Министерство образования
Российской Федерации
Петрозаводский государственный университет
Кольский филиал

СТРУКТУРЫ И АЛГОРИТМЫ ОБРАБОТКИ ДАННЫХ

Учебное пособие

Апатиты
2000

Матьяш В.А., Путилов В.А., Фильчаков В.В. , Щёкин С.В.Структуры и алгоритмы обработки данных - Апатиты, КФ ПетрГУ, 2000. - 80 с.

СОДЕРЖАНИЕ


Министерство образования
Российской Федерации
Петрозаводский государственный университет
Кольский филиал

СТРУКТУРЫ И АЛГОРИТМЫ ОБРАБОТКИ ДАННЫХ

Учебное пособие

Апатиты
2000

Матьяш В.А., Путилов В.А., Фильчаков В.В. , Щёкин С.В.Структуры и алгоритмы обработки данных - Апатиты, КФ ПетрГУ, 2000. - 80 с.


1. Основные структуры данных

Программируя решение любой задачи, необходимо выбрать уровень абстрагирования. Иными словами, определить множество данных, представляющих предметную область решаемой задачи. При выборе следует руководствоваться проблематикой решаемой задачи и способами представления информации. Здесь необходимо ориентироваться на те средства, которые предоставляют системы программирования и вычислительная техника, на которой будут выполняться программы. Во многих случаях эти критерии не являются полностью независимыми.

Вопросы представления данных часто разбиваются на различные уровни детализации. Уровню языков программирования соответствуют абстрактные типы и структуры данных. Рассмотрим их реализацию в языке программирования Turbo-Pascal. Простейшим типом данных является переменная. Существуют несколько встроенных типов данных. Например, описание

```
Var

  i, j : integer;

  x : real;

  s: string;
```

объявляет переменные i, j целочисленного типа, x - вещественного и s - строкового.

Переменной можно присвоить значение, соответствующее ее типу

```
I:=46;

X:=3.14;

S:="строка символов";
```

Такие переменные представляют собой лишь отдельные элементы. Для того чтобы можно было говорить о структурах данных, необходимо иметь некоторую агрегацию переменных. Примером такой агрегации является массив.

1.1. Массивы

Массив объединяет элементы одного типа данных. Более формально его можно определить как упорядоченную совокупность элементов некоторого типа, адресуемых при помощи одного или нескольких индексов. Частным случаем является одномерный массив

```
Var

l : array [1..100] of integer;
```

В приведенном примере описан массив l, состоящий из элементов целочисленного типа. Элементы могут быть адресованы при помощи индекса в диапазоне значений от 1 до 100. В математических расчетах такой массив соответствует вектору. Массив не обязательно должен быть одномерным. Можно описать в виде массива матрицу 100*100

```
Var

  M : array [1..100,1..100] of real;
```

В этом случае можно говорить уже о двумерном массиве. Аналогичным образом можно описать массив с большим числом измерений, например трехмерный

```
Var

  M_3_d : array [0..10,0..10,0..10] of real;
```

Теперь можно говорить уже о многомерном массиве. Доступ к элементам любого массива

осуществляется при помощи индексов как к обычной переменной.

```
M_3_d [0,0,10]:=0.25;

M[10,30]:=m_3_d[0,0,10]+0.5;

L[i]:=300;
```

1.2. Записи

Более сложным типом является запись. Основное отличие записи заключается в том, что она может объединять элементы данных разных типов.

Рассмотрим пример простейшей записи

```
Type

    Person = record

    Name: string;

    Address: string;

    Index: longint;

end;
```

Запись описанного типа объединяет четыре поля. Первые три из них символьного типа, а четвертое - целочисленного. Приведенная конструкция описывает тип записи. Для того чтобы использовать данные описанного типа, необходимо описать сами данные. Один из вариантов использования отдельных записей - объединение их в массив, тогда описание массива будет выглядеть следующим образом

```
Var

   Persons : array[1..30] of person;
```

Следует заметить, что в Turbo-pascal эти два описания можно объединить в виде описания так называемого массива записей

```
Var

   Persons : array[1..30] of record

   Name: string;

   Address: string;

   Index: longint;

end;
```

Доступ к полям отдельной записи осуществляется через имя переменной и имя поля.

```
Persons[1] . Name:="Иванов";

Persons[1] . Adress:='город Санкт-Петербург ┘ ";

Persons[2] . Name:="Петров";

Persons[2] . Adress:='город Москва ┘ ";
```

Разумеется, что запись можно использовать в качестве отдельной переменной, для этого соответствующая переменная должна иметь тип, который присвоен описанию записи

```
Type

     Person = record

          Name: string;
```

```
        Address: string;
        Index: Longint;
    end;
Var
    Person1: person;
Begin
    Person1.index:=190000;
```

1.3. Множества

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

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

```
Var
    RGB, YIQ, CMY : Set of string;
```

Здесь мы привели описание двух множеств, элементами которых являются строки. В отличие от массивов и записей здесь отсутствует возможность индексирования отдельных элементов.

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

Операции выполняются по отношению ко всей совокупности элементов множества. Можно лишь добавить, исключить или выбрать элементы, выполняя допустимые операции.

1.4. Динамические структуры данных

Мы ввели базовые структуры данных: массивы, записи, множества. Мы назвали их базовыми, поскольку из них можно образовывать более сложные структуры. Цель описания типа данных и определения некоторых переменных как относящихся к этому типу состоит в том, чтобы зафиксировать диапазон значений, присваиваемых этим переменным, и соответственно размер выделяемой для них памяти. Поэтому такие переменные называются статическими. Однако существует возможность создавать более сложные структуры данных. Для них характерно, что в процессе обработки данных изменяются не только значения переменных, но и сама их структура. Соответственно динамически изменяется и размер памяти, занимаемый такими структурами. Поэтому такие данные получили название данных с динамической структурой.

1.4.1. Линейные списки

Наиболее простой способ связать некоторое множество элементов - это организовать линейный список. При такой организации элементы некоторого типа образуют цепочку. Для связывания элементов в списке используют систему указателей. В рассматриваемом случае любой элемент линейного списка имеет один указатель, который указывает на следующий элемент в списке или является пустым указателем, что означает конец списка.

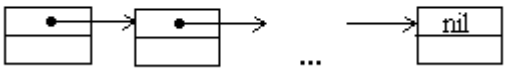


Рис. 1.1. Линейный список (связанный список)

В языке Turbo Pascal предусмотрены два типа указателей - типизированные и не типизированные указатели. В случае линейного списка описание данных может выглядеть следующим образом.

```
type
    element = record
        data:string;
        next: pointer;
    end;
var
    head: pointer;
    current, last : ^element;
```

В данном примере элемент списка описан как запись, содержащая два поля. Поле data строкового типа служит для размещения данных в элементе списка. Другое поле next представляет собой не типизированный указатель, который служит для организации списковой структуры.

В описании переменных описаны три указателя head, last и current. Head является не типизированным указателем. Указатель current является типизированным указателем, что позволяет через него организовать доступ к полям внутри элемента, имеющего тип element. Для объявления типизированного указателя обычно используется символ ^, размещаемый непосредственно перед соответствующим типом данных. Однако описание типизированного указателя еще не означает создание элемента списка. Рассмотрим, как можно осуществить создание элементов и их объединение в список.

В системе программирования Turbo Pascal для размещения динамических переменных используется специальная область памяти "heap-область". Heap-область размещается в памяти компьютера следом за областью памяти, которую занимает программа и статические данные, и может рассматриваться как сплошной массив, состоящий из байтов.

Попробуем создать первый элемент списка. Это можно осуществить при помощи процедуры New

```
New(Current);
```

После выполнения данной процедуры в динамической области памяти создается динамическая переменная, тип которой определяется типом указателя. Сам указатель можно рассматривать как адрес переменной в динамической памяти. Доступ к переменной может быть осуществлен через указатель. Заполним поля элемента списка.

```
Current^.data:= "данные в первом элементе списка " ;
Current^.next:=nil;
```

Значение указателя nil означает пустой указатель. Обратим внимание на разницу между присвоением значения указателю и данным, на которые он указывает. Для указателей допустимы только операции присваивания и сравнения. Указателю можно присваивать значение указателя того же типа или константу nil. Данным можно присвоить значения, соответствующие декларируемым типам данных. Для того чтобы присвоить значение данным, после указателя используется символ ^. Поле Current^.next само является указателем, доступ к его содержимому осуществляется через указатель Current.

В результате выполнения описанных действий мы получили список из одного элемента. Создадим еще один элемент и свяжем его с первым элементом.

Head:=Current;

New(last);

Last^.data:= "данные во втором элементе списка " ;

Last^.next:=nil;

Current^.next:=nil;

Непосредственно перед созданием первого элемента мы присвоили указателю Head значение указателя Current. Это связано с тем, что линейный список должен иметь заголовок. Другими словами, первый элемент списка должен быть отмечен указателем. В противном случае, если значение указателя Current в дальнейшем будет переопределено, то мы навсегда потеряем возможность доступа к данным, хранящимся в первом элементе списка.

Динамическая структура данных предусматривает не только добавление элементов в список, но и их удаление по мере надобности. Самым простым способом удаления элемента из списка является переопределение указателей, связанных с данным элементом (указывающих на него). Однако сам элемент данных при этом продолжает занимать память, хотя доступ к нему будет навсегда утерян. Для корректной работы с динамическими структурами следует освобождать память при удалении элемента структуры. В языке TurboPascal для этого можно использовать функцию Dispose. Покажем, как следует корректно удалить первый элемент нашего списка.

Head:=last;

Dispose(current);

Рассмотрим пример более сложной организации списка. Линейный список неудобен тем, что при попытке вставить некоторый элемент перед текущим элементом требуется обойти почти весь список, начиная с заголовка, чтобы изменить значение указателя в предыдущем элементе списка. Чтобы устранить данный недостаток введем второй указатель в каждом элементе списка. Первый указатель связывает данный элемент со следующим, а второй - с предыдущим. Такая организация динамической структуры данных получила название линейного двунаправленного списка (двусвязанного списка).

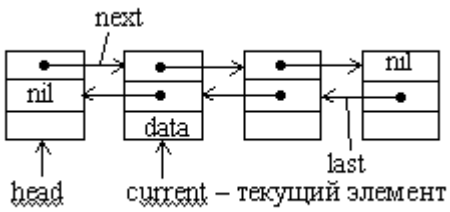


Рис.1.2. Двунаправленный список

Интересным свойством такого списка является то, что для доступа к его элементам вовсе не обязательно хранить указатель на первый элемент. Достаточно иметь указатель на любой элемент списка. Первый элемент всегда можно найти по цепочке указателей на предыдущие элементы, а последний - по цепочке указателей на следующие. Но наличие указателя на заголовок списка в ряде случаев ускоряет работу со списком.

Приведем пример программы, которая выполняет следующие операции с двунаправленным линейным списком:

добавление (справа и слева от текущего);

удаление (справа и слева от текущего);

поиск;

вывод списка.

program;

```
        type   element = record
                data:string;
                last, next: pointer;
         end;
var
      i,n: integer;
     head: pointer;
     current, pnt, pnt2: ^element;
     s:string;
begin
new(current);
head:=current;
current^.data:=head;
current^.next:=nil;
current^.last:=nil;
repeat
      writeln("1 - сделать текущим");
      writeln("2 - список элементов");
      writeln("3 - добавить справа");
      writeln("4 - добавить слева");
      writeln("5 - удалить текущий");
      writeln("6 - удалить справа от текущего");
      writeln("7 - удалить слева от текущего");
      writeln("0 - выход");
      writeln("текущий элемент: ", current^.data);
      readln(n);
     if n=1 then
{Выбор нового текущего элемента}
     begin
         writeln(""); readln(s);
         pnt:=head;
        repeat
              if pnt^.data=s then current:=pnt;
             pnt:=pnt^.next;
        until pnt=nil;
```

```
    end;
    if n=2 then
    {Вывод всех элементов}
    begin
        pnt:=head; i:=1
        repeat
                writeln(i, " - ", pnt^.data);
                pnt:=pnt^.next;
                i:=i+1;
       until pnt=nil;
    end;
    if n=3 then
{Добавление нового элемента справа от текущего}
    begin
         writeln("элемент"); readln(s);
         new(pnt);
         pnt^.data:=s;
         pnt^.last:=current;
         pnt^.next:=current^.next;
         pnt2:=current^.next;
         if not(pnt2=nil) then pnt2^.last:=pnt;
    end;
    if n=4 then
{Добавление нового элемента слева от текущего}
     begin
         writeln("элемент"); readln(s);
         new(pnt);
         pnt^.data:=s;
         pnt^.last:=current^.last;
         pnt^.next:=current;
         pnt2:=current^.last;
         if not(pnt2=nil) then pnt2^.next:=pnt;
     end;
     if n=5 and not(current=head) then
  {Удаление текущего элемента}
```

```
      begin
           pnt:=current^.last;
           pnt^.next:=current^next;
           pnt2:=current^.next;
           if not(pnt2=nil) then pnt2^.last:=current^.last;
           dispose(current);
      end;
      if n=6 and not(current^.next=nil) then
{Удаление элемента справа от текущего}
      begin
           pnt:=current^.next;
           current^.next:=pnt^next;
           pnt2:=pnt^.next;
           if not(pnt2=nil) then pnt2^.last:=current;
          dispose(pnt);
      end;
      if n=7 and not(current^.last=head) and not(current^.last=nil) then
{Удаление элемента слева от текущего}
     begin
           pnt:=current^.last;
           current^.last:=pnt^.last;
           pnt2:=pnt^.last;
           if not(pnt2=nil) then pnt2^.next:=current;
           dispose(pnt);
     end;
until n=0;
end.
```

В данной программе реализован алгоритм поиска элемента в списке (сделать текущим). В процессе поиска происходит обход с начала списка. Наличие указателя на заголовок списка ускоряет процесс поиска, так как не требуется сначала найти первый элемент, а затем - сделать обход списка.

1.4.2. Циклические списки

Линейные списки характерны тем, что в них можно выделить первый и последний элементы, причем для однонаправленного линейного списка обязательно нужно иметь указатель на первый элемент.

Циклические списки также как и линейные бывают однонаправленными и двунаправленными. Основное отличие циклического списка состоит в том, что в списке нет пустых указателей.

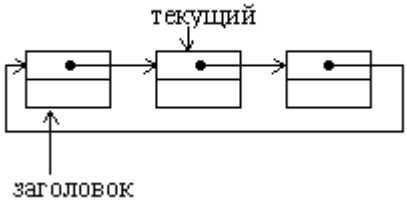


Рис.1.3. Однонаправленный циклический список

Последний элемент списка содержит указатель, связывающий его с первым элементом. Для полного обхода такого списка достаточно иметь указатель только на текущий элемент.

В двунаправленном циклическом списке система указателей аналогична системе указателей двунаправленного линейного списка.

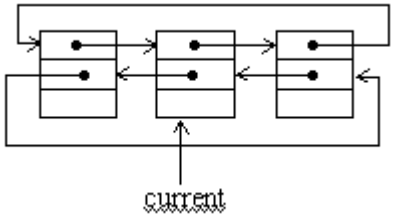


Рис.1.4. Двунаправленный циклический список

Двунаправленный циклический список позволяет достаточно просто осуществлять вставки и удаления элементов слева и справа от текущего элемента. В отличие от линейного списка, элементы являются равноправными и для выделения первого элемента необходимо иметь указатель на заголовок. Однако во многих случаях нет необходимости выделять первый элемент списка и достаточно иметь указатель на текущий элемент. Рассмотрим пример программы, которая осуществляет следующие операции с двунаправленным циклическим списком:

добавление (справа и слева от текущего);

удаление (справа и слева от текущего);

поиск;

вывод списка.

```
program;
type element = record
        data:string;
        last, next: pointer;
    end;
var
     i,n: integer;
     point: pointer;
     current, pnt, pnt2: ^element;
     s:string;
begin
```

```
new(current);
current^.data:="first";
current^.next:=current
current^.last:=current;
repeat
      writeln("1 - сделать текущим");
      writeln("2 - список элементов");
      writeln("3 - добавить справа");
      writeln("4 - добавить слева");
      writeln("5 - удалить текущий");
      writeln("6 - удалить справа от текущего");
       writeln("7 - удалить слева от текущего");
       writeln("0 - выход");
       writeln("текущий элемент: ", current^.data);
       readln(n);
       if n=1 then
{Выбор нового текущего элемента}
      begin
            writeln(""); readln(s);
            pnt:=current; point:=current;
            repeat
                  if pnt^.data=s then current:=pnt;
                  pnt:=pnt^.next;
           until pnt=point;
       end;
       if n=2 then
{Вывод всех элементов}
       begin
            pnt:=curent; i:=1
            repeat
                  writeln(i, " - ", pnt^.data);
                  pnt:=pnt^.next; i:=i+1;
            until pnt=current;
      end;
      if n=3 then
```

```
{Добавление нового элемента справа от текущего}
    begin
        writeln("элемент"); readln(s);
        new(pnt);
        pnt^.data:=s;
        pnt^.last:=current;
        pnt^.next:=current^.next;
        pnt2:=current^.next;
        pnt2^.last:=pnt;
        current^.next:=pnt;
    end;
    if n=4 then
{Добавление нового элемента слева от текущего}
    begin
        writeln("элемент"); readln(s);
        new(pnt);
        pnt^.data:=s;
        pnt^.last:=current^.last;
        pnt^.next:=current;
        pnt2:=current^.last;
        pnt2^.next:=pnt;
    end;
    if n=5 and not(current^.next=current) then
    begin
{Удаление текущего элемента}
        pnt:=current^.last;
        pnt2^.next:=current^next;
        pnt2^.last:=current^.last;
        pnt2:=current^next;
        dispose(current);
        current:=pnt2;
    end;
    if n=6 and not(current^.next=current) then
{Удаление элемента справа от текущего}
    begin
```

```
            pnt:=current^.next;
            current^.next:=pnt^next;
            pnt2:=pnt^.next;
            pnt2^.last:=current;
            dispose(pnt);
        end;
        if n=7 and not(current^.next=current)then
{Удаление элемента слева от текущего}
        begin
            pnt:=current^.last;
            current^.last:=pnt^.last;
            pnt2:=pnt^.last;
            pnt2^.next:=current;
            dispose(pnt);
        end;
until n=0;
end.
```

В данном примере указатель на первый элемент списка отсутствует. Для предотвращения зацикливания при обходе списка во время поиска указатель на текущий элемент предварительно копируется и служит ограничителем.

1.4.3. Мультисписки

Иногда возникают ситуации, когда имеется несколько разных списков, которые включают в свой состав одинаковые элементы. В таком случае при использовании традиционных списков происходит многократное дублирование динамических переменных и нерациональное использование памяти. Списки фактически используются не для хранения элементов данных, а для организации их в различные структуры. Использование мультисписков позволяет упростить эту задачу.

Мультисписок состоит из элементов, содержащих такое число указателей, которое позволяет организовать их одновременно в виде нескольких различных списков. За счет отсутствия дублирования данных память используется более рационально.

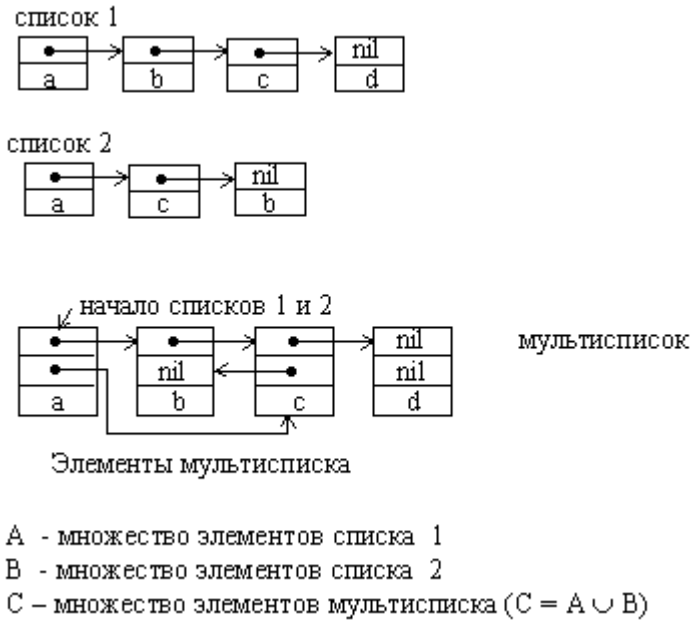


Рис.1.5. Объединение двух линейных списков в один мультисписок.

Экономия памяти - далеко не единственная причина, по которой применяют мультисписки. Многие реальные структуры данных не сводятся к типовым структурам, а представляют собой некоторую комбинацию из них. Причем комбинируются в мультисписках самые различные списки - линейные и циклические, односвязанные и двунаправленные.

1.5. Представление стека и очередей в виде списков 1.5.1. Стек

Стек представляет собой структуру данных, из которой первым извлекается тот элемент, который был добавлен в нее последним. Стек как динамическую структуру данных легко организовать на основе линейного списка. Для такого списка достаточно хранить указатель вершины стека, который указывает на первый элемент списка. Если стек пуст, то списка не существует и указатель принимает значение nil.

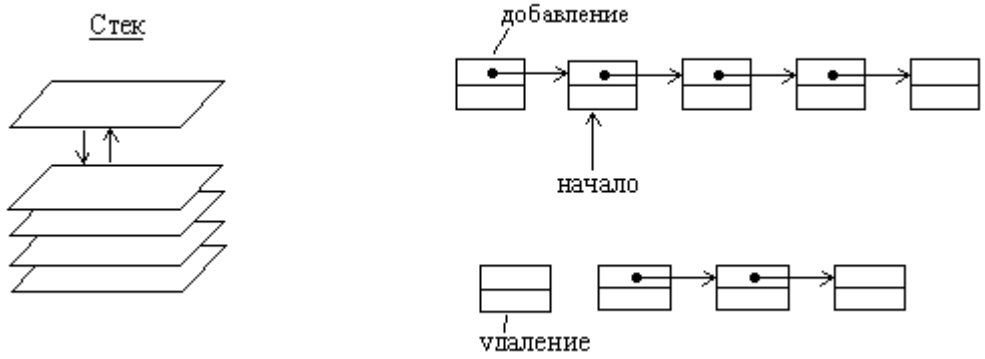


Рис.1.6. Организация стека на основе линейного списка.

Приведем пример программы, реализующей стек как динамическую структуру.

```
Program stack;

type

    element=record

        data:string;

        next:pointer;

    end;
```

```
var
    n: integer;
    current:^element;
    pnt:^element;
procedure put_element(s:string); {занесение элемента в стек}
begin
    new(pnt);
    x^.data:=s;
    x^.next:=current;
    current:=pnt;
end;
procedure get_element(var s:string); {получение элемента из стека}
begin
    if current=nil then s:="пусто" else
    begin
        pnt:=current;
        s:=pnt^.data;
        current:=pnt^.next;
        dispose(pnt);
     end;
end;
{------------program--------------}
begin
current:=nil;
repeat
      writeln("1 - добавить в стек");
      writeln("2 - удалить из стека");
      writeln("0 - выход");
      readln(n);
      if n=1 then
      begin
            write("элемент ? ");
            readln(s);
            put_element(s);
      end;
```

```
    if n=2 then
    begin
        get_element(s);
        writeln(s);
    end;
until n=0;
end.
```

В программе добавление нового элемента в стек оформлено в виде процедуры put_element, а получение элемента из стека - как процедура get_element.

1.5.2. Очереди

Дек или двусторонняя очередь, представляет собой структуру данных, в которой можно добавлять и удалять элементы с двух сторон. Дек достаточно просто можно организовать в виде двусвязанного ациклического списка. При этом первый и последний элементы списка соответствуют входам (выходам) дека.

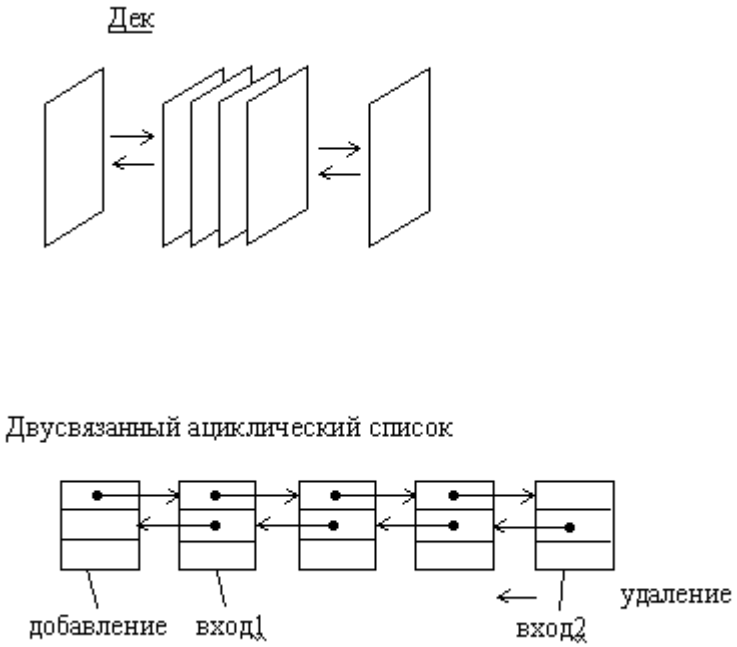


Рис.1.7. Организация дека на основе двусвязанного линейного списка

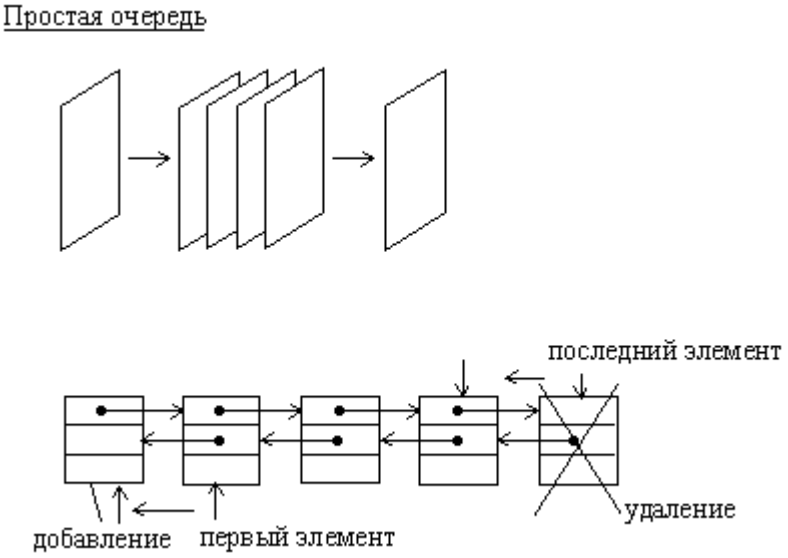


Рис.1.8. Организация дека на основе двусвязанного линейного списка

Простая очередь может быть организована на основе двусвязанного линейного списка. В отличие от дека простая очередь имеет один вход и один выход. При этом тот элемент, который был добавлен в очередь первым, будет удален из нее также первым.

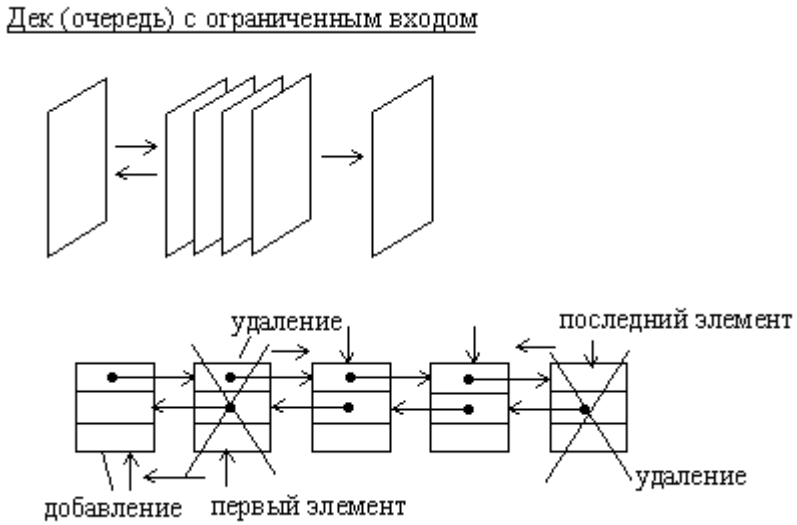
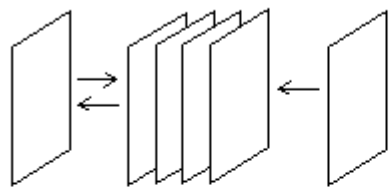


Рис.1.9. Организация дека с ограниченным входом на основе двусвязанного линейного списка

Дек (очередь) с ограниченным выходом



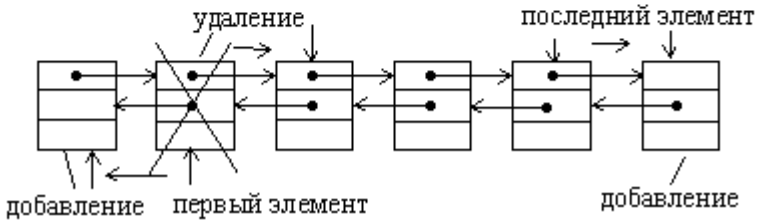


Рис.1.10. Организация дека с ограниченным выходом на основе двусвязанного линейного списка

Очередь с ограниченным входом или с ограниченным выходом также как дек или очередь можно организовать на основе линейного двунаправленного списка.

Разновидностями очередей являются очередь с ограниченным входом и очередь с ограниченным выходом. Они занимают промежуточное положение между деком и простой очередью.

Причем дек с ограниченным входом может быть использован как простая очередь или как стек.

2. Задачи поиска в структурах данных

Одно из наиболее часто встречающихся в программировании действий - поиск. Существует несколько основных вариантов поиска, и для них создано много различных алгоритмов. При дальнейшем рассмотрении делается принципиальное допущение: группа данных, в которой необходимо найти заданный элемент, фиксирована. Будет считаться, что множество из N элементов задано в виде такого массива

a: array[0..N-1] of Item

Обычно тип Item описывает запись с некоторым полем, играющим роль ключа. Задача заключается в поиске элемента, ключ которого равен заданному "аргументу поиска" x. Полученный в результате индекс i, удовлетворяющий условию a[i].key = x, обеспечивает доступ к другим полям обнаруженного элемента. Так как здесь рассматривается, прежде всего, сам процесс поиска, то мы будем считать, что тип Item включает только ключ.

2.1. Линейный поиск

Если нет никакой дополнительной информации о разыскиваемых данных, то очевидный подход - простой последовательный просмотр массива с увеличением шаг за шагом той его части, где желаемого элемента не обнаружено. Такой метод называется линейным поиском. Условия окончания поиска таковы:

Элемент найден, т. е. ai = x.

Весь массив просмотрен и совпадения не обнаружено.

Это дает нам линейный алгоритм:

Алгоритм 1.

i:=0;

while (i<N) and (a[i]<>x) do i:=i+1

Следует обратить внимание, что если элемент найден, то он найден вместе с минимально возможным индексом, т. е. это первый из таких элементов. Равенство i=N свидетельствует, что совпадения не существует.

Очевидно, что окончание цикла гарантировано, поскольку на каждом шаге значение i увеличивается, и, следовательно, оно достигнет за конечное число шагов предела N; фактически же, если совпадения не было, это произойдет после N шагов.

На каждом шаге алгоритма осуществляется увеличение индекса и вычисление логического выражения. Можно упростить шаг алгоритма, если упростить логическое выражение, которое состоит из двух членов. Это упрощение осуществляется путем формулирования логического выражения из одного члена, но при этом необходимо гарантировать, что совпадение произойдет всегда. Для этого достаточно в конец массива поместить дополнительный элемент со значением x. Такой вспомогательный элемент называется "барьером". Теперь массив будет описан так:

a: array[0..N] of integer

и алгоритм линейного поиска с барьером выглядит следующим образом:

Алгоритм 1".

a[N]:=x; i:=0;

while a[i]<>x do i:=i+1

Ясно, что равенство i=N свидетельствует о том, что совпадения (если не считать совпадения с барьером) не было.

2.2. Поиск делением пополам (двоичный поиск)

Совершенно очевидно, что других способов убыстрения поиска не существует, если, конечно, нет еще какой-либо информации о

данных, среди которых идет поиск. Хорошо известно, что поиск можно сделать значительно более эффективным, если данные будут упорядочены. Поэтому приведем алгоритм (он называется "поиском делением пополам"), основанный на знании того, что массив A упорядочен, т. е. удовлетворяет условию

$$a_{k-1} \le a_k, \text{ где } 1 \le k < N$$

Основная идея - выбрать случайно некоторый элемент, предположим am, и сравнить его с аргументом поиска x. Если он равен x, то поиск заканчивается, если он меньше x, то делается вывод, что все элементы с индексами, меньшими или равными m, можно исключить из дальнейшего поиска; если же он больше x, то исключаются индексы больше и равные m. Выбор m совершенно не влияет на корректность алгоритма, но влияет на его эффективность. Очевидно, что чем большее количество элементов исключается на каждом шаге алгоритма, тем этот алгоритм эффективнее. Оптимальным решением будет выбор среднего элемента, так как при этом в любом случае будет исключаться половина массива.

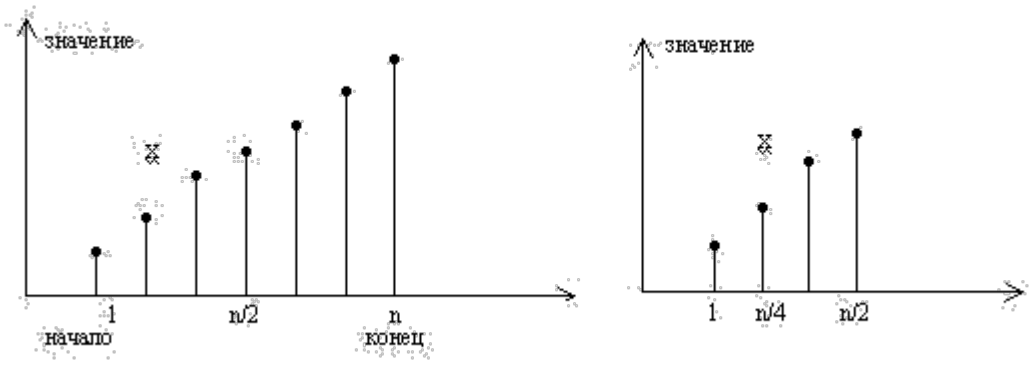


Рис.2.1. Поиск делением пополам

В этом алгоритме используются две индексные переменные L и R, которые отмечают соответственно левый и правый конец секции массива a, где еще может быть обнаружен требуемый элемент.

Алгоритм 2.

```
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 begin

            Found:=true

        end else begin

            if a[m]<x then L:=m+1 else R:=m-1

        end

end;
```

Максимальное число сравнений для этого алгоритма равно log2n, округленному до ближайшего целого. Таким образом, приведенный алгоритм существенно выигрывает по сравнению с линейным поиском, ведь там ожидаемое число сравнений - N/2.

Эффективность несколько улучшается, если поменять местами заголовки условных операторов. Проверку на равенство можно выполнять во вторую очередь, так как она встречается лишь единожды и приводит к окончанию работы. Но более существенный выигрыш даст отказ от окончания поиска при фиксации совпадения. На первый взгляд это кажется странным, однако, при внимательном рассмотрении обнаруживается, что выигрыш в эффективности на каждом шаге превосходит потери от сравнения с несколькими дополнительными элементами (число шагов в худшем случае равно logN).

Алгоритм 2".

```
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
```

Окончание цикла гарантировано. Это объясняется следующим. В начале каждого шага L<R. Для среднего арифметического m справедливо условие L face="Symbol" =< m < R. Следовательно, разность L-R действительно убывает, ведь либо L увеличивается при присваивании ему значения m+1, либо R уменьшается при присваивании значения m. При L face="Symbol" =< R повторение цикла заканчивается.

Выполнение условия L=R еще не свидетельствует о нахождении требуемого элемента. Здесь требуется дополнительная проверка. Также, необходимо учитывать, что элемент a[R] в сравнениях никогда не участвует. Следовательно, и здесь необходима дополнительная проверка на равенство a[R]=x. Следует отметить, что эти проверки выполняются однократно.

Приведенный алгоритм, как и в случае линейного поиска, находит совпадающий элемент с наименьшим индексом.

2.3. Поиск в таблице

Поиск в массиве иногда называют поиском в таблице, особенно если ключ сам является составным объектом, таким, как массив чисел или символов. Часто встречается именно последний случай, когда массивы символов называют строками или словами. Строковый тип определяется так:

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

соответственно определяется и отношение порядка для строк x и y:

x = y, если xj = yj для 0 face="Symbol" =< j < M

x < y, если xi < yi для 0 face="Symbol" =< i < M и xj = yj для 0 face="Symbol" =< j < i

Для того чтобы установить факт совпадения, необходимо установить, что все символы сравниваемых строк соответственно равны один другому. Поэтому сравнение составных операндов сводится к поиску их несовпадающих частей, т. е. к поиску "на неравенство". Если неравных частей не существует, то можно говорить о равенстве. Предположим, что размер слов достаточно мал, скажем, меньше 30. В этом случае можно использовать линейный поиск и поступать таким образом.

Для большинства практических приложений желательно исходить из того, что строки имеют переменный размер. Это предполагает, что размер указывается в каждой отдельной строке. Если исходить из ранее описанного типа, то размер не должен превосходить максимального размера M. Такая схема достаточно гибка и подходит для многих случаев, в то же время она позволяет избежать сложностей динамического распределения памяти. Чаще всего используются два таких представления размера строк:

Размер неявно указывается путем добавления концевого символа, больше этот символ нигде не употребляется. Обычно для этой цели используется "непечатаемый" символ со значением 00h. (Для дальнейшего важно, что это минимальный символ из всего множества символов.)

Размер явно хранится в качестве первого элемента массива, т. е. строка s имеет следующий вид: s = s0, s1, s2, ..., sM-1. Здесь s1, ..., sM-1 - фактические символы строки, а s0 = Chr(M). Такой прием имеет то преимущество, что размер явно доступен, недостаток же в том, что этот размер ограничен размером множества символов (256).

В последующем алгоритме поиска отдается предпочтение первой схеме. В этом случае сравнение строк выполняется так:

```
i:=0;

while (x[i]=y[i]) and (x[i]<>00h) do i:=i+1
```

Концевой символ работает здесь как барьер.

Теперь вернемся к задаче поиска в таблице. Он требует "вложенных" поисков, а именно: поиска по строчкам таблицы, а для каждой строчки последовательных сравнений - между компонентами. Например, пусть таблица T и аргумент поиска x определяются таким

образом:

```
T: array[0..N-1] of String;

x: String
```

Допустим, N достаточно велико, а таблица упорядочена в алфавитном порядке. При использовании алгоритма поиска делением пополам и алгоритма сравнения строк, речь о которых шла выше, получаем такой фрагмент программы:

```
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]<>00h) 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]<>00h) do i:=i+1

end

{(R<N) and (T[R,i]=x[i]) фиксирует совпадение}
```

2.3.1. Прямой поиск строки

Часто приходится сталкиваться со специфическим поиском, так называемым поиском строки. Его можно определить следующим образом. Пусть задан массив s из N элементов и массив p из M элементов, причем 0 < M face="Symbol" =< N. Описаны они так:

```
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 символы, т.е. s можно считать некоторым текстом, а p - словом, и необходимо найти первое вхождение этого слова в указанном тексте. Это действие типично для любых систем обработки текстов, отсюда и очевидная заинтересованность в поиске эффективного алгоритма для этой задачи. Разберем алгоритм поиска, который будем называть прямым поиском строки.

Алгоритм 3.

```
i:=-1;

repeat

       i:=i+1; j:=0;

       while (j<M) and (s[i+j]=p[j]) do j:=j+1;

until (j=M) or (i=N-M)
```

Вложенный цикл с предусловием начинает выполняться тогда, когда первый символ слова p совпадает с очередным, i-м, символом текста s. Этот цикл повторяется столько раз, сколько совпадает символов текста s, начиная с i-го символа, с символами слова p (максимальное количество повторений равно M). Цикл завершается при исчерпании символов слова p (перестает выполняться условие j<M) или при несовпадении очередных символов s и p (перестает выполняться условие s[i+j]=p[j]). Количество совпадений подсчитывается с использованием j. Если совпадение произошло со всеми символами слова p (т.е. слово p найдено), то выполняется условие j=M, и алгоритм завершается. В противном случае поиск продолжается до тех пор, пока не просмотренной останется часть текста s, которая содержит символов, меньше, чем есть в слове p (т.е. этот остаток уже не может совпасть со словом p). В этом случае

выполняется условие i=N-M, что тоже приводит к завершению алгоритма. Это показывает гарантированность окончания алгоритма.

Этот алгоритм работает достаточно эффективно, если допустить, что несовпадение пары символов происходит после незначительного количества сравнений во внутреннем цикле. При большой мощности типа Item это достаточно частый случай. Можно предполагать, что для текстов, составленных из 128 символов, несовпадение будет обнаруживаться после одной или двух проверок. Тем не менее, в худшем случае производительность будет внушать опасение.

2.3.2. Алгоритм Кнута, Мориса и Пратта

Приблизительно в 1970 г. Д. Кнут, Д. Морис и В. Пратт изобрели алгоритм, фактически требующий только N сравнений даже в самом плохом случае. Новый алгоритм основывается на том соображении, что после частичного совпадения начальной части слова с соответствующими символами текста фактически известна пройденная часть текста и можно "вычислить" некоторые сведения (на основе самого слова), с помощью которых потом можно быстро продвинуться по тексту. Приведенный пример поиска слова ABCABD показывает принцип работы такого алгоритма. Символы, подвергшиеся сравнению, здесь подчеркнуты. Обратите внимание: при каждом несовпадении пары символов слово сдвигается на все пройденное расстояние, поскольку меньшие сдвиги не могут привести к полному совпадению.

```
ABCABCABAABCABD
ABCABD
  ABCABD
   ABCABD
    ABCABD
     ABCABD
```

Основным отличием КМП-алгоритма от алгоритма прямого поиска является осуществления сдвига слова не на один символ на каждом шаге алгоритма, а на некоторое переменное количество символов. Таким образом, перед тем как осуществлять очередной сдвиг, необходимо определить величину сдвига. Для повышения эффективности алгоритма необходимо, чтобы сдвиг на каждом шаге был бы как можно большим.

Если j определяет позицию в слове, содержащую первый несовпадающий символ (как в алгоритме прямого поиска), то величина сдвига определяется как j-D. Значение D определяется как размер самой длинной последовательности символов слова, непосредственно предшествующих позиции j, которая полностью совпадает с началом слова. D зависит только от слова и не зависит от текста. Для каждого j будет своя величина D, которую обозначим dj.

Так как величины dj зависят только от слова, то перед началом фактического поиска можно вычислить вспомогательную таблицу d; эти вычисления сводятся к некоторой предтрансляции слова. Соответствующие усилия будут оправданными, если размер текста значительно превышает размер слова (M<<N). Если нужно искать многие вхождения одного и того же слова, то можно пользоваться одними и теми же d. Приведенные примеры объясняют функцию d.

Последний пример на рис. 2.2 показывает: так как pj равно A вместо F, то соответствующий символ текста не может быть символом A из-за того, что условие si<>pj заканчивает цикл. Следовательно, сдвиг на 5 не приведет к последующему совпадению, и поэтому можно увеличить размер сдвига до шести. Учитывая это, предопределяем вычисление dj как поиск самой длинной совпадающей последовательности с дополнительным ограничением pdj<>pj. Если никаких совпадений нет,

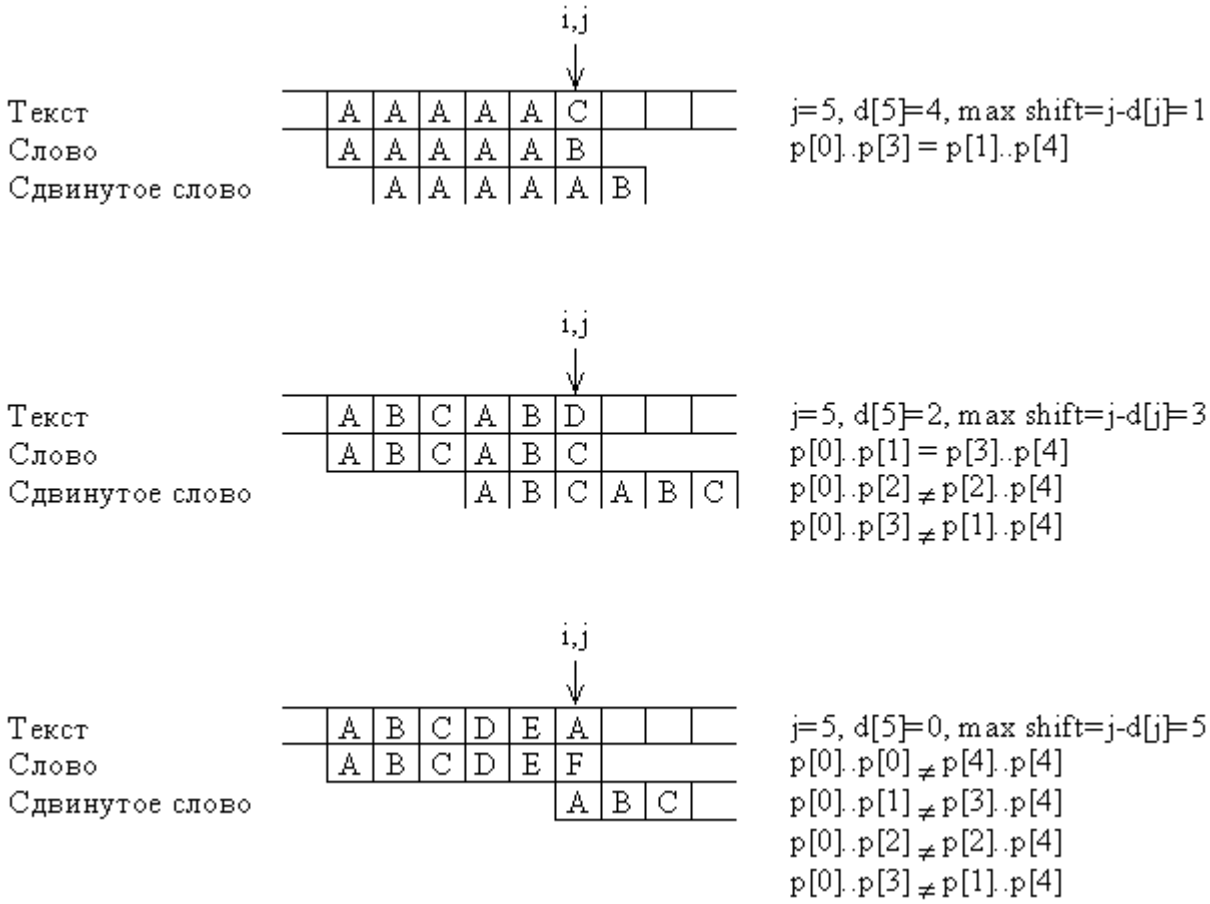


Рис. 2.2. Частичное совпадение со словом и вычисление d_j.

то считается dj=-1, что указывает на сдвиг "на целое" слово относительно его текущей позиции. Следующая программа демонстрирует КМП-алгоритм.

```
Program KMP;

const

    Mmax = 100; Nmax = 10000;

var

    i, j, k, M, N: integer;

    p: array[0..Mmax-1] of char; {слово}

    s: array[0..Mmax-1] of char; {текст}

    d: array[0..Mmax-1] of integer;

begin

{Ввод текста s и слова p}

Write('N:'); Readln(N);

Write('s:'); Readln(s);

Write('M:'); Readln(M);

Write('p:'); Readln(p);

{Заполнение массива d}
```

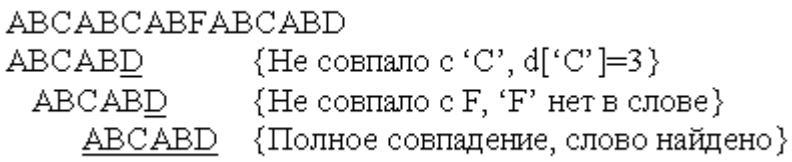
```
j:=0; k:=-1; d[0]:=-1;

while j<(M-1) do begin

        while(k>=0) and (p[j]<>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 слова p в тексте s}

i:=0; j:=0;

while (j<M) and (i<N) do begin

      while (j>=0) and (s[i]<>p[j]) do j:=d[j]; {Сдвиг слова}

       i:=i+1; j:=j+1;

end;

{Вывод результата поиска}

if j=M then Writeln('Yes') {найден }

else Writeln('No'); {не найден}

Readln;

end.
```

Точный анализ КМП-поиска, как и сам его алгоритм, весьма сложен. Его изобретатели доказывают, что требуется порядка M+N сравнений символов, что значительно лучше, чем M*N сравнений из прямого поиска. Они так же отмечают то положительное свойство, что указатель сканирования i никогда не возвращается назад, в то время как при прямом поиске после несовпадения просмотр всегда начинается с первого символа слова и поэтому может включать символы, которые ранее уже просматривались. Это может привести к негативным последствиям, если текст читается из вторичной памяти, ведь в этом случае возврат обходится дорого. Даже при буферизованном вводе может встретиться столь большое слово, что возврат превысит емкость буфера.

2.3.3. Алгоритм Боуера и Мура

КМП-поиск дает подлинный выигрыш только тогда, когда неудаче предшествовало некоторое число совпадений. Лишь в этом случае слово сдвигается более чем на единицу. К несчастью, это скорее исключение, чем правило: совпадения встречаются значительно реже, чем несовпадения. Поэтому выигрыш от использования КМП-стратегии в большинстве случаев поиска в обычных текстах весьма незначителен. Метод же, предложенный Р. Боуером и Д. Муром в 1975 г., не только улучшает обработку самого плохого случая, но дает выигрыш в промежуточных ситуациях.

БМ-поиск основывается на необычном соображении - сравнение символов начинается с конца слова, а не с начала. Как и в случае КМП-поиска, слово перед фактическим поиском трансформируется в некоторую таблицу. Пусть для каждого символа x из алфавита величина dx - расстояние от самого правого в слове вхождения x до правого конца слова. Представим себе, что обнаружено расхождение между словом и текстом. В этом случае слово сразу же можно сдвинуть вправо на dpM-1 позиций, т.е. на число позиций, скорее всего большее единицы. Если несовпадающий символ текста в слове вообще не встречается, то сдвиг становится даже больше, а именно сдвигать можно на длину всего слова. Вот пример, иллюстрирующий этот процесс:



Ниже приводится программа с упрощенной стратегией Боуера-Мура, построенная так же, как и предыдущая программа с КМП-алгоритмом. Обратите внимание на такую деталь: во внутреннем цикле используется цикл с repeat, где перед сравнением s и p увеличиваются значения k и j. Это позволяет исключить в индексных выражениях составляющую -1.

```
Program BM;

const

    Mmax = 100; Nmax = 10000;

var

    i, j, k, M, N: integer;

    ch: char;

    p: array[0..Mmax-1] of char; {слово}

    s: array[0..Nmax-1] of char; {текст}

    d: array[' '..'z'] of integer;

begin

{Ввод текста s и слова p}

Write('N:'); Readln(N);

Write('s:'); Readln(s);

Write('M:'); Readln(M);

Write('p:'); Readln(p);

{Заполнение массива d}

for ch:=' ' to 'z' do d[ch]:=M;

for j:=0 to M-2 do d[p[j]]:=M-j-1;

{Поиск слова p в тексте s}

i:=M;

repeat

      j:=M; k:=i;

      repeat {Цикл сравнения символов }

            k:=k-1; j:=j-1; {слова, начиная с правого.}

      until (j<0) or (p[j]<>s[k]); {Выход, если сравнили все}

      {слово или несовпадение. }

      i:=i+d[s[i-1]]; {Сдвиг слова вправо }
```

```
until (j<0) or (i>N);

{Вывод результата поиска}

if j<0 then WriteIn('Yes') {найден }

else WriteIn('No'); {не найден}

ReadIn;

end.
```

Почти всегда, кроме специально построенных примеров, данный алгоритм требует значительно меньше N сравнений. В самых же благоприятных обстоятельствах, когда последний символ слова всегда попадает на несовпадающий символ текста, число сравнений равно N/M.

Авторы алгоритма приводят и несколько соображений по поводу дальнейших усовершенствований алгоритма. Одно из них - объединить приведенную только что стратегию, обеспечивающую большие сдвиги в случае несовпадения, со стратегией Кнута, Морриса и Пратта, допускающей "ощутимые" сдвиги при обнаружении совпадения (частичного). Такой метод требует двух таблиц, получаемых при предтрансляции: d1 - только что упомянутая таблица, а d2 - таблица, соответствующая КМП-алгоритму. Из двух сдвигов выбирается больший, причем и тот и другой "говорят", что никакой меньший сдвиг не может привести к совпадению. Дальнейшее обсуждение этого предмета приводить не будем, поскольку дополнительное усложнение формирования таблиц и самого поиска, кажется, не оправдывает видимого выигрыша в производительности. Фактические дополнительные расходы будут высокими и неизвестно, приведут ли все эти ухищрения к выигрышу или проигрышу.

3. Методы ускорения доступа к данным

3.1. 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.

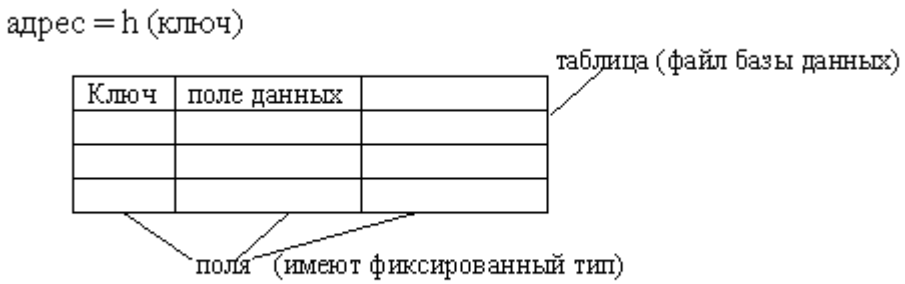


Рис.3.1. Хеш-таблица

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".

3.1.1. 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.3.2. Разновидности методов разрешение коллизий

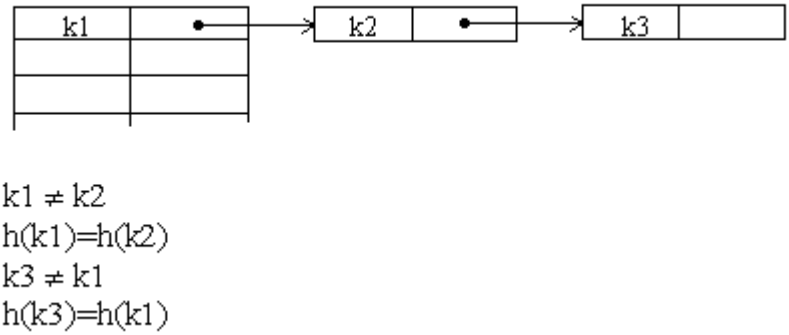


Рис.3.3. Разрешение коллизий при добавлении элементов методом цепочек

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

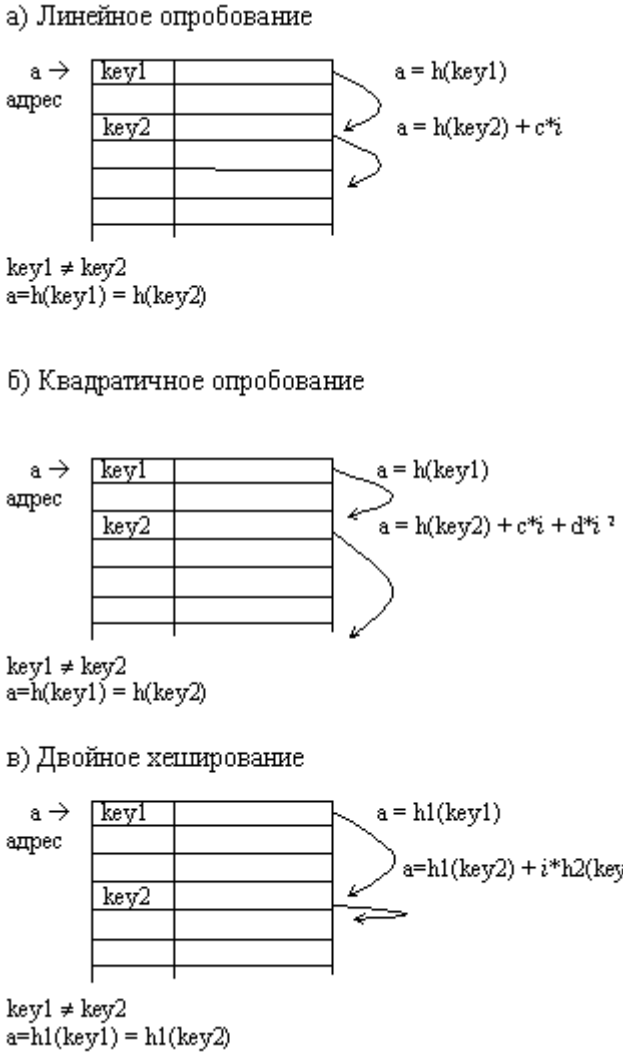


Рис.3.4. Разрешение коллизий при добавлении элементов методами открытой адресации.

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

$a = h(key2) + c*i + d*i^2$

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(key) + i*c
- Если t(a) = свободно, то t(a) = key, записать элемент, ***стоп элемент добавлен***
- 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

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

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

Удаление

- i = 0
- a = h(key) + i*c
- Если t(a) = key, то t(a) =удалено, ***стоп элемент удален***
- Если t(a) = 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.

3.1.2. 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.

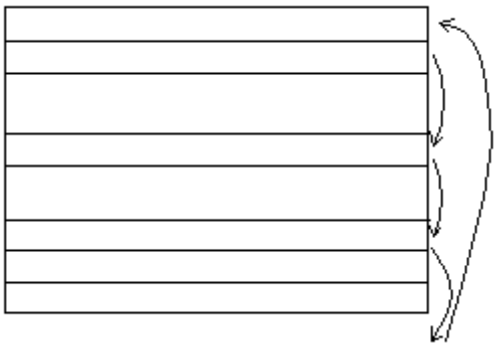


Рис.3.5. Циклический переход к началу таблицы.

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(key) + c*i) div n + (h(key) + c*i) mod n) 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
- a = ((h(key) + c*i) div n + (h(key) + c*i) mod n) 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 зацикливанию. 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 понадобиться 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 зацикливания алгоритма в случае повторного просмотра элементов таблицы.

Рассмотрим алгоритм вставки, реализующий предлагаемый подход.

Вставка

- i = 0
- a = ((h(key) + c*i) div n + (h(key) + c*i) mod n) mod n
- Если t(a) = свободно или t(a) = удалено, то t(a) = 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(key) + c*i) div n + (h(key) + c*i) mod n) mod n
- Если t(a) = key, то t(a) =удалено, ***стоп элемент удален***
- Если t(a) = свободно или i>m, то ***стоп элемент не найден***
- i = i + 1, перейти к шагу 2

Поиск

- i = 0
- a = ((h(key) + c*i) div n + (h(key) + c*i) mod n) mod n
- Если t(a) = key, то ***стоп элемент найден***
- Если t(a) = свободно или i>m, то ***стоп элемент не найден***
- i = i + 1, перейти к шагу 2

3.1.3. 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. 3.6. 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

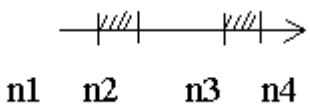
(диапазон непрерывный)

```
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

1. 3 большие латинские (коды 65-90)
2. 2 цифры (коды 48-57)
3. 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)+57);

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.

3.2. 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.

3.2.1. 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.

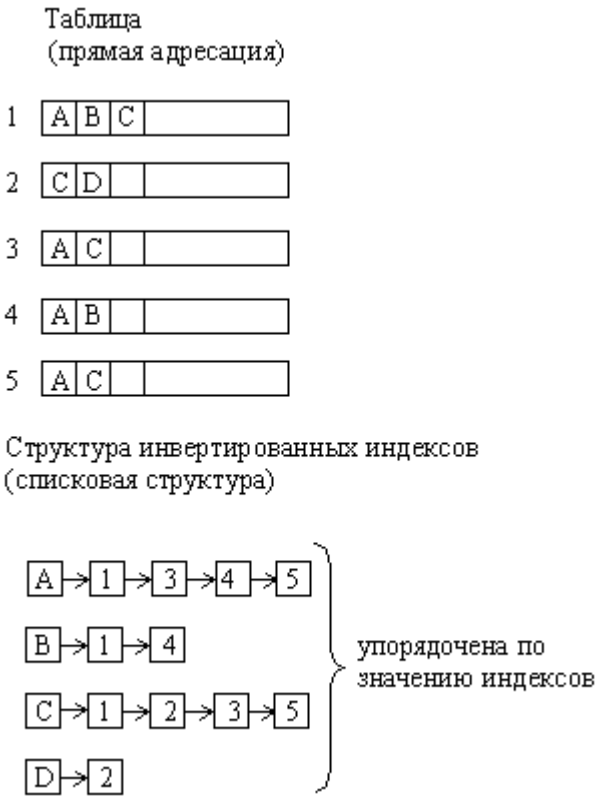


Рис.3.7. Метод организации таблицы с инвертированными индексами

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.

3.2.2. 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.

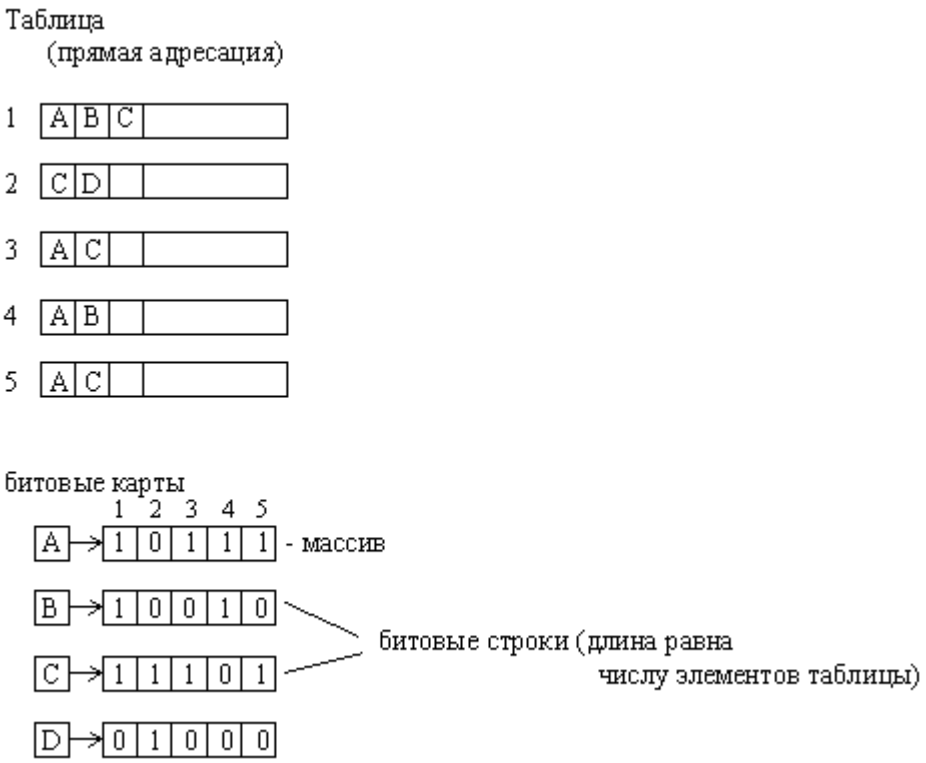


Рис.3.8. Организация вспомогательной структуры данных в виде битовых карт

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 длине таблицы и редко встречающихся ключах битовая карта превращается в большую разреженную матрицу, состоящую в основном из одних нулей.

4. Представление графов и деревьев

Теория графов является важной частью вычислительной математики. С помощью этой теории решаются большое количество задач из различных областей. Граф состоит из множества вершин и множества ребер, которые соединяют между собой вершины. С точки зрения теории графов не имеет значения, какой смысл вкладывается в вершины и ребра. Вершинами могут быть населенными пункты, а ребрами дороги, соединяющие их, или вершинами являться подпрограммы, соединенные вершин ребрами означает взаимодействие подпрограмм. Часто имеет значение направления дуги в графе. Если ребро имеет направление, оно называется дугой, а граф с ориентированными ребрами называется орграфом.

Дадим теперь более формально основное определение теории графов. Граф G есть упорядоченная пара (V,E), где V - непустое множество вершин, E - множество пар элементов множества V, пара элементов из V называется ребром. Упорядоченная пара элементов из V называется дугой. Если все пары в E - упорядочены, то граф называется ориентированным.

Путь - это любая последовательность вершин орграфа такая, что в этой последовательности вершина b может следовать за вершиной a, только если существует дуга, следующая из a в b. Аналогично можно определить путь, состоящий из дуг. Путь начинающийся в одной вершине и заканчивающийся в одной вершине называется циклом. Граф в котором отсутствуют циклы, называется ациклическим.

Важным частным случаем графа является дерево. Деревом называется орграф для которого :

1. Существует узел, в которой не входит не одной дуги. Этот узел называется корнем.

2. В каждую вершину, кроме корня, входит одна дуга.

С точки зрения представления в памяти важно различать два типа деревьев: бинарные и сильноветвящиеся.

В бинарном дереве из каждой вершины выходит не более двух дуг. В сильноветвящемся дереве количество дуг может быть произвольным.

4.1. Бинарные деревья

Бинарные деревья классифицируются по нескольким признакам. Введем понятия степени узла и степени дерева. Степенью узла в дереве называется количество дуг, которое из него выходит. Степень дерева равна максимальной степени узла, входящего в дерево. Исходя из определения степени понятно, что степень узла бинарного дерева не превышает числа два. При этом листьями в дереве являются вершины, имеющие степень ноль.

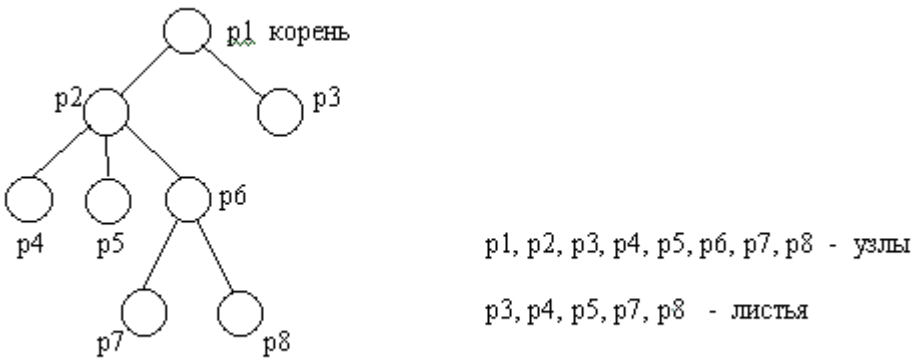


Рис.4.1. Бинарное дерево

Другим важным признаком структурной классификации бинарных деревьев является строгость бинарного дерева. Строго бинарное дерево состоит только из узлов, имеющих степень два или степень ноль. Нестрого бинарное дерево содержит узлы со степенью равной одному.

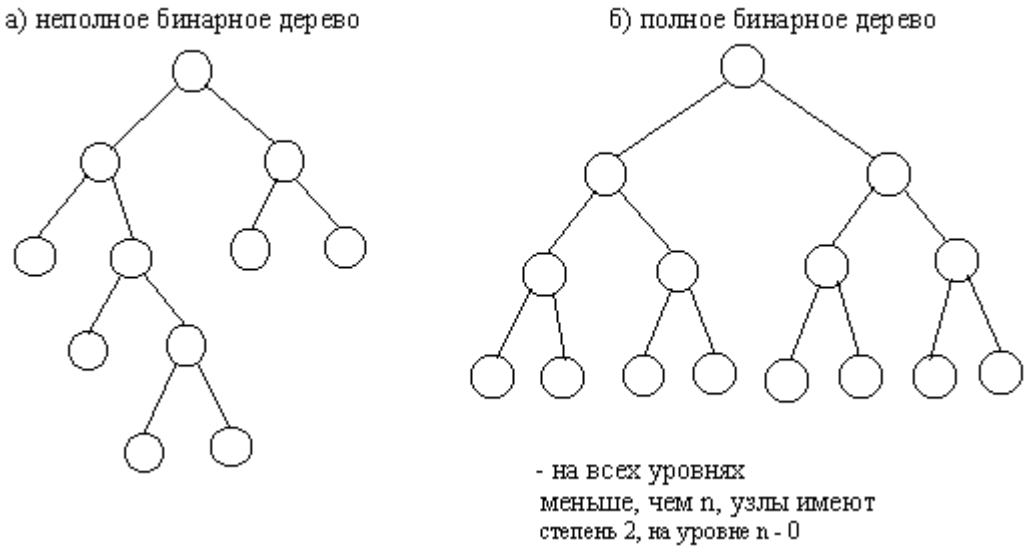


Рис.4.2. Полное и неполное бинарные деревья

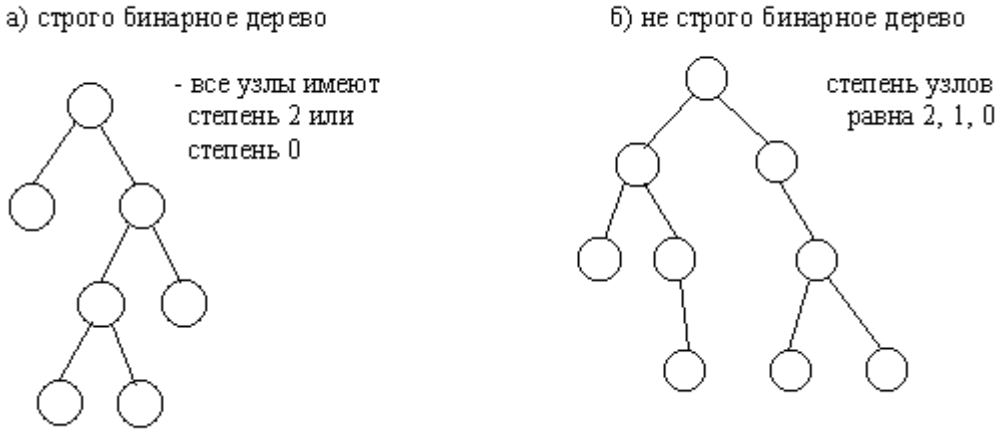


Рис.4.3. Строго и не строго бинарные деревья

4.2. Представление бинарных деревьев

Бинарные деревья достаточно просто могут быть представлены в виде списков или массивов. Списочное представление бинарных деревьев основано на элементах, соответствующих узлам дерева. Каждый элемент имеет поле данных и два поля указателей. Один указатель используется для связывания элемента с правым потомком, а другой - с левым. Листья имеют пустые указатели потомков. При таком способе представления дерева обязательно следует сохранять указатель на узел, являющийся корнем дерева.

Можно заметить, что такой способ представления имеет сходство с простыми линейными списками. И это сходство не случайно. На самом деле рассмотренный способ представления бинарного дерева является разновидностью мультисписка, образованного комбинацией множества линейных списков. Каждый линейный список объединяет узлы, входящие в путь от корня дерева к одному из листьев.

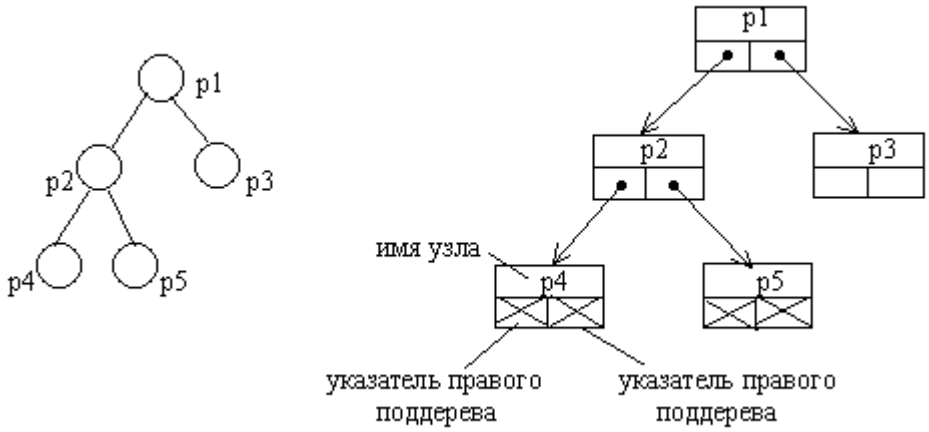


Рис.4.4. Представление бинарного дерева в виде списковой структуры

Приведем пример программы, которая осуществляет создание и редактирование бинарного дерева, представленного в виде списковой структуры

```
program bin_tree_edit;
type node=record
        name: string;
        left, right: pointer;
    end;
var
     n:integer;
     pnt_s,current_s,root: pointer;
     pnt,current:^node;
     s: string;
procedure node_search (pnt_s:pointer; var current_s:pointer);
{Поиск узла по содержимому}
var
      pnt_n:^node;
begin
pnt_n:=pnt_s; writeln(pnt_n^.name);
if not (pnt_n^.name=s) then
      begin
         if pnt_n^.left <> nil then
               node_search (pnt_n^.left,current_s);
        if pnt_n^.right <> nil then
```

```
                node_search (pnt_n^.right,current_s);
        end
else current_s:=pnt_n;
end;
procedure node_list (pnt_s:pointer);
{Вывод списка всех узлов дерева}
var
        pnt_n:^node;
begin
pnt_n:=pnt_s; writeln(pnt_n^.name);
if pnt_n^.left <> nil then node_list (pnt_n^.left);
if pnt_n^.right <> nil then node_list (pnt_n^.right);
end;
procedure node_dispose (pnt_s:pointer);
{Удаление узла и всех его потомков в дереве}
var
        pnt_n:^node;
begin
if pnt_s <> nil then
        begin
            pnt_n:=pnt_s; writeln(pnt_n^.name);
            if pnt_n^.left <> nil then
                node_dispose (pnt_n^.left);
            if pnt_n^.right <> nil then
                node_dispose (pnt_n^.right);
           dispose(pnt_n);
        end
end;
begin
new(current);root:=current;
current^.name:='root';
current^.left:=nil;
current^.right:=nil;
```

```
repeat
        writeln('текущий узел -',current^.name);
        writeln('1-присвоить имя левому потомоку');
        writeln('2-присвоить имя правому потомку');
        writeln('3-сделать узел текущим');
        writeln('4-вывести список всех узлов');
        writeln('5-удалить потомков текущего узла');
        read(n);
        if n=1 then
        begin {Создание левого потомка}
             if current^.left= nil then new(pnt)
             else pnt:= current^.left;
             writeln('left ?');
             readln;
             read(s);
             pnt^.name:=s;
             pnt^.left:=nil;
             pnt^.right:=nil;
            current^.left:= pnt;
         end;
        if n=2 then
         begin {Создание правого потомка}
              if current^.right= nil then new(pnt)
             else pnt:= current^.right;
             writeln('right ?');
             readln;
             read(s);
             pnt^.name:=s;
             pnt^.left:=nil;
             pnt^.right:=nil;
             current^.right:= pnt;
         end;
         if n=3 then
```

```
        begin {Поиск узла}
            writeln('name ?');
            readln;
            read(s);
            current_s:=nil; pnt_s:=root;
            node_search (pnt_s, current_s);
            if current_s <> nil then current:=current_s;
        end;
        if n=4 then
        begin {Вывод списка узлов}
            pnt_s:=root;
            node_list(pnt_s);
        end;
        if n=5 then
        begin {Удаление поддерева}
             writeln('l,r ?');
             readln;
             read(s);
             if (s='l') then
                  begin {Удаление левого поддерева}
                  pnt_s:=current^.left;
                  current^.left:=nil;
                  node_dispose(pnt_s);
                  end
             else
                 begin {Удаление правого поддерева}
                 pnt_s:=current^.right;
                 current^.right:=nil;
                 node_dispose(pnt_s);
                  end;
        end;
until n=0
end.
```

В виде массива проще всего представляется полное бинарное дерево, так как оно всегда имеет строго определенное число вершин на каждом уровне. Вершины можно пронумеровать слева направо последовательно по уровням и использовать эти номера в качестве индексов в одномерном массиве.

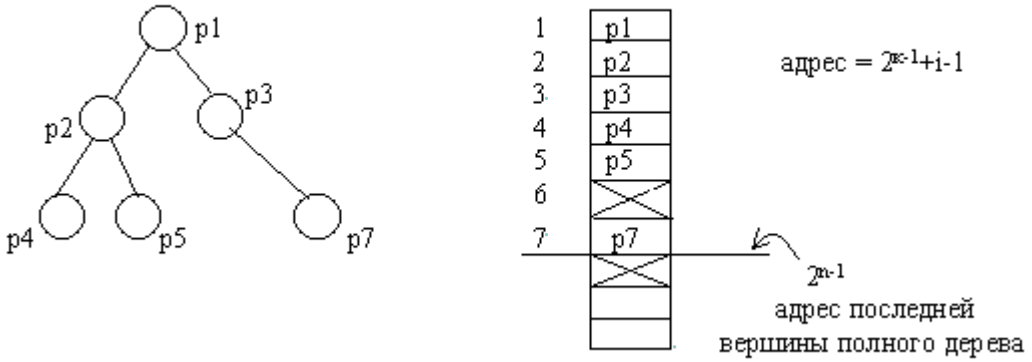


Рис.4.5. Представление бинарного дерева в виде массива

Если число уровней дерева в процессе обработки не будет существенно изменяться, то такой способ представления полного бинарного дерева будет значительно более экономичным, чем любая списковая структура.

Однако далеко не все бинарные деревья являются полными. Для неполных бинарных деревьев применяют следующий способ представления. Бинарное дерево дополняется до полного дерева, вершины последовательно нумеруются. В массив заносятся только те вершины, которые были в исходном неполном дереве. При таком представлении элемент массива выделяется независимо от того, будет ли он содержать узел исходного дерева. Следовательно, необходимо отметить неиспользуемые элементы массива. Это можно сделать занесением специального значения в соответствующие элементы массива. В результате структура дерева переносится в одномерный массив. Адрес любой вершины в массиве вычисляется как

адрес = 2к-1+i-1,

где k-номер уровня вершины, i- номер на уровне k в полном бинарном дереве. Адрес корня будет равен единице. Для любой вершины можно вычислить адреса левого и правого потомков

адрес_L = 2к+2(i-1)

адрес_R = 2к+2(i-1)+1

Главным недостатком рассмотренного способа представления бинарного дерева является то, что структура данных является статической. Размер массива выбирается исходя из максимально возможного количества уровней бинарного дерева. Причем чем менее полным является дерево, тем менее рационально используется память.

4.3. Прохождение бинарных деревьев

В ряде алгоритмов обработки деревьев используется так называемое прохождение дерева. Под прохождением бинарного дерева понимают определенный порядок обхода всех вершин дерева. Различают несколько методов прохождения.

Прямой порядок прохождения бинарного дерева можно определить следующим образом

1. попасть в корень
2. пройти в прямом порядке левое поддерево
3. пройти в прямом порядке правое поддерево

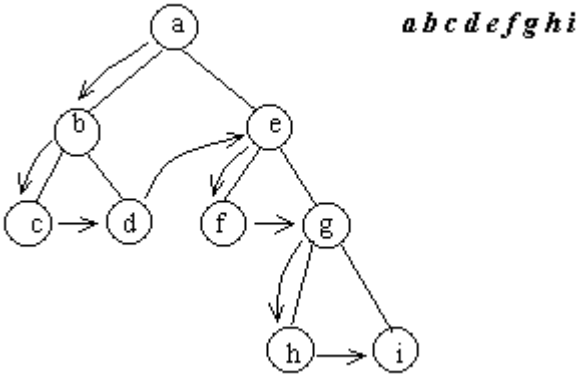


Рис.4.6. Прямой порядок прохождения бинарного дерева

Прохождение бинарного дерева в обратном порядке можно определить в аналогичной форме

1. пройти в обратном порядке левое поддерево
2. пройти в обратном порядке правое поддерево
3. попасть в корень

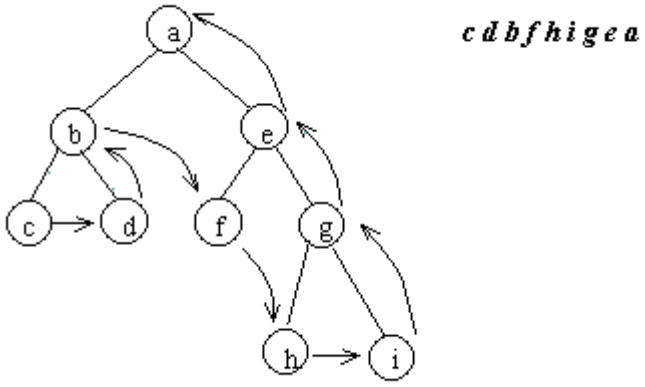


Рис.4.7. Обратный порядок прохождения бинарного дерева

Определим еще один порядок прохождения бинарного дерева, называемый симметричным.

1. пройти в симметричном порядке левое поддерево
2. попасть в корень
3. пройти в симметричном порядке правое поддерево

Порядок обхода бинарного дерева можно хранить непосредственно в структуре данных. Для этого достаточно ввести дополнительное поле указателя в элементе списковой структуры и хранить в нем указатель на вершину, следующую за данной вершиной при обходе дерева.

Представление деревьев в виде массивов также допускает хранение порядка прохождения дерева. Для этого вводится дополнительный массив, в который записываются адрес вершины в основном массиве, следующей за данной вершиной.

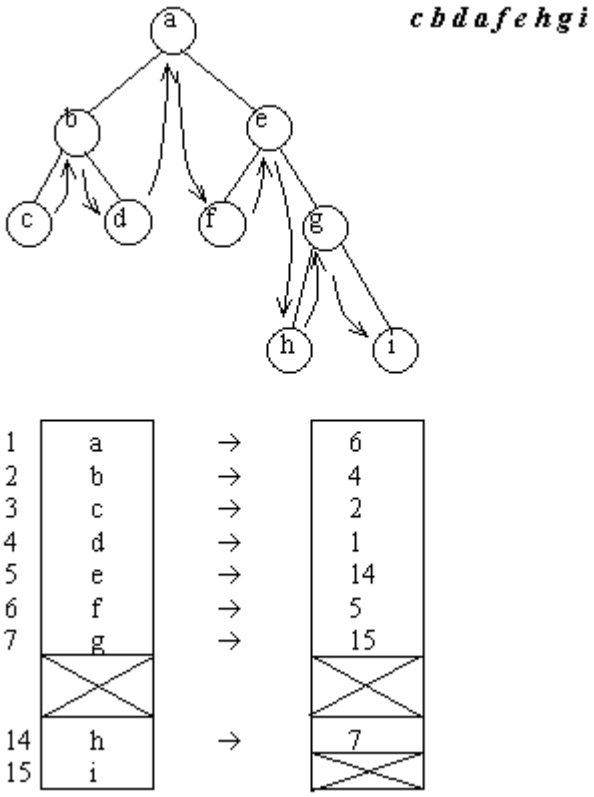


Рис.4.8. Представление симметрично прошитого бинарного дерева в виде массивов

Такие структуры данных получили название прошитых бинарных деревьев. Указатели или адреса, определяющие порядок обхода называют нитями. При этом в соответствии с порядком прохождения вершин

различают право прошитые, лево прошитые и симметрично прошитые бинарные деревья.

4.4. Алгоритмы на деревьях

4.4.1. Сортировка с прохождением бинарного дерева

В качестве примера использования прохождения бинарного дерева можно привести один из способов сортировки. Допустим, мы имеем некоторый массив и пытаемся упорядочить его элементы по возрастанию. Сама сортировка при этом распадается на две фазы

1. построение дерева
2. прохождение дерева

Дерево строится по следующим принципам. В качестве корня создается узел, в который записывается первый элемент массива. Для каждого очередного элемента создается новый лист. Если элемент меньше значения в текущем узле, то для него выбирается левое поддерево, если больше или равен - правое.

Рис.4.9. Сортировка по возрастанию с прохождением бинарного дерева

Для создания очередного узла происходят сравнения элемента со значениями существующих узлов, начиная с корня.

Во время второй фазы происходит прохождение дерева в симметричном порядке. Результатом сортировки является последовательность значений элементов, извлекаемых их пройденных узлов.

Для того чтобы сделать сортировку по убыванию, необходимо изменить только условия выбора поддерева при создании нового узла во время построения дерева.

4.4.2. Сортировка методом турнира с выбыванием

Приведем другой алгоритм сортировки, основанный на использовании бинарных деревьев. Данный метод получил название турнира с выбыванием. Пусть мы имеем исходный массив

10, 20, 3, 1, 5, 0, 4, 8

Сортировка начинается с создания листьев дерева. В качестве листьев бинарного дерева создаются узлы, в которых записаны значения элементов исходного массива.

Дерево строится от листьев к корню. Для двух соседних узлов строится общий предок, до тех пор, пока не будет создан корень. В узел-предок заносится значение, являющееся наименьшим из значений в узлах-потомках.

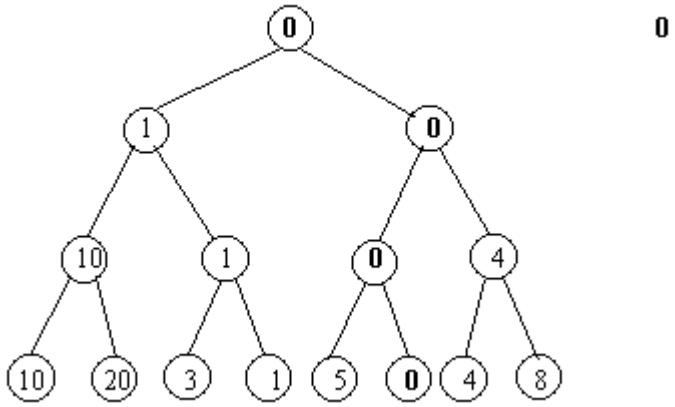
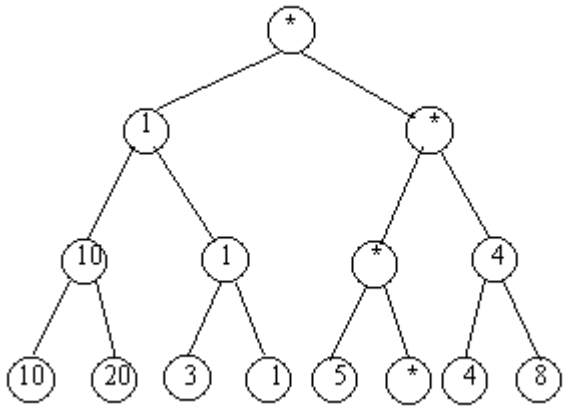


Рис.4.10. Построение дерева сортировки

В результате построения такого дерева наименьший элемент попадает сразу в корень. Далее начинается извлечение элементов из дерева. Извлекается значение из корня. Данное значение является первым элементом в результирующем массиве. Извлеченное значение помещается в отсортированный массив и заменяется в дереве на специальный символ.

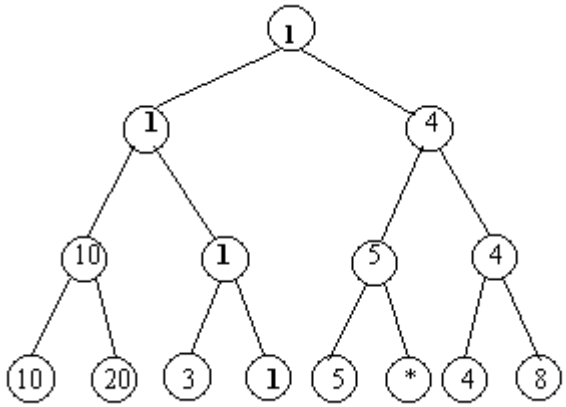
После этого происходит повторное занесение значений в родительские элементы от листьев к корню. При сравнениях специальный символ считается большим по отношению к любому другому значению.

После повторного заполнения из корня извлекается очередной элемент и итерация повторяется. Извлечения элементов продолжаются до тех пор, пока в дереве не останутся одни специальные символы.



0

Рис.4.11. Замена извлекаемого элемента на специальный символ



0, **1**

Рис.4.12. Повторное заполнение дерева сортировки

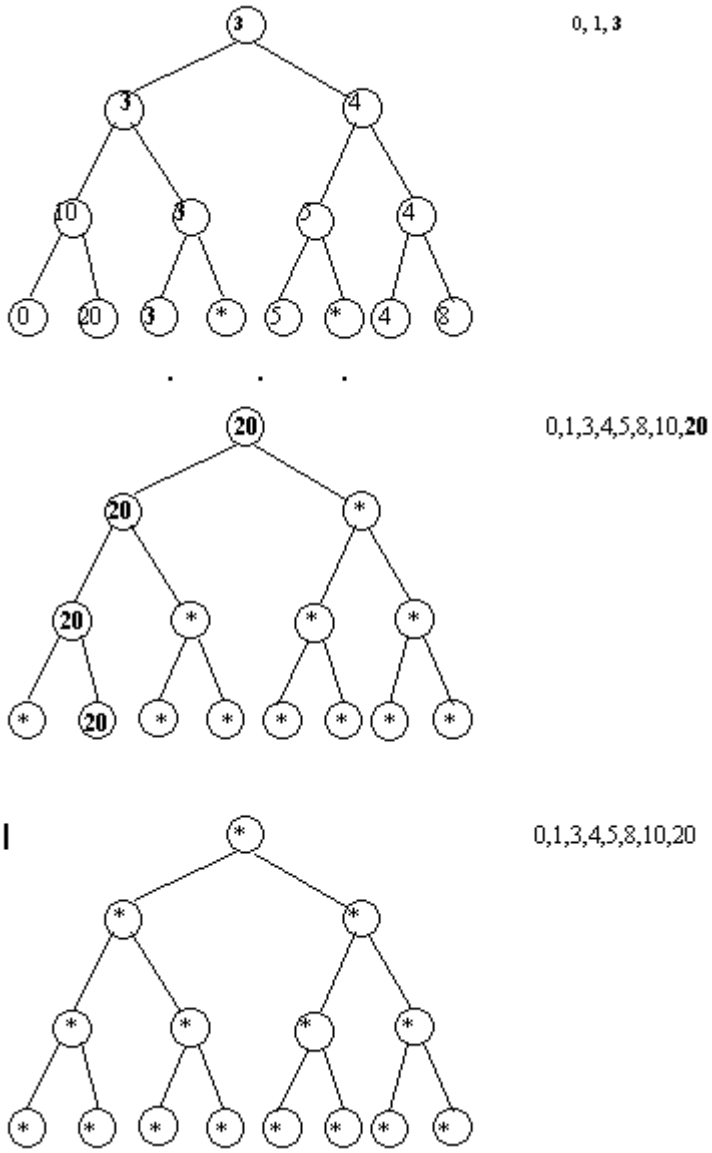


Рис.4.13. Извлечения элементов из дерева сортировки

В результате получим отсортированный массив

0, 1, 3, 4, 5, 8, 10, 20

4.4.3. Применение бинарных деревьев для сжатия информации

Рассмотрим применение деревьев для сжатия информации. Под сжатием мы будем понимать получение более компактного кода.

Рассмотрим следующий пример. Имеется текстовая строка S, состоящая из 10 символов

S = ABCCCDDDDD

При кодировании одного символа одним байтом для строки потребуется 10 байт.

Попробуем сократить требуемую память. Рассмотрим, какие символы действительно требуется кодировать. В данной строке используются всего 4 символа. Поэтому можно использовать укороченный код.

A 00

B 01

C 10

D 11

S = 00, 01, 10, 10, 10, 11, 11, 11, 11, 11 (20 бит)

В данном случае мы проанализировали текст на предмет использования символов. Можно заметить, что различные символы имеют различную частоту повторения. Существуют методы кодирования, позволяющие использовать этот факт для уменьшения длины кода.

Одним из таких методов является кодирование Хафмена. Он основан на использовании кодов различной длины для различных символов. Для максимально повторяющихся символов используют коды минимальной длины.

Построение кодовой таблицы происходит с использованием бинарного дерева. В корне дерева помещаются все символы и их суммарная частота повторения. Далее выбирается наиболее часто используемый символ и помещается со своей частотой повторения в левое поддерево. В правое поддерево помещаются оставшиеся символы с их суммарной частотой. Затем описанная операция проводится для всех вершин дерева, которые содержат более одного символа.

Само дерево может быть использовано в качестве кодовой таблицы для кодирования и декодирования текста. Кодирование осуществляется следующим образом. Для очередного символа в качестве кода используется путь от листа соответствующего символа к корню дерева. Причем каждому левому поддереву приписывается ноль, а каждому правому - единица.

Символ	частота	код
D	5	0
C	3	10
A	1	110
B	1	111

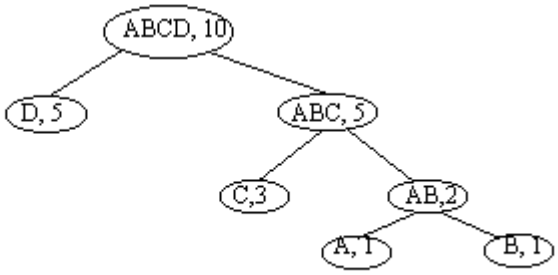


Рис. 4.14. Построение кодовой таблицы

Тогда для строки S будет получен следующий код

S=11011110101000000

Длина кода составляет 17 бит, что меньше по сравнению с укороченным кодом.

Теперь рассмотрим процесс декодирования. Алгоритм распаковки кода можно сформулировать следующим образом.

Распаковка

1. i:=0, j:=0;

2. если i>n, то ***стоп строка распакована***, иначе i:=i+1;

3. node:= root;

4. если b(i)=0, то node:=left(node), иначе node:=right(node)

5. если left(node)=0 и right(node)=0, то j:=j+1, s(j):= str(node), перейти к шагу 2, иначе i:=i+1, перейти к шагу 4

В алгоритме корень дерева обозначен как root, а Left(node) и right(node) обозначают левый и правый потомки узла node.

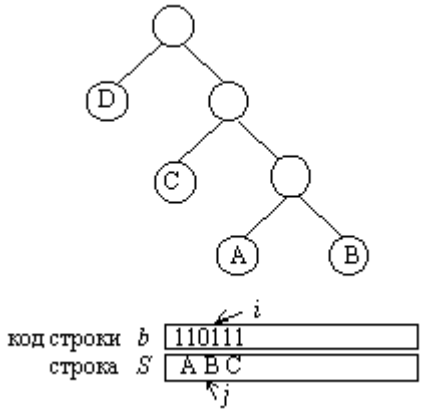


Рис. 4.15. Процесс распаковки кода

На практике такие способы упаковки используются не только для текстов, но и для произвольных двоичных данных. Дело в том, что любой файл можно рассматривать как последовательность байт. Тогда дерево кодирования можно построить не для символов, а для значений байт, встречающихся в кодируемом файле. Поскольку байт может принимать 256 значений, то соответствующее дерево будет иметь не более 256 листьев. В узлах дерева после его полного построения нет необходимости хранить несколько значений кодов и частоты повторения. Для кодирования и декодирования достаточно хранить только одно значение кода и только для листового узла. Поэтому такой способ представления кодовой таблицы является достаточно компактным.

Схемы кодирования подобного типа используются в программах архивации данных и сжатия растровых изображений в форматах графических файлов.

4.4.4. Представление выражений с помощью деревьев

С помощью деревьев можно представлять произвольные арифметические выражения. Каждому листу в таком дереве соответствует операнд, а каждому родительскому узлу - операция. В общем случае дерево при этом может оказаться не бинарным.

Однако если число операндов любой операции будет меньше или равно двум, то дерево будет бинарным. Причем если все операции будут иметь два операнда, то дерево окажется строго бинарным.

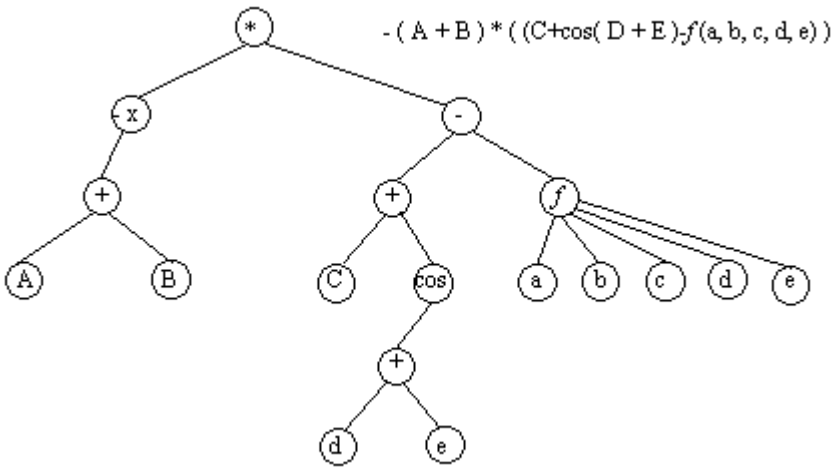


Рис.4.16. Представление арифметического выражения произвольного вида в виде дерева.

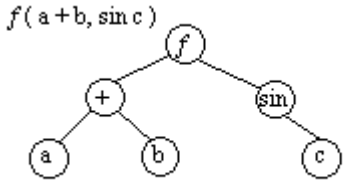


Рис. 4.17. Представление арифметического выражения в виде бинарного дерева

Бинарные деревья могут быть использованы не только для представления выражений, но и для их вычисления. Для того чтобы выражение можно было вычислить, в листьях записываются значения операндов.

Затем от листьев к корню производится выполнение операций. В процессе выполнения в узел операции записывается результат ее выполнения. В конце вычислений в корень будет записано значение, которое и будет являться результатом вычисления выражения.

Помимо арифметических выражений с помощью деревьев можно представлять выражения других типов. Примером являются логические выражения. Поскольку функции алгебры логики определены над двумя или одним операндом, то дерево для представления логического выражения будет бинарным

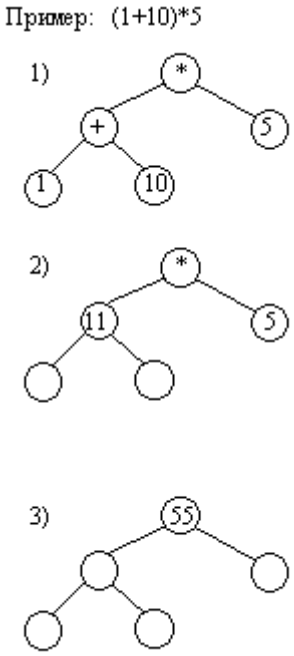


Рис.4.18. Вычисление арифметического выражения с помощью бинарного дерева

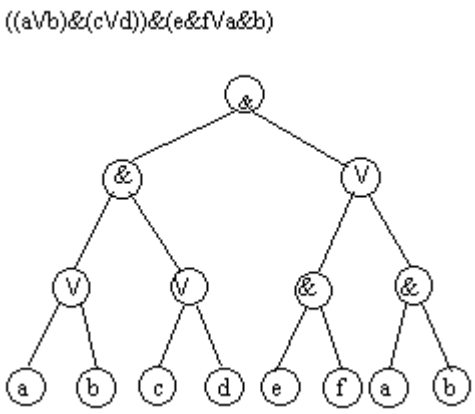


Рис. 4.19. Представление логического выражения в виде бинарного дерева

4.5. Представление сильноветвящихся деревьев

До сих пор мы рассматривали только способы представления бинарных деревьев. В ряде задач используются сильноветвящиеся деревья. Каждый элемент для представления бинарного дерева должен содержать как минимум три поля - значение или имя узла, указатель левого поддерева, указатель правого поддерева. Произвольные деревья могут быть бинарными или сильноветвящимися. Причем число потомков различных узлов не ограничено и заранее не известно.

Тем не менее, для представления таких деревьев достаточно иметь элементы, аналогичные элементам списковой структуры бинарного дерева. Элемент такой структуры содержит минимум три поля: значение узла, указатель на начало списка потомков узла, указатель на следующий элемент в списке потомков текущего уровня. Также как и для бинарного дерева необходимо хранить

указатель на корень дерева. При этом дерево представлено в виде структуры, связывающей списки потомков различных вершин. Такой способ представления вполне пригоден и для бинарных деревьев.

Представление деревьев с произвольной структурой в виде массивов может быть основано на матричных способах представления графов.

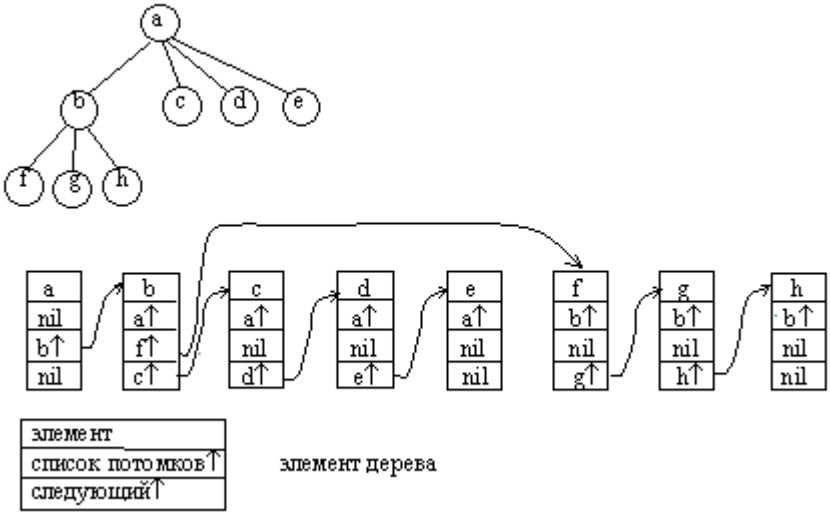


Рис.4.20. Представление сильноветвящихся деревьев в виде списков

4.6. Применение сильноветвящихся деревьев

Один из примеров применения сильноветвящихся деревьев был связан с представлением арифметических выражений произвольного вида. Рассмотрим использование таких деревьев для представления иерархической структуры каталогов файловой системы. Во многих файловых системах структура каталогов и файлов, как правило, представляет собой одно или несколько сильноветвящихся деревьев. В файловой системе MS Dos корень дерева соответствует логическому диску. Листья дерева соответствуют файлам и пустым каталогам, а узлы с ненулевой степенью - непустым каталогам.

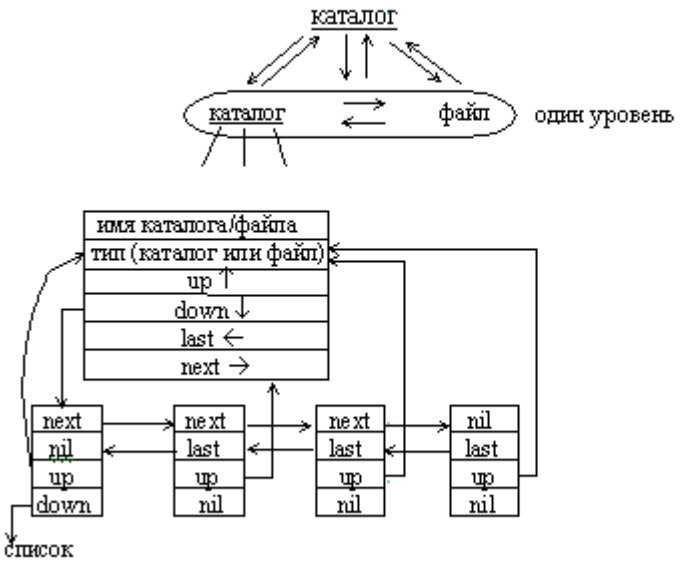


Рис.4.21. Представление логической структуры каталогов и файлов в виде сильноветвящегося дерева.

Для представления такой структуры используем расширение спискового представления сильноветвящихся деревьев. Способы представления деревьев, рассмотренные ранее, являются предельно экономичными, но не очень удобными для перемещения по дереву в разных направлениях. Именно такая задача встает при просмотре структуры каталогов. Необходимо осуществлять "навигацию" - перемещаться из текущего каталога в каталог верхнего или нижнего уровня, или от файла к файлу в пределах одного

каталога.

Для облегчения этой задачи сделаем списки потомков двунаправленными. Для этого достаточно ввести дополнительный указатель на предыдущий узел "last". С целью упрощения перемещения по дереву от листьев к корню введем дополнительный указатель на предок текущего узла "up". Общими с традиционными способами представления являются указатели на список потомков узла "down" и следующий узел "next".

Для представления оглавления диска служат поля имя и тип файла/каталога. Рассмотрим программу, которая осуществляет чтение структуры заданного каталога или диска, позволяет осуществлять навигацию и подсчет места занимаемого любым каталогом.

```
program dir_tree;

uses dos;

type node=record

        name: string[50]; {Имя каталога/файла}

        size: longint; {Размер файла (байт) }

        node_type: char; {Тип узла (файл -'f' / каталог-'c')}

        up,down: pointer; {Указатели на предка и список потомков}

l       last,next: pointer; {Указатели на соседние узлы}

    end;

var

   n,i,l:integer;

   root, current_root: pointer;

   pnt, current:^node;

    s : searchrec;

   str: string;

procedure create_tree(local_root:pointer);

{Отображение физического оглавления диска в логическую структуру}

var

     local_node, local_r_node, local_last : ^node;

procedure new_node;

{Создание нового узла в дереве каталогов и файлов}

begin

new(local_node);

local_node^.last:=local_last;

if not(local_last=nil) then local_last^.next:=local_node;

local_node^.next:=nil;

local_node^.down:=nil;

local_node^.up:=local_r_node;
```

```
if local_r_node^.down =nil then local_r_node^.down:=local_node;
local_node^.name:=local_r_node^.name+'\'+s.name;
if s.attr and Directory = 0 then local_node^.node_type:='f'
else local_node^.node_type:='c';
local_node^.size:=s.size;
local_last:=local_node;
end;
begin {Собственно процедура}
local_r_node:=local_root;
local_last:=nil;
findfirst(local_r_node^.name+'\*.*',anyfile,s);
if doserror = 0 then
    begin
    if (s.name<>'.') and (s.name<>'..') and (s.attr and VolumeID = 0)
    then new_node;
    while doserror=0 do begin
        findnext(s);
        if (doserror = 0) and (s.name<>'.') and (s.name<>'..') and (s.attr and VolumeID = 0)
        then new_node;
    end
    end;
    if not (local_r_node^.down=nil) then
    begin
    local_node:=local_r_node^.down;
    repeat
        if local_node^.node_type='c' then create_tree(local_node);{Рекурсия}
        local_node:=local_node^.next
    until local_node=nil
end
end;
procedure current_list;
{Вывод оглавления текущего каталога}
begin
```

```
current:=current_root;
writeln('текущий каталог - ', current^.name);
if current^.node_type='c'then
begin
pnt:=current^.down;
i:=1;
repeat {Проходим каталог в дереве}
     writeln (i:4,'-',pnt^.name);
     pnt:=pnt^.next;
     i:=i+1
until pnt=nil
end;
end;
procedure down;
{Навигация в дереве каталогов. Перемещение на один уровень вниз}
begin
current:=current_root;
if not (current^.down=nil) then
     begin
     current:= current^.down;
     writeln('номер в оглавлении'); readln; read(l);
     i:=1;
    while (i<l) and not (current^.next=nil) do
    begin
         current:=current^.next;
         i:=i+1
   end;
   if (current^.node_type='c') and not (current^.down=nil)
   then current_root:= current;
   end;
end;
procedure up;
{Навигация в дереве каталогов. Перемещение на один уровень вверх}
```

```
begin
current:=current_root;
if not (current^.up=nil) then current_root:=current^.up;
end;
procedure count;
{Расчет числа файлов и подкаталогов иерархической структуры каталога}
var
    n_files, n_cats :integer;
procedure count_in (local_root : pointer);
var
    local_node, local_r_node: ^node;
begin
local_r_node:=local_root;
if not (local_r_node^.down=nil) then
    begin
    local_node:=local_r_node^.down;
    repeat
        if local_node^.node_type='f' then
            n_files:=n_files+1
        else
        begin
            n_cats:=n_cats+1;
            count_in (local_node)
        end;
        local_node:=local_node^.next
    until local_node=nil
    end
end;
begin {Собственно процедура}
n_files:=0; n_cats:=0;
count_in (current_root);
writeln ('файлы : ',n_files, ' каталоги: ', n_cats);
end;
```

```
procedure count_mem;
{Расчет физического объема иерархической структуры каталога}
var
    mem :longint;
procedure count_m_in (local_root : pointer);
var
    local_node, local_r_node: ^node;
begin
local_r_node:=local_root;
if not (local_r_node^.down=nil) then
     begin
      local_node:=local_r_node^.down;
      repeat
           if  local_node^.node_type='f' then
                mem:=mem+local_node^.size
           else
                count_m_in (local_node);
     local_node:=local_node^.next
     until local_node=nil
     end
end;
begin  {Собственно процедура}
mem:=0;
count_m_in (current_root);
writeln ('mem ', mem, ' bytes');
end;
{----------основная программа----------}
begin
new(current);
{Инициализация корня дерева каталогов и указателей для навигации}
root:=current; current_root:=current;
writeln('каталог ?'); read(str); writeln(str);
current^.name:=str;
```

```
current^.last:=nil; current^.next:=nil;

current^.up:=nil; current^.down:=nil;

current^.node_type:='c';

{Создание дерева каталогов}

create_tree(current);

if current^.down=nil then current^.node_type:=' ';

repeat

{ Интерактивная навигация }

      writeln ('1-список');

      writeln('2-вниз');

      writeln('3-вверх');

      writeln('4-число файлов');

      writeln('5-объем');

      readln(n);

      if n=1 then current_list;

      if n=2 then down;

      if n=3 then up;

      if n=4 then count;

      if n=5 then count_mem;

until n=0

end.
```

Для чтения оглавления диска в данной программе используются стандартные процедуры findfirst и findnext, которые возвращают сведения о первом и последующих элементах в оглавлении текущего каталога.

В процессе чтения корневого каталога строится первый уровень потомков в списковой структуре дерева. Далее процедура построения поддерева вызывается для каждого узла в корневом каталоге. Затем процесс повторяется для всех непустых какталогов до тех пор, пока не будет построена полная структура оглавления.

Все операции по просмотру содержимого каталогов и подсчету занимаемого объема производятся не с физическими каталогами и файлами, а с созданным динамическим списковым представлением их логической структуры в виде сильноветвящегося дерева.

В данном примере программы для каждого файла или каталога хранится его полное имя в MS-DOS, которое включает имя диска и путь. Если программу несколько усложнить, то можно добиться более эффективного использования динамической памяти. Для этого потребуется хранить в узле дерева только имя каталога или файла, а полое имя - вычислять при помощи цепочки имен каталогов до корневого узла.

4.7. Представление графов

Граф можно представить в виде списочной структуры, состоящей из списков двух типов - списка вершин и списков ребер. Элемент списка вершин содержит поле данных и два указателя. Один указатель связывает данный элемент с элементом другой вершины. Другой указатель связывает элемент списка вершин со списком ребер, связанных с данной вершиной. Для ориентированного графа используют список дуг, исходящих из вершины. Элемент списка дуг состоит только из двух указателей. Первый указатель используется для того, чтобы показать в какую вершину дуга входит, а второй - для связи элементов в списке дуг вершины.

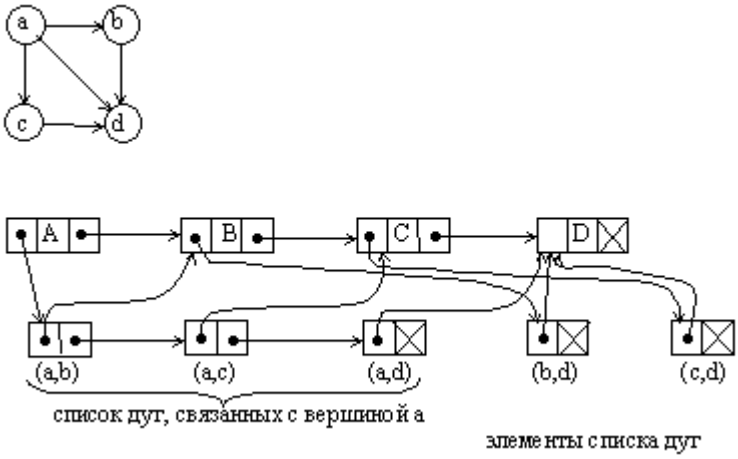
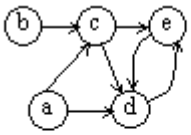


Рис. 4.22. Представление графа в виде списочной структуры

Очень распространенным является матричный способ представления графов. Для представления ненаправленных графов обычно используют матрицы смежности, а для ориентированных графов - матрицы инцидентности. Обе матрицы имеют размерность n*n, где n-число вершин в графе. Вершины графа последовательно нумеруются.

Матрица смежности имеет значение ноль в позиции m (i,j), если не существует ребра, связывающего вершину i с вершиной j, или имеет единичное значение в позиции m (i,j), если такое ребро существует.

Правила построения матрицы инцидентности аналогичны правилам построения матрицы инцидентности. Разница состоит в том, что единица в позиции m (i,j) означает выход дуги из вершины i и вход дуги в вершину j.



Матрица смежности

	a	b	c	d	e
a	0	0	1	1	0
b	0	0	1	0	0
c	0	1	0	1	1
d	1	0	1	0	1
e	1	0	1	1	0

Матрица инцидентности

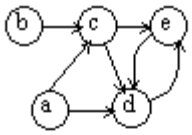
	a	b	c	d	e
a	0	0	1	1	0
b	0	0	1	0	0
c	0	0	0	1	1
d	0	0	0	0	1
e	0	0	0	1	0

Рис.4.23. Матричное представление графа

Поскольку матрица смежности симметрична относительно главной диагонали, то можно хранить и обрабатывать только часть матрицы. Можно заметить, что для хранения одного элемента матрицы достаточно выделить всего один бит.

4.8. Алгоритмы на графах

В некоторых матричных алгоритмах обработки графов используются так называемые матрицы путей. Определим понятие пути в ориентированном графе. Под путем длиной k из вершины i в вершину j мы будем понимать возможность попасть из вершины i в вершину j за k переходов, каждому из которых соответствует одна дуга. Одна матрица путей m_k содержит сведения о наличии всех путей одной длины k в графе. Единичное значение в позиции (i,j) означает наличие пути длины k из вершины i в вершину j.



m_1

	a	b	c	d	e
a	0	0	1	1	0
b	0	0	1	0	0
c	0	0	0	1	1
d	0	0	0	0	1
e	0	0	0	1	0

m_2

	a	b	c	d	e
a	0	0	0	1	1
b	0	0	0	1	1
c	0	0	0	1	1
d	0	0	0	1	0
e	0	0	0	0	1

m_3

	a	b	c	d	e
a	0	0	0	1	1
b	0	0	0	1	1
c	0	0	0	1	1
d	0	0	0	0	1
e	0	0	0	1	0

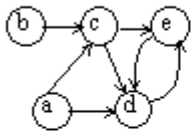
Рис.4.24. Матрицы путей

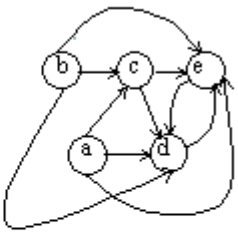
Матрица m_1 полностью совпадает с матрицей инцидентности. По матрице m_1 можно построить m_2. По матрице m_2 можно построить m_3 и т. д. Если граф является ациклическим, то число мариц путей ограничено. В противном случае матрицы будут повторяться до бесконечности с некоторым периодом, связанным с длиной циклов. Матрицы путей не имеет смысла вычислять до бесконечности. Достаточно остановиться в случае повторения матриц.

Если выполнить логическое сложение всех матриц путей, то получится транзитивное замыкание графа.

$M_{tr} = m_1 \text{ OR } m_2 \text{ OR } m_3$

В результате матрица будет содержать все возможные пути в графе.





m_{tr}

	a	b	c	d	e
a	0	0	1	1	1
b	0	0	1	1	1
c	0	0	0	1	1
d	0	0	0	1	1
e	0	0	0	1	1

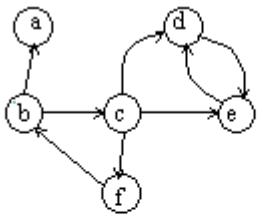
Рис. 4.25. Транзитивное замыкание в графе

Наличие циклов в графе можно определить с помощью эффективного алгоритма. Алгоритм может быть реализован как для матричного, так и для спискового способа представления графа.

Принцип выделения циклов следующий. Если вершина имеет только входные или только выходные дуги, то она явно не входит ни в один цикл. Можно удалить все такие вершины из графа вместе со связанными с ними дугами. В результате появятся новые вершины, имеющие только входные или выходные дуги. Они также удаляются. Итерации повторяются до тех пор, пока граф не перестанет изменяться. Отсутствие изменений свидетельствует об отсутствии циклов, если все вершины были удалены. Все оставшиеся вершины обязательно принадлежат циклам.

Сформулируем алгоритм в матричном виде

1. Для **i** от **1** до **n** выполнить шаги 1-2
2. Если строка **M(*i*,*) = 0**, то обнулить столбец ***i***
3. Если столбец **M(*, i) = 0**, то обнулить строку ***i***
4. Если матрица изменилась, то выполнить шаг 1
5. Если матрица нулевая, то ***стоп, граф ациклический***, иначе ***матрица содержит вершины, входящие в циклы***



начало	a	b	c	d	e	f
итерация 1	b	c	d	e	f	
итерация 2		c	d	e	f	
итерация 3		c	d	e		
итерация 4			d	e		
итерация 5			d	e		

итерация 1

	a	b	**c**	d	e	f	
a	0	0	0	0	0	0	< обнуляемая строка
b	1	0	0	0	0	0	
c	0	0	0	1	1	1	< обнуляемая строка
d	0	0	0	0	1	0	
e	0	0	0	1	0	0	
f	0	1	0	0	0	0	

Рис. 4.26. Поиск циклов в графе

Достоинством данного алгоритма является то, что происходит одновременное определение цикличности или ацикличности графа и формирование списка вершин, входящих в циклы. В матричной реализации после завершения алгоритма остается матрица инцидентности, соответствующая подграфу, содержащему все циклы исходного графа.

ЛИТЕРАТУРА

- Вирт Н. **Алгоритмы и структуры данных**. - М: Мир, 1989 -360с.
- Сибуя М., Ямамото Т. **Алгоритмы обработки данных**. - М: Мир, 1986 -218с.
- Лэгсам Й, Огенстайн М. **Структуры данных для персональных ЭВМ** - М: Мир, 1989 -586с.
- **Турбо Паскаль 7.0** - Киев: BHV, 1998 - 448с.