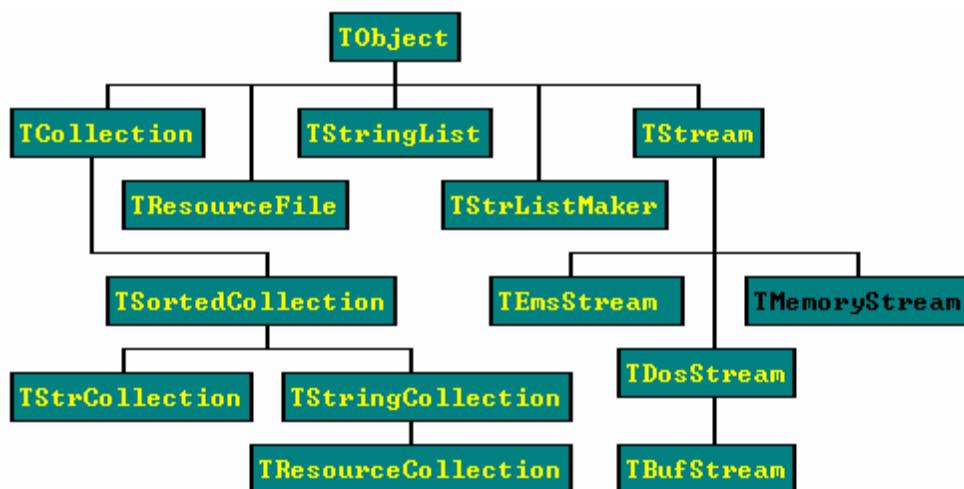
Особый интерес представляет процедура Run, которую новый объект “перекрывает” в своем описании, т.е. заменяет метод из родительского объекта другим методом с тем же именем, но другим содержанием.

```

Procedure Tautomobil.Run;
Begin
    Write("Перемещение относительно начальной точки: ",
        abs(v*t+a*t*t/2));
End;

```

Построив новый класс, мы расширили возможности начальной модели, увеличив ее точность. Но уточнение модели – это не единственное преимущество наследования. С помощью механизма наследования можно расширять возможности исходных объектов, получая совершенно новые объекты. На основе этого строятся иерархии объектов:



**Рис.1 Иерархия объектов библиотеки TurboVision**

Изначально ООП применялся при построении интерфейсов программ, т.к. большинство программ имеют одинаковые части, обеспечивающие ввод и вывод данных. При вводе данных вне зависимости от приложения необходимо проверять данные на корректность, например, при вводе числовых данных пользователь может ввести буквы. При применении стандартного подхода с использованием функции Read программа в этом случае выдаст ошибку периода исполнения и завершит работу. Используя объектно-ориентированный подход, мы можем избежать такой ситуации.

Т.к. построение окон, диалогов, меню, хранилищ данных – это общие задачи для любой программы, то создателями средств разработки было решено построить библиотеки на основе ООП, реализующие интерфейсные части программ, в частности TurboVision. Затем этот подход был распространен на другие части разработки программ. В дальнейшем ООП распространилось целиком на все части компьютерной деятельности: операционные системы,

базы данных, прикладные программы.

## **Различное поведение объектов. Полиморфизм.**

Объекты более высоких уровней иерархии как правило разительно отличаются от объектов более низших уровней и по своим свойствам, и по функциям. Помимо того, что они имеют значительно сложное строение, чем их первоначальный предок, их поведение также будет различным.

*Полиморфизм* – это свойство объекта потомка изменять методы, унаследованные от объекта родителя.

С полиморфизмом мы уже встречались в примере с объектом `Tautomobil`. Функция `Run` выполнялась по-разному в потомке и в родителе.

Полиморфизм делает более гибким наследование, более того наследование без полиморфизма в разработке объектов-потомков ограничивает возможности потомка или требует необоснованного роста количества методов. Т.е. В приведенном выше примере нам потребовалось бы создать новый метод `Run1`.

При описании объекта `Tauto` мы использовали ключевые слова: `public` и `private`. Настало время объяснить их назначение. Мы уже говорили о том, что инкапсуляция скрывает данные внутри объекта, защищая их от прямого воздействия. Это обеспечивается заданием методов доступа к компонентам объектов. Методы доступа могут носить различный характер и описываются спецификаторами `private`, `public` и `protected` (закрытый, открытый, защищенный).

Спецификатор `public`, предшествующий объявлению поля или метода сигнализирует о том, что доступ к этому полю или методу разрешен из любой точки программы через имя соответствующего объекта.

Спецификатор `private`, означает, что доступ к полю или методу объекта из вне запрещен, а вызов закрытых процедур и функций, а также изменение значений закрытых полей возможен только внутри методов самого объекта. Кроме того объекты – наследники также не имеют доступ к закрытым частям объекта – родителя.

Спецификатор `protected` аналогичным образом защищает компоненты объекта от доступа из вне, но разрешает объектам-потомкам пользоваться защищенными компонентами.

В языке паскаль спецификаторы доступа к компонентам объектов имеют значение лишь на уровне модулей, в то время как в языке C++ они действуют независимо от этого.

## **Виртуальные методы. Позднее связывание.**

Пример.

**Type**

`A = object`

`constructor Init;`

`procedure writeself;`

`procedure print;`

`destruvtor Done;`

`end;`

`b = object (a)`

`constructor Init;`

`procedure writeself;`

`destructor Done;`

```

end;

constructora.Init;
begin
end;
destructora.Done;
begin
end;
procedurea.writeself;
begin
    writeln('объект a');
end;
procedurea.print;
begin
    writeself;
end;
constructorb.Init;
begin
end;
destructorb.Done;
begin
end;
procedureb.writeself;
begin
    writeln('объект b');
end;

```

Рассмотрим кусок программы.

```

Var
    object_a:a;
    object_b:b;
begin
    object_a.print;
    object_b.print;
end;

```



Результатом ее работы будет:

**объект a**

**объект a**

Исправим описание объекта a:

```
a = object
...
    procedure writeself; virtual;
...
end;
```

Результат работы программы изменится:

**объект a**

**объект b**

При описании объекта *a* в первом случае компилятор на этапе компиляции произведет подстановку адреса подпрограммы `writeself` из этого же объекта в функцию `print`. Следовательно, объект *b* унаследует функцию `writeself` в том же виде.

Во втором случае спецификатор `virtual` покажет компилятору, что в месте вызова функции `writeself` из `print` следует оставить указатель на метод неопределенным (ссылающимся на относительный адрес строки таблицы виртуальных методов) и проинициализировать его только на этапе исполнения. Это свойство называется поздним связыванием.

Таблица виртуальных методов создается специальным методом – конструктором объекта. Каждая строка таблицы содержит физический адрес соответствующей виртуальной функции. В момент вызова виртуальной функции в программе происходит обращение к виртуальному методу по относительному адресу в таблице виртуальных функций данного объекта.

Основное преимущество ООП состоит в *повторном использовании ранее написанного кода*, что обеспечивает динамичное развитие программных продуктов, накопление опыта и аккумуляцию наиболее удачных решений в базовых объектах иерархии.

Следующим важным преимуществом является сокрытие данных и разделение содержимого объекта на интерфейс и внутреннюю часть. Это позволяет защитить данные от случайного или преднамеренного разрушения.

## Лекция 9

### Поиск

Одна из распространенных задач программирования – поиск. Предположим, что группа данных, в которой следует отыскать заданный элемент фиксирована и задана, например, в виде массива

A:array [0..N-1] of item,

где item – запись с некоторым полем – ключом. Задача заключается в поиске элемента, ключ которого соответствует заданному «аргументу поиска» x:

A[i].key = x. Предположим для начала, что тип item это и есть ключ (key).

#### **Линейный поиск**

Представляет самый простой метод поиска, когда не имеется никакой дополнительной информации об элементах поиска. При этом последовательно просматривается весь массив элементов:

```
I := 0;
While (i < N) and (a[i] <> x) do
  i := i+1;
```

При этом значение i = N свидетельствует о том, что искомый элемент не найден.

Алгоритм можно упростить, если ввести в массив a дополнительный N элемент – «барьер», равный x. Тогда алгоритм поиска будет выглядеть следующим образом:

```
I := 0;
A[N] := x;
While (a[i] <> x) do
  i := i+1;
```

При этом значение i = N также свидетельствует о том, что искомый элемент не найден.

#### **Анализ линейного поиска**

В самом плохом случае данному алгоритму потребуется N сравнений, чтобы найти искомый элемент.

#### **Поиск делением пополам**

Очевидно, что поиск элемента будет гораздо эффективнее, если данные упорядочены. Т.е массив a удовлетворяет условию:

$\forall k: 0 < k < N: a_{k-1} \leq a_k.$

Отсюда поиск можно организовать следующим образом: выбирается случайный элемент  $a_m$  и его ключ сравнивается с аргументом поиска x. Если он равен x, то поиск заканчивается, если меньше, то исключаем из поиска все элементы массива a с индексами меньшими или равными m; если же он больше x, то исключаем индексы большие m.

Обозначим L левую границу поиска, а R – правую. M – индекс среднего элемента.

```
L:=0; R := N-1; found := false;
While (L <= R) and (not found) do
Begin
  M := (L+R) div 2;
  If (a[m] = x) then found:=true;
  Else
    If a[m] < x then L:=m+1
    Else R:=m-1;
```

**End;**

Алгоритм можно улучшить, используя утверждение:

$(\forall k: 0 \leq k < L: a_k < x) \ \& \ (\forall k: R \leq k < N: a_k \geq x)$

**L:=0; R := N;**

**While** (L < R) **do**

**Begin**

**M** := (L+R) div 2;

**If** a[m] < x **then** L:=m+1

**Else** R:=m;

**End;**

При R = N никаких совпадений нет. В остальных же случаях следует проверить равенство a[R] = x, что даст окончательный ответ.

### **Анализ алгоритма бинарного поиска**

Оптимальное решение при двоичном поиске дает выбор среднего элемента. В этом случае на каждом шаге отсекается ровно половина массива. В результате максимальное число сравнений равно  $\lceil \log_2 N \rceil + 1$ .

### **Однородный бинарный поиск с вычислениями**

Модифицировать представленный выше алгоритм можно, если вместо трех индексов m, L, R хранить только два: текущее положение i и величину его изменения  $\delta$ . После каждого сравнения, **недавшего равенства**, можно установить  $i=i+\delta$  и  $\delta=\delta/2$ .

Алгоритм:

1. *Инициализация:*

$i = N/2, m = N/2.$

2. *Сравнение:*

    если  $x < a[i]$ , то перейти к шагу 3;

    если  $x > a[i]$ , то перейти к шагу 4;

    если  $x = a[i]$ , то алгоритм закончен удачно.

3. *Уменьшение i:*

    {мы определили интервал продолжения поиска, содержащий m или m-1 записей; i указывает на первый элемент справа от интервала}

    если  $m = 0$  алгоритм закончен неудачно;

    в противном случае  $i = i-m/2; m = m/2$ ; перейти к шагу 2.

4. *Увеличение i:*

    {мы определили интервал продолжения поиска, содержащий m или m-1 записей; i указывает на первый элемент слева от интервала}

    если  $m = 0$  алгоритм закончен неудачно;

    в противном случае  $i = i+m/2; m = m/2$ ; перейти к шагу 2.

### **Однородный бинарный поиск со вспомогательной таблицей**

Этот алгоритм подобен предыдущему, но использует для вычисления вспомогательную таблицу:

$\text{delta}[j] = \lceil (N+2^{j-1})/2^j \rceil$ , для  $1 \leq j \leq \lceil \log_2 N \rceil + 2$

Алгоритм:

1. *Инициализация:*

- $i = \text{delta}[1], j=2.$
2. *Сравнение:*  
 если  $x < a[i]$ , то перейти к шагу 3;  
 если  $x > a[i]$ , то перейти к шагу 4;  
 если  $x = a[i]$ , то алгоритм закончен удачно.
  3. *Уменьшение  $i$ :*  
 если  $\text{delta}[j] = 0$  алгоритм закончен неудачно;  
 в противном случае  $i = i - \text{delta}[j]; j = j+1$ ; перейти к шагу 2.
  4. *Увеличение  $i$ :*  
 если  $\text{delta}[j] = 0$  алгоритм закончен неудачно;  
 в противном случае  $i = i + \text{delta}[j]; j = j+1$ ; перейти к шагу 2.

## Поиск в таблице

Рассмотрим поиск в более сложных массивах данных, элементы которых являются составными объектами. Например, массив строк, оканчивающихся нулем. Строковый тип определен так:

```
String: array [0..M-1] of char;
```

Определим отношение порядка для строк:

$(x = y) = (\forall j: 0 \leq j < M: x_j = y_j),$

$(x < y) = \exists i: 0 \leq i < N : ((\forall j: 0 \leq j < i: x_j = y_j) \ \& \ (x_i < y_i)),$

Сравнение строк в этом случае выполняется по следующему алгоритму:

```
I:=0;
While (x[i] = y[i]) and (x[i] <> #0) do
    I:= i+1;
```

Если цикл закончил свою работу по причине  $x[i] = \#0$ , то сравниваемые строки равны, в противном случае сравнение  $x[i]$  с  $y[i]$  даст окончательный результат. Заметим, что определенные выше отношения порядка для строк имеют лексикографический смысл.

Поиск в таблице требует вложенных поисков. Допустим, что  $N$  достаточно велико, и таблица упорядочена в алфавитном порядке. Используем алгоритм деления пополам:

```
T: array [0 .. N-1] of String;
X: string;
...
L:= 0 ; R := N;
While L < R do
Begin
    M:= (L + R) div 2; i:= 0;
    While (T[m,i] = x[i]) and (x[i] <> #0) do
        I:= i+1;
    If T[m,i] < x[i] then L := m+1
        else R := m;
End;
If R < N then
begin
    i:= 0;
    While (T[R,i] = x[i]) and (x[i] <> #0) do
        I:= i+1;
End;
```

В данном алгоритме совпадение строк удовлетворяет условию :  
 $(R < N) \ \& \ (T[R,i] = x[i])$

### **Прямой поиск строки**

Во многих случаях приходится сталкиваться с такой задачей, как поиск строки. Ее можно определить следующим образом: пусть заданы два массива  $s$  из  $N$  элементов и  $p$  из  $M$  элементов. Причем,  $0 < M \leq N$ .

```
S : array [0 .. N-1] of item;  
P : array [0 .. M-1] of item;
```

Поиск строки обнаруживает первое вхождение  $p$  в  $s$ . Обычно  $item$  – это символ,  $s$  можно считать некоторым текстом, а  $p$  – образом или словом. Результатом задачи будет индекс  $i$ , указывающий на первое совпадение с образом.

Введем предикат  $P(i,j)$ :

$$P(i,j) = \forall k: 0 \leq k < j: s_{i+k} = p_k. \quad (1)$$

Найденный индекс должен удовлетворять этому предикату,  $P(i,M)$ . Но этого не достаточно. Т.к. поиск обнаруживает первое вхождение образа, то  $P(k,M)$  должен быть ложным для всех  $k < i$ . Обозначим это условие через  $Q(i)$ :

$$Q(i) = \forall k: 0 \leq k < i: \sim P(k,M). \quad (2)$$

Задача сводится к такому алгоритму:

```
I := -1;  
Repeat i := i+1 { Q(i) }  
    Found := P(i,M)  
Until found or (i = N-M)
```

Вычисление  $P$  сводится к повторяющемуся сравнению отдельных символов. Применив правило Моргана к (1) получим:

$$P(i,j) = (\sim \exists k: 0 \leq k < j: s_{i+k} \neq p_k) \quad (3)$$

Т.о. алгоритм ищет «несовпадение» между образом и строкой символов. В результате этого уточнения получаем программу:

```
I := -1;  
Repeat  
    i := i+1; j := 0 ; { Q(i) }  
    while (j < M) and (s[i+j] = p[j]) do  
        j := j+1; { P(i, j+1) }  
    { Q(i) & P(i, j) & ((j = M) or (s[i+j] <> p[j])) }  
Until (j = M) or (i = N - M);
```

$J = M$  в условии окончания соответствует условию  $found$ , т.к. из него следует  $P(i,M)$ .  $I = N - M$  говорит об отсутствии совпадений, т.е.  $Q(N-M)$ .

### **Анализ прямого поиска строки.**

Алгоритм работает достаточно эффективно, если допустить, что несовпадение происходит спустя всего лишь нескольких сравнений. Но в худшем случае производительность будет низкой. Пример, возьмем строку из  $N-1$  символов 'А' и одного символа 'В' и образ, состоящий из  $M-1$  символа 'А' и одного 'В'. Чтобы обнаружить совпадение в конце строки следует провести порядка  $N \cdot M$  сравнений.

## Поиск в строке. Алгоритм Кнута, Морриса и Пратта (КМП - алгоритм)

Алгоритм КМП основан на том факте, что начиная каждый раз сравнивать с начала образа мы можем уничтожать ценную информацию. После частичного совпадения начальной части образа с соответствующими символами строки мы можем вычислить некоторые сведения, с помощью которых быстро продвинемся по тексту.

Пример.

```
Н о о l а - Н о о l а   g i r l s   l i k e   Н о о l i g a n s
Н о о l i g a n
      Н о о l i g a n
        Н о о l i g a n
          Н о о l i g a n
            Н о о l i g a n
              . . .
                                Н о о l i g a n
```

Обратите внимание, что в данном примере при каждом несовпадении двух символов, образ сдвигается на все пройденное расстояние.

КМП-алгоритм записывается так:

```
I:=0; j:=0;
While (j < M) and (i < N) do
begin
  { Q(i-j) & P(i-j, j) }
  while (j >= 0) and (s[i] <> p[j]) do
    j := D;
  i:= i+1;
  j:= j+1;
end;
```

D определяет сдвиг образа на неопределенную пока величину.

В КМП-алгоритме индекс  $i$  не отмечает позицию первого символа образа в тексте. Выровненное положение образа теперь  $i-j$ . При этом  $0 \leq i < N$  и  $0 \leq j < M$ . Условия  $Q(i-j)$  и  $P(i-j, j)$  сохраняются как глобальные.

Если алгоритм заканчивает работу по причине  $j = M$ , то из  $P(i-j, j)$  следует справедливость  $P(i-M, M)$ , т.е. совпадение начинается с позиции  $i-M$ . Если работа окончена по причине  $i = N$ , то т.к.  $j < M$ , то из  $Q(i)$  следует, что совпадений нет.

Прежде чем найти D докажем, что алгоритм никогда не делает инвариант  $Q(i-j) \& P(i-j, j)$  ложным.

Легко видеть, что в начале  $i=j=0$ , откуда следует истинность  $Q(i-j)$  и  $P(i-j, j)$ .

Операторы, увеличивающие  $i$  и  $j$  на 1 не сдвигают образ вправо, т.к. разность  $i-j$  остается неизменной. Значит  $Q(i-j)$  истинно.

$P(i-j, j)$  также истинно в этом случае, т.к. из условия цикла следует, что либо  $j < 0$ , либо  $s_i = p_j$  (по правилу Моргана  $(j \geq 0) \text{ and } (s[i] \neq p[j]) = (j < 0) \text{ or } (s[i] = p[j])$ ). Если  $j < 0$ , то  $P(i-j, j+1)$  истинно из (2), если  $s_i = p_j$   $P(i-j, j+1)$  истинно из (1).

И наконец, постулируем, что присваивание  $j = D$  не делает инвариант ложным.

Найдем выражение для D.

При  $D < j$  присваивание  $j = D$  соответствует сдвигу образа вправо на  $j-D$  позиций. Желательно, чтобы сдвиг был бы настолько большим, насколько это возможно, т.е. D было бы малым (Рис.1).

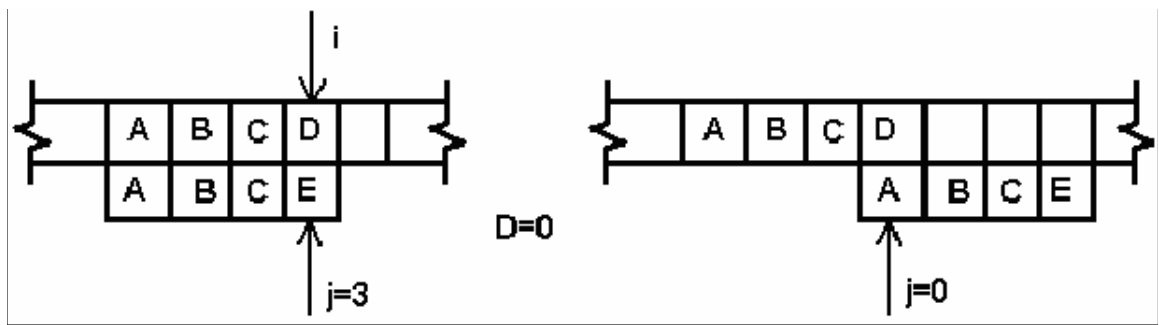


Рис.1 Присваивание  $j = D$  сдвигает образ на  $j - D$  позиций

Если инвариант  $Q(i-j) \ \& \ P(i-j,j)$  после присваивания  $j$  значения  $D$  истинен, то перед ним должно быть истинно  $Q(i-D) \ \& \ P(i-D,D)$ .

Из (1) следует:

$$S_{i-j} \dots S_{i-1} = p_0 \dots p_{j-1}$$

Первые рассмотренные нами  $j$  символов образа совпадают с текстом и из  $P(i-D,D)$  с  $D \leq j$  следует:

$$p_0 \dots p_{D-1} = S_{i-D} \dots S_{i-1}$$

Т.к.  $S_{i-1}$  соответствует  $p_{j-1}$ ,  $S_{i-2}$  соответствует  $p_{j-2}$  ...  $S_{i-D}$  соответствует  $p_{j-D}$ , то

$$p_0 \dots p_{D-1} = p_{j-D} \dots p_{j-1}, \quad (4)$$

и предикат  $\sim P(i-k, M)$  для  $k=1, \dots, j-D$ .  $Q(i-D)$  превращается в

$$p_0 \dots p_{k-1} \neq p_{j-k} \dots p_{j-1} \text{ для всех } k = 1, \dots, j-D. \quad (5)$$

Эти рассуждения подводят нас к важному выводу: значение  $D$  определяется одним лишь образом и не зависит от строки текста.

Из (4-5) следует. Что для определения  $D$  каждого  $j$  следует найти наименьшее  $D$ , т.е. **самую длинную последовательность символов образа, предшествующих позиции  $j$ , которая полностью совпадает с началом образа.**

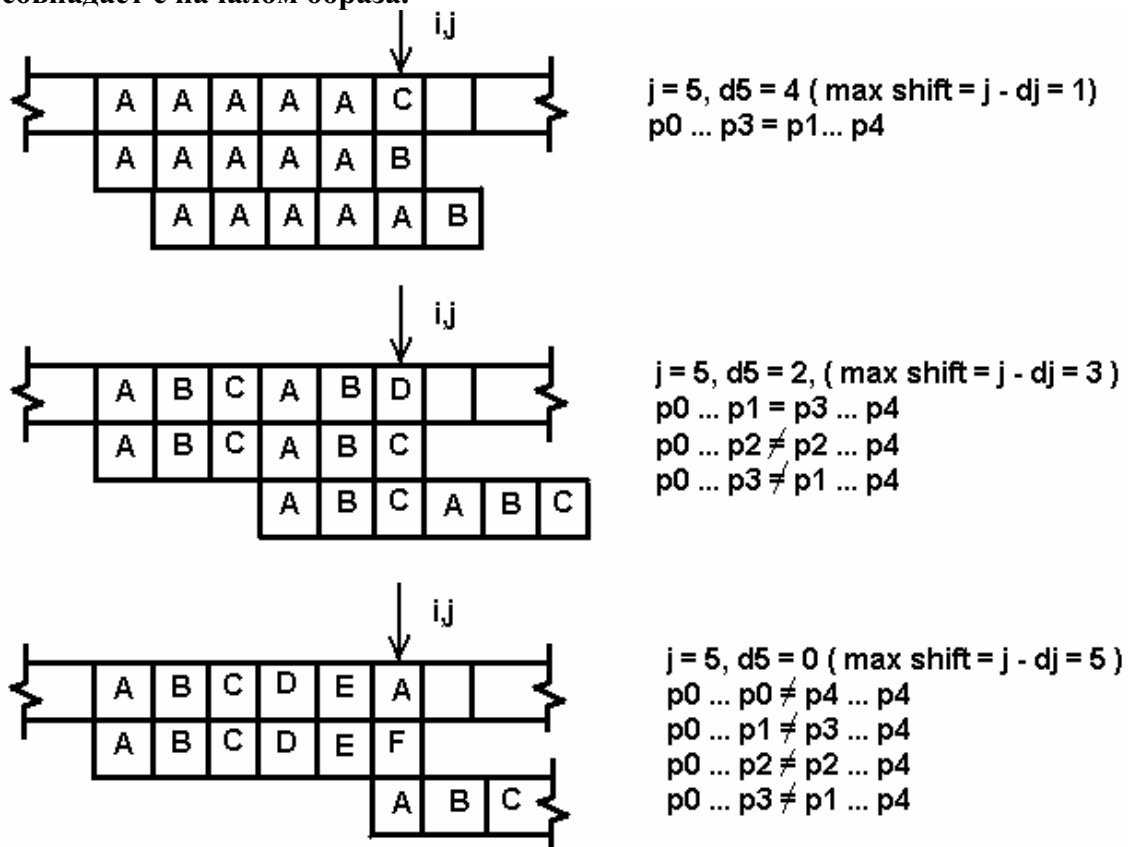
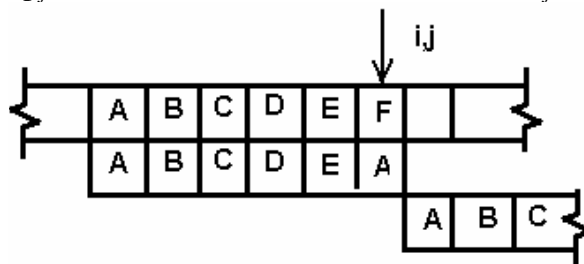


Рис.2 Частичное совпадение с образом и вычисление  $d_j$

Такое  $D$  для каждого  $j$  будем обозначать  $d_j$  (Рис.2). Рассмотрим пример (Рис.3). Т.к.  $p_j$  равно  $A$  вместо  $F$ , то соответствующий символ строки не может быть символом  $A$ , т.к. условие  $s_i \neq p_j$  заканчивает цикл. Следовательно, сдвиг на 5 не приведет к последующему совпадению, и поэтому размер сдвига можно увеличить до 6. Учитывая это, запишем выражения для  $d_j$ :

$p_0 \dots p_{d_j-1} = p_{j-d_j} \dots p_{j-1}$ ,  $p_{d_j} \neq p_j$ . Если никаких совпадений нет, то  $d_j = -1$ .



$j = 5, d_5 = 0$  ( max shift =  $j - d_j = 5$  )

Рис.3 Образ сдвигается за его последний символ

**Const**

$M_{max} = 100$ ; {максимальный размер образа}

$N_{max} = 1000$ ; {максимальный размер строки}

**Var**

$I, j, k$ :integer;

$m, n$ : integer; {фактическое число символов}  
{образа и строки соответственно}

$p$ :array [0..  $M_{max}-1$ ] of char;

$s$ :array [0..  $N_{max}-1$ ] of char;

$d$ :array [0..  $M_{max}-1$ ] of integer;

**begin**

{заполнение массивов  $p$  и  $s$ }

...

{формирование массива  $d$ }

$j := 0$ ;  $k := -1$ ;  $d[0] := -1$ ;

**while**  $j < M-1$  **do**

**begin**

**while**  $(k \geq 0)$  **and**  $(p[j] \neq p[k])$  **do**

$k := d[k]$ ;

$j := j+1$ ;

$k := k+1$ ;

**if**  $p[j] = p[k]$  **then**

$d[j] := d[k]$

**else**

$d[j] := k$ ;

**end**;

{поиск}

$i := 0$ ;  $j := 0$ ;

**while**  $(j < M)$  **and**  $(i < N)$  **do**

**begin**

**while**  $(j \geq 0)$  **and**  $(s[i] \neq p[j])$  **do**  $j := d[j]$ ;

$i := i+1$ ;

$j := j+1$

**end**;

**if**  $j = M$  **then**

$\text{write}(\text{"Найдено!"})$ ;

**end.**



## Анализ КМП-поиска

Точный анализ КМП-поиска, как и сам алгоритм сложен. Его создатели доказывают, что требуется порядка  $M+N$  сравнений, а не  $N \cdot M$  как в прямом поиске.

### Поиск в строке. Алгоритм Боуэра и Мура

Схема КМП-алгоритма дает подлинный выигрыш лишь в случае, если неудаче предшествовало некоторое количество совпадений образа со строкой. В этом случае образ сдвигается более чем на 1. Однако, в большинстве случаев совпадения встречаются значительно реже, чем несовпадения.

Рассмотрим алгоритм поиска подстроки в строке, предложенный Р.Боуэром и Д.Муром. Он основывается на сравнении символов с конца образа, а не с начала. Как и в предыдущем случае перед поиском формируется вспомогательная таблица  $D$ , в которой для каждого символа из алфавита содержится величина  $d_x$  – расстояние от самого правого в образе вхождения символа  $x$  до конца образа.

```
Н о о l а - Н о о l а   g i r l s   l i k e   Н о о l i g a n s
Н о о l i g a n
      Н о о l i g a n
            Н о о l i g a n
                  Н о о l i g a n
                        Н о о l i g a n
                              Н о о l i g a n
```

Предположим, что сразу же обнаружено расхождение образа со строкой. В этом случае образ можно сдвинуть вправо на  $d_{s_{m-1}}$  позиций. Если  $s_{m-1}$  в образе вообще не встречается, то сдвиг происходит на всю длину образа.

БМ-алгоритм использует несколько модифицированные версии предикатов (1-2):

$$P(i,j) = \forall k: j \leq k < M: s_{i+j+k} = p_k \quad (6)$$

$$Q(i) = \forall k: 0 \leq k < i: \sim P(k,0) \quad (7)$$

Запишем алгоритм Боуэра-Мура:

```
I:=M; j:=M;
While (j>0) and (i < N) do
begin
  J:=M; k:=i; { Q(i-M) }
  While (j>0) and (s[k-i] = p[j-1]) do
  Begin
    K:=k-1; j:= j -1; { P(k-j, j) & (k-j = i-M) }
  End;
  I:=i+d[s[i-1]];
End;
```

Индексы удовлетворяют условиям:  $0 \leq j \leq M$  и  $0 \leq i, k \leq M$ . Поэтому из окончания при  $j = 0$  вместе с  $P(k-j,j)$  следует  $P(k,0)$ , т.е. совпадение в позиции  $k$ .

При окончании с  $j > 0$  необходимо, чтобы  $i = N$ , следовательно, из  $Q(i-M)$  следует  $Q(N-M)$ , сигнализирующее о том, что совпадений нет.

$Q(i-M)$  и  $P(k-j,j)$  инварианты двух циклов. В начале они удовлетворяются, т.к.  $Q(0)$  и  $P(k, M)$  всегда истинны.

Выясним эффект от двух операторов, уменьшающих  $i$  и  $j$  на 1. На  $Q(i-M)$  они не действуют, и т.к. было выяснено, что  $s_{k-j} = p_{j-1}$ , то справедливость  $P(k-j, j-1)$  как предусловия гарантирует справедливость  $P(k-j,j)$  в качестве постусловия. Если внутренний цикл заканчивается про  $j>0$ , то из  $s_{k-j} \neq p_{j-1}$  следует  $\sim P(k-j,0)$ , т.к.

$$\sim P(i,0) = \exists k: 0 \leq k < M: s_{i+k} \neq p_k.$$

Более того, из  $k-j = M-i$  следует  $Q(i-M) \& \sim P(k-j,0) = Q(i+1-M)$ , фиксирующее несовпадение в позиции  $i-M+1$ .

Теперь покажем, что оператор  $i:=i+ds_{i-1}$  никогда не делает инвариант ложным. Гарантированно, что перед этим оператором  $Q(i+ds_{i-1}-M)$  истинно. Т.к. мы знаем, что справедливость  $Q(i+1-M)$ , то достаточно установить справедливость  $\sim P(i+h-M)$  для  $h = 2, 3, \dots, ds_{i-1}$ .

Напоминаем, что  $dx$  – это расстояние от самого правого в образе вхождения  $x$  до его конца. Это записывается так:

$\forall k: M - d_x \leq k < M-1: p_k \neq x.$

Подставляем  $s_i$  вместо  $x$ , получаем:

$\forall h: M - ds_{i-1} \leq h < M-1: d_{i-1} \neq p_h.$

$\forall h: 1 < h \leq ds_{i-1}: s_{i-1} \neq p_{h-M}.$

$\forall h: 1 < h \leq ds_{i-1}: \sim P(i+h-M)$

**CONST**

$M_{max} = 100;$

$N_{max} = 10000;$

**VAR**

$i, j, k, M, N: \text{INTEGER};$

$ch: \text{char};$

$p: \text{ARRAY } [0..M_{max}-1] \text{ OF CHAR}; \{ \text{образ} \}$

$s: \text{ARRAY } [0..N_{max}-1] \text{ OF CHAR}; \{ \text{строка} \}$

$d: \text{ARRAY } [0C..177C] \text{ OF INTEGER};$

**BEGIN**

*{формирование образа и строки}*

...

*{формирование массива d}*

**FOR**  $ch := 0C \text{ TO } 177C$  **DO**

$d[ch] := M;$

**FOR**  $j := 0 \text{ TO } M-2$  **DO**

$d[p[j]] := M-j-1;$

*{поиск}*

$i := M;$

**REPEAT**

$j := M; k := i;$

**REPEAT**

$k := k-1;$

$j := j-1;$

**until**  $(j < 0) \text{ or } (p[j] \neq s[i]);$

$i := i + d[s[i-1]]$

**UNTIL**  $(j < 0) \text{ OR } (i \geq N);$

**IF**  $j < 0$  **THEN**  $\text{Writeln}(\text{"Найдено"});$

**END.**

## Анализ БМ-алгоритма

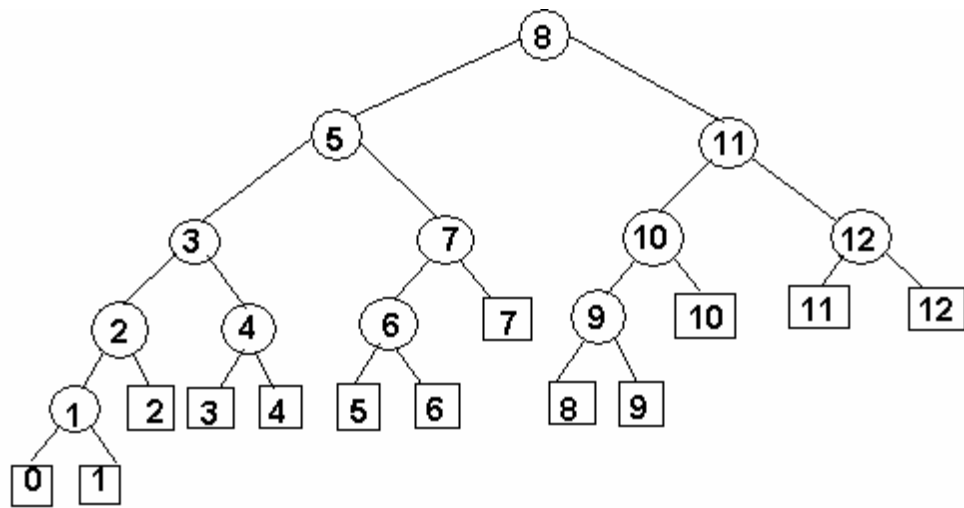
Замечательное свойство в БМ-алгоритма в том, что почти всегда, кроме специально построенных примеров, он требует значительно меньше  $N$  сравнений. В самых же благоприятных обстоятельствах, когда последний символ образа всегда попадает на несовпадающий символ текста, число сравнений равно  $N/M$ .

## Поиск Фибоначчи

Перед описанием алгоритма дадим определение дереву Фибоначчи. В целом, дерево Фибоначчи порядка  $k$  имеет  $F_{k+1} - 1$  внутренних и  $F_{k+1}$  внешних узлов. Строится оно следующим образом:

Если  $k = 0$  или  $k = 1$ , дерево вырождается в единственный узел 0.

Если  $k \geq 2$ , корнем является  $F_k$ ; левое поддерево представляет собой дерево Фибоначчи порядка  $k-1$ ; правое поддерево представляет собой дерево Фибоначчи порядка  $k-2$  с числами, увеличенными на  $F_k$ .



**Рис.1 Дерево Фибоначчи порядка 6**

Для удобства описания алгоритма предположим, что  $N+1$  – число Фибоначчи  $F_{k+1}$ .

#### Алгоритм.

1. Инициализация:  
 $i = F_k$ ,  $p = F_{k-1}$ ,  $q = F_{k-2}$  { $p$  и  $q$  – последовательные числа Фибоначчи}
2. если  $x < a[i]$ , то перейти к шагу 3;  
 если  $x > a[i]$ , то перейти к шагу 4;  
 если  $x = a[i]$ , то алгоритм успешно завершен.
3. Уменьшение  $i$ :  
 если  $q = 0$ , то алгоритм завершен неудачно;  
 в противном случае  $i = i - q$ ,  $p = q$ ,  $q = p - q$ , перейти к шагу 2.
4. Увеличение  $i$ :  
 если  $p = 1$ , то алгоритм завершен неудачно;  
 в противном случае  $i = i + q$ ,  $p = p - q$ ,  $q = q - p$ , перейти к шагу 2.

### **Поиск по бинарному дереву**

Суть метода состоит в том, что каждый элемент исходного массива имеет дополнительное поле – ключ. По набору ключей строится бинарное дерево поиска, каждый элемент которого, представляет собой запись с четырьмя полями: собственно ключом, ссылкой на левое поддерево, ссылкой на правое поддерево и поле индекса, т.е. номер записи, соответствующей данному ключу в исходном массиве. Т.о. неупорядоченный исходный массив преобразуется в упорядоченный.

#### Алгоритм.

1. Инициализация:  
 $p = \text{root}$ ; {указатель на текущий элемент дерева }
2. Сравнение:  
 если  $K < p.^{\text{key}}$ , перейти к шагу 3;  
 если  $K > p.^{\text{key}}$ , перейти к шагу 4;  
 если  $K = p.^{\text{key}}$  поиск успешно завершен, индекс искомой записи в массиве  $p.^{\text{index}}$ ;
3. Перемещение влево:  
 если  $p.^{\text{left}} \neq \text{nil}$ , то  $p = p.^{\text{left}}$ , перейти к шагу 2;  
 в противном случае поиск оказался неудачным.
4. Перемещение вправо:  
 если  $p.^{\text{right}} \neq \text{nil}$ , то  $p = p.^{\text{right}}$ , перейти к шагу 2;  
 в противном случае поиск оказался неудачным.

## Лекция 10

### Сортировка массивов

Сортировку следует понимать как процесс перегруппировки заданного множества объектов в некотором определенном порядке. Цель сортировки – упростить последующий поиск элементов в отсортированном множестве.

Выбор сортировки очень зависит от структуры обрабатываемых данных. В связи с этим методы сортировки разделяют на два класса: сортировка структур с прямым доступом (массивов) и сортировку структур с последовательным доступом (файлов, списков).

Основное условие, которое должно выполняться для всех методов сортировки массивов – экономное использование доступной памяти. Это предполагает, что перестановки, приводящие элементы в порядок должны осуществляться *на том же месте*, т.е. алгоритмы, в которых элементы исходного массива *a* передаются в результирующий массив *b* представляют небольшой интерес. Хорошей мерой эффективности того или иного метода сортировки может быть число необходимых сравнений - *S* и число пересылок (перестановок) элементов - *M*.

Исходный массив определим следующим образом:

```
Type
    Index = integer;
Var
    A: array [1..n] of item;
```

### Сортировка с помощью прямого включения

Алгоритм сортировки таков:

```
For i:=2 to n do
begin
    X:=a[i];
    {включение x на соответствующее место среди a[1]...a[i]}
End;
```

Пример.

Сортировка с помощью прямого включения

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

```
Procedure StrightInsertion;
```

```
    Var
```

```
        I, j:index; x:item;
```

```
Begin
```

```
    For i :=2 to n do
```

```
    begin
```

```
        X:=a[i];
```

```

        A[0] := x; {барьер}
        J:=i;
        While x<a[j-1] do
        Begin
            A[j] := a[j-1];
            J:=j-1;
        End;
        A[j]:=x;
    End;
End;

```

### Анализ метода прямого включения

```

Cmin = n-1;
Cave = (n2+n-2)/4;
Cmax = (n2+n-4)/4;
Mmin = 3·(n-1);
Mave = (n2+9n-10)/4;
Mmax = (n2+3n-4)/2;

```

Минимальные оценки встречаются в случае уже упорядоченной исходной последовательности элементов, наихудшие – когда исходная последовательность упорядочена в обратном порядке.

### Сортировка с помощью двоичного включения

Предыдущий алгоритм можно улучшить, если обратить внимание, что последовательность  $a_1...a_{i-1}$ , в которую надо вставить новый элемент, уже упорядочена. Поэтому целесообразно включать новый элемент в данную последовательность с помощью двоичного поиска.

```

Procedure BinaryInsertion;
    Var
        I, j, m, R, L:index; x:item;
    Begin
        For i :=2 to n do
        begin
            X:=a[i];
            L := 1;
            R:=i;
            While L<R do
            Begin
                M := (L+R)div 2;
                If a[m] <= x then L:=m+1 else R := m;
            End;
            For j:=i downto R+1 do
                A[j]:=a[j-1];
            A[R]:=x;
        End;
    End;

```

### Анализ метода прямого включения

Число сравнений при данном методе практически не зависит от начального порядка элементов и равно  $C = n(\log_2 n - c) + c$ , где  $c = 1/\ln 2$ .

## Сортировка с помощью прямого выбора

Этот прием основан на следующих принципах:

1. Выбирается элемент с наименьшим ключом
2. Он меняется местами с первым элементом  $a_1$
3. Затем этот процесс повторяется с оставшимися  $n-1$  элементами,  $n-2$  элементами и т.д. до тех пор, пока не останется один самый большой элемент

Алгоритм формулируется следующим образом:

**For**  $i:=1$  **to**  $n-1$  **do**

**Begin**

    Присвоить  $k$  индекс наименьшего элемента из  $a[i] \dots a[n]$ ;

    Поменять местами  $a[i]$  и  $a[k]$ ;

**end;**

Пример.

Сортировка прямым выбором

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

**procedure** StraightSelection;

**var**

$i, j, k$ :index;  $x$ :item;

**begin**

**for**  $i:=1$  **to**  $n-1$  **do**

**begin**

$k:=i$ ;

$x:=a[i]$ ;

**for**  $j:=i+1$  **to**  $n$  **do**

**if**  $a[j] < x$  **then**

**begin**

$k:=j$ ;

$x:=a[k]$ ;

**end;**

$a[k]:=a[i]$ ;

$a[i]:=x$ ;

**end;**

**end;**

## Анализ прямого выбора

Число сравнений ключей ( $C$ ) очевидно, не зависит от начального порядка ключей.

$$C = (n^2 - n)/2.$$

Минимальное число перестановок в случае изначальной упорядоченности ключей:

$$M_{\min} = 3 \cdot (n-1)$$

Максимальное число перестановок в случае упорядоченности ключей в обратном порядке:

$$M_{\max} = n^2/4 + 3 \cdot (n-1).$$

## Сортировка с помощью прямого обмена

Изложенный ниже алгоритм прямого обмена основывается на сравнении и смене мест для пары соседних элементов и продолжении этого процесса тех пор, пока не будут упорядочены все элементы.

Как и в упоминавшемся методе прямого выбора, мы повторяем проходы по массиву, сдвигая каждый раз наименьший элемент оставшейся последовательности к левому концу массива. Если будем рассматривать массивы как вертикальные, а не горизонтальные построения, то элементы можно интерпретировать как пузырьки в чане с водой, причем вес каждого соответствует его ключу. В этом случае при каждом проходе один пузырек как бы поднимается до уровня, соответствующего его весу. Такой метод широко известен под именем «пузырьковая сортировка».

Пример.

Метод «пузырька»

i =	1	2	3	4	5	6	7	8
	44	06	06	06	06	06	06	06
	55	44	12	12	12	12	12	12
	12	55	44	18	18	18	18	18
	42	12	55	44	42	42	42	42
	94	42	18	55	44	44	44	44
	18	94	42	42	55	55	55	55
	06	18	94	67	67	67	67	67
	67	67	67	94	94	94	94	94

```

PROCEDURE BubbleSort;
var
    i, j, index;
    x: item;
BEGIN
    FOR i = 2 TO n DO
        FOR j := n downto i DO
            IF a[j-1] > a[j] THEN
                begin
                    x := a[j-1];
                    a[j-1] := a[j];
                    a[j] := x ;
                end
            END
        END
    END;

```

## Шейкерная сортировка

Очевидный прием улучшения предыдущего алгоритма – запоминать, были или не были перестановки в процессе некоторого прохода. Если перестановок не было, то алгоритм можно заканчивать. Другой способ улучшить сортировку – запоминать индекс последнего обмена. Ясно, что все пары соседних элементов, выше данного индекса  $k$  уже находятся в желаемом порядке. Поэтому просмотры можно заканчивать на этом индексе, а не идти до заранее определенного нижнего предела  $i$ .

Шейкерная сортировка с успехом используется в случаях, когда известно, что элементы почти упорядочены.

Пример.

Шейкерная сортировка.

L =	2	3	3	4	4
R =	8	8	7	7	4
dir =	↑	↓	↑	↓	↑
	44	06	06	06	06
	55	44	44	12	12
	12	55	12	44	18
	42	12	42	18	42
	94	42	55	42	44
	18	94	18	55	55
	06	18	67	67	67
	67	67	94	94	94

```

PROCEDURE ShakerSort;
VAR
    j, k, L, R: index;
    x : item;
BEGIN
    L := 2,
    R := n;
    K := n;
    REPEAT
        FOR j := R DOWNTO L DO
            IF a[j-1] > a[j] THEN
                begin
                    x := a[j-1];
                    a[j-1] := a[j];
                    a[j] := x;
                    k := j;
                END
            L := k+1;
            FOR j:= L TO R DO
                IF a[j-1] > a[j] THEN
                    begin
                        x := a[j-1];
                        a[j-1] := a[j];
                        a[j] := x;
                        k:=j;
                    end;
                R := k-1;
            UNTIL L > R
    END;

```

## Анализ пузырьковой и шейкерной сортировок

Число сравнений в строго обменном алгоритме

$$C = (n^2 - n)/2,$$

а минимальное, среднее и максимальное число перемещений (присваиваний) элементов будет соответственно

$$M_{\min} = 0;$$

$$M_{\text{avg}} = 3 \cdot (n^2 - n)/2.$$

$$M_{\max} = 3 \cdot (n^2 - n)/4.$$

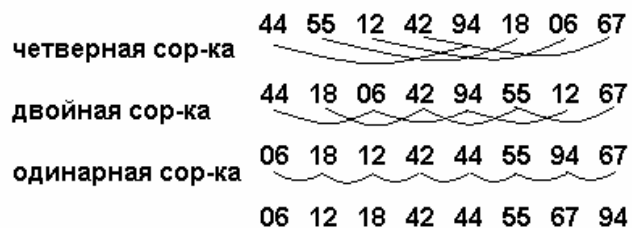
Минимальное число сравнений для шейкерной сортировки  $C = n-1$ .



## Улучшенные методы сортировки

### Сортировка с помощью включений с уменьшающимися расстояниями (сортировка Шелла)

Метод прямого включения можно улучшить, используя следующий алгоритм. Сначала отдельно группируются и сортируются элементы, отстоящие друг от друга на расстоянии 4. Такой процесс называется четверной сортировкой. В нашем примере восемь элементов, и каждая группа состоит точно из двух элементов. После первого прохода элементы перегруппировываются - теперь каждый элемент группы отстоит от другого на две позиции - и вновь сортируются. Это называется двойной сортировкой. И, наконец, на третьем проходе идет обычная, или одинарная, сортировка.



Такой метод дает упорядоченный массив. Каждая  $i$ -сортировка объединяет две группы, уже отсортированные  $2i$  сортировкой, что дает преимущество для каждого нового прохода. Расстояния в группах можно уменьшать по-разному, лишь бы последнее было единичным, ведь в самом плохом случае последний проход и сделает всю работу. Однако совсем не очевидно, что такой прием "уменьшающихся расстояний" может дать лучшие результаты, если расстояния не будут степенями двойки. Поэтому приводимая ниже программа не ориентирована на такую определенную последовательность расстояний. Все  $t$  расстояний обозначаются соответственно  $h_1, h_2, \dots, h_t$ , и для них выполняются условия:

$$h_t = 1, h_{i+1} < h_i$$

Каждая  $h$ -сортировка программируется как сортировка с помощью прямого включения. Причем простота условия окончания поиска места для включения обеспечивается методом барьеров. Это подразумевает, что каждая из сортировок нуждается в постановке своего собственного барьера, поэтому приходится расширять массив не только на одну единственную компоненту  $a_0$ , а на  $H_1$  компонент. Его описание выглядит так:

```
A: ARRAY [-h1..n] OF item;
```

```
PROCEDURE ShellSort;
CONST
    t = 4;
VAR
    I, j, k, s: index;
    X: item;
    M: 1..t;
    H: ARRAY [1..t] OF INTEGER;
BEGIN
    h[1] := 9;
    h[2] := 5;
    h[3] := 3;
    h[4] := 1;
    FOR m := 1 TO t do
        begin
            k := h[m];
            s := -k; { место барьера }
            FOR i := k+1 TO n do
```

```

Begin
  x := a[i];
  j := i-k;
  IF s = 0 THEN
    S := -K;
  s := s+1;
  a[s] := x; {установка барьера}
  WHILE x < a[j] DO
  begin
    a[j+k] := a[j];
    j := j-k
  end;
  a[j+k] := x;
END;
END;
End;

```

## Анализ сортировки Шелла

Анализ этого алгоритма поставил несколько весьма трудных математических проблем, многие из которых так еще и не решены. В частности, неизвестно, какие расстояния дают наилучшие результаты. Но они не должны быть множителями один другого. Это позволяет избежать явления, когда при каждом проходе взаимодействуют две цепочки, которые до этого нигде еще не пересекались.

Д.Кнут считает, что имеет смысл использовать такую последовательность: 1, 4, 13, 40, 121, ..., где  $h_{k-1} = 3h_k + 1$ ,  $h_t = 1$  и  $t = \lceil \log_3 n \rceil - 1$ . Есть также другая последовательность: 1, 3, 7, 15, 31, ..., где  $h_{k-1} = 2h_k + 1$ ,  $h_t = 1$  и  $t = \lceil \log_2 n \rceil - 1$ . В последнем случае для сортировки  $n$  элементов затраты пропорциональны  $n^{1.2}$ .

## Сортировка с помощью дерева (HeapSort)

Метод сортировки с помощью прямого выбора основан на повторяющихся поисках наименьшего ключа среди  $n$  элементов, среди оставшихся  $n-1$  элементов и т.д. Обнаружение наименьшего среди  $n$  элементов требует  $n-1$  сравнений, среди  $n-1$  уже нужно  $n-2$  сравнений и т.д. Сумма первых  $n-1$  целых равна  $1/2(n^2 - n)$ . Усовершенствовать упомянутый метод сортировки можно, оставляя после каждого прохода больше информации, чем просто идентификация единственного минимального элемента. Например, сделав  $n/2$  сравнений, можно определить в каждой паре ключей меньший. С помощью  $n/4$  сравнений - меньший из пары уже выбранных меньших и т.д. Прделав  $n-1$  сравнений, мы можем построить дерево выбора, вроде представленного на Рис.1, и идентифицировать его корень как нужный нам наименьший ключ.

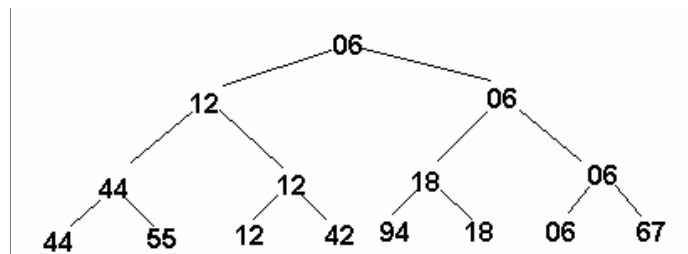


Рис.1 Повторяющиеся выборы среди двух ключей

Второй этап сортировки - спуск вдоль пути, отмеченного наименьшим элементом, и исключение его из дерева путем замены либо на пустой элемент (дырку) в самом низу, либо на элемент соседней ветви в промежуточных вершинах (Рис.2). Элемент, передвинувшийся в корень дерева, вновь будет наименьшим (теперь уже вторым) ключом и его можно исключить (Рис.3). После таких шагов дерево станет пустым (т.е. в нем останутся только дырки), и

процесс сортировки заканчивается. На каждом из  $n$  шагов выбора требуется только  $\log_2 n$  сравнений. Поэтому на весь процесс понадобится  $n \cdot \log_2 n$  сравнений плюс  $n$  шагов на построение дерева.

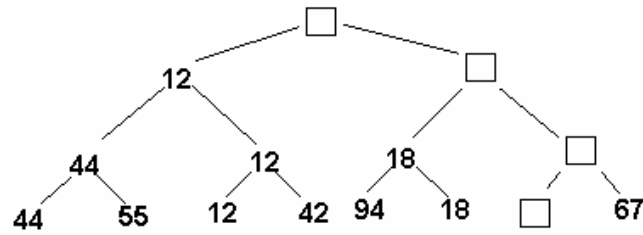


Рис.2 Выбор наименьшего ключа

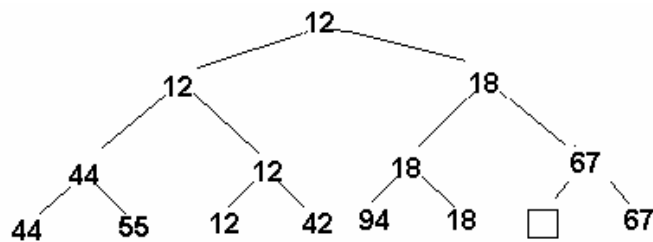


Рис.3 Заполнение «дырок»

Алгоритм, реализующий данную сортировку называется HeapSort.

Дадим определение: *пирамида* понимается как последовательность ключей  $h_L, h_{L+1}, \dots, h_R$ , такая, что

$$h_i \leq h_{2i} \text{ и } h_i \leq h_{2i+1} \text{ для } i = L, \dots, R/2.$$

Любое двоичное дерево можно рассматривать как массив по схеме (Рис.4), где  $h_1$  – наименьший элемент массива:  $h_1 = \min(h_1, h_2, \dots, h_n)$ . Предположим, что есть некоторая пирамида с элементами  $h_{L+1}, \dots, h_R$  и для некоторых значений  $L$  и  $R$  нужно добавить новый элемент  $x$ . Возьмем в качестве исходной пирамиду на Рис.4. и добавим к ней слева элемент  $h_1=44$ . Новая пирамида получается так: сначала новый элемент ставится в вершину пирамиды, а затем постепенно опускается вниз по направлению к наименьшему из двух примыкающих к нему элементов. Причем сам этот элемент перемещается вверх (Рис.5).

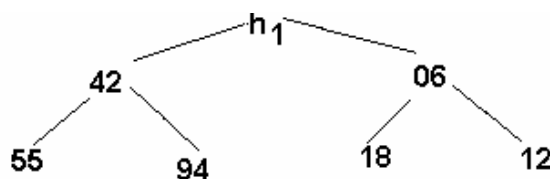


Рис.4 Пирамида из семи элементов

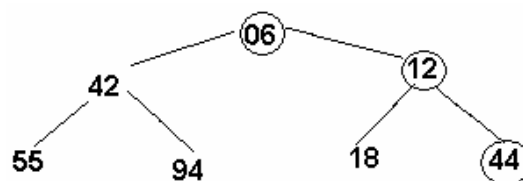


Рис.5 Просеивание ключа 44 через пирамиду

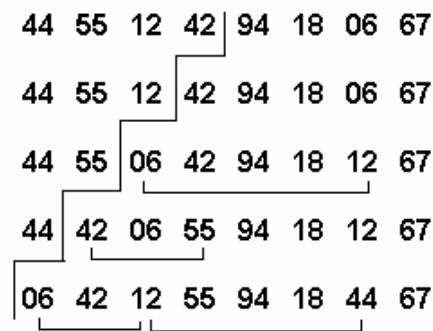
Р.Флойд предложил способ формирования пирамиды *на том же месте*. Для некоторого массива  $h_1 \dots h_n$  элементы  $h_m \dots h_n$  ( $m = (n \div 2) + 1$ ) уже образуют пирамиду. Далее

пирамида расширяется влево, и каждый раз добавляется и сдвигается в нужную позицию новый элемент.

Для получения не только частичной, но и полной упорядоченности среди элементов, нужно проделать  $n$  сдвигающих шагов, причем после каждого шага на вершину пирамиды выталкивается очередной наименьший элемент. Всплывающие верхние элементы хранятся так: каждый раз берется последняя компонента пирамиды (например,  $x$ ), прячется верхний элемент пирамиды в освободившемся теперь месте, а  $x$  сдвигается в нужное место.

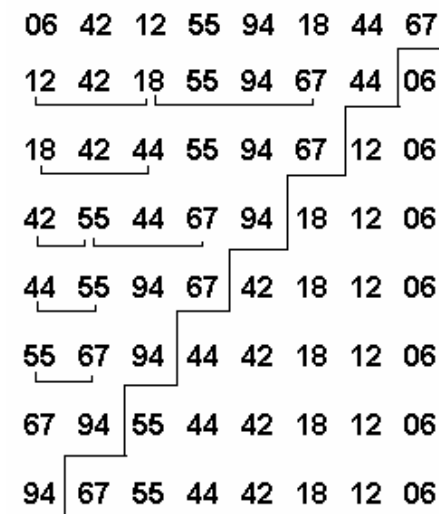
Пример.

Построение пирамиды.



Пример.

Сортировка с помощью HeapSort



**PROCEDURE** HeapSort;

**var**

L,R:index;

X:item;

**PROCEDURE** sift(L,R:index); {процедура сдвига}

**VAR**

i,j:index;

X:item;

**BEGIN**

i := L; j:= 2L;

x := a[L];

**IF** (j < R) **and** (a[j] < a[j+1]) **THEN** j:= j+1;

**WHILE** (j<= R) **and** (x < a[j]) **DO**

**Begin**

a[i] := a[j];

```

        i := j;
        j := 2j;
        IF (j < R) and (a[j] < a[j+1]) THEN j:=j+1;
    END;
    a[i] := x;
END; {end sift}
BEGIN
    L := (n DIV 2) + 1;
    R := n;
    WHILE L > 1 DO { формирование пирамиды на том же месте }
    Begin
        L := L-1;
        sift(L, R)
    END;
    While R>1 do { сортировка }
    begin
        x := a[1];
        a[1] := a[R];
        a[R] := x;
        R := R-1;
        Sift(L, R);
    END;
END;

```

## Анализ HeapSort

В самом плохом из возможных случаев для сортировки потребуется  $n \cdot \log_2 n$  шагов. Среднее число перемещений приблизительно равно  $\log_2 n \cdot n/2$ .

## Сортировка с помощью разделения (Quicksort)

Третий улучшенный метод сортировки основан на обмене. Этот метод приводит к самому лучшему из известных методу сортировки. Метод быстрой сортировки (Quicksort) основан на том соображении, что для достижения лучшего эффекта сначала лучше производить перестановки элементов на большие расстояния.

Выберем наугад какой либо элемент (назовем его  $x$ ) и будем просматривать слева наш массив до тех пор пока не обнаружим элемент  $a_i > x$ . Затем будем просматривать массив справа, пока не встретим  $a_j < x$ . Теперь поменяем местами эти два элемента и продолжим наш процесс просмотра и обмена, пока оба просмотра не встретятся где-то в середине массива. В результате массив окажется разбитым на левую часть с ключами меньше (или равными)  $x$  и правую - с ключами больше (или равными)  $x$ .

Теперь нужно применить этот процесс к получившимся двум частям, затем - к частям частей, и так до тех пор, пока каждая из частей не будет состоять из одного единственного элемента.

```

Procedure QuickSort;
  Procedure sort(L,R:index)
  Var
    I,j:index; w,x:item;
  Begin
    I:=L;
    J:=R;
    X:=a[(L+R) div 2];
    Repeat
      While a[i]<x do i:=i+1;
      While x<a[j] do j:=j-1;

```

```

        If i<=j then
        Begin
            W:=a[i];
            A[i]:=a[j];
            A[j]:=w;
            I:=i+1;
            J:=j-1;

        End;
    Until i>j;
    If L<j then sort(L,j)
    If i<R then sort(i,R)
End;
BEGIN
    Sort(1,n);
END;

```

## Анализ QuickSort

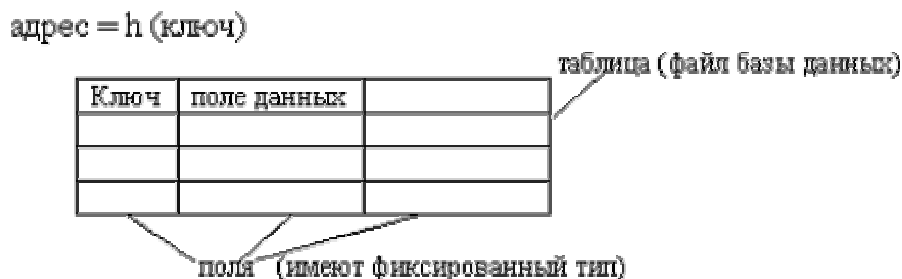
В самом хорошем случае общее число сравнений будет равно  $n \cdot \log_2 n$ , а общее число обменов -  $n \cdot \log_2(n)/6$ .

# Методы ускорения доступа к данным

## Хеширование данных

Для ускорения доступа к данным в таблицах можно использовать предварительное упорядочивание таблицы в соответствии со значениями ключей.

При этом могут быть использованы методы поиска в упорядоченных структурах данных, например, метод половинного деления, что существенно сокращает время поиска данных по значению ключа. Однако при добавлении новой записи требуется переупорядочить таблицу. Потери времени на повторное упорядочивание таблицы могут значительно превышать выигрыш от сокращения времени поиска. Поэтому для сокращения времени доступа к данным в таблицах используется так называемое случайное упорядочивание или хеширование. При этом данные организуются в виде таблицы при помощи хеш-функции  $h$ , используемой для “вычисления” адреса по значению ключа.



**Рис.1 Хеш-таблица**

Идеальной хеш-функцией является такая hash-функция, которая для любых двух неодинаковых ключей дает неодинаковые адреса.

$$k1 \neq k2 \Rightarrow h(k1) \neq h(k2)$$

Подобрать такую функцию можно в случае, если все возможные значения ключей заранее известны. Такая организация данных носит название “совершенное хеширование”. В случае заранее неопределенного множества значений ключей и ограниченной длины таблицы подбор совершенной функции затруднителен. Поэтому часто используют хеш-функции, которые не гарантируют выполнение условия.

Рассмотрим пример реализации несовершенной хеш-функции на языке TurboPascal. Предположим, что ключ состоит из четырех символов. При этом таблица имеет диапазон адресов от 0 до 10000.

```
function hash (key : string[4]): integer;  
var  
  f: longint;  
begin  
  f:=ord (key[1]) - ord (key[2]) + ord (key[3]) -ord (key[4]);  
  {вычисление функции по значению ключа}  
  f:=f+255*2;  
  {совмещение начала области значений функции с начальным  
адресом хеш-таблицы (a=1)}  
  f:=(f*10000) div (255*4);  
  {совмещение конца области значений функции с конечным адресом  
хеш-таблицы (a=10 000)}
```

```

hash:=f
end;

```

При заполнении таблицы возникают ситуации, когда для двух неодинаковых ключей функция вычисляет один и тот же адрес. Данный случай носит название “коллизия”, а такие ключи называются “ключи-синонимы”.

## Методы разрешения коллизий

Для разрешения коллизий используются различные методы, которые в основном сводятся к методам “цепочек” и “открытой адресации”.

Методом цепочек называется метод, в котором для разрешения коллизий во все записи вводятся указатели, используемые для организации списков – “цепочек переполнения”. В случае возникновения коллизии при заполнении таблицы в список для требуемого адреса хеш-таблицы добавляется еще один элемент.

Поиск в хеш-таблице с цепочками переполнения осуществляется следующим образом. Сначала вычисляется адрес по значению ключа. Затем осуществляется последовательный поиск в списке, связанном с вычисленным адресом.

Процедура удаления из таблицы сводится к поиску элемента и его удалению из цепочки переполнения.



Рис.2. Разновидности методов разрешения коллизий

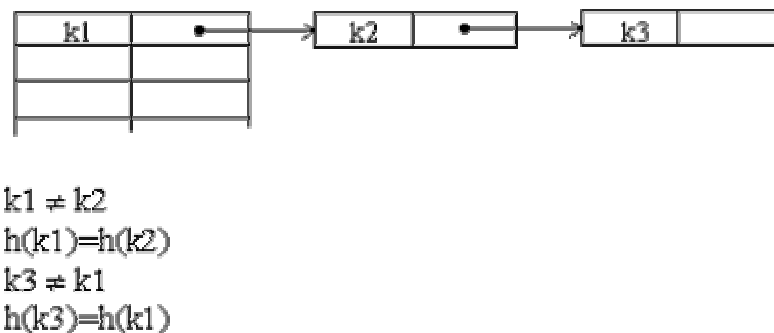
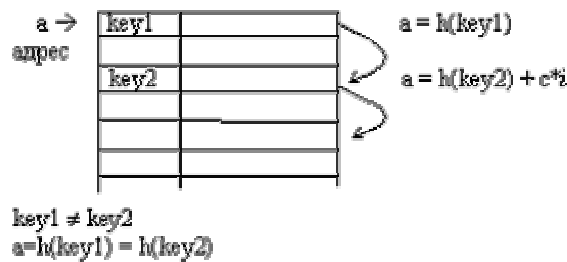


Рис.3. Разрешение коллизий при добавлении элементов методом цепочек.

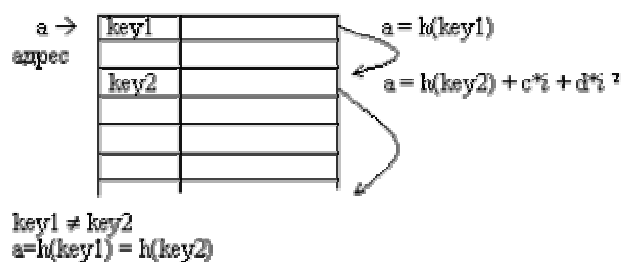


Метод открытой адресации состоит в том, чтобы, пользуясь каким-либо алгоритмом, обеспечивающим перебор элементов таблицы, просматривать их в поисках свободного места для новой записи.

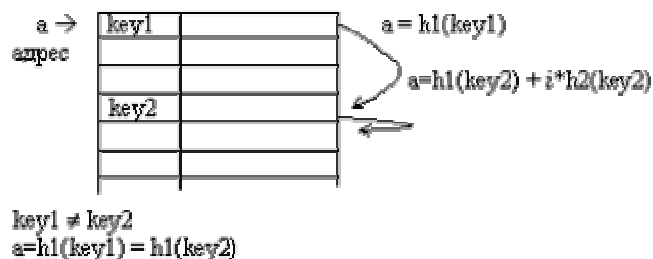
а) Линейное опробование



б) Квадратичное опробование



в) Двойное хеширование



**Рис.4. Разрешение коллизий при добавлении элементов методами открытой адресации.**

Линейное опробование сводится к последовательному перебору элементов таблицы с некоторым фиксированным шагом

$$a = h(key) + c*i,$$

где  $i$  – номер попытки разрешить коллизию. При шаге равном единице происходит последовательный перебор всех элементов после текущего.

Квадратичное опробование отличается от линейного тем, что шаг перебора элементов не линейно зависит от номера попытки найти свободный элемент

$$a = h(key) + c*i + d*i^2$$

Благодаря нелинейности такой адресации уменьшается число проб при большом числе ключей-синонимов.

Однако даже относительно небольшое число проб может быстро привести к выходу за адресное пространство небольшой таблицы вследствие квадратичной зависимости адреса от номера попытки.

Еще одна разновидность метода открытой адресации, которая называется двойным хешированием, основана на нелинейной адресации, достигаемой за счет суммирования значений основной и дополнительной хеш-функций

$$a = h1(key) + i*h2(key).$$

Опишем алгоритмы вставки и поиска для метода линейное опробование.

Вставка

$i = 0$

$a = h(\text{key}) + i * c$

Если  $t(a) = \text{свободно}$ , то  $t(a) = \text{key}$ , записать элемент, стоп элемент добавлен

$i = i + 1$ , перейти к шагу 2

Поиск

$i = 0$

$a = h(\text{key}) + i * c$

Если  $t(a) = \text{key}$ , то стоп элемент найден

Если  $t(a) = \text{свободно}$ , то стоп элемент не найден

$i = i + 1$ , перейти к шагу 2

Аналогичным образом можно было бы сформулировать алгоритмы добавления и поиска элементов для любой схемы открытой адресации. Отличия будут только в выражении, используемом для вычисления адреса

(шаг 2). С процедурой удаления дело обстоит не так просто, так как она в данном случае не будет являться обратной процедуре вставки.

Дело в том, что элементы таблицы находятся в двух состояниях: свободно и занято. Если удалить элемент, переведя его в состояние свободно, то после такого удаления алгоритм поиска будет работать некорректно. Предположим, что ключ удаляемого элемента имеет в таблице ключи синонимы. В том случае, если за удаляемым элементом в результате разрешения коллизий были размещены элементы с другими ключами, то поиск этих элементов после удаления всегда будет давать отрицательный результат, так как алгоритм поиска останавливается на первом элементе, находящемся в состоянии свободно.

Скорректировать эту ситуацию можно различными способами. Самый простой из них заключается в том, чтобы производить поиск элемента не до первого свободного места, а до конца таблицы. Однако такая модификация алгоритма сведет на нет весь выигрыш в ускорении доступа к данным, который достигается в результате хеширования.

Другой способ сводится к тому, чтобы проследить адреса всех ключей-синонимов для ключа удаляемого элемента и при необходимости пере разместить соответствующие записи в таблице. Скорость поиска после такой операции не уменьшится, но затраты времени на само пере размещение элементов могут оказаться очень значительными.

Существует подход, который свободен от перечисленных недостатков. Его суть состоит в том, что для элементов хеш-таблицы добавляется состояние “удалено”. Данное состояние в процессе поиска интерпретируется, как занято, а в процессе записи как свободно.

Сформулируем алгоритмы вставки поиска и удаления для хеш-таблицы, имеющей три состояния элементов.

Вставка

$i = 0$

$a = h(\text{key}) + i * c$

Если  $t(a) = \text{свободно}$  или  $t(a) = \text{удалено}$ , то  $t(a) = \text{key}$ , записать элемент, стоп элемент добавлен

$i = i + 1$ , перейти к шагу 2

Удаление

$i = 0$

$a = h(\text{key}) + i * c$

Если  $t(a) = \text{key}$ , то  $t(a) = \text{удалено}$ , стоп элемент удален

Если  $t(a) = \text{свободно}$ , то стоп элемент не найден

$i = i + 1$ , перейти к шагу 2

Поиск

```

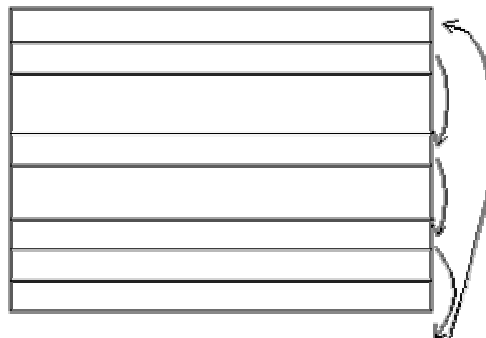
i = 0
a = h(key) + i*c
Если t(a) = key, то стоп элемент найден
Если t(a) = свободно, то стоп элемент не найден
i = i + 1, перейти к шагу 2

```

Алгоритм поиска для хеш-таблицы, имеющей три состояния, практически не отличается от алгоритма поиска без учета удалений. Разница заключается в том, что при организации самой таблицы необходимо отмечать свободные и удаленные элементы. Это можно сделать, зарезервировав два значения ключевого поля. Другой вариант реализации может предусматривать введение дополнительного поля, в котором фиксируется состояние элемента. Длина такого поля может составлять всего два бита, что вполне достаточно для фиксации одного из трех состояний. На языке TurboPascal данное поле удобно описать типом Byte или Char.

### Переполнение таблицы и рехеширование

Очевидно, что по мере заполнения хеш-таблицы будут происходить коллизии и в результате их разрешения методами открытой адресации очередной адрес может выйти за пределы адресного пространства таблицы. Что бы это явление происходило реже, можно пойти на увеличение длины таблицы по сравнению с диапазоном адресов, выдаваемым хеш-функцией.



**Рис.5. Циклический переход к началу таблицы.**

С одной стороны это приведет к сокращению числа коллизий и ускорению работы с хеш-таблицей, а с другой – к нерациональному расходованию адресного пространства. Даже при увеличении длины таблицы в два раза по сравнению с областью значений хеш-функции нет гарантии того, что в результате коллизий адрес не превысит длину таблицы. При этом в начальной части таблицы может оставаться достаточно свободных элементов. Поэтому на практике используют циклический переход к началу таблицы.

Рассмотрим данный способ на примере метода линейного опробования. При вычислении адреса очередного элемента можно ограничить адрес, взяв в качестве такового остаток от целочисленного деления адреса на длину таблицы  $n$ .

```

Вставка
i = 0
a = (h(key) + c*i) mod n
Если t(a) = свободно или t(a) = удалено, то t(a) = key, записать элемент, стоп
элемент добавлен
i = i + 1, перейти к шагу 2

```

В данном алгоритме мы не учитываем возможность многократного превышения адресного пространства. Более корректным будет алгоритм, использующий сдвиг адреса на 1 элемент в случае каждого повторного превышения адресного пространства. Это

повышает вероятность найти свободные элементы в случае повторных циклических переходов к началу таблицы.

Вставка

$i = 0$

$a = ((h(\text{key}) + c*i) \text{ div } n + (h(\text{key}) + c*i) \text{ mod } n) \text{ mod } n$

Если  $t(a) = \text{свободно}$  или  $t(a) = \text{удалено}$ , то  $t(a) = \text{key}$ , записать элемент, стоп элемент добавлен

$i = i + 1$ , перейти к шагу 2

$a = ((h(\text{key}) + c*i) \text{ div } n + (h(\text{key}) + c*i) \text{ mod } n) \text{ mod } n$

Рассматривая возможность выхода за пределы адресного пространства таблицы, мы не учитывали факторы заполненности таблицы и удачного выбора хеш-функции. При большой заполненности таблицы возникают частые коллизии и циклические переходы в начало таблицы. При неудачном выборе хеш-функции происходят аналогичные явления. В наихудшем варианте при полном заполнении таблицы алгоритмы циклического поиска свободного места приведут к заикливлению. Поэтому при использовании хеш-таблиц необходимо стараться избегать очень плотного заполнения таблиц. Обычно длину таблицы выбирают из расчета двукратного превышения предполагаемого максимального числа записей. Не всегда при организации хеширования можно правильно оценить требуемую длину таблицы, поэтому в случае большой заполненности таблицы может потребоваться рехеширование. В этом случае увеличивают длину таблицы, изменяют хеш-функцию и переупорядочивают данные.

Производить отдельную оценку плотности заполнения таблицы после каждой операции вставки нецелесообразно, поэтому можно производить такую оценку косвенным образом – по числу коллизий во время одной вставки. Достаточно определить некоторый порог числа коллизий, при превышении которого следует произвести рехеширование. Кроме того, такая проверка гарантирует невозможность заикливления алгоритма в случае повторного просмотра элементов таблицы.

Рассмотрим алгоритм вставки, реализующий предлагаемый подход.

Вставка

$i = 0$

$a = ((h(\text{key}) + c*i) \text{ div } n + (h(\text{key}) + c*i) \text{ mod } n) \text{ mod } n$

Если  $t(a) = \text{свободно}$  или  $t(a) = \text{удалено}$ , то  $t(a) = \text{key}$ , записать элемент, стоп элемент добавлен

Если  $i > m$ , то стоп требуется рехеширование

$i = i + 1$ , перейти к шагу 2

В данном алгоритме номер итерации сравнивается с пороговым числом  $m$ . Следует заметить, что алгоритмы вставки, поиска и удаления должны использовать идентичное образование адреса очередной записи.

Удаление

$i = 0$

$a = ((h(\text{key}) + c*i) \text{ div } n + (h(\text{key}) + c*i) \text{ mod } n) \text{ mod } n$

Если  $t(a) = \text{key}$ , то  $t(a) = \text{удалено}$ , стоп элемент удален

Если  $t(a) = \text{свободно}$  или  $i > m$ , то стоп элемент не найден

$i = i + 1$ , перейти к шагу 2

Поиск

$i = 0$

$a = ((h(\text{key}) + c*i) \text{ div } n + (h(\text{key}) + c*i) \text{ mod } n) \text{ mod } n$

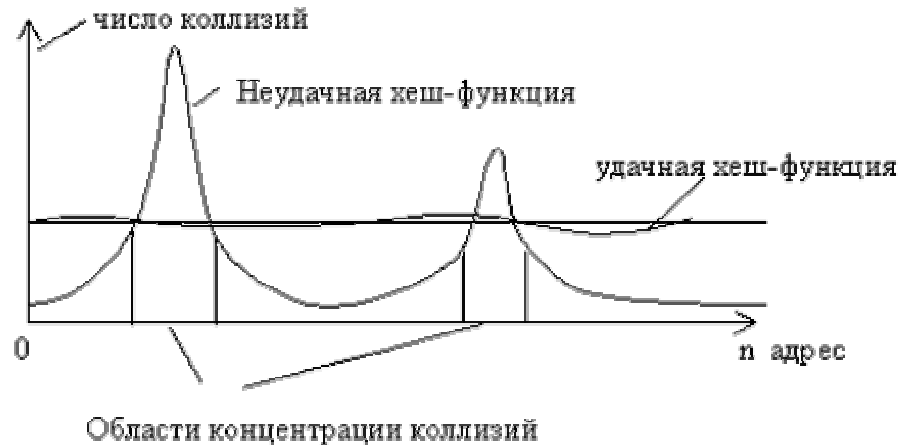
Если  $t(a) = \text{key}$ , то стоп элемент найден

Если  $t(a) = \text{свободно}$  или  $i > m$ , то стоп элемент не найден

$i = i + 1$ , перейти к шагу 2

## Оценка качества хеш-функции

Как уже было отмечено, очень важен правильный выбор хеш-функции. При удачном построении хеш-функции таблица заполняется более равномерно, уменьшается число коллизий и уменьшается время выполнения операций поиска, вставки и удаления. Для того чтобы предварительно оценить качество хеш-функции можно провести имитационное моделирование. Моделирование проводится следующим образом. Формируется целочисленный массив, длина которого совпадает с длиной хеш-таблицы. Случайно генерируется достаточно большое число ключей, для каждого ключа вычисляется хеш-функция. В элементах массива просчитывается число генераций данного адреса. По результатам такого моделирования можно построить график распределения значений хеш-функции. Для получения корректных оценок число генерируемых ключей должно в несколько раз превышать длину таблицы.



**Рис. 6. Распределение коллизий в адресном пространстве таблицы**

Если число элементов таблицы достаточно велико, то график строится не для отдельных адресов, а для групп адресов. Например, все адресное пространство разбивается на 100 фрагментов и подсчитывается число попаданий адреса для каждого фрагмента. Большие неравномерности свидетельствуют о высокой вероятности коллизий в отдельных местах таблицы. Разумеется, такая оценка является приближенной, но она позволяет предварительно оценить качество хеш-функции и избежать грубых ошибок при ее построении.

Оценка будет более точной, если генерируемые ключи будут более близки к реальным ключам, используемым при заполнении хеш-таблицы. Для символьных ключей очень важно добиться соответствия генерируемых кодов символов тем кодам символов, которые имеются в реальном ключе. Для этого стоит проанализировать, какие символы могут быть использованы в ключе.

Например, если ключ представляет собой фамилию на русском языке, то будут использованы русские буквы. Причем первый символ может быть большой буквой, а остальные — малыми. Если ключ представляет собой номерной знак автомобиля, то также несложно определить допустимые коды символов в определенных позициях ключа.

Рассмотрим пример генерации ключа из десяти латинских букв, первая из которых является большой, а остальные — малыми.

Пример

: ключ — 10 символов, 1-й большая латинская буква

2-10 малые латинские буквы

```
var i:integer; s:string[10];
```

```
begin
```

```
s[1]:=chr(random(90-65)+65);
```

```
for i:=2 to 10 do s[i]:=chr(random(122-97)+97);
```

end

В данном фрагменте используется тот факт, что допустимые коды символов располагаются последовательными непрерывными участками в кодовой таблице. Рассмотрим более общий случай. Допустим, необходимо сгенерировать ключ из  $m$  символов с кодами в диапазоне от  $n1$  до  $n2$ .

Генерация ключа из  $m$  символов с кодами в диапазоне от  $n1$  до  $n2$

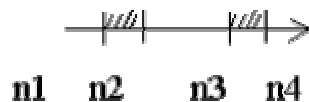
(диапазон непрерывный)

```
for i:=1 to m do str[i]:=chr(random(n2-n1)+n1);
```

На практике возможны варианты, когда символы в одних позициях ключа могут принадлежать к разным диапазонам кодов, причем между этими диапазонами может существовать разрыв.

Генерация ключа из  $m$  символов с кодами

в диапазоне от  $n1$  до  $n4$  (диапазон имеет разрыв от  $n2$  до  $n3$ )



```
for i:=1 to m do
```

```
begin
```

```
x:=random((n4 - n3) + (n2 - n1));
```

```
if x<=(n2 - n1) then str[i]:=chr(x + n1)
```

```
else str[i]:=chr(x + n1 + n3 - n2)
```

```
end;
```

Рассмотрим еще один конкретный пример. Допустим известно, что ключ состоит из 7 символов. Из них три первые символа – большие латинские буквы, далее идут две цифры, остальные – малые латинские.

Пример: длина ключа 7 символов

3 большие латинские (коды 65-90)

2 цифры (коды 48-57)

2 малые латинские (коды 97-122)

```
var
```

```
key: string[7];
```

```
begin
```

```
for i:=1 to 3 do key[i]:=chr(random(90-65)+65);
```

```
for i:=4 to 5 do key[i]:=chr(random(57-48)+48);
```

```
for i:=6 to 7 do key[i]:=chr(random(122-97)+97);
```

```
end;
```

В рассматриваемых примерах мы исходили из предположения, что хеширование будет реализовано на языке Turbo Pascal, а коды символов соответствуют альтернативной кодировке.

## **Организация данных для ускорения поиска по вторичным ключам**

До сих пор рассматривались способы поиска в таблице по ключам, позволяющим однозначно идентифицировать запись. Мы будем называть такие ключи первичными ключами. Возможен вариант организации таблицы, при котором отдельный ключ не позволяет однозначно идентифицировать запись. Такая ситуация часто встречается в базах данных. Идентификация записи осуществляется по некоторой совокупности ключей. Ключи, не позволяющие однозначно идентифицировать запись в таблице, называются вторичными ключами.

Даже при наличии первичного ключа, для поиска записи могут быть использованы вторичные. Например, поисковые системы internet часто организованы как наборы записей, соответствующих Web-страницам. В качестве вторичных ключей для поиска выступают ключевые слова, а сама задача поиска сводится к выборке из таблицы некоторого множества записей, содержащих требуемые вторичные ключи.

## Инвертированные индексы

Рассмотрим метод организации таблицы с инвертированными индексами. Для таблицы строится отдельный набор данных, содержащий так называемые инвертированные индексы. Вспомогательный набор содержит для каждого значения вторичного ключа отсортированный список адресов записей таблицы, которые содержат данный ключ.

Поиск осуществляется по вспомогательной структуре достаточно быстро, так как фактически отсутствует необходимость обращения к основной структуре данных. Область памяти, используемая для индексов,

является относительно небольшой по сравнению с другими методами организации таблиц.



**Рис.7. Метод организации таблицы с инвертированными индексами**

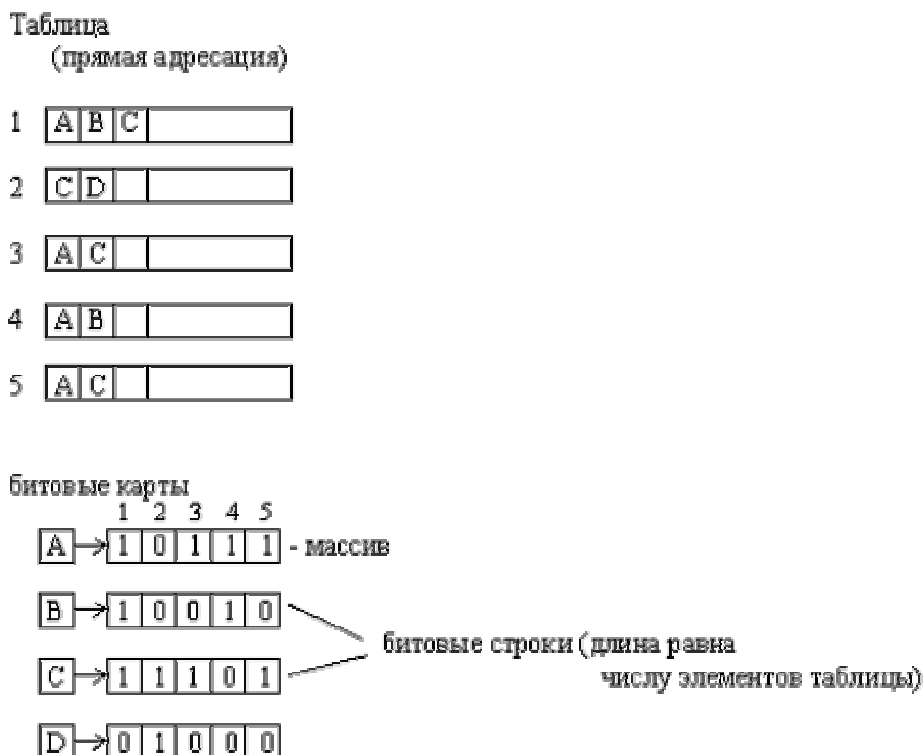
Недостатками данной системы являются большие затраты времени на составление вспомогательной структуры данных и ее обновление. Причем эти затраты возрастают с увеличение объема базы данных.

Система инвертированных индексов является чрезвычайно удобной и эффективной при организации поиска в больших таблицах.

## Битовые карты

Для таблиц небольшого объема используют организацию вспомогательной структуры данных в виде битовых карт. Для каждого значения вторичного ключа записей

основного набора данных записывается последовательность битов. Длина последовательности битов равна числу записей. Каждый бит в битовой карте соответствует одному значению вторичного ключа и одной записи. Единица означает наличие ключа в записи, а ноль – отсутствие.



**Рис.8. Организация вспомогательной структуры данных в виде битовых карт**

Основным преимуществом такой организации является очень простая и эффективная организация обработки сложных запросов, которые могут объединять значения ключей различными логическими предикатами. В этом случае поиск сводится к выполнению логических операций запроса непосредственно над битовыми строками и интерпретации результирующей битовой строки. Другим преимуществом является простота обновления карты при добавлении записей.

К недостаткам битовых карт следует отнести увеличение длины строки пропорционально длине файла. При этом заполненность карты единицами уменьшается с увеличением длины файла. Для большой длины таблицы и редко встречающихся ключах битовая карта превращается в большую разреженную матрицу, состоящую в основном из одних нулей.