МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

СЕВЕРО-КАВКАЗСКАЯ ГОСУДАРСТВЕННАЯ АКАДЕМИЯ

КАФЕДРА «ПРИКЛАДНАЯ ИНФОРМАТИКА»

М.У. Эркенова

АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ

Учебно-методическое пособие для
обучающихся II курса по специальности
09.03.04 «Программная инженерия»

Черкесск, 2025

УДК 004.6
ББК 32.973.05
Э 78

Рассмотрено на заседании кафедры «Прикладная информатика»
Протокол № 3 от «16» октября 2024 г.
Рекомендовано к изданию редакционно-издательским советом СКГА.
Протокол № 27 от «07» ноября 2024 г.

Рецензенты: Кочкарова П.А. – к.ф.-м.н., доцент кафедры ПИ

Э 78 **Эркенова, М.У.** Алгоритмы и структуры данных: учебно-методическое пособие обучающихся II курса по специальности 09.03.04 «Алгоритмы и структуры данных» / М.У. Эркенова. – Черкесск: БИЦ СКГА, 2025. –136 с.

УДК 004.6
ББК 32.973.05

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Изучение дисциплины «Алгоритмы и структуры данных» является одним из основных моментов в процессе подготовки специалистов по разработке программного обеспечения для компьютерных систем. Это связано с тем, что первичная задача программиста заключается в применении решения о форме представления данных и выборе алгоритмов, применяемых к этим данным. И лишь затем выбранная структура программы и данных реализуется на конкретном языке программирования. В связи с этим знание классических методов и приемов обработки данных позволяет избежать ошибок, которые могут возникать при чисто интуитивной разработки программ.

Данное пособие содержит необходимый теоретический материал по четырем основным разделам курса «Алгоритмы и структуры данных»: методы сортировки данных, древовидные структуры данных, хеширование и кодирование информации. Все рассмотренные методы проиллюстрированы наглядными примерами. Также для каждого метода приведен конкретный алгоритм, позволяющий легко программировать данный метод. Кроме того, в каждой главе даны варианты заданий, выполнив которые, можно окончательно уяснить все особенности изучаемых методов.

ТЕОРЕТИЧЕСКИЙ КУРС

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.

Существует 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.

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;
```

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;
```

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, так же, как и массивы, объединяют однотипные элементы. Поэтому в описании множества обязательно должен быть указан тип его элементов:

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

Здесь приведено 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.

Динамические структуры данных

Введены базовые структуры данных: массивы, записи, множества. Названы 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.

Линейные списки

Наиболее простой способ связать некоторое множество элементов – 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.).

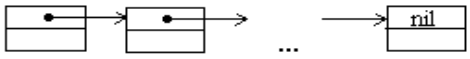


Рисунок 1.1– Линейный список (связанный список)

В языке Turbo Pascal предусмотрены два типа указателей – типизированные и не типизированные. 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 создать элементы и объединить их 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 занимают 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 Turbo Pascal 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.).

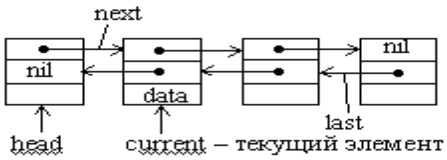


Рисунок 1.2 – Двунаправленный список

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.

Циклические списки

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

Циклические списки, как и линейные, бывают однонаправленными (рис. 1.3.) и двунаправленными (рис.1.4). Основное отличие циклического списка состоит в том, что в списке нет пустых указателей.

Рисунок 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. 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.

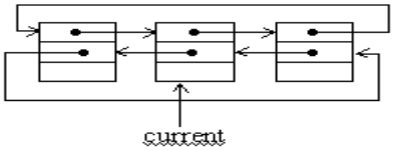


Рисунок 1.4 – Двунаправленный циклический список

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.

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.

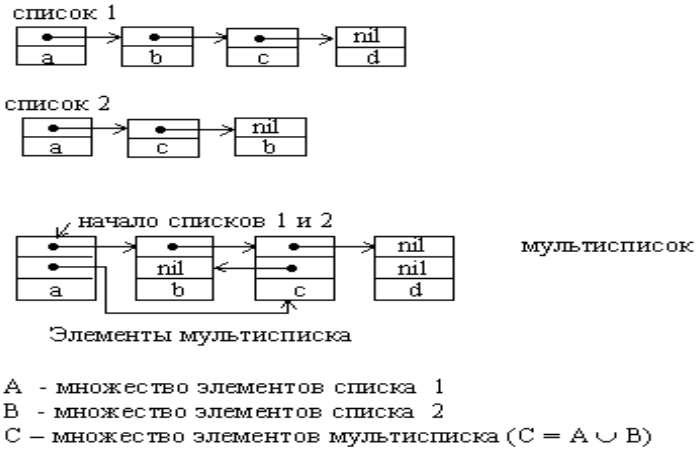


Рисунок 1.5 – Объединение двух линейных списков в один мультисписок

Мультисписок состоит из элементов, содержащих такое число указателей, которое позволяет организовать их одновременно в виде нескольких различных списков (рис.1.5.). За счет отсутствия дублирования данных память используется более рационально.

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.

Представление стека и очередей в виде списков

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 (рис. 1.6.). 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.

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;

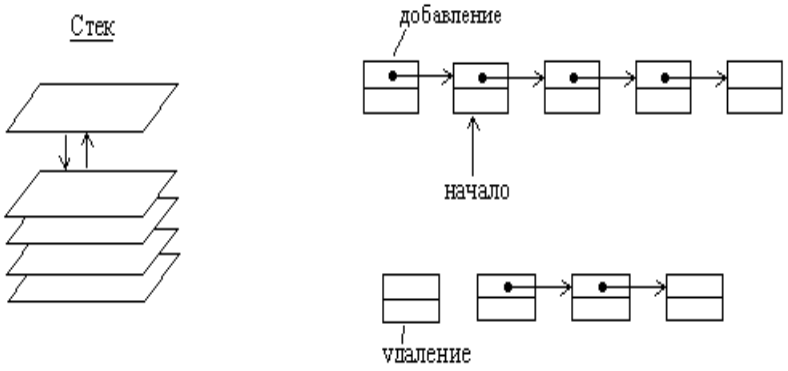


Рисунок 1.6 – Организация стека на основе линейного списка.

```
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.

Очередь

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 (рис. 1.7.). При этом первый и последний элементы списка соответствуют входам (выходам) дека.

Простая очередь может быть организована на основе двусвязанного линейного списка (рис. 1.8.). 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.

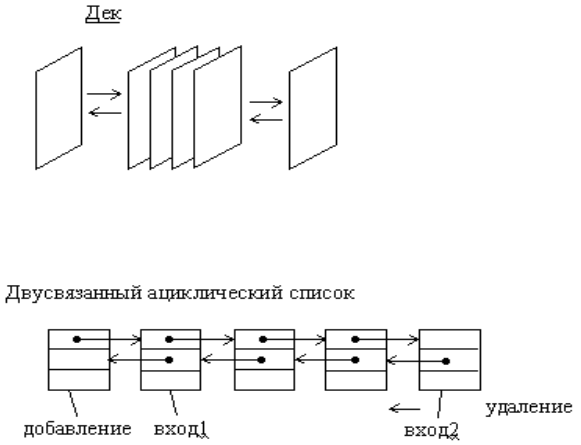


Рисунок 1.7 – Организация дека на основе двусвязанного линейного списка

Очередь с ограниченным входом или с ограниченным выходом также, как дек или очередь, можно организовать на основе линейного двунаправленного списка (рис.1.9).

Разновидностями очереди являются очередь с ограниченным входом и очередь с ограниченным выходом (рис.1.10). Они занимают промежуточное положение между деком и простой очередью.

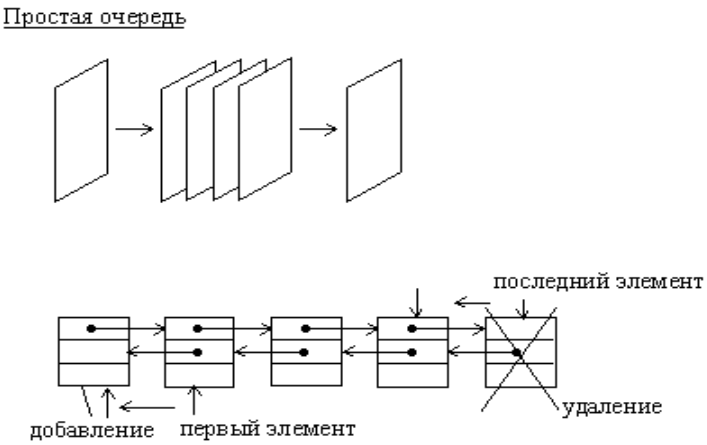


Рисунок 1.8 – Организация дека на основе двусвязанного линейного списка

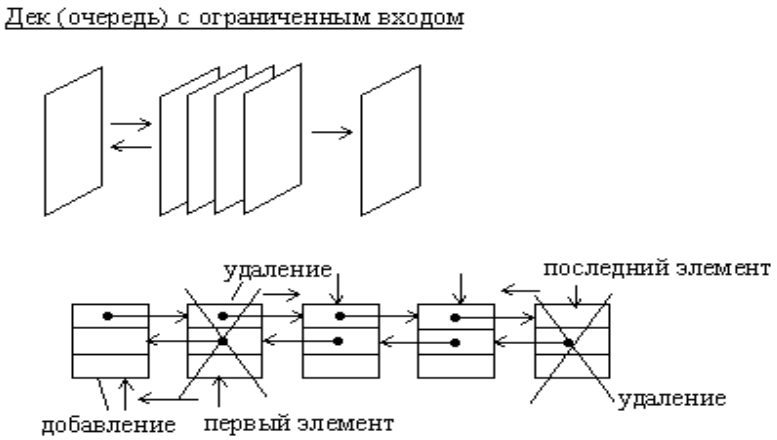


Рисунок 1.9 – Организация дека с ограниченным входом на основе двусвязанного линейного списка

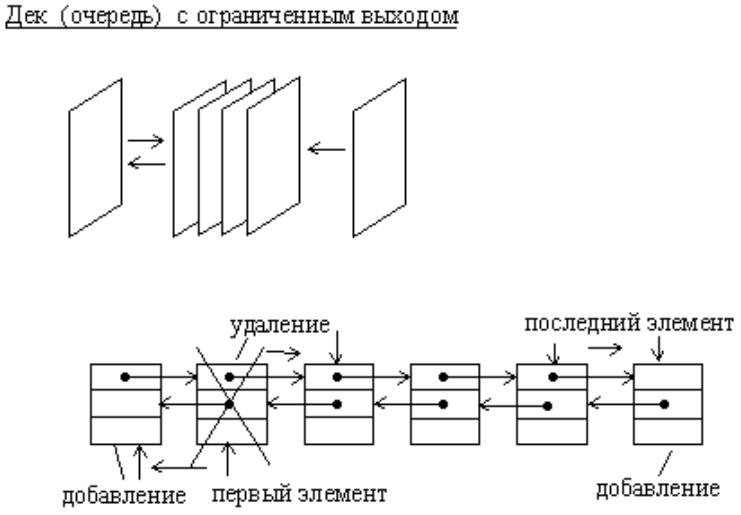


Рисунок 1.10 – Организация дека с ограниченным выходом на основе двусвязанного линейного списка

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 “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 будем считать, что тип Item включает только ключ.

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 обязательно. 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.

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”) (рис.2.1), основанный на знании того, что массив A упорядочен, т. е. удовлетворяет условию:

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.

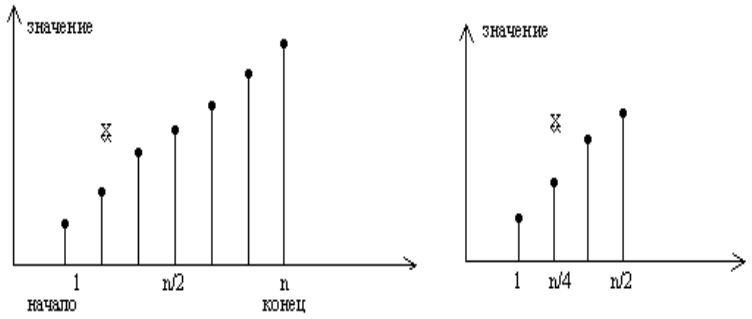


Рисунок 2.1 – Поиск делением пополам

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.

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, меньше 30. В этом случае можно использовать линейный поиск.

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}
```

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. В худшем случае производительность будет внушать опасение.

3. Задачи сортировки в структурах данных

Здесь будем обсуждать методы сортировки информации. В общей постановке задача ставится следующим образом. Имеется последовательность однотипных записей, одно из полей которых выбрано в качестве ключевого (далее будем называть его ключом сортировки). Тип данных ключа должен содержать операции сравнения (=, >, <, >= и <=). Задачей сортировки является преобразование исходной последовательности в последовательность, содержащую те же записи, но в порядке возрастания (или убывания) значений ключа. Метод сортировки называется устойчивым, если при его применении не изменяется относительное положение записей с равными значениями ключа.

Различают сортировку массивов записей, целиком расположенных в основной памяти (внутреннюю сортировку), и сортировку файлов, хранящихся во внешней памяти и не помещающихся полностью в основной памяти (внешнюю сортировку). Для внутренней и внешней сортировки требуются существенно разные методы. В этой части рассмотрим наиболее известные методы внутренней сортировки, начиная с простых и понятных, но не слишком быстрых, и заканчивая не столь просто понимаемыми усложненными методами.

Естественным условием, предъявляемым к любому методу внутренней сортировки, является то, что эти методы не должны требовать дополнительной памяти: все перестановки с целью упорядочения элементов массива должны производиться в пределах того же массива. Мерой эффективности алгоритма внутренней сортировки является число требуемых сравнений значений ключа (C) и число перестановок элементов (M).

Заметим, что поскольку сортировка основана только на значениях ключа и никак не затрагивает оставшиеся поля записей, можно говорить о сортировке массивов ключей. В следующих разделах, чтобы не привязываться к конкретному языку программирования и его синтаксическим особенностям, будем описывать алгоритмы словами и иллюстрировать их на простых примерах.

Сортировка включением

Одним из наиболее простых и естественных методов внутренней сортировки является сортировка с простыми включениями. Идея алгоритма очень проста. Пусть имеется массив ключей a[1], a[2], ..., a[n]. Для каждого элемента массива, начиная со второго, производится сравнение с элементами с меньшим индексом (элемент a[i] последовательно сравнивается с элементами a[i-1], a[i-2] ...) и до тех пор, пока для очередного элемента a[j] выполняется соотношение a[j] > a[i], a[i] и a[j] меняются местами. Если удается встретить такой элемент a[j], что a[j] <= a[i], или если достигнута нижняя граница массива, производится переход к обработке элемента a[i+1] (пока не будет достигнута верхняя граница массива).

Легко видеть, что в лучшем случае (когда массив уже упорядочен) для выполнения алгоритма с массивом из n элементов потребуется n-1 сравнение и 0 пересылок. В худшем случае (когда массив упорядочен в обратном порядке) потребуется n(n-1)/2 сравнений и столько же пересылок. Таким образом, можно оценивать сложность метода простых включений как n2.

Можно сократить число сравнений, применяемых в методе простых включений, если воспользоваться тем фактом, что при обработке элемента a[i] массива элементы a[1], a[2], ..., a[i-1] уже упорядочены, и воспользоваться для поиска элемента, с которым должна быть произведена перестановка, методом двоичного деления. В этом случае оценка числа требуемых сравнений становится nlog(n). Заметим, что поскольку при выполнении перестановки требуется сдвижка на один элемент нескольких элементов, то оценка числа пересылок остается n2.

Пример сортировки методом простого включения

Начальное состояние массива : 8 23 5 65 44 33 1 6
Шаг 1: 8 23 5 65 44 33 1 6
Шаг 2:
8 5 23 65 44 33 1 6
5 8 23 65 44 33 1 6
Шаг 3: 5 8 23 65 44 33 1 6
Шаг 4: 5 8 23 44 65 33 1 6
Шаг 5:
5 8 23 44 33 65 1 6
5 8 23 33 44 65 1 6
Шаг 6:
5 8 23 33 44 1 65 6
5 8 23 33 1 44 65 6
5 8 23 1 33 44 65 6
5 8 1 23 33 44 65 6
5 1 8 23 33 44 65 6
1 5 8 23 33 44 65 6
Шаг 7:

1 5 8 23 33 44 6 65
1 5 8 23 33 6 44 65
1 5 8 23 6 33 44 65
1 5 8 6 23 33 44 65
1 5 6 8 23 33 44 65

Дальнейшим развитием метода сортировки с включениями является сортировка методом Шелла, называемая по-другому сортировкой включениями с уменьшающимся расстоянием. Мы не будем описывать алгоритм в общем виде, а ограничимся случаем, когда число элементов в сортируемом массиве является степенью числа 2. Для массива с 2n элементами алгоритм работает следующим образом. В первой фазе производится сортировка включением всех пар элементов массива, расстояние между которыми есть 2(n-1). Во второй фазе производится сортировка включением элементов полученного массива, расстояние между которыми есть 2(n-2). И так далее, пока мы не дойдем до фазы с расстоянием между элементами, равным единице, и не выполним завершающую сортировку с включениями. Применение метода Шелла к массиву, используемому в наших примерах, показано ниже.

Пример сортировки методом Шелла
Начальное состояние массива :
8 23 5 65 44 33 1 6
Фаза 1 (сортируются элементы, расстояние между которыми четыре):
8 23 5 65 44 33 1 6
8 23 5 65 44 33 1 6
8 23 1 65 44 33 5 6
8 23 1 6 44 33 5 65
Фаза 2 (сортируются элементы, расстояние между которыми два):
1 23 8 6 44 33 5 65
1 23 8 6 44 33 5 65
1 23 8 6 5 33 44 65
1 23 5 6 8 33 44 65
1 6 5 23 8 33 44 65
1 6 5 23 8 33 44 65
1 6 5 23 8 33 44 65
Фаза 3 (сортируются элементы, расстояние между которыми один):
1 6 5 23 8 33 44 65
1 5 6 23 8 33 44 65
1 5 6 23 8 33 44 65
1 5 6 8 23 33 44 65
1 5 6 8 23 33 44 65
1 5 6 8 23 33 44 65
1 5 6 8 23 33 44 65

В общем случае алгоритм Шелла естественно переформулируется для заданной последовательности из t расстояний между элементами h1, h2, ..., ht, для которых выполняются условия h1 = 1 и h(i+1) < hi. Дональд Кнут показал, что при правильно подобранных t и h сложность алгоритма Шелла является 1,2n, что существенно меньше сложности простых алгоритмов сортировки.

Обменная сортировка

Простая обменная сортировка (в просторечии называемая "методом пузырька") для массива a[1], a[2], ..., a[n] работает следующим образом. Начиная с конца массива сравниваются два соседних элемента (a[n] и a[n-1]). Если выполняется условие a[n-1] > a[n], то значения элементов меняются местами. Процесс продолжается для a[n-1] и a[n-2] и т.д., пока не будет произведено сравнение a[2] и a[1]. Понятно, что после этого на месте a[1] окажется элемент массива с наименьшим значением. На втором шаге процесс повторяется, но последними сравниваются a[3] и a[2]. На последнем шаге будут сравниваться только текущие значения a[n] и a[n-1]. Понятна аналогия с пузырьком, поскольку наименьшие элементы (самые "легкие") постепенно "всплывают" к верхней границе массива.

Пример сортировки методом пузырька

Начальное состояние массива :

8 23 5 65 44 33 1 6

Шаг 1:

8 23 5 65 44 33 1 6
8 23 5 65 44 1 33 6
8 23 5 65 1 44 33 6
8 23 5 1 65 44 33 6
8 23 1 5 65 44 33 6
8 1 23 5 65 44 33 6
1 8 23 5 65 44 33 6

Шаг 2:

1 8 23 5 65 44 6 33
1 8 23 5 65 6 44 33
1 8 23 5 6 65 44 33
1 8 23 5 6 65 44 33
1 8 5 23 6 65 44 33
1 5 8 23 6 65 44 33

Шаг 3:

1 5 8 23 6 65 33 44
1 5 8 23 6 33 65 44
1 5 8 23 6 33 65 44
1 5 8 6 23 33 65 44
1 5 6 8 23 33 65 44

Шаг 4:

1 5 6 8 23 33 44 65
1 5 6 8 23 33 44 65
1 5 6 8 23 33 44 65
1 5 6 8 23 33 44 65
Шаг 5:
1 5 6 8 23 33 44 65
1 5 6 8 23 33 44 65
1 5 6 8 23 33 44 65
Шаг 6:
1 5 6 8 23 33 44 65
1 5 6 8 23 33 44 65
Шаг 7:
1 5 6 8 23 33 44 65

Для метода простой обменной сортировки требуется число сравнений n(n-1)/2, минимальное число пересылок 0, а среднее и максимальное число пересылок n2.

Метод пузырька допускает три простых усовершенствования. Во-первых, на четырех последних шагах расположение значений элементов не менялось (массив оказался уже упорядоченным). Поэтому если на некотором шаге не было произведено ни одного обмена, то выполнение алгоритма можно прекращать. Во-вторых, можно запоминать наименьшее значение индекса массива, для которого на текущем шаге выполнялись перестановки. Очевидно, что верхняя часть массива до элемента с этим индексом уже отсортирована, и на следующем шаге можно прекращать сравнения значений соседних элементов при достижении такого значения индекса. В-третьих, метод пузырька работает неравноправно для «легких» и «тяжелых» значений. Легкое значение попадает на нужное место за один шаг, а тяжелое на каждом шаге опускается по направлению к нужному месту на одну позицию. На этих наблюдениях основан метод шейкерной сортировки (ShakerSort). При его применении на каждом следующем шаге меняется направление последовательного просмотра. В результате на одном шаге "всплывает" очередной наиболее легкий элемент, а на другом "тонет" очередной самый тяжелый. Пример шейкерной сортировки приведен ниже.

Пример шейкерной сортировки

Начальное состояние массива :
8 23 5 65 44 33 1 6
Шаг 1:
8 23 5 65 44 33 1 6
8 23 5 65 44 1 33 6
8 23 5 65 1 44 33 6
8 23 5 1 65 44 33 6
8 23 1 5 65 44 33 6
8 1 23 5 65 44 33 6
1 8 23 5 65 44 33 6

Шаг 2:
1 8 23 5 65 44 33 6
1 8 5 23 65 44 33 6
1 8 5 23 65 44 33 6
1 8 5 23 44 65 33 6
1 8 5 23 44 33 65 6
1 8 5 23 44 33 6 65
Шаг 3:
1 8 5 23 44 6 33 65
1 8 5 23 6 44 33 65
1 8 5 6 23 44 33 65
1 8 5 6 23 44 33 65
1 5 8 6 23 44 33 65
Шаг 4:
1 5 6 8 23 44 33 65
1 5 6 8 23 44 33 65
1 5 6 8 23 44 33 65
1 5 6 8 23 33 44 65
Шаг 5:
1 5 6 8 23 33 44 65
1 5 6 8 23 33 44 65
1 5 6 8 23 33 44 65

Шейкерная сортировка позволяет сократить число сравнений (по оценке Кнута средним числом сравнений является (n2 - n (const + ln(n))), хотя порядком оценки по-прежнему остается n2. Число же пересылок, вообще говоря, не меняется.

Шейкерную сортировку рекомендуется использовать в тех случаях, когда известно, что массив "почти упорядочен".

Сортировка выбором

При сортировке массива a[1], a[2], ..., a[n] методом простого выбора среди всех элементов находится элемент с наименьшим значением a[i], a[1] и a[i] обмениваются значениями. Затем этот процесс повторяется для получаемых подмассивов a[2], a[3], ..., a[n], ... a[j], a[j+1], ..., a[n] до тех пор, пока мы не дойдем до подмассива a[n], содержащего к этому моменту наибольшее значение. Работа алгоритма иллюстрируется следующим примером.

Пример сортировки простым выбором
Начальное состояние массива :
8 23 5 65 44 33 1 6
Шаг 1:
1 23 5 65 44 33 8 6
Шаг 2:
1 5 23 65 44 33 8 6

Шаг 3:
1 5 6 65 44 33 8 23
Шаг 4:
1 5 6 8 44 33 65 23
Шаг 5:
1 5 6 8 33 44 65 23
Шаг 6:
1 5 6 8 23 44 65 33
Шаг 7:
1 5 6 8 23 33 65 44
Шаг 8:
1 5 6 8 23 33 44 65

Для метода сортировки простым выбором требуемое число сравнений – n(n-1)/2. Порядок требуемого числа пересылок (включая те, которые требуются для выбора минимального элемента) в худшем случае составляет n2. Однако порядок среднего числа пересылок есть nln(n), что в ряде случаев делает этот метод предпочтительным.

Сортировка разделением (Quicksort)

Метод сортировки разделением был предложен Чарльзом Хоаром в 1962 г. Этот метод является развитием метода простого обмена и настолько эффективен, что его стали называть "методом быстрой сортировки – Quicksort". Основная идея алгоритма состоит в том, что случайным образом выбирается некоторый элемент массива x, после чего массив просматривается слева, пока не встретится элемент a[i] такой, что a[i] > x, а затем массив просматривается справа, пока не встретится элемент a[j] такой, что a[j] < x. Эти два элемента меняются местами, и процесс просмотра, сравнения и обмена продолжается, пока мы не дойдем до элемента x. В результате массив оказывается разбитым на две части – левую, в которой значения ключей будут меньше x, и правую со значениями ключей, большими x. Далее процесс рекурсивно продолжается для левой и правой частей массива до тех пор, пока каждая часть не будет содержать в точности один элемент. Понятно, что, как обычно, рекурсию можно заменить итерациями, если запоминать соответствующие индексы массива. Проследим этот процесс на примере нашего стандартного массива.

Пример быстрой сортировки
Начальное состояние массива :
8 23 5 65 |44| 33 1 6
Шаг 1 (в качестве x выбирается a[5]):
|--------|
8 23 5 6 44 33 1 65
8 23 5 6 1 33 44 65
Шаг 2 (в подмассиве a[1], a[5] в качестве x выбирается a[3]):
8 23 |5| 6 1 33 44 65

|--------|
1 23 5 6 8 33 44 65
|--|
1 5 23 6 8 33 44 65
Шаг 3 (в подмассиве a[3], a[5] в качестве x выбирается a[4]) :
1 5 23 |6| 8 33 44 65
|----|
1 5 8 6 23 33 44 65
Шаг 4 (в подмассиве a[3], a[4] выбирается a[4]):
1 5 8 |6| 23 33 44 65
|--|
1 5 6 8 23 33 44 65

Алгоритм недаром называется быстрой сортировкой, поскольку для него оценкой числа сравнений и обменов является nlog(n). На самом деле в большинстве утилит, выполняющих сортировку массивов, используется именно этот алгоритм.

Сортировка с помощью дерева (Heapsort)

Начнем с простого метода сортировки с помощью дерева, при использовании которого явно строится двоичное дерево сравнения ключей. Построение дерева начинается с листьев, которые содержат все элементы массива. Из каждой соседней пары выбирается наименьший элемент, и эти элементы образуют следующий (ближе к корню уровень дерева). Из каждой соседней пары выбирается наименьший элемент и т.д., пока не будет построен корень, содержащий наименьший элемент массива. Двоичное дерево сравнения для массива, используемого в наших примерах, показано на рис. 3.1. Итак, имеем наименьшее значение элементов массива. Для того чтобы получить следующий элемент, спустимся от корня по пути, ведущему к листу с наименьшим значением. В этой листовой вершине проставляется фиктивный ключ с «бесконечно большим» значением, а во все промежуточные узлы, занимавшиеся наименьшим элементом, заносится наименьшее значение из узлов – непосредственных потомков (рис. 3.2). Процесс продолжается до тех пор, пока все узлы дерева не будут заполнены фиктивными ключами (рис. 3.3-3.8).

На каждом из n шагов, требуемых для сортировки массива, нужно log(n) (двоичный) сравнений. Следовательно, всего потребуется nlog(n) сравнений, но для представления дерева понадобится 2n - 1 дополнительных единиц памяти.

Имеется более совершенный алгоритм, который принято называть пирамидальной сортировкой (Heapsort). Его идея состоит в том, что вместо полного дерева сравнения исходный массив a[1], a[2], ..., a[n] преобразуется в пирамиду, обладающую тем свойством, что для каждого a[i] выполняются условия a[i] <= a[2i] и a[i] <= a[2i+1]. Затем пирамида используется для сортировки.

Наиболее наглядно метод построения пирамиды выглядит при древовидном представлении массива, показанном на рис. 3.9. Массив представляется в виде двоичного дерева, корень которого соответствует элементу массива a[1]. На втором ярусе находятся элементы a[2] и a[3]. На третьем – a[4], a[5], a[6], a[7] и т.д.

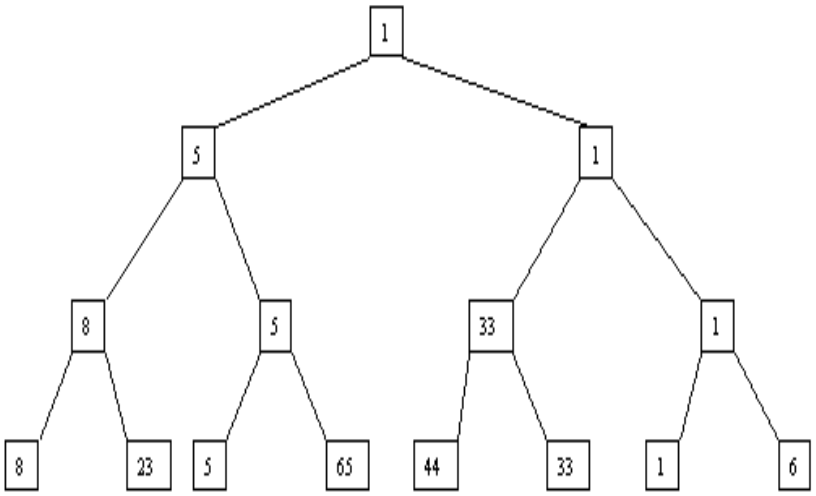


Рисунок 3.1 – Двоичное дерево сравнения для массива

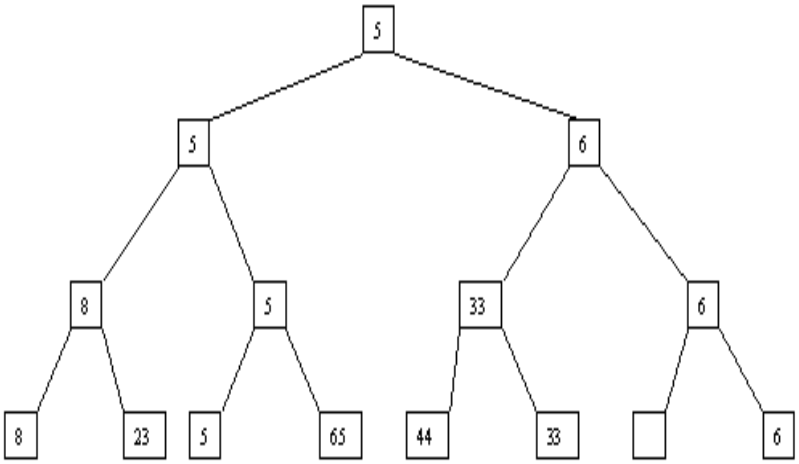


Рисунок 3.2 – Второй шаг сортировки

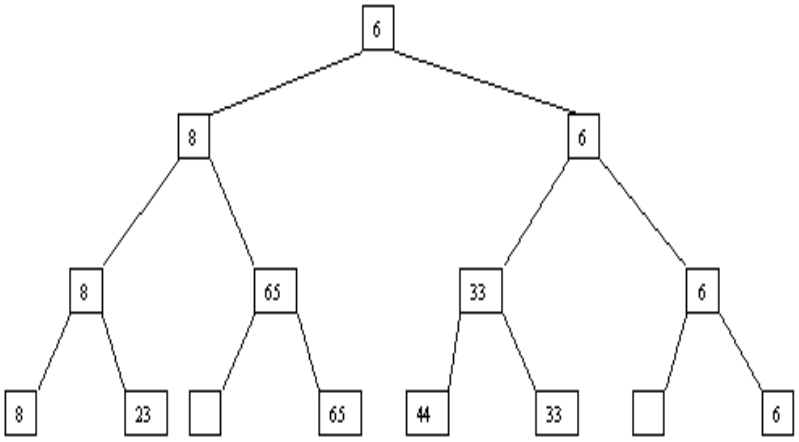


Рисунок 3.3 – Третий шаг сортировки

Как видно, для массива с нечетным количеством элементов соответствующее дерево будет сбалансированным, а для массива с четным количеством элементов n элемент a[n] будет единственным (самым левым) листом "почти" сбалансированного дерева. Очевидно, что при построении пирамиды нас будут интересовать элементы a[n/2], a[n/2-1], ..., a[1] для массивов с четным числом элементов и элементы a[(n-1)/2], a[(n-1)/2-1], ..., a[1] для массивов с нечетным числом элементов (поскольку только для таких элементов существенны ограничения пирамиды).

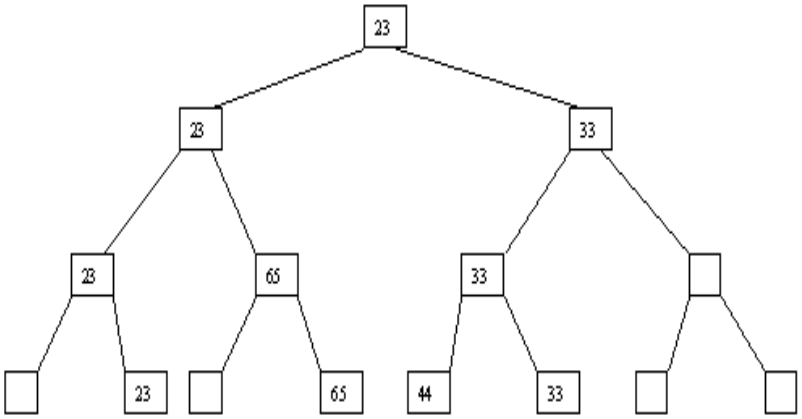


Рисунок 3.4 – Четвертый шаг сортировки

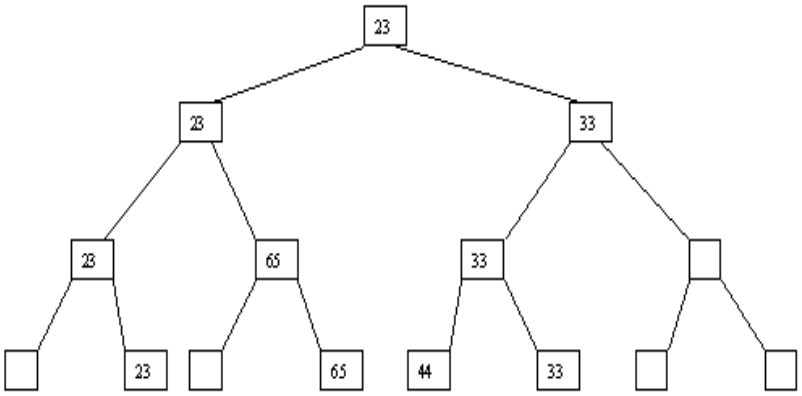


Рисунок 3.5 – Пятый шаг сортировки

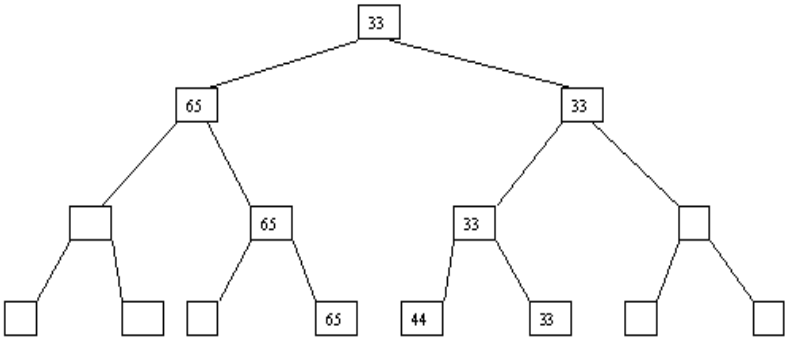


Рисунок 3.6 – Шестой шаг сортировки

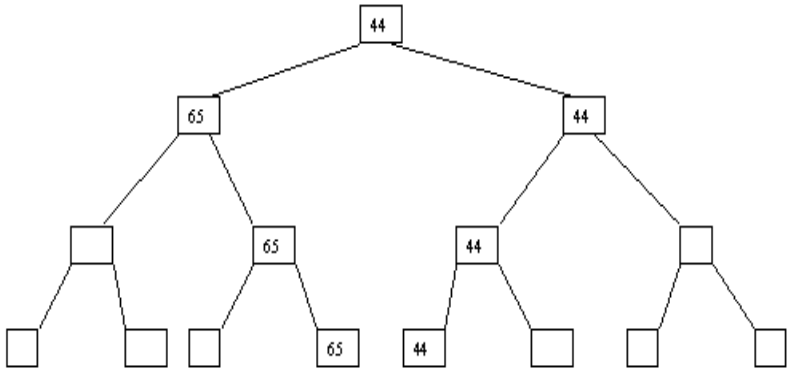


Рисунок 3.7 – Седьмой шаг сортировки

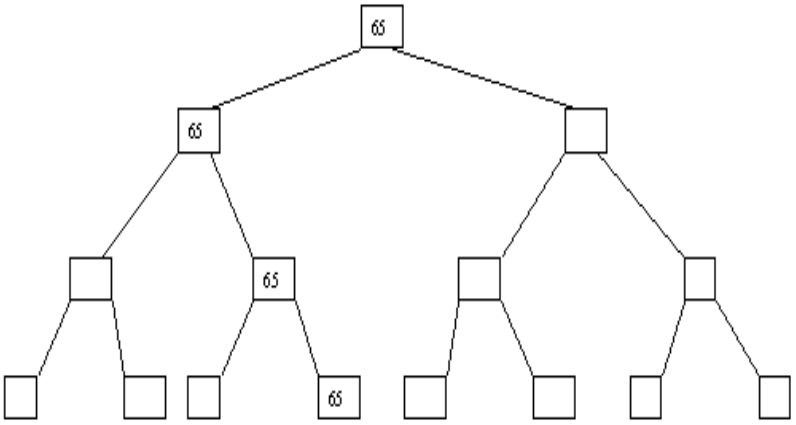


Рисунок 3.8 – Восьмой шаг сортировки

Пусть i – наибольший индекс из числа индексов элементов, для которых существенны ограничения пирамиды. Тогда берется элемент a[i] в построенном дереве, и для него выполняется процедура просеивания, состоящая в том, что выбирается ветвь дерева, соответствующая min(a[2*i], a[2*i+1]), и значение a[i] меняется местами со значением соответствующего элемента. Если этот элемент не является листом дерева, для него выполняется аналогичная процедура и т.д. Такие действия выполняются последовательно для a[i], a[i-1], ..., a[1]. Легко видеть, что в результате мы получим древовидное представление пирамиды для исходного массива (последовательность шагов для используемого в наших примерах массива показана на рис. 3.10-3.13).

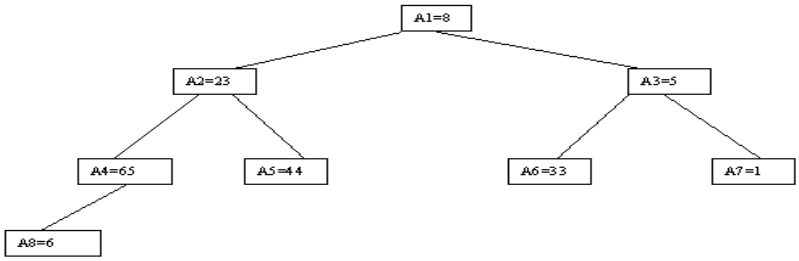


Рисунок 3.9 – Первый шаг сортировки

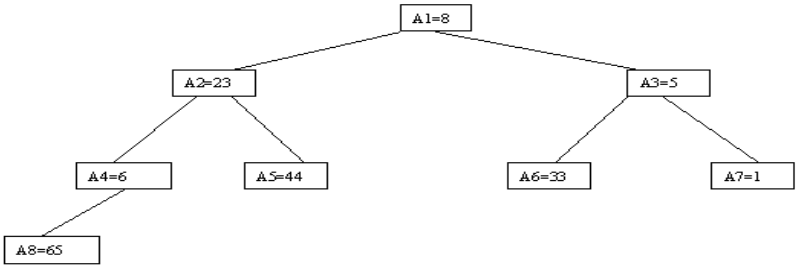


Рис. 3.10. Второй шаг сортировки

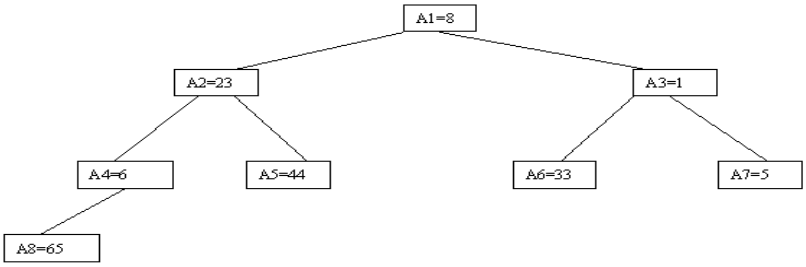


Рисунок 3.11 – Третий шаг сортировки

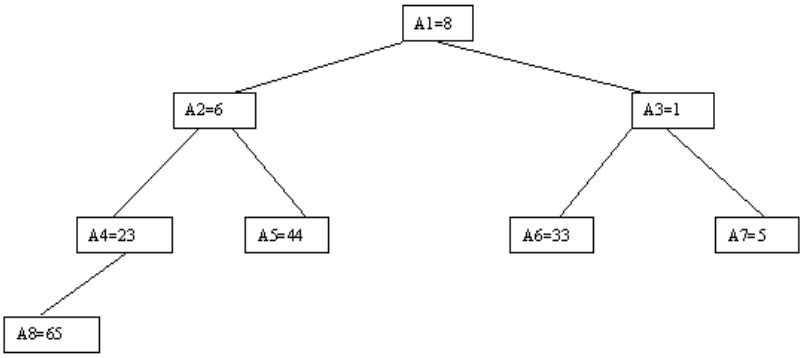


Рисунок 3.12 – Четвёртый шаг сортировки

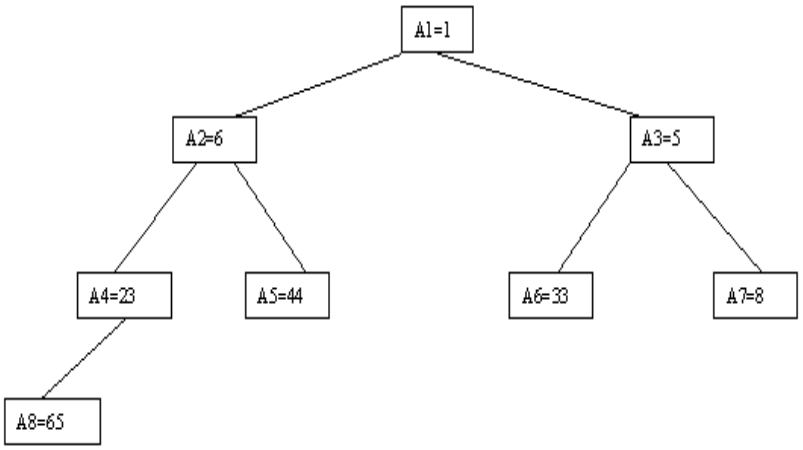


Рисунок 3.13 – Пятый шаг сортировки

В 1964 г. Флойд предложил метод построения пирамиды без явного построения дерева (хотя метод основан на тех же идеях). Построение пирамиды методом Флойда для нашего стандартного массива показано ниже.

Пример построения пирамиды

Начальное состояние массива :

8 23 5 |65| 44 33 1 6

Шаг 1:
8 23 |5| 6 44 33 1 65
Шаг 2:
8 |23| 1 6 44 33 5 65
Шаг 3:
|8| 6 1 23 44 33 5 65
Шаг 4:
1 6 8 23 44 33 5 65
1 6 5 23 44 33 8 65

Ниже показано, как производится сортировка с использованием построенной пирамиды. Суть алгоритма заключается в следующем. Пусть i – наибольший индекс массива, для которого существенны условия пирамиды. Тогда начиная с a[1] до a[i] выполняются следующие действия. На каждом шаге выбирается последний элемент пирамиды (в нашем случае первым будет выбран элемент a[8]). Его значение меняется со значением a[1], после чего для a[1] выполняется просеивание. При этом на каждом шаге число элементов в пирамиде уменьшается на 1 (после первого шага в качестве элементов пирамиды рассматриваются a[1], a[2], ..., a[n-1]; после второго – a[1], a[2], ..., a[n-2] и т.д., пока в пирамиде не останется один элемент). Легко видеть, что в результате мы получим массив, упорядоченный по убыванию. Можно модифицировать метод построения пирамиды и сортировки, чтобы получить упорядочение по возрастанию, если изменить условие пирамиды на a[i] >= a[2i] и a[1] >= a[2i+1] для всех осмысленных значений индекса i.

Сортировка с помощью пирамиды

Исходная пирамида:
1 6 5 23 44 33 8 65
Шаг 1:
65 6 5 23 44 33 8 1
5 6 65 23 44 33 8 1
5 6 8 23 44 33 65 1
Шаг 2:
65 6 8 23 44 33 5 1
6 65 8 23 44 33 5 1
6 23 8 65 44 33 5 1
Шаг 3:
33 23 8 65 44 6 5 1
8 23 33 65 44 6 5 1

Шаг 4:
44 23 33 65 8 6 5 1
23 44 33 65 8 6 5 1
Шаг 5:
65 44 33 23 8 6 5 1
33 44 65 23 8 6 5 1

Шаг 6:
65 44 33 23 8 6 5 1
44 65 33 23 8 6 5 1
Шаг 7:
65 44 33 23 8 6 5 1

Процедура сортировки с использованием пирамиды требует выполнения порядка nxlog n шагов (логарифм – двоичный) в худшем случае, что делает ее особо привлекательной для сортировки больших массивов.

Сортировка со слиянием

Сортировки со слиянием, как правило, применяются в тех случаях, когда требуется отсортировать последовательный файл, не помещающийся целиком в основной памяти. Методам внешней сортировки посвящается следующая часть книги, в которой основное внимание будет уделяться методам минимизации числа обменов с внешней памятью. Однако существуют и эффективные методы внутренней сортировки, основанные на разбиениях и слияниях.

Один из популярных алгоритмов внутренней сортировки со слияниями основан на следующих идеях (для простоты будем считать, что число элементов в массиве, как и в нашем примере, является степенью числа 2). Сначала поясним, что такое слияние. Пусть имеются два отсортированных в порядке возрастания массива p[1], p[2], ..., p[n] и q[1], q[2], ..., q[n] и имеется пустой массив r[1], r[2], ..., r[2n], который мы хотим заполнить значениями массивов p и q в порядке возрастания. Для слияния выполняются следующие действия: сравниваются p[1] и q[1], и меньшее из значений записывается в r[1]. Предположим, что это значение p[1]. Тогда p[2] сравнивается с q[1] и меньшее из значений заносится в r[2]. Предположим, что это значение q[1]. Тогда на следующем шаге сравниваются значения p[2] и q[2] и т.д., пока мы не достигнем границ одного из массивов. Тогда остаток другого массива просто дописывается в "хвост" массива r.

Для сортировки со слиянием массива a[1], a[2], ..., a[n] заводится парный массив b[1], b[2], ..., b[n]. На первом шаге производится слияние a[1] и a[n] с размещением результата в b[1], b[2], слияние a[2] и a[n-1] с размещением результата в b[3], b[4], ..., слияние a[n/2] и a[n/2+1] с помещением результата в b[n-1], b[n]. На втором шаге производится слияние пар b[1], b[2] и b[n-1], b[n] с помещением результата в a[1], a[2], a[3], a[4], слияние пар b[3], b[4] и b[n-3], b[n-2] с помещением результата в a[5], a[6], a[7], a[8], ..., слияние пар b[n/2-1], b[n/2] и b[n/2+1], b[n/2+2] с помещением результата в a[n-3], a[n-2], a[n-1], a[n]. И так далее. На последнем шаге, например (в зависимости от значения n), производится слияние последовательностей элементов массива длиной n/2 a[1], a[2], ..., a[n/2] и a[n/2+1], a[n/2+2], ..., a[n] с помещением результата в b[1], b[2], ..., b[n].

Пример слияния двух массивов показан на рис. 3.14.

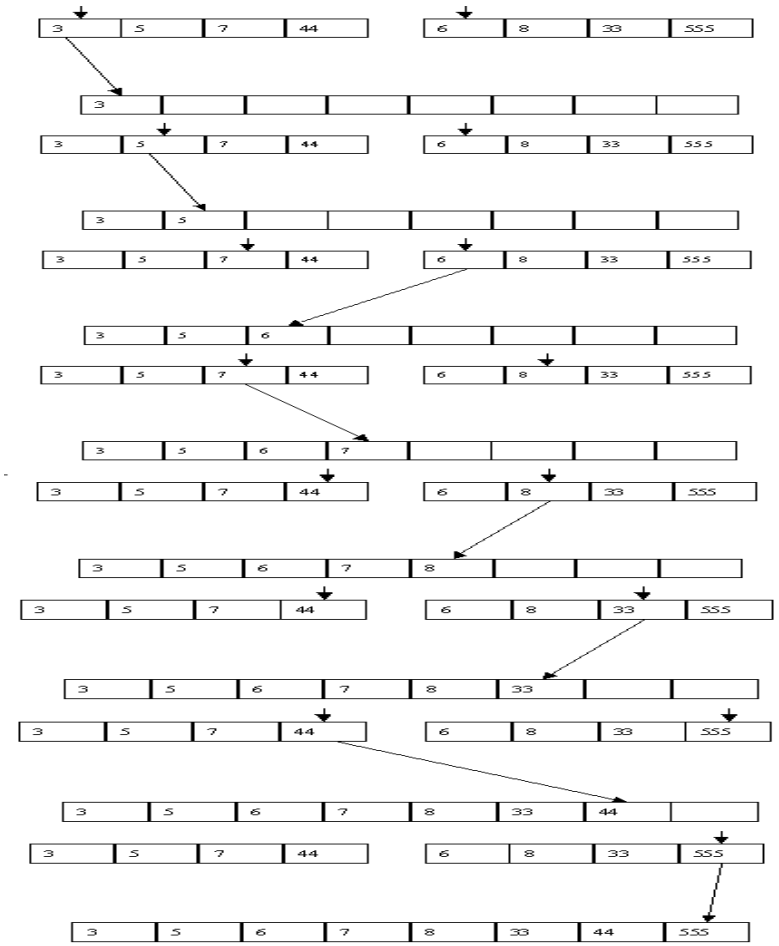


Рисунок 3.14 – Слияние двух массивов

Для случая массива, используемого в наших примерах, последовательность шагов показана ниже.

Пример сортировки со слиянием

Начальное состояние массива:

8 23 5 65 44 33 1 6

Шаг 1:

6 8 1 23 5 33 44 65

Шаг 2:

6 8 44 65 1 5 23 33

Шаг 3:

1 5 6 8 23 33 44 65

При применении сортировки со слиянием число сравнений ключей и число пересылок оценивается как nlog(n). Но следует учитывать, что для выполнения алгоритма для сортировки массива размера n требуется 2n элементов памяти.

Методы внешней сортировки.

Принято называть "внешней" сортировку последовательных файлов, располагающихся во внешней памяти и слишком больших, чтобы можно было целиком переместить их в основную память и применить один из рассмотренных в предыдущем разделе методов внутренней сортировки.

Наиболее часто внешняя сортировка применяется в системах управления базами данных при выполнении запросов, и от эффективности применяемых методов существенно зависит производительность СУБД.

Следует пояснить, почему речь идет именно о последовательных файлах, т.е. о файлах, которые можно читать запись за записью в последовательном режиме, а писать можно только после последней записи. Методы внешней сортировки появились, когда наиболее распространенными устройствами внешней памяти были магнитные ленты. Для лент чисто последовательный доступ был абсолютно естественным. Когда произошел переход к запоминающим устройствам с магнитными дисками, обеспечивающими "прямой" доступ к любому блоку информации, казалось, что чисто последовательные файлы потеряли свою актуальность. Однако это ощущение было ошибочным.

Все дело в том, что практически все используемые в настоящее время дисковые устройства снабжены подвижными магнитными головками. При выполнении обмена с дисковым накопителем выполняется подвод головок к нужному цилиндру, выбор нужной головки (дорожки), прокрутка дискового пакета до начала требуемого блока и, наконец, чтение или запись блока. Среди всех этих действий самое большое время занимает подвод головок. Именно это время определяет общее время выполнения операции. Единственным доступным приемом оптимизации доступа к магнитным дискам является как можно более «близкое» расположение на накопителе последовательно адресуемых блоков файла. Но и в этом случае движение головок будет минимизировано только в том случае, когда файл читается или пишется в чисто последовательном режиме. Именно с такими файлами при потребности сортировки работают современные СУБД.

Следует также заметить, что на самом деле скорость выполнения внешней сортировки зависит от размера буфера (или буферов) основной памяти, которая может быть использована для этих целей. Рассмотрим основные методы внешней сортировки, работающие при минимальных расходах основной памяти.

Прямое слияние

Начнем с того, как можно использовать в качестве метода внешней сортировки алгоритм простого слияния, обсуждавшийся в конце предыдущей части. Предположим, что имеется последовательный файл A, состоящий из записей a1, a2, ..., an (снова для простоты предположим, что n представляет собой степень числа 2). Будем считать, что каждая запись состоит ровно из одного элемента, представляющего собой ключ сортировки. Для сортировки используются два вспомогательных файла B и C (размер каждого из них будет n/2).

Сортировка состоит из последовательности шагов, в каждом из которых выполняется распределение состояния файла A в файлы B и C, а затем слияние файлов B и C в файл A. (Заметим, что процедура слияния для

файлов полностью иллюстрируется рис. 3.14.) На первом шаге для распределения последовательно читается файл A, записи a1, a3, ..., a(n-1) пишутся в файл B, а записи a2, a4, ..., an – в файл C (начальное распределение). Начальное слияние производится над парами (a1, a2), (a3, a4), ..., (a(n-1), an), и результат записывается в файл A. На втором шаге снова последовательно читается файл A, в файл B записываются последовательные пары с нечетными номерами, а в файл C – с четными. При слиянии образуются и пишутся в файл A упорядоченные четверки записей. И так далее. Перед выполнением последнего шага файл A будет содержать две упорядоченные подпоследовательности размером n/2 каждая. При распределении первая из них попадет в файл B, а вторая – в файл C. После слияния файл A будет содержать полностью упорядоченную последовательность записей. Ниже показан пример внешней сортировки простым слиянием.

Пример внешней сортировки прямым слиянием
Начальное состояние файла A:
8 23 5 65 44 33 1 6
Первый шаг:
Распределение
Файл B 8 5 44 1
Файл C 23 65 33 6
Слияние: файл A
8 23 5 65 33 44 1 6
Второй шаг:
Распределение
Файл B 8 23 33 44
Файл C 5 65 1 6
Слияние: файл A
5 8 23 65 1 6 33 44
Третий шаг:
Распределение
Файл B 5 8 23 65
Файл C 1 6 33 44
Слияние: файл A
1 5 6 8 23 33 44 65

Заметим, что для выполнения внешней сортировки методом прямого слияния в основной памяти требуется расположить всего лишь две переменные – для размещения очередных записей из файлов B и C. Файлы A, B и C будут log(n) раз прочитаны и столько же раз записаны.

Естественное слияние

При использовании метода прямого слияния не принимается во внимание то, что исходный файл может быть частично отсортированным, т.е.

содержать упорядоченные подпоследовательности записей. Серией называется подпоследовательность записей ai, a(i+1), ..., aj такая, что ak <= a(k+1) для всех i <= k < j, ai < a(i-1) и aj > a(j+1). Метод естественного слияния основывается на распознавании серий при распределении и их использовании при последующем слиянии.

Как и в случае прямого слияния, сортировка выполняется за несколько шагов, на каждом из которых сначала выполняется распределение файла A по файлам B и C, а потом слияние B и C в файл A. При распределении распознается первая серия записей и переписывается в файл B, вторая – в файл C и т.д. При слиянии первая серия записей файла B сливается с первой серией файла C, вторая серия B – со второй серией C и т.д. Если просмотр одного файла заканчивается раньше, чем просмотр другого (по причине разного числа серий), то остаток недопросмотренного файла целиком копируется в конец файла A. Процесс завершается, когда в файле A остается только одна серия. Пример сортировки файла показан на рис. 3.15 и 3.16.

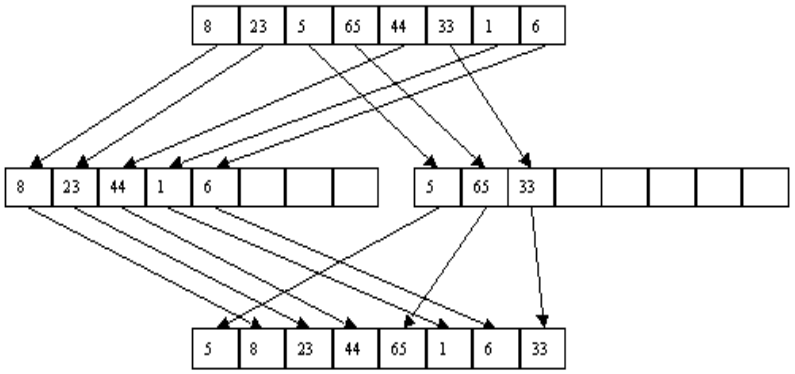


Рисунок 3.15 – Первый шаг

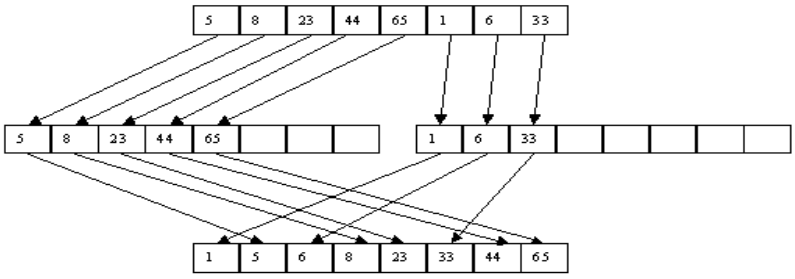


Рисунок 3.16 – Второй шаг

Очевидно, что число чтений/перезаписей файлов при использовании этого метода будет не хуже, чем при применении метода прямого слияния, а в среднем – лучше. С другой стороны, увеличивается число сравнений за счет тех, которые требуются для распознавания концов серий. Кроме того, поскольку длина серий может быть произвольной, то максимальный размер файлов B и C может быть близок к размеру файла A.

4. Хеширование данных

Для ускорения доступа к данным в таблицах можно использовать предварительное упорядочение 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 упорядочение таблицы могут значительно превышать выигрыш от сокращения времени поиска. Поэтому для сокращения времени доступа к данным в таблицах используется так называемое случайное упорядочение, или хеширование. При этом данные организуются в виде таблицы при помощи хеш-функции h, используемой для “вычисления” адреса по значению ключа (рис 4.1).

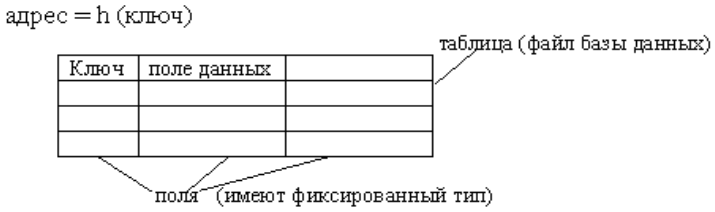


Рисунок 4.1 – Хеш-таблица

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

Подобрать такую функцию можно в случае, если все возможные значения ключей заранее известны. Такая организация данных носит название “совершенное хеширование“. В случае заранее не определенного 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 Turbo Pascal. 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”.

5. Представление графов и деревьев

Теория графов является важной частью вычислительной математики. С помощью этой теории решается большое количество задач из различных областей. Граф состоит из множества вершин и множества ребер, которые соединяют между собой вершины. С точки зрения теории графов не имеет значения, какой смысл вкладывается в вершины и ребра. Вершинами могут быть населенные пункты, а ребрами – дороги, соединяющие их, или вершины – подпрограммы, а соединение вершин ребрами – взаимодействие подпрограмм. Часто имеет значение направление 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 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 который не входит ни 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* (рис 5.1).

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

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 (рис.5.2). Исходя из определения степени понятно, что степень узла бинарного дерева не превышает числа *два.* При этом листьями в дереве являются вершины, имеющие степень *ноль*.

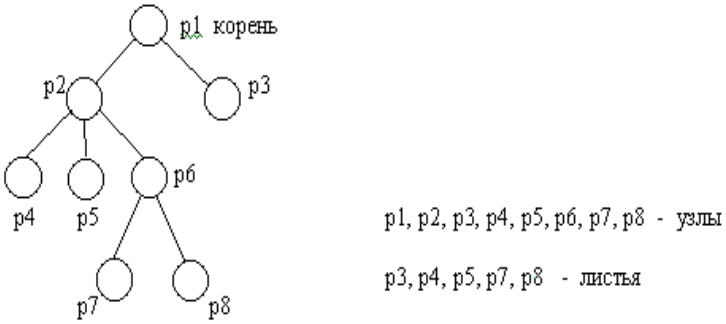


Рисунок 5.1 – Бинарное дерево

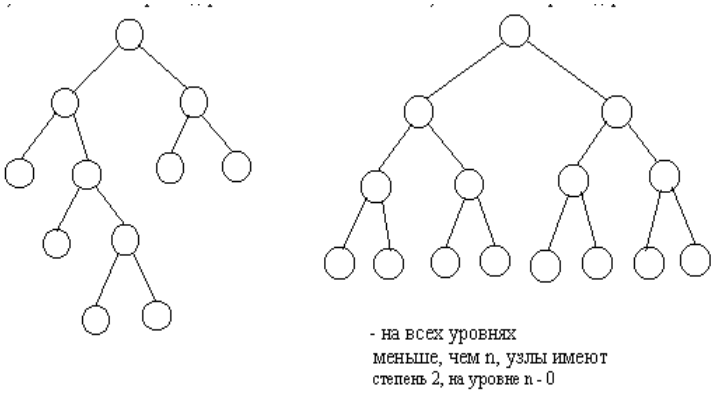


Рисунок 5.2 – Полное *(а)* и неполное *(б)* бинарные деревья

Другим важным признаком структурной классификации бинарных деревьев является строгость бинарного дерева (рис.5.3.). Строго бинарное дерево состоит только из узлов, имеющих степень *два* или степень *ноль*. Не строго бинарное дерево содержит узлы со степенью единица.

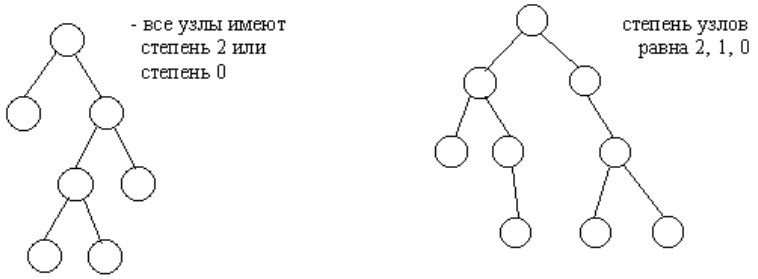


Рисунок 5.3 – Строго *(а)* и не строго *(б)* бинарные деревья

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.

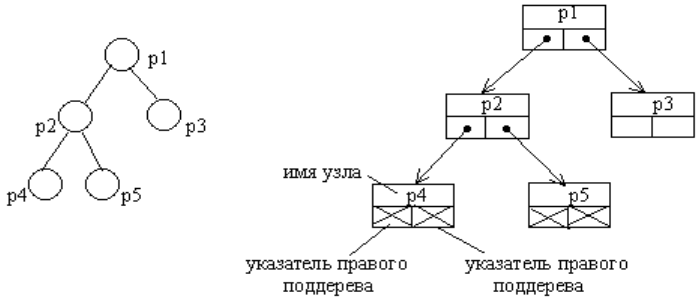


Рисунок 5.4 – Представление бинарного дерева в виде списковой структуры

Приведем пример программы, которая осуществляет создание и редактирование бинарного дерева, представленного в виде списковой структуры (рис.5.4):

```
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 потомку');
          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 (рис 5.5.). Вершины можно пронумеровать слева направо последовательно по уровням и использовать эти номера в качестве индексов в одномерном массиве.

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

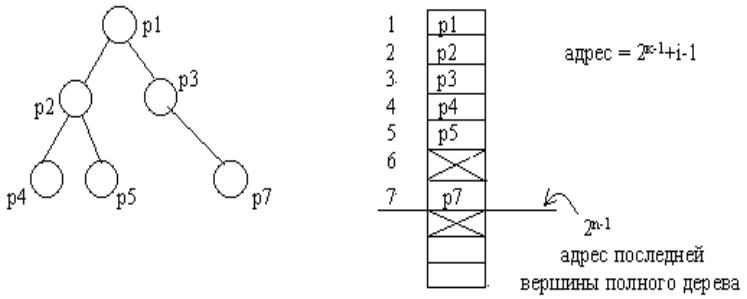


Рисунок 5.5 – Представление бинарного дерева в виде массива

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.

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 (рис 5.6.) можно определить следующим образом:

- попасть в корень;
- пройти в прямом порядке левое поддерево;
- пройти в прямом порядке правое поддерево.

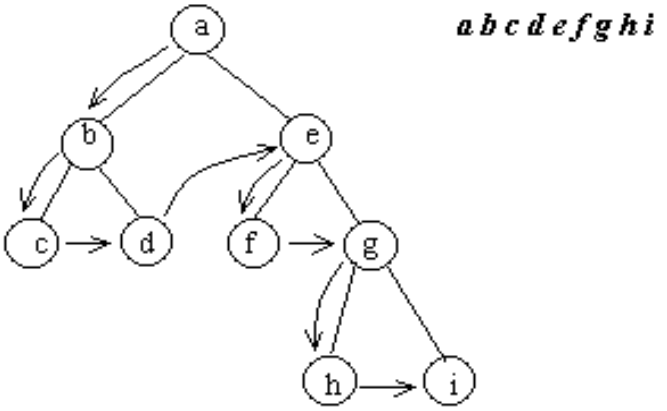


Рисунок 5.6– Прямой порядок прохождения бинарного дерева

Прохождение бинарного дерева в обратном порядке (рис.5.7) можно определить в аналогичной форме:

- пройти в обратном порядке левое поддерево;
- пройти в обратном порядке правое поддерево;
- попасть в корень.

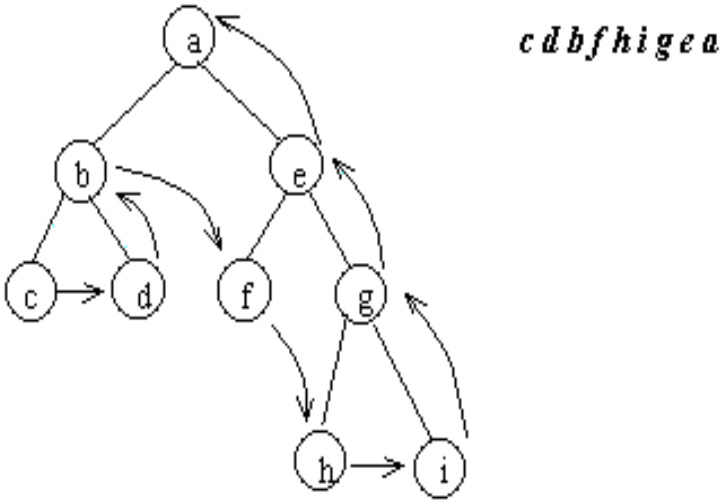


Рисунок 5.7 – Обратный порядок прохождения бинарного дерева

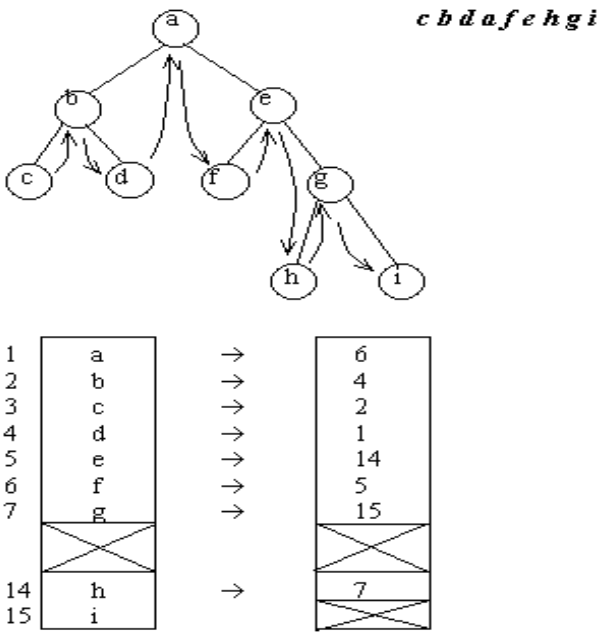


Рисунок 5.8 – Представление симметрично прошитого бинарного дерева в виде массивов

Определим еще один порядок прохождения бинарного дерева, называемый симметричным:

- пройти в симметричном порядке левое поддерево;
- попасть в корень;
- 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 записывается адрес вершины в основном массиве, следующей за данной вершиной.

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 правопрошитые, левопрошитые и симметрично прошитые бинарные деревья (рис 5.8.).

Алгоритмы на деревьях

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 (рис 5.9.). 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.

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 из пройденных узлов.

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

Рисунок 5.9 – Сортировка по возрастанию с прохождением бинарного дерева

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 (рис. 5.10). 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.

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 (рис. 5.11).

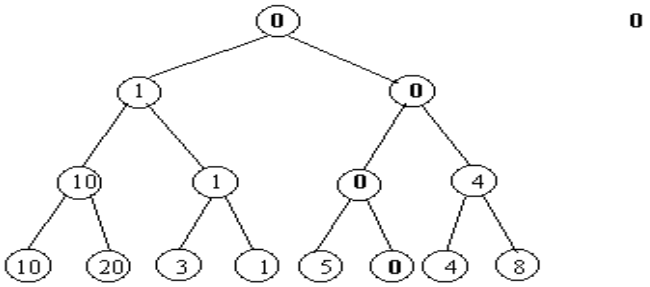


Рисунок 5.10 – Построение дерева сортировки

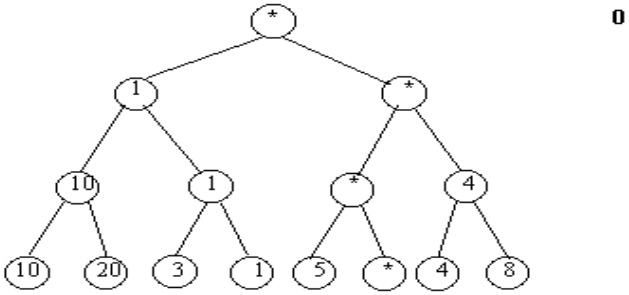


Рисунок 5.11 – Замена извлекаемого элемента на специальный символ

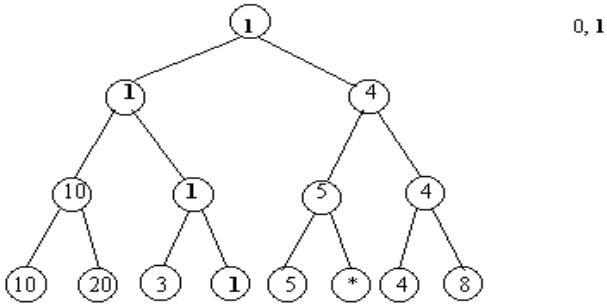


Рисунок 5.12 – Повторное заполнение дерева сортировки

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 (рис. 5.13) и итерация повторяется. Извлечения элементов продолжаются до тех пор, пока в дереве не останутся одни специальные символы.

В результате получим отсортированный массив

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

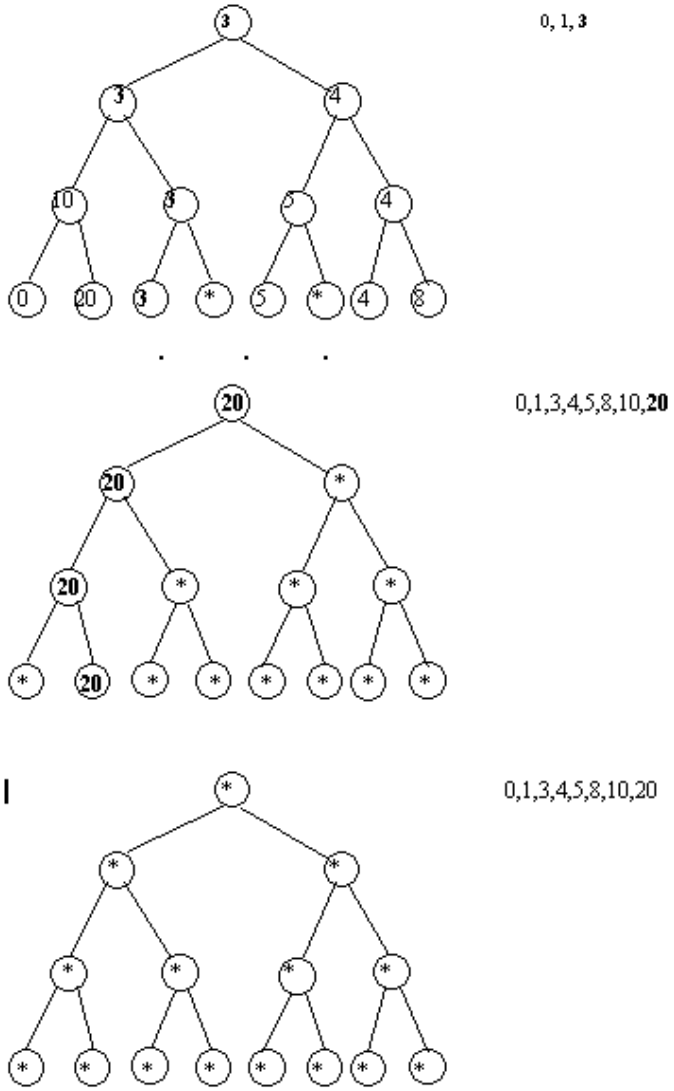


Рисунок 5.13 – Извлечения элементов из дерева сортировки

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 десяти 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 используется всего четыре 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. Метод 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 (рис. 5.14). 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

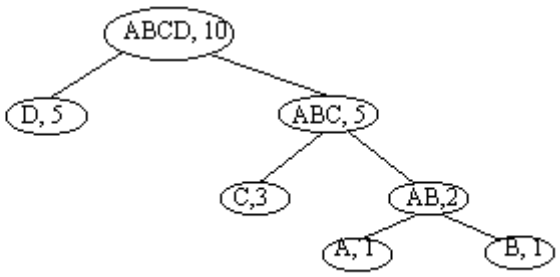


Рисунок 5.14 – Построение кодовой таблицы

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 (рис 5.15). 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.

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.

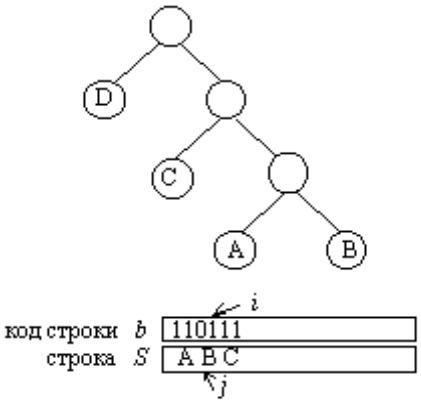


Рисунок 5.1 – Процесс распаковки кода

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

Представление выражений с помощью деревьев

С помощью деревьев можно представлять произвольные арифметические выражения (рис. 5.16). Каждому листу в таком дереве соответствует операнд, а каждому родительскому узлу – операция. В общем случае дерево при этом может оказаться не бинарным (рис. 5.17).

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.

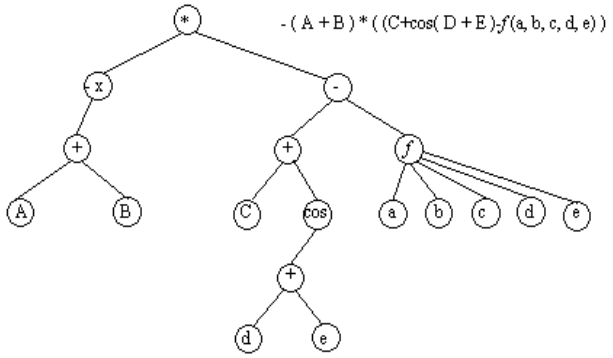


Рисунок 5. – Представление арифметического выражения произвольного вида в виде дерева

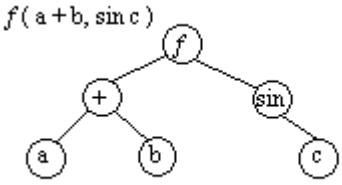


Рисунок 5.17 – Представление арифметического выражения в виде бинарного дерева

Бинарные деревья могут быть использованы не только для представления выражений, но и для их вычисления (рис 5.18). 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.5.19).

Пример: (1+10)*5

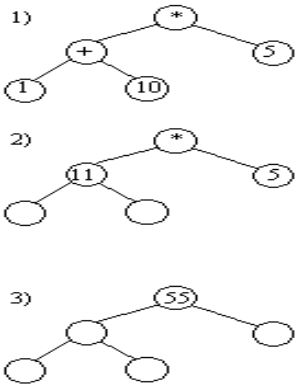


Рисунок 5.18 – Вычисление арифметического выражения с помощью бинарного дерева

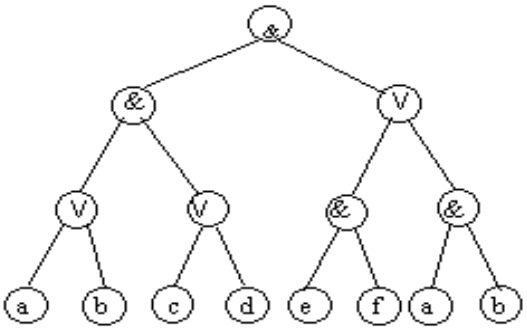


Рисунок 5.19 – Представление логического выражения
в виде бинарного дерева

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 (рис 5.20).

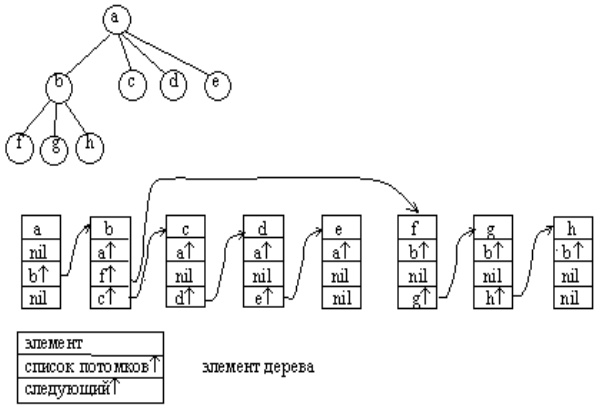


Рисунок 5.20 – Представление сильноветвящихся деревьев в виде списков

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.

ЛАБОРАТОРНЫЕ РАБОТЫ

Выполнение лабораторных работ предполагает четыре этапа: подготовку, непосредственно выполнение, оформление отчета и защиту работы.

Подготовка к работе выполняется в часы лабораторных и практических занятий и заключается в следующем.

. Выбор и описание структур данных и разработка алгоритмов. Следует помнить, что алгоритм является формальным описанием вычислительной процедуры и в нем необходимо использовать обозначения, определенные в формальной постановке задачи. Поэтому алгоритм не должен представляться текстом программы; излишние подробности, связанные с требованиями синтаксиса языка программирования, обычно заслоняют существо дела.

Программная реализация. Заключается в представлении структур данных и алгоритмов соответствующими конструкциями выбранного языка программирования. При этом должны быть указаны идентификаторы переменных, подпрограмм и т. п., сопоставленные с соответствующими элементами формального алгоритма. Тексты программ должны содержать подробные комментарии, поясняющие назначение процедур, их параметров, использование переменных, смысл и особенности реализации отдельных программных блоков.

Выполнение работы заключается в вводе текстов программ, их трансляции и отладке на тестовых примерах. Порядок выполнения приводится в описании каждой работы. Выполнение работы подтверждается подписью преподавателя в отчете, что означает допуск к защите работы.

Отчет о работе должен содержать:

1. Титульный лист.
2. Цель работы.
3. Информацию в соответствии с подготовкой к работе.
4. Результаты выполнения работы (результаты решения задач, таблицы, графики, аналитические и/или экспериментальные зависимости и т. д.).
5. Выводы по работе, отражающие анализ результатов и предпочтительные области применения разработанных структур данных и алгоритмов.
6. Тексты программ в виде текстовых файлов.

Защита работы заключается в опросе студента преподавателем. Преподаватель вправе задавать вопросы по всем разделам отчета: теоретической части, аспектам выбора структур данных и разработки алгоритмов, программной реализации, результатам исследований. По итогам опроса решается вопрос о защите лабораторной работы.

Лабораторная работа № 1
Разработка программы с использованием записей

Цель работы: формирование знаний и умений по работе с записями. Приобретение навыков работы с записями.

Краткие теоретические сведения

При решении научно - технических и экономических задач обработки совокупностей большого количества значений используются *массивы*. Но при работе с массивами основное ограничение заключается в том, что каждый элемент массива должен иметь *один и тот же тип данных*.

Иногда для решения задач, в которых возникает необходимость хранить и обрабатывать совокупности данных различного типа, используются записи.

Запись представляет собой наиболее общий и гибкий структурированный тип данных, так как она может быть образована из не однотипных компонентов и в ней явным образом выражена связь между элементами данных, характеризующими реальный объект.

Запись-это структурированный тип данных, состоящий из фиксированного числа компонентов, называемых *полями записи*. Определение типа записи начинается идентификатором ***Record*** и заканчивается зарезервированным словом ***end***. Между ними заключен список компонентов, называемых ***полями***, с указанием имен полей и типа каждого поля.

Структура объявления типа записи такова:

```
Type
  <имя типа> = Record
  <имя поля>: <тип компонентов>;
  ...
  <имя поля>: <тип компонентов>;
End;
VAR
  <имя переменной>: <имя типа>;
```

Пример описания записи:

```
Type
  Car=Record
  Number:Integer;
  Marka:String[20];
  FIO:String[40];
  Address:String[60];
  End;
Var
Mashina: Car;
```

В данном примере была объявлена запись с именем Car, у которой имеется 4 поля: номер, название марки машины, ФИО владельца и его адрес.

Идентификатор поля должен быть уникален только в пределах записи, однако лучше его сделать уникальным в пределах всей программы. Объем памяти, необходимый для записи, складывается из длин полей. Значения полей записи могут быть использованы в выражениях. ***Обращение к значению поля*** осуществляется с помощью идентификатора переменной и идентификатора поля, разделенных точкой. Такая комбинация называется ***составным именем***.

Например, доступ к полям записи Car осуществляется как: *Mashina.Marka, Mashina.FIO, Mashina.Number*. Составное имя можно использовать везде, где допустимо применение типа поля. Для присваивания полям значений используется оператор присваивания.

Пример присваивания полям записи Mashina:

```
Mashina. Number:=164536l;
Mashina.Marka:='ГАЗ-24';
Mashina.FIO:='Иванов И.И';
Mashina. Address:='ул.Пушкина 12-30';
```

Составные имена можно использовать в операторах ввода-вывода:

```
Read (Mashina. Number, Mashina. FIO, Mashina. Address);
Write(Mashina. Number:4, Mashina. FIO:12, Mashina. Address:25);
```

Допускается применение оператора присваивания и к записям в целом, если они имеют один и тот же тип. Например,

```
Mash:=Mashina;
```

После выполнения этого оператора значения полей записи *Mash* станут равны значениям соответствующих полей записи *Mashina.*

В ряде задач удобно пользоваться ***массивами из записей***. Их можно описать следующим образом:

```
 Type
Car=Record
        Number:Integer;
        Marka:String[20];
        FIO:String[40];
        Address:String[60];
        End;
Var
  Mashins: array [1..20] of Car;
```

Обращение к полям такой записи имеет громоздкий вид, для решения этой проблемы в языке Паскаль предназначен оператор **With**, который имеет следующий формат:

With <переменная типа запись> do <оператор>;

Один раз, указав переменную типа запись в операторе *With*, можно работать с именами полей как с обычными переменными.

Пример присвоения значения полям записи Car с помощью оператора With.

```
  With Mashina do
        Begin
 Number:=164536l;
  Marka:=’ГАЗ-24’;
   FIO:=’Иванов И.И’;
     Address:=’ул.Пушкина 12-30’;
End;
```

Пример программы работы с записями

Пусть необходимо составить программу, которая создает каталог компьютерных программ и обеспечивает поиск программ по фамилии автора.

Для описания сведений о компьютерных программах в разделе типов введем тип *Prog_Type* –запись следующей структуры:

```
Prog_Type=Record
Title:String[50];
Author:String[50];
Entry:Integer;
Firma:String[40];
End;
```

где *Title*- поле для записи названия программы, *Author*-поле для записи фамилии автора, *Entry*- поле для записи года разработки, *Firma*-поле для записи фирмы-разработчика.

В разделе описания переменных введем массив *Prog_Katalog* из 10 записей типа *Prog_Type*.Переменную *Num_Array*, принимающую значения от 1до 10 введем для указания на порядковый номер записи в массиве *Prog_Katalog*. Для критерия поиска введем переменную *Author* строкового типа. Результат поиска записывается в переменную логического типа *Yes_Prog.*

В целом текст программы может выглядеть так:

```
Program Katalog;
Type
Prog_Type=record
          Title:string[50];

          Author:String[50];
          Entry:Integer;
          Firma:String[40];
end;
Var
Prog_Katalog:Array[1..10] of Prog_Type;
Num_Array:1..10;
```

```
Author:String[50];
Yes_Prog:Boolean;
Procedure Input_Data;
        Begin
Writeln('Введите данные о ',Num_Array,'-й программе:');
With Prog_Katalog[Num_Array] do
        begin
        Write('Название программы: ');
        Readln(Title);
        Write('Автор:');
        Readln(Author);
        Write('Год разработки:');
        Readln(Entry);
        Write('Фирма:');
Readln(Firma);

Writeln;
        end;
        end;
Procedure Write_Data(Num:Integer);
        begin
Writeln('Программа № ',Num);
With Prog_Katalog[Num_Array] do
begin
Writeln('Название:',Title);
Writeln('Фамилия автора:',Author);
Writeln('Год разработки:',Entry);
Writeln('Фирма:',Firma);
end;
        end;
{Основная программа}
Begin
{У пользователя запрашивается 3 раза ввод данных о программах}
for Num_Array:=1 to 3 do
        Input_Data;
Writeln;
Writeln('Поиск информации(программы) по фамилии автора: ');
Writeln;
Write('Введите фамилия автора: ');
Readln(Author);
Yes_Prog:=False;
for Num_array:=1 to 10 do
        if Prog_Katalog[Num_Array].Author=Author then
        begin
```

```
Write_Data(Num_Array);
Yes_Prog:=True;
        end;
if not Yes_Prog then Write('Нет программ данного автора ',Author);
end.
```

Порядок выполнения работы

1. Изучить теоретические сведения по теме “ Работа с записями”.
2. Разработать программу для работы с записями согласно полученному заданию.
3. Показать работающую программу преподавателю.
4. Ответить на контрольные вопросы.

Варианты заданий

1. Составить программу, выводящую на экран меню детского кафе (наименование изделия, вес, стоимость).
2. Составить программу, выводящую на экран студенческую ведомость (Ф. И. О., оценки за три экзамена, средний балл).
3. Составить программу, выводящую на экран расписание движения поездов (станция отправления, станция прибытия, время прибытия, время в пути).
4. Составить программу, выводящую на экран меню ресторана "Дракон" (наименование изделия, вес, стоимость).
5. Составить программу, выводящую на экран анкетные данные учеников (Ф. И. О., год рождения, адрес, сведения о родителях).
6. Составить программу, выводящую на экран список книг домашней библиотеки (автор, название книги, издательство, год издания, стоимость).
7. Составить программу, выводящую на экран расписание экзаменов и зачетов (предмет, вид отчетности, число, преподаватель).
8. Составить программу, выводящую на экран сведения о студентах (Ф. И. О., курс, группа, номер зачетки, средний балл).
9. Составить программу, выводящую на экран сведения о периодических изданиях (наименование издания, тираж, годовая стоимость).
10. Составить программу, выводящую на экран расписание учителя (номер урока, время начала урока, класс, предмет, номер кабинета).
11. Составить программу, выводящую на экран расписание полетов самолетов (пункт посадки, время отправления, время прибытия, время полета, стоимость билета).
12. Составить программу, выводящую на экран перечень товаров, имеющихся в продаже в магазине "Океан" (наименование, единица измерения, цена, количество).
13. Составить программу, выводящую на экран информацию о наличии товаров на складе (наименование, артикул, дата получения, единица измерения, количество, цена).

14. Составить программу, выводящую на экран "Телефонный справочник" (Ф. И. О., адрес, номер телефона).

15. Составить программу, выводящую на экран график отпусков (Ф. И. О., дата начала отпуска, дата выхода на работу, количество дней).

Контрольные вопросы

1. Понятие записи. Структура объявления типа записи.
2. Обращение к значению поля. Составные имена.
3. Присвоение полям записи значений. Массивы записей.
4. Пример программы с использованием записей.

Лабораторная работа № 2
Разработка программы с использованием записей с вариантами

Цель работы: формирование знаний и умений по работе с записями с вариантами. Приобретение навыков работы с вариантными полями.

Краткие теоретические сведения

Записи, рассмотренные в предыдущей лабораторной работе, имеют строго определенную структуру. В некоторых случаях это резко ограничивает возможности их применения. Поэтому в языке Паскаль имеется возможность задать тип записи, содержащий произвольное число вариантов структуры. Такие записи называются *записями с вариантами.* Записи с вариантами обеспечивают средства объединения записей, которые похожи, но не идентичны по форме. Они состоят из *фиксированной* и *вариантной частей*.

Вариантная часть формируется с помощью оператора *Case*. Он задает особое поле записи-поле признака, которое определяет, какой из вариантов в данный момент будет активизирован. Значением признака в каждый текущий момент выполнения программы должна быть одна из расположенных далее констант. Константа, служащая признаком, задает вариант записи и называется константой выбора.

Формат:

Type

<имя типа> = ***Record***

Case *<поле признака>: <имя типа>* ***of***

<константа выбора 1>: (поле, ...: тип);

...

<константа выбора n>: (поле, ...: тип);

End;

Компоненты каждого варианта (идентификаторы полей и их типы) заключаются в круглые скобки. *У части Case нет отдельного end*. Одно слово *end* заканчивает всю конструкцию записи с вариантами. Количество полей каждого из вариантов не ограничено. Объем памяти, необходимый для

записи с вариантами, складывается из объемов полей фиксированной части и максимального по объему поля переменной части.

Пример:

```
Type
        Rec=Record
              Number:Byte;
             Code:Integer;
     Case Flag :Boolean of
     True:(Proce1:Integer);
     False:(Price2:Real);
       End;
   Var
 PRec: Rec;
```

В данном примере была объявлена запись с именем *Rec*, у которой поля *Number* и *Code* расположены в фиксированной части записи, они доступны в программе в любой текущий момент независимо от значения поля признака. Поле *Price 1* может использоваться только в том случае, если значение поля признака *Flag* равно *True*. Поле *Price 2* доступно в противоположном случае, т.е. если значение *Flag* равно *False.*

При использовании записей с вариантами необходимо придерживаться следующих правил:

3 все имена полей должны отличаться друг от друга, по крайней мере, одним символом, даже если они встречаются в разных вариантах

3 запись может иметь только одну вариантную часть, причем вариантная часть должна размещаться в конце записи

3 если поле, соответствующее какой-либо метке, является пустым, то оно записывается следующим образом: *<метка>: ();*

Пример программы работы с записями

Пусть необходимо составить программу, которая создает каталог изданий в библиотеке, обеспечивает ввод данных о литературе, поиск и подсчет количества книг данного издания.

Литературу в библиотеке можно разделить на три типа изданий: книги, журналы, газеты. Для описания сведений о типе изданий в разделе типов введем перечисляемый тип:

Type_Publ=(Book,Journal,Newspaper);

Для описания сведений о литературе в разделе типов ведем тип *Liter*. Для разного типа изданий в каталоге требуется хранить разную информацию, например: если для поиска книги нужно знать год издания, то для журнала помимо года издания, нужно знать его номер, а для газеты не только год, номер, но и день выпуска. В связи с необходимостью хранения разной информации в структуре записи *Liter* наряду с неизменной частью - полями *Title* и *Author*, в которых отображается название публикации и фамилия автора будет вариантная часть, отражающая дату издания по-разному в

зависимости от типа издания. Запись *Liter* будет иметь следующую структуру:

```
Liter=Record
Title:String[50];
Author:String[50];
Case V: Type_Publ of
Book: (YearB:Integer);
Journal: (Num:1..12;
          YearJ:1900..2001);
Newspaper: (Day:1..31;
            Month:1..12;
           YearN:Integer);
End;
```

где *V*- признак выбора вариантов, который может принимать значение *Book, Journal, Newspaper*. Для типа *Book* предусмотрено хранение года издания (поле *YearB*), для издания типа *Journal*-номера(*Num*) и год издания (поле *YearJ*), для издания типа *Newspaper* –дня (поле *Day*), месяца (поле *Month*) и года выпуска (поле *YearN*).

В разделе описания констант зададим значение максимального числа записей в каталоге *Count=10*.

Текст программы:

```
Program Kat_Library;
Type
    Type_Publ=(Book,Journal,Newspaper);
    Liter=Record
            Title:string[50];
            Author:String[50];
            case V:Type_Publ of
            Book:(YearB:Integer);
            Journal:(Num:1..12;
                    YearJ:1900..2000);
            Newspaper:(Day:1..31;
                     Month:1..12;
                     YearN:Integer);
        end;
Const Count=10;
Var
Katalog:Array[1..Count] of Liter;
Num_Array:1..Count;
Yes_Liter:Boolean;
Vybor:Byte;
Edition:Type_Publ;
Count_Find:Integer;
Procedure Input_Data;
```

```
Begin
Writeln('Введите данные о литературе ',Num_Array,':');
Write('Введите число, указывающее вид издания:');
Write('1-книга, 2-журнал, 3-газета: ');
Readln(Vybor);
Case Vybor of
1:Katalog[Num_Array].V:=Book;
2:Katalog[Num_Array].V:=Journal;
3:Katalog[Num_Array].V:=Newspaper;
end;
With Katalog[Num_Array] do
begin
Write('Фамилия автора: ');
Readln(Author);
Write('Название:');
Readln(Title);
Case V of
Book:begin
Write('Год издания: ');
Readln(YearB);
end;
Journal:begin
Write('Номер: ');
Readln(Num);
Write('Год издания: ');
Readln(YearJ);
        end;
Newspaper:begin
        Write('Дата издания: День ');
        Readln(Day);
        Write('Месяц: ');
        Readln(Month);
        Write('Год: ');
        Readln(YearN);
end;
end;
end;
end;
Procedure Write_Data;
begin
        Writeln;
        Writeln('Литература № ',Num_Array);
With Katalog[Num_Array] do
        begin
```

```
Writeln('Название:',Title);
Writeln('Фамилия автора:',Author);
Case V of
Book:begin
    Writeln('Год издания:',YearB);
    end;
Journal:begin
     Write('Номер: ',Num);
     Writeln('Год издания: ',YearJ);
     end;
Newspaper:begin
Writeln('Дата издания: День:',Day,'Месяц:',Month,'Год:',YearN);
end;
end;
end;
end;
procedure Find_Liter;
begin
Writeln('Поиск литературы по типу издания: ');
Writeln;
Write('Введите число, указывающее вид издания: ');
Write('1-книга, 2-журнал, 3-газета');
Readln(Vybor);
case Vybor of
1:Edition:=Book;
2:Edition:=Journal;
3:Edition:=Newspaper;
end;
Yes_Liter:=False;
Count_Find:=0;
for Num_Array:=1 to Count do
If Katalog[Num_Array].V=Edition then
begin
Count_Find:=Count_Find+1;
Write_Data;
Yes_Liter:=True;
end;
if not Yes_Liter then
Writeln('В библиотеке нет такой литературы ')
else
Writeln('Всего в библиотеке',Count_Find,' таких изданий');
end;
begin
for Num_Array:=1 to Count do
```

```
Input_Data;
Writeln;
Find_Liter;
end.
```

Порядок выполнения работы

1. Изучить теоретические сведения по теме “ Работа с вариантными записями”.

2. Разработать программу для работы с записями с вариантами согласно полученному заданию.

3. Показать работающую программу преподавателю.

4. Ответить на контрольные вопросы.

Варианты заданий

1.Сформировать переменную типа запись, в которой расположены данные о каждом отдельном ученике в следующем порядке: имя (15 символов), фамилия (15 символов), год обучения (целое число), буква (символ). Требуется перенести эти данные в другую переменную выводя первую букву имени и фамилию ученика:

И. Петров

П. Иванов

и т. д.

2.Переменная содержит сведения об учениках некоторой школы (см. задачу 1).

а) Собрать в сведения об учениках девятых классов школы,

б) Выяснить, на сколько человек в восьмых классах больше, чем в девятых.

3.Багаж пассажира характеризуется количеством вещей и общим весом вещей. Сформировать переменную Bagaj, содержащую сведения о багаже нескольких пассажиров. Сведения о багаже каждого пассажира представляют собой запись с двумя полями: одно поле целого типа (количество вещей) и одно-действительное (вес в килограммах).

а) Найти багаж, средний вес одной вещи в котором отличается не более, чем на 0,3 кг от общего среднего веса одной вещи.

б) Найти число пассажиров, имеющих более двух вещей и число пассажиров, количество вещей которых превосходит среднее число вещей.

в) Выяснить, имеется ли пассажир, багаж которого состоит из одной вещи весом менее 30 кг.

4.Упорядочить сведения о багаже, записанные в переменной bagaje(см. предыдущую задачу) по не возрастанию веса багажа. Предполагается, что число пассажиров, зарегистрировавших багаж, известно заранее и равно n (некоторая константа), при этом n – не слишком велико. Указание. Перенести сведения о багаже из переменной багаж в массив B1,...Bn, затем упорядочить этот массив, используя то, что для переменных x, y одного и того же

комбинированного типа можно использовать оператор присваивания x:=y После этого переписать элементы массива BI, ..., Bn в переменную Bagaje.

5. Требуется удалить из данной переменной Bagaje сведения о багаже, общий вес вещей в котором меньше, чем 10 кг. Использовать вспомогательную переменную F.

6.Переписать сведения о багаже из переменной Bagaj в переменную Bag. В переменной Bag сведения о багаже каждого пассажира представляются массивом из двух целых чисел - числа вещей и общего веса вещей, выраженного в граммах. Составить также программу обратного преобразования: переписи сведений о багаже из переменной Bag в переменную Bagaj.

7. Сформирована переменная bibl, содержащий сведения о книгах. Сведения о каждой из книг – это фамилия автора, название и год издания.

а) Найти названия книг данного автора, изданных с 1960 года

б) Определить имеется ли книга с названием "Информатика". Если да, то сообщить фамилию автора и год издания. Если таких книг несколько, то сообщить сведения обо всех этих книгах.

8.Дана переменная T, которая содержит номера телефонов сотрудников учреждения: Указывается фамилия сотрудника, его инициалы и номер телефона. Найти номер телефона сотрудника по его фамилии и инициалам.

9. Сформирована переменная типа запись, содержащая различные даты. Каждая дата - это число, месяц и год. Найти:

а) год с наименьшим номером.

б) все весенние даты.

в) самую позднюю дату.

10.Сформировать переменную Tovar, содержащую сведения об экспортируемых товарах: Указывается наименование товара, страна импортирующая товар, и объем поставляемой партии в штуках. Составить список стран, в которые экспортируется данный товар, и общий объем его экспорта.

11.Сформирована переменная Assortim, содержащая сведения об игрушках: указано название игрушки, ее стоимость в рублях, и возрастные границы. Получить следующие сведения:

а) название игрушек цена которых не превышает 4 руб., и которые подходят детям 5 лет.

б) цену самого дорогого конструктора.

в) Название наиболее дорогих игрушек. (цена которых отличается не более чем на 1 руб. от самой дорогой.)

г) название игрушек которые подходят как детям 4 лет так и детям 10 лет.

д) можно ли подобрать игрушку, любую кроме мяча, подходящую ребенку 3 лет, и дополнительно мяч, так, чтобы суммарная стоимость игрушек не превосходила 5 руб.?

Контрольные вопросы

1. Понятие записи с вариантами. Структура объявления.
2. Правила использования записей с вариантами.
3. Пример программы с использованием записей с вариантами.

Лабораторная работа № 3
Программа с использованием множеств

Цель работы: формирование знаний и умений по работе с множествами. Приобретение навыков обработки множеств.

Краткие теоретические сведения

Множество – это структурированный тип данных, представляющий собой набор взаимосвязанных по какому-либо признаку или группе признаков объектов, которые можно рассматривать как единое целое. Каждый объект во множестве называется *элементом множества*.

Все элементы множества должны принадлежать одному из скалярных типов, кроме вещественного. Этот тип называется *базовым типом множества*. Базовый тип задается диапазоном или перечислением. *Область значений* типа множество – набор всевозможных подмножеств, составленных из элементов базового типа.

В выражениях на языке Паскаль значения элементов множества указываются в квадратных скобках:[1,2,3,4], [‘a’,’b’,’c’], [‘a’..’z’]. Если множество не имеет элементов, оно называется *пустым* и обозначается, как [ ]. Количество элементов множества называется его *мощностью*.

Формат записи множественных типов:

```
Type
  <имя типа> = set of <элемент 1,…., элементN>;
Var
<идентификатор, ….> :  <имя типа>;
```

Можно задать множественный тип и без предварительного описания:

```
Var
<идентификатор, …> :  set of <элемент1, …>;
```

Пример:

```
Type
   Simply  = set of  ‘a’ ..’h’;
   Number = set of  1..31;
Var
  Pr : Simply;
  N : Number;
  Letter : set of char; {определение множества без предварительного описания в разделе типов}
```

В данном примере переменная *Pr* может принимать значения символов латинского алфавита от ‘a’ до ‘h’; *N* – любое значение в диапазоне 1..31;

Letter – любой символ. Попытка присвоить другие значения вызовет программное прерывание.

Количество элементов множества не должно превышать 256, соответственно номера значений базового типа должны находиться в диапазоне 0..255. Контроль диапазонов осуществляется включением директивы {$R+}. Объем памяти, занимаемый одним элементом множества, составляет 1 бит.

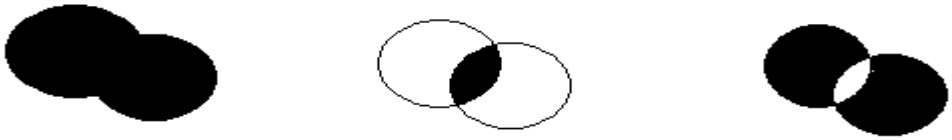
Объем памяти для переменной типа множество вычисляется по форму

Объем памяти = (Max DIV 8) – (Min DIV 8) + 1,

где *Max* и *Min* – верхняя и нижняя границы базового типа.

Результат операций рад двумя множествами можно наглядно представить с помощью закрашенных частей двух кружочков:

Объединение *Пересечение* *Разность*



Использование в программе данных типа **set** дает ряд преимуществ: значительно упрощаются сложные операторы if , увеличивается степень наглядности программы и понимания алгоритма решения задачи, экономятся память, время компиляции и выполнения.

Имеются и отрицательные моменты, основной из них – отсутствие в языке Паскаль средств ввода-вывода элементов множества, поэтому программист сам должен писать соответствующие процедуры.

Иллюстрацией описания и операций над множествами может служить следующий фрагмент программы:

```
Program Dem_Mno;   {демонстрация операций над множествами}
Type
   Digits=set of 0..9;
Var
   D1, D2, D3, D:Digits;
Begin
    D1:=[2,4,6,8];       {заполнение множеств}
    D2:=[0..3,5];
    D3:=[1,3,5,7,9];
    D:=D1+D2;      {объединение множеств D1 и D2}
    D:=D+D3;         {объединение множеств D и D3 }
    D:=D-D2;         {разность множеств D и D2 }
    D:=D*D1;         {пересечение множеств D и D1}
end.
```

Как видно из текста программы, сначала описан тип *Digits=set of 0..9,* затем описаны переменные *D1,D2,D3,D* этого типа. В первой части программы осуществляется заполнение множеств, а затем над множествами выполняются операции объединения, пересечения, разности.

Вторым примером работы с множествами может служить следующая задача: описать множество M (1..50) и сделать его пустым. Вводя целые числа с клавиатуры, заполнить множество 10 элементами.

```
Program Input_Mno;
Var
    M:set of 1..50;
    X,I:integer;
Begin
    M:= [ ];                {M – пустое множество}
    for I:=1 to 10 do
        begin
          write('введите ', I, ' –й элемент множества: ');
          readln (X);
          if (X in M) then    {если введенное число входит в множество M}
                    begin
                    writeln(X,' помещен в множество 1..50');
                    M:=M+[X];
                    end;
        end;
writeln;
end.
```

В разделе описания переменных описано множество целых чисел от 1 до 50, переменная X целого типа, которая используется для считывания числа-кандидата в множество, и целая переменная I, используемая для подсчета количества введенных чисел. В начале программы применена операция инициализации множества M, так как оно не имеет элементов и является пустым:

```
M:= [ ];
```

Заполнение множества элементами производится с использованием оператора повтора for, параметр которого I будет указывать порядковый номер вводимого элемента. Операция заполнения множества записывается оператором присваивания:

```
M:=M+[X];
```

Контроль заполнения множества записан операцией проверки принадлежности in. Если условие X in M выполняется, выводится сообщение о том, что число X помещено в множество.

Третий пример демонстрирует описание множества гласных и согласных букв русского языка и определяет количество гласных и согласных букв в предложении, введенном с клавиатуры пользователем.

Зададим тип Letters –множество букв русского языка, затем опишем переменные этого типа: Glasn-множество гласных букв, Sogl-множество согласных букв. Вводимое с клавиатуры предложение опишем переменной Text типа String. Для указания символа в строке Text применим переменную I типа Byte.Для подсчета количества гласных и согласных букв опишем

переменные Gи S. Проверку принадлежности символов, составляющих предложение множествам гласных или согласных букв русского языка запишем с использованием цикла for , параметр I которого, изменяясь от 1 до значения длины предложения, будет указывать порядковый номер символа в предложении. Принадлежность очередного символа предложения множеству гласных или согласных букв запишем операцией in. Если символ является гласной буквой (Text[I] in Glasn), то счетчик гласных букв G увеличивается на 1. Аналогично с согласными буквами. Текст программы может выглядеть так:

```
Program Glasn_Sogl;
Type
    Letters=set of ‘А’..’я’;
Var
    Glasn, Sogl:Letters;
    Text:String;
    I:Byte;
    G,S:Byte;
Begin
    Glasn:=[‘А’,’а’, ‘Е’,’е’,’И’,’и’,’О’,’о’,’У’,’у’,’Э’,’Ю’,’ю’, ‘Я’,’я’];
Sogl:=[‘Б’..’Д’,’б’..’д’,’Ж’,’ж’,’З’,’з’,’К’..’Н’,’к’..’н’,’П’..’Т’,’п’..’т’,’Ф’..’Щ’,’
‘ф’..’щ’,’Ъ’,’ъ’,’Ь’,’ь’];
    Write(‘Введите предложение’);
    Readln(Text);
    G:=0;
    S:=0;
    For I:=1 to Length(Text) do
      Begin
        If Text[I] in Glasn then G:=G+1;
        If Text[I] in Sogl then S:=S+1;
      End;
Writeln(‘В предложении” ’,Text,’ ”‘,G,’гласных и ‘,S,’согласных букв’);
End.
```

Порядок выполнения работы

1. Изучить теоретические сведения по теме “ Работа с множествами”.
2. Разработать программу для работы с множествами, используя возможные операции.
3. Показать работающую программу преподавателю.
4. Ответить на контрольные вопросы.

Варианты заданий

1. Заданы имена девочек. Определить, какие из этих имен встречаются во всех классах данной параллели, которые есть только в некоторых классах и какие из этих имен не встречаются ни в одном классе.

2. Задан некоторый набор товаров. Определить для каждого из товаров, какие из них имеются в каждом магазине и каких товаров нет ни в одном магазине.

3. Дан текст, за которым следует точка. В алфавитном порядке напечатать все строчные русские гласные буквы (а, е, и, о, у, ы, э, ю, я), входящие в этот текст.

4. Имеется множество, содержащее натуральные числа из некоторого диапазона . Сформировать два множества, первые из которых содержит все простые числа из данного множества, а второе – все составные.

5. Известны марки машин, изготовляемых в данной стране и импортируемых за рубеж. Даны некоторые К стран. Определить для каждой из марок, какие из них были:1) доставлены во все страны; 2) доставлены в некоторые из стран; 3) не доставлены ни в одну страну.

6. В озере водится несколько видов рыб. Три рыбака поймали рыб, представляющих некоторые из имеющихся видов. Определить: какие рыбы есть в озере, но нет ни у одного их рыбаков.

7. Дан текст из строчных латинских букв и цифр. Определить чего – букв или цифр – больше в этом тексте.

8. В N колхозах выращивают некоторые сельскохозяйственные культуры из имеющегося перечня. Определить культуры: возделываемые во всех колхозах; возделываемые только в некоторых колхозах.

9. Подсчитать количество различных цифр в десятичной записи натурального числа.

10. Есть список игрушек, некоторые из которых имеются в n детских садах. Определить игрушки из списка: которых нет ни в одном из детсадов; которые есть в каждом из детсадов.

11. Напечатать в порядке убывания все цифры , входящие в запись данного натурального числа.

12. Составить программу, которая вычисляет сумму тех элементов двумерного массива, номера строк и столбцов которых принадлежат соответственно непустым множествам S1иS2

13. Задан год рождения. Определить, сколько человек в списке жильцов студенческого общежития родились в этот год.

14. Задано некоторое множество M и множество T того же типа. Подсчитать, сколько элементов из множеств T и M совпадает.

Контрольные вопросы

1. Множества. Основные понятия и определения.
2. Операции над множествами.
3. Фрагменты программ с использованием множеств.

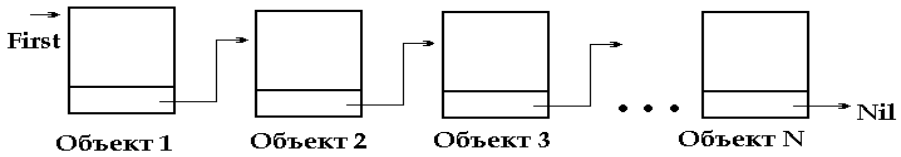
Лабораторная работа № 4
Разработка программы создания связанного списка

Цель работы: формирование знаний и умений по работе с динамической памятью. Приобретение навыков работы с динамическими структурами данных.

Краткие теоретические сведения

Использование указателей для организации связанных списков

Чаще всего указатели используются для ссылки на записи, тем самым достигается значительная экономия памяти. Если же сама запись содержит в



себе поле-указатель, указывающий на следующую за ней запись, то это позволяет образовать *связанные списки* – структуру, в которой отдельные записи последовательно связаны друг с другом. В приведенном ниже примере используются записи, в которых наряду с данными об автомобиле имеется указатель на следующую запись, благодаря чему получается связанный список. Одно из полей каждого объекта связанного списка имеет тип указатель и указывает на очередной объект в списке. Указатель на первый объект содержится в переменной *First*, а последний объект имеет указатель на *Nil*.

Пример программы создания и использования связанного списка

Пусть требуется создать связанный список из записей, содержащих сведения об автомобилях, а также реализовать операции со связанным списком: запись первым в список, удаление первого объекта из списка, просмотр всего списка, удаление объекта, следующего за указанным.

```
{Пример использования указателей для обработки связанного списка}
Program point;
Uses Crt;
Type
NameStr = String [20];
Link = ^Auto;
Auto = record
Name : NameStr; {Марка автомобиля}
Speed : real;   {Скорость}
Next : Link;    {Поле для связи со следующим объектом в списке}
end;
Var
P,First : Link;   {Указатели на запись: текущую, первую}
NamFind : NameStr;     {Марка автомобиля для поиска}
V : 0..4;              {Селектор меню}
```

```
EndMenu : boolean;      {Окончание вывода меню}
{Поиск объекта с именем FN, по результатам поиска возвращает указатель на найденный объект или Nil, если объект не найден}
Function FindName(FN:NameStr) : Link;
Var
Curr : Link;
begin
Curr:=First;    {Установить указатель на первом объекте в списке }
{Повторять пока не дойдем до конца списка}
while Curr <>       Nil do
if Curr^.Name=FN then {Если нашли заданный объект}
begin
FindName:=Curr;       {Возвращаем значение указателя на него}
Exit;       {Завершаем функцию}
end
else
Curr:=Curr^ .Next;      {Перейти к следующей записи}
FindName:=Nil;       {В списке нет искомого объекта}
end; {Конец FindName}
{Добавление записи первой в связанный список}
procedure AddFirst(A:Link);
begin
A^. Next:=First;     {Новый объект первый в списке}
First:=A;             {Голова списка ссылается на новый объект}
end;                 {Конец AddFirst}
{Удаление первого объекта из списка}
procedure DelFirst(var A:Link);
begin
A:=First;
First:=First^. Next; {Теперь First указывает на тот объект, на который ранее ссылался объект A}
end; {Конец DelFirst}
{Удаление из списка объекта, стоящего после объекта Old}
procedure DelAfter(Old:Link; var A:Link);
begin
A:=Old^.Next;    {Переменной A присваивается значение указателя на удаляемый объект}
Old^.Next:=Old^.Next^.Next; {Теперь Old указывает на тот объект, на который ранее ссылался следующий за ним объект, а объект A исключен из связанного списка}
end; {Конец DelAfter}
{Ввести данные об объекте}
procedure InpAvto;
  begin
```

```
P:=New(Link);      {Создать очередной объект типа Auto}
Write('Введите марку автомобиля :');
Readln(P^.Name) ;
Write('Максимальная скорость :');
Readln(P^.Speed);
AddFirst(P);      {Вызов процедуры добавления записи, на которую ссылается указатель P (P- фактический параметр, A - формальный параметр-значение) }
end;          {Конец InpAvto}
{Вывести на экран все объекты из связанного списка}
procedure MyList;
var
 Curr : Link; {Локальная переменная - указатель на очередной объект}
begin
Curr:=First;  {Установить указатель на первом объекте в списке}
{Повторять, пока не дойдем до конца списка}
while Curr <> Nil do
begin
Writeln( 'Марка : ' , Curr^. Name,' скорость : ', Curr^. Speed) ;
Curr:=Curr^.Next;  {Перейти к очередному объекту связанного списка}
end ;
Write('Вывод списка окончен. Нажмите Enter');
Readln;
end;     {Конец MyList}
Begin     {Основная программа}
New(P);  {Создать новую динамическую переменную и установить на нее переменную-указатель}
EndMenu:=False ;
repeat        {Очищать экран и выводить меню до тех пор, пока EndMenu<>True}
CIrScr;
Writeln('Укажите вид работы:');
Writein('1. Запись первым в список');
Writeln('2. Удаление первого объекта из списка');
Writein('3. Просмотр всего списка') ;
Writein('4. Удаление объекта, следующего в списке за указанным') ;
Writein('0. Окончание работы');
Readin(V) ;
Case V of {Вызов разных процедур в зависимости от выбора пункта меню}
1 : InpAvto;       {Ввод данных об объекте}
2 : DelFirst(P);   {Удаление первого в списке}
3 : MyList; {Вывод списка всех элементов связанного списка}
4 : begin {Удаление объекта, следующего за указанным}
```

```
Write('Введите марку автомобиля, за которым следует удаляемый из списка :');
Readln(NamFind) ;
DelAfter(FindHame(NamFind),P); {Вызов процедуры
DelAfter с параметрами: функцией FindName(NamFind) и указателем P}
end
else
EndMenu:=True; {Завершить вывод меню}
end;
until EndMenu;  {Если EndMenu=True, то завершить вывод меню на экран}
Dispose(P); {Уничтожить динамическую переменную P и освободить память в куче}
end.
```

Порядок выполнения работы

1. Изучить теоретические сведения по теме ” Разработка программы создания связанного списка”.
2. Получить у преподавателя индивидуальное задание и разработать программу для работы со связанным списком согласно заданному варианту.
3. Показать работающую программу преподавателю.
4. Ответить на контрольные вопросы.

Варианты заданий

1. Напишите программу подсчета суммарного числа букв '**a**' и букв '**b**' в данной строковой переменной. Вывести на экран каких букв больше.
2. Задано предложение **y**, состоящее из слов-строк. Проверить, встречается ли данное слово **x** в предложении **y**.
3. Предложение содержит буквы латинского и русского алфавитов. Написать программу, которая выводит буквы только латинского алфавита в порядке их следования в предложении.
4. Дано предложение-строка. Подсчитать количество слов, начинающихся с буквы '**a**'.
5. Написать программу, подсчитывающую, сколько раз в данном слове **x** встречается (в качестве его части) слово **y**.
6. Написать программу, которая каждый встречающийся в строке заданный символ заменяет на заданную последовательность символов, расширяя при этом строку.
7. Задано предложение-строка. Написать программу, которая находит самое длинное слово, встречающееся в предложении.
8. Написать программу, вычеркивающую из данного текста все буквы '**a**'.

9. Написать программу, которая проверяет в строке баланс открывающихся и закрывающихся круглых скобок (строка содержит арифметическое выражение).

10. Написать программу, которая каждую встреченную букву '**б**' заменяет сочетанием '**ку**'.

11. Задано предложение, состоящее из слов-строк. Написать программу, которая находит самое короткое слово в предложении.

12. Предложение состоит из слов-строк. Написать программу, которая подсчитывает количество слов в предложении.

13. Написать программу, проверяющую, является ли частью данного слова слово '**сок**'. Ответ должен быть '**да**' или '**нет**'.

14. Даны две строки. Вычеркнуть из строки **A** символы, встречающиеся в строке **B**.

15. Из данного предложения вычеркнуть слова, встречающиеся больше одного раза.

Контрольные вопросы

1. Связанные списки.
2. Применение указателей для организации связанных списков.

Лабораторная работа 5.
Стеки и очереди

Цель работы: 1) ознакомиться со способами реализации стеков и очередей и выполнения над ними операций; 2) получить практические навыки в программировании с использованием стеков и очередей.

Необходимые исходные сведения

Стеки и очереди – это динамические структуры данных с ограниченным видом операций, включением новых и выключением старых переменных.

Стек представляет собой последовательность элементов с одной точкой доступа, называемой вершиной стека. Все элементы последовательности, кроме элемента, расположенного в вершине стека, недоступны. Новый элемент добавляется только в вершину стека, сдвигая остальные элементы последовательности. Исключить можно только элемент из вершины стека. Остальные элементы при этом сдвигаются в сторону вершины. Таким образом, стек работает по принципу «последним пришел – первым ушел» и часто называется структурой LIFO (Last In – First Out). В алгоритмах операцию добавления элемента в стек записывают в виде S←X (элемент X поместить в вершину стека S); операцию исключения в виде X←S (исключить элемент X из вершины стека S и присвоить его значение переменной X). В литературе эти операции традиционно называются PUSH (добавление) и POP (исключение). Очередь представляет собой последовательность элементов с двумя точками доступа: начало (голова) и

конец (хвост). Новый элемент добавляется всегда в конец очереди, исключается всегда элемент, расположенный в начале очереди. Таким образом, очередь работает по принципу «первым зашел – первым ушел» и часто называется структурой FIFO (First In – First Out). В алгоритмах операцию включения элемента в очередь записывают в виде Q←X (элемент X помещается в конец очереди Q), а операцию исключения из очереди – в виде X←Q (исключить элемент X из начала очереди Q и присвоить его значение переменной X). Стеки и очереди удобно реализовывать с помощью указателей, т.к. указатели имеют весьма гибкую структуру.

Описание типа стек:

```
type
  ct=^celltype;
  celltype=record
             element:byte;
             next:ct;
             end;
  stack=record
          top:ct;
          fin:ct;
          end;
```

где top – указатель на вершину стека.

Описание типа очередь:

```
type
  ct=^celltype;
  celltype=record
             element:byte;
             next:ct;
             end;
  queue=record
          front:ct;
          rear:ct;
        end;
```

где front – указатель на начало очереди, rear – указатель на конец очереди.

Пример выполнения

Задание: написать программы, реализующие операции над стеками и очередями, с помощью указателей:

1) функция STACKTOP(Q), которая определяет значение элемента в вершине стека Q без его удаления (не определена для пустого стека);

2) процедура INSERT(S,X), которая помещает элемент X в конец очереди S;

3) функция REMOVE(S), которая удаляет элемент из начала очереди S, т.е. операция X:=REMOVE (S) удаляет элемент из начала очереди S и присваивает его значение переменной X (не определена для пустой очереди).

Реализация функции STACKTOP(Q):

```
function stacktop(var s:stack):byte;
begin
    stacktop:=s.top^.next^.element;
end;
```

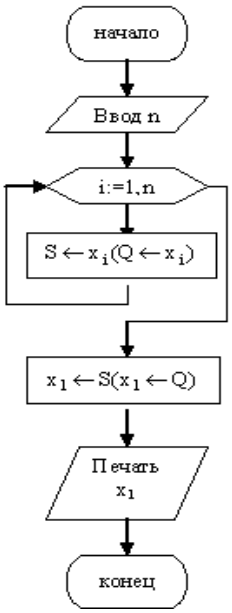
Реализация процедуры INSERT(S,X):

```
procedure insert(var q:queue;x:byte);
begin
    new(q.rear^.next);
    q.rear:=q.rear^.next;
    q.rear^.element:=x;
    q.rear^.next:=nil;
end;
```

Реализация функции REMOVE(S):

```
function remove(var q:queue):byte;
begin
    remove:=q.front^.next^.element;
    delstart(q);
end;
```

Блок-схема алгоритма



Текст программы:

Очередь:

```
uses
   crt;
const
    n=10;
type
   ct=^celltype;
   celltype=record
```

```
                element:byte;
                next:ct;
                end;
   queue=record
            front:ct;
            rear:ct;

            end;
var
  s:queue;
  x:byte;

procedure insert(var q:queue;x:byte);
begin
    new(q.rear^.next);
    q.rear:=q.rear^.next;
    q.rear^.element:=x;
    q.rear^.next:=nil;
end;
procedure create(var q:queue);
var
  i,x:byte;
begin
    new(q.front);
    q.front^.next:=nil;
    q.rear:=q.front;
    for i:=1 to n do
    begin
        x:=round(random*100);
        insert(q,x);
    end;
    writeln('QUEUE IS CREATE.');
end;

procedure delstart(var q:queue);
begin
    q.front:=q.front^.next;
end;

function remove(var q:queue):byte;
begin
    remove:=q.front^.next^.element;
    delstart(q);
end;
```

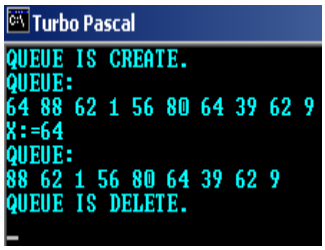
```
procedure printqueue(var q:queue);
var
  temp:queue;
begin
    writeln('QUEUE:');
    temp:=q;
    while temp.front^.next<>nil do
    begin
        write(temp.front^.next^.element,' ');
        temp.front:=temp.front^.next;
    end;
    writeln;
end;
procedure delqueue(var q:queue);
begin
    while q.front^.next<>nil do delstart(q);
    writeln('QUEUE IS DELETE.');
end;

begin
    clrscr;
    randomize;
    create(s);
    printqueue(s);
    x:=remove(s);
    writeln('X:=',x);
    printqueue(s);
    delqueue(s);
    readkey;
end.
```

Стек:

```
uses
   crt;
const
    n=10;
type
   ct=^celltype;
   celltype=record
                element:byte;
                next:ct;
                end;
   stack=record
            top:ct;
            fin:ct;
```

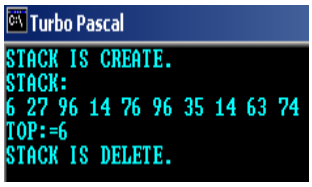
```
            end;
var
  q:stack;
  x:byte;

procedure push(var s:stack);
begin
    new(s.fin^.next);
    s.fin:=s.fin^.next;
    s.fin^.element:=round(random*100);
    s.fin^.next:=nil;
end;

procedure create(var s:stack);
var
  i:byte;
begin
    new(s.top);
    s.top^.next:=nil;
    s.fin:=s.top;
    for i:=1 to n do push(s);
    writeln('STACK IS CREATE.');
end;
procedure pop(var s:stack);
var
  temp:stack;
begin
    temp:=s;
    while temp.top^.next<>s.fin do temp.top:=temp.top^.next;
    s.fin:=temp.top;
end;

function stacktop(var s:stack):byte;
begin
    stacktop:=s.top^.next^.element;
end;

procedure printstack(var s:stack);
var
  temp:stack;
begin
    writeln('STACK:');
    temp:=s;
    while temp.top^.next<>nil do
```

```
        begin
            write(temp.top^.next^.element,' ');
            temp.top:=temp.top^.next;
        end;
        writeln;
    end;

    procedure delstack(var s:stack);
    begin
        s.top^.next:=nil;
        writeln('STACK IS DELETE.');
    end;

    begin
        clrscr;
        randomize;
        create(q);
        printstack(q);
        x:=stacktop(q);
        writeln('TOP:=',x);
        delstack(q);
        readkey;
    end.
```



Результаты расчетов

Для очереди S: 64,88,62,1,56,80,64,39,62,9 результат работы функции REMOVE(S) будет следующим: X=64, S: 88,62,1,56,80,64,39,62,9.



Для стека Q: 6,27,96,14,76,96,35,14,63,74 результат работы функции STACKTOP(Q) будет следующим: TOP=6.

Варианты заданий

1.Разработать реализации для очередей, состоящих из действительных чисел, с использованием массивов и указателей

2.Реализовать двухстороннюю очередь, если элементами являются действительные числа

3.Написать программу поиска элементов в позициях p для стека

4.Разработать реализации для стека, состоящего из действительных чисел, с использованием массивов и указателей

5.Написать программу поиска элементов в позициях p для очереди

6.Написать программу сортировки элементов для очереди

7.Написать программу обмена элементами в позициях p и NEXT(p) для стека

8.Написать программу обмена элементами в позициях p и NEXT(p) для очереди

9.Написать программу сложения и умножения двух стеков

10.Написать программу сложения и умножения двух очередей

Контрольные вопросы

1. Дайте характеристику основных структур данных. В каких алгоритмах используются основные типы данных?
2. Назовите отличительные особенности типа данных Массив, Запись и Множество. В чем их различия?
3. Как реализуются динамические структуры данных? В чем их отличие от статических структур?
4. В каких алгоритмах целесообразнее использовать динамические структуры данных?
5. Назовите отличительные особенности типа данных Линейный список. Как реализуются алгоритмы, использующие линейные списки?
6. Назовите отличительные особенности типа данных Циклический список. Как реализуются алгоритмы, использующие циклические списки?
7. Для чего создаётся пользовательский тип данных Мультисписок?
8. Как представить стек и очередь в виде списков?
9. Какие особенности у алгоритмов, использующих тип данных Стек?

Лабораторная работа № 6
Реализация алгоритма бинарного поиска

Цель работы: формирование знаний и умений по изучению метода бинарного поиска. Приобретение навыков реализации алгоритма бинарного поиска.

Основные понятия и определения

Поиском называется алгоритм, который на входе воспринимает некоторое значение *X* и определяет запись, ключ которой совпадает со значением *X*. При этом на выходе такой алгоритм может выдать либо найденную запись, либо указатель на нее. *X* называется *аргументом поиска.*

Если в таблице имеется запись с ключом, равным *X*, то поиск называется *удачным* или *результативным*. В противном случае поиск называется *неудачным (безрезультатным).*

В дальнейшем будем рассматривать массив *A* с элементами *A[1], A[2] ... A[N]*, в котором индекс изменяется от 1 до N. Кроме того, считаем, что *A[i]*- и есть ключ i-того элемента массива.

Линейный поиск

Применяется в тех случаях, когда о характере расположения ключей нет никакой информации. Считается, что ключи не упорядочены. Тогда, единственный способ поиска это сравнение *X* с *A[1]*. Если они не равны, тогда *X* сравнивается с *A[2]*. Если *X=A[i]*, и ключ первичный (каждое его значение уникально в массиве), то поиск прекращается.

Линейный поиск в упорядоченном массиве данных

Пусть для элементов массива *A* выполняется условие:

A[1]<A[2]<A[3]<…< A[n] *(1)*

Ключи упорядочены по *возрастанию*. В произвольном массиве в случае неудачного поиска приходится просматривать массив от начала до конца. В упорядоченном массиве достаточно определить момент выполнения неравенства *X< A[i]*, после чего поиск следует закончить, т.к. *A[i+1] … A[n]* будут заведомо больше *X*.

Бинарный (двоичный) поиск

Данный поиск называется *делением пополам (метод дихотомии*). Тоже выполняется в упорядоченных массивах, для элементов которого выполняется условие (1). *X*- аргумент поиска.

Определяется *m=N/2* - целая часть. Рассматривается *A[m]*-срединный элемент исходной последовательности. Если *A[m]=X*, то поиск закончен, он результативен. Если *A[m]<X*, то из поиска исключаются все элементы, от *A[1] до A[m]*, т.к. они заведомо меньше *X*. Если *A[m]>X*, то из поиска исключаются все элементы, *от A[m] до A[n]*. Поиск продолжается в оставшейся подпоследовательности (половине): определяется значение *m*-индекс элемента, и если его ключ не равен *X*, то из поиска исключаются одна из половинок и т.д.

Пусть дан упорядоченный массив из 10 элементов (N=10):

1 2 3 4 5 6 7 8 9 10

Пусть необходимо в данном массиве найти цифру 6. Определяем m=10/2=5.

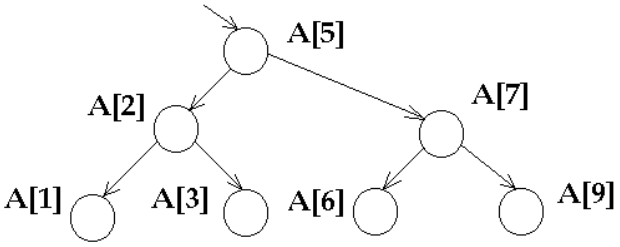
Рассматриваем A[5]=5 (A[m]). В данном случае A[m] не равно искомому элементу (5 не равно 6), и проверяем A[m]<X ? Условие выполняется (5<6) и из поиска исключаются все элементы от 1 до 5. Поиск продолжается в оставшейся половинке.

6 7 8 9 10 (N=5)

Определяем m=5/2=2 (целая часть). Рассматриваем A[2]=7. Так как A[m]>X (7>6), то из поиска исключаются все элементы от 7 до 10. Поиск продолжается в последовательности элементов, в данном случае состоящей из одного элемента равного 6, который и является аргументом поиска. Поиск закончен.

Описанную процедуру можно представить в виде дерева, в корень которого помещаем срединный элемент всей последовательности, его

сыновьями будут срединные элементы половинок. Сыновьями вершин первого уровня будут срединные элементы четвертинок.



Тогда поиск выполняется, начиная с корня. По достижении каждого узла определяется направление дальнейшего поиска. Если аргумент поиска меньше ключа, размещенного в узле, то поиск выполняется в левом поддереве. Если больше , то в правом поддереве. Подобного рода древовидная структура называется *деревом поиска.*

Пример 1 программы с использованием алгоритма бинарного поиска

```
Program DemoSearch_2;
Uses Crt;
Const
  Maximum=100;{максимальное значение элементов массива}
Type
  DataType=Integer;
  DataIn=array [1..Maximum] of Integer;{пользовательский тип DataIn,
который обозначает массив из элементов типа Integer}
Var
  WorkArray:DataIn;{объявляем массив типа DataIn}
  i:Byte;
{Объявляем функцию Poisk2, у которой 3 формальных параметра:
1-массив, 2-количество элементов массива, 3-ключ поиска}
Function Poisk2(Vx:DataIn;N:Integer;K1:DataType):Integer;
{Poisk2}
Var
   L,U,Middle:Integer;
   Rez:Boolean;
   j:Integer;
begin
L:=1;
U:=N;
Rez:=False;
While (L<=U) and (not Rez) do
      begin
 {Средний элемент массива}
 Middle:=(L+U)div 2;
{если ключ меньше среднего элемента массива, то поиск происходит
в правой оставшейся части массива}
```

```
        If K1<Vx[Middle] then U:=Middle-1
        else
          If K1>Vx[Middle] then L:=Middle+1
          else
                    Rez:=True;
                  end;
                    If Rez=True then Poisk2:=Middle
                    else
                    Poisk2:=0;
                  end;{Poisk2}
      {Основная программа}
Begin
ClrScr;
for i:=1 to 10 do
begin
       writeln('Введите ',i,'-й элемент массива');
        readln(WorkArray[i]);
end;
Case Poisk2(WorkArray,10,6) of
0:Writeln('Такого значения нет')
else
Writeln('Впервые цифра 6 появилась под номером',Poisk2(WorkArray,10,6));
end;
repeat until keypressed;
End.
```

Пример 2

Пусть $a_1, a_2, \ldots, a_n$ –целочисленный массив и $a_1, a_2, \ldots, a_n$; b – целое число. Рассмотрим задачу: выяснить, входит ли данное число в массив $k = ceil(\log_2(n+1))$ и если входит, то каково значение p, для которого ap = b? Эту задачу мы назовем задачей поиска элемента.

Областью поиска служит n-элементный ряд, предварительно отсортированный. Вначале диапазон поиска определяется граничными значениями Low = 1 и High = n. Строим первую «гипотезу», полагая, что искомое значение находится в позиции Test – где-то посередине между границами Low и High. Если проверка подтверждает это, то поиск успешно завершен. Если значение в позиции Test меньше искомого, нижней границей (Low) становится Test + 1, а если больше, то модифицируется верхняя (High) граница – для нее принимается значение, равное

Test – 1. Описанный процесс постепенного сближения границ поиска с последующей проверкой среднего элемента повторяется многократно, завершаясь одним из двух исходов: либо нужное значение обнаруживается (успех), либо Low становится больше High (неудача).

Максимальное количество проверок, необходимое для завершения бинарного поиска (при любом исходе – удачном или неудачном), определяется по формуле, где ceil обозначает функцию, возвращающую ближайшее целое число, большее или равное ее аргументу. Другими словами, k есть наименьшее целое, такое, что 2k равно или больше, чем n + 1. Например, если n = 100, для сортировки потребуется не более семи проверок, поскольку 27 = 128.

Пусть b = 6, а массив a состоит из 10 элементов: 3 5 6 8 12 15 17 18 20 25.

Шаг 1. Найдем номер среднего элемента: $Test = \left[\frac{1+10}{2}\right] = 5$ (Low=1, High=10). Так как 6<="" div="">

Шаг 2. Рассматриваем лишь первые 4 элемента массива; находим индекс среднего элемента этой части $Test = \left[\frac{1+4}{2}\right] = 2$ (Low=1, High=4). 6>a[2], следовательно, первый и второй элементы из рассмотрения исключаем: 3 5 6 8 12 15 17 18 20 25.

Шаг 3. Рассматриваем два элемента; значение $Test = \left[\frac{3+4}{2}\right] = 3$ (Low=3, High=4): 3 5 6 8 12 15 17 18 20 25. a[3] = 6! Элемент найден, его номер – 3.

Порядок выполнения работы

1. Изучить теоретические сведения по теме “Алгоритм бинарного поиска”.
2. Получить у преподавателя индивидуальное задание и разработать программу по заданному варианту.
3. Показать работающую программу преподавателю.
4. Ответить на контрольные вопросы.

Варианты заданий

Для всех вариантов предварительно написать процедуру поиска

1. Опишите массив записей, содержащих фамилию абонента и номер его телефона. Запрограммируйте двоичный поиск в телефонном справочнике.

2. Индексом называется таблица, содержащая отсортированные значения некоторых ключей и их местоположение в массиве записей. Индексом пользуются для ускорения поиска в массиве (сам массив может быть не отсортирован). Запрограммируйте процедуру составления индекса и

2.1. Последовательного поиска при помощи индекса.

2.2. Бинарного поиска при помощи индекса.

3. Пусть задан в виде последовательности символов некоторый текст T, состоящий из слов и есть два списка из нескольких слов в виде двух массивов A и B. Написать программу, преобразующую текст T в текст S, путем замены каждого вхождения слова A[i] на соответствующее слово B[i].

4. В нашем примере поиск проводился в массиве, упорядоченном по возрастанию. Напишите процедуру бинарного поиска в массиве, упорядоченном по убыванию.

5. Дан массив из строк (например, фамилий). Отсортировать его по алфавиту и написать процедуру вставки новой фамилии после заданной так, чтобы алфавитный порядок не нарушился. Предусмотреть ситуацию, когда массив заполнен «до отказа» и вставка нового элемента невозможна.

6. Имеется железнодорожное расписание, содержащее номер рейса поезда, времена отправления и прибытия и станцию прибытия. Организовать поиск номера поезда, время отправления и прибытия, если задана станция.

7. Компания с целью определения спроса на свою продукцию организует некоторый опрос. Продукция – компакт-диски с записями шлягеров. Все опрашиваемые делятся на две группы в соответствии с полом. Каждый опрашиваемый должен назвать три песни, которые идентифицируются номерами от 1 до N (пусть N = 10). Файл с данными обрабатывается программой, которая должна печатать:

7.1. Список песен в порядке их популярности. Каждая строка содержит название песни и число упоминаний ее при опросе. Песни, которые ни разу не упоминались, в список не включаются.

7.2. Три наиболее популярные песни (хиты) и два списка (в соответствии с группами) с именами всех тех ответивших, которые поставили на первое место один из этих трех хитов.

Контрольные вопросы

1. Основные понятия поиска данных.
2. Различные методы поиска данных. Краткое описание алгоритмов.
3. Бинарный поиск. Описание алгоритма и пример программы.

Лабораторная работа 7.
Сортировка значений в таблице

Цель работы: изучить методы сортировки массивов.

Краткие теоретические сведения

Сортировка представляет собой процесс упорядочения множества подобных информационных объектов в порядке возрастания или убывания их значений. Например, список i из n элементов будет отсортирован в порядке возрастания значений элементов, если i <= i <= ... <= I.

В общем случае при сортировке данных только часть информации используется в качестве ключа сортировки, который используется в сравнениях. Когда выполняется обмен, передается вся структура данных. Например, в списке почтовых отправлений в качестве ключа сортировки может использоваться почтовый индекс, а в операциях обмена к почтовому индексу добавляется полное имя и адрес.

Классы алгоритмов сортировки

Имеется три способа сортировки массивов:

– сортировка выбором;

– сортировка вставкой.

Представьте, что лежит колода карт. Для сортировки карт *обменом* необходимо разложить карты на столе лицевой стороной вверх и затем менять местами те карты, которые расположены в неправильном порядке, делая это до тех пор, пока колода карт не станет упорядоченной.

Для сортировки *выбором* необходимо разложить карты на столе, выбрать самую младшую карту и взять ее в свою руку. Затем из оставшихся на столе карт вновь выбрать наименьшую по значению карту и поместить ее позади той карты, которая уже имеется в руке. Этот процесс надо продолжать до тех пор, пока все карты не окажутся в руках. Поскольку каждый раз выбирается наименьшая по значению карта из оставшихся на столе, по завершении такого процесса карты в руке будут отсортированы.

Для сортировки *вставкой* необходимо держать карты в своей руке, поочередно снимая карту с колоды. Каждая взятая вами карта помещается в новую колоду на столе, причем она ставится на соответствующее место. Колода будет отсортирована, когда у вас в руке не окажется ни одной карты.

Сортировка Шелла

Сортировка Шелла получила свое название по имени ее создателя Д.Л. Шелла. Однако это название можно считать удачным, так как выполняемые при сортировке действия напоминают укладывание морских ракушек друг на друга ("ракушка" – одно из значений слова shell).

Общий метод, который использует сортировку вставкой, применяет принцип уменьшения расстояния между сравниваемыми элементами. Далее показана схема выполнения сортировки Шелла для массива "оасве". Сначала сортируются все элементы, которые смещены друг от друга на три позиции. Затем сортируются все элементы, которые смещены на две позиции. И, наконец, упорядочиваются все соседние элементы.

- проход 1: f d a c b e
- проход 2: c b a f d e
- проход 3: a b c e d f
- полученный результат: a b c d e f

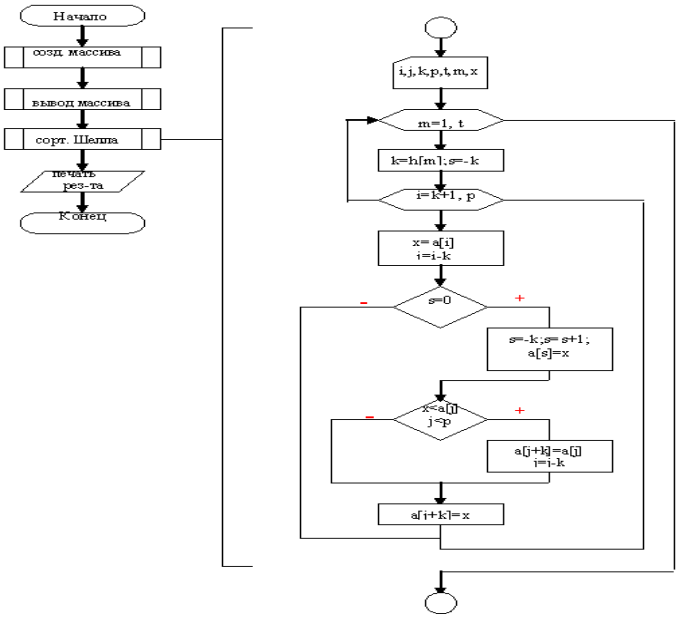
```
procedure shell(var a:massiv;p:word);
const
    t=5;
var
  i,j,k,s,m:integer;
  h:array[1..t] of integer;
  x:data;
begin
    writeln('Сортировка Шелла:');
    h[1]:=9; h[2]:=5; h[3]:=3; h[4]:=2; h[5]:=1;
```

```
    for m:=1 to t do
    begin
        k:=h[m];
        s:=-k;
        for i:=k+1 to p do
        begin
            x:=a[i];
            j:=i-k;
            if s=0 then
            begin
                s:=-k;
                s:=s+1;
                a[s]:=x;
            end;
            while (x<a[j]) and (j<p) do
              begin
                  a[j+k]:=a[j];
                  j:=j-k;
              end;
            a[j+k]:=x;
        end;
    end;
end;
```

Пример выполнения

Задание: составить алгоритм и программу сортировки массива методом Шелла.

Блок-схема алгоритма



Текст программы:

```
uses crt;
type
   data=integer;
   massiv=array[1..15] of integer;
var
  a,b:massiv;
  n,i:word;
procedure create_massive(var a:massiv;p:word);
var
  i:word;
begin
    writeln('Массив создан:');
    for i:=1 to p do
    begin
        a[i]:=2*(9+random(100));
        write(a[i]:5);
    end;
end;
procedure vivod_massive(var a:massiv;p:word);
var
  i:word;
begin
    for i:=1 to p do write(a[i]:5);
end;

procedure shell(var a:massiv;p:word);
const
    t=5;
var
  i,j,k,s,m:integer;
  h:array[1..t] of integer;
  x:data;
begin
    writeln('Сортировка Шелла:');
    h[1]:=9; h[2]:=5; h[3]:=3; h[4]:=2; h[5]:=1;
    for m:=1 to t do
    begin
        k:=h[m];
        s:=-k;
        for i:=k+1 to p do
        begin
            x:=a[i];
            j:=i-k;
```

```
            if s=0 then
            begin
                s:=-k;
                s:=s+1;
                a[s]:=x;
            end;
            while (x<a[j]) and (j<p) do
              begin
                  a[j+k]:=a[j];
                  j:=j-k;
              end;
            a[j+k]:=x;
          end;
      end;
  end;
  begin
      clrscr;
      write('Введите число элементов массива 2..15: ');
      readln(n);
      create_massive(b,n);
      writeln;
      shell(b,n);
      vivod_massive(b,n);
      readkey;
  end.
```

Варианты заданий

1.Разработать алгоритм и программу внутренней сортировки значений в таблице (пузырьковая сортировка)

2.Разработать алгоритм и программу внутренней сортировки значений в таблице (сортировка простыми вставками)

3. Разработать алгоритм и программу внутренней сортировки значений в таблице (простая сортировка выбором)

4. Разработать алгоритм и программу внутренней сортировки значений в таблице (метод Шелла)

5. Разработать алгоритм и программу внутренней сортировки значений в таблице (бинарные вставки)

6. Разработать алгоритм и программу внутренней сортировки значений в таблице (быстрая сортировка)

7. Разработать алгоритм и программу внутренней сортировки значений в таблице (сортировка выбором)

8. Разработать алгоритм и программу внешней сортировки значений в таблице (простое слияние)

9. Разработать алгоритм и программу внешней сортировки значений в таблице (естественное слияние)

10.Разработать алгоритм и программу внешней сортировки значений в таблице (улучшенные методы сортировки)

Контрольные вопросы

1. Для чего применяется сортировка данных?
2. Какие существуют алгоритмы внешней сортировки?
3. Какие существуют алгоритмы внутренней сортировки?
4. В чем заключается смысл алгоритма сортировки Хоара?
5. В чем заключается смысл алгоритма сортировки слиянием?
6. В чем заключается смысл алгоритма сортировки выбором?
7. Какие существуют алгоритмы усовершенствованной сортировки?
8. В чем заключается смысл алгоритма сортировки вставками?
9. В чем заключается смысл алгоритма сортировки Шелла?
10. В чем заключается смысл алгоритма обменной сортировки?
11. В чем заключается смысл алгоритма сортировки методом «пузырька»?

Лабораторная работа № 8
Реализация алгоритмов сортировок включением и выбором

Цель работы: формирование знаний и умений по изучению методов внутренних сортировок. Приобретение навыков реализации алгоритмов сортировки.

Краткие теоретические сведения
Методы внутренней сортировки

В данных лабораторно-практических работах будут рассмотрены методы, у которых на входе – вектор с произвольным положением ключей, а на выходе - этот же вектор упорядочен по *возрастанию*.

Существует три группы методов внутренней сортировки (сортировка включением, сортировка выбором, обменная сортировка). В данной лабораторной работе рассмотрены методы сортировки включением и выбором.

Сортировки включением

Сортировка прямым включением. Элементы условно разделяют на *готовую* a[1],...,a[i-1] и *входную последовательности* a[i],...,a[n]. Сначала i=2. На каждом шаге, увеличивая i на единицу, берут i-й элемент входной последовательности и передают в готовую последовательность, вставляя на подходящее место.

Итак, в начале рассматривается второй элемент *(i=2)* последовательности ключей. Он сравнивается с первым элементом *(a[i-1])* и если он его меньше, то они меняются местами. В противном случае проход

заканчивается, *i* увеличивается на *1* и процесс повторяется. Заметим, что упорядоченная последовательность *a[i-1]* называется *готовой последовательностью* и новый элемент вставляется в готовую последовательность на свое место.

На каждом проходе происходит перемещение *i*-того элемента в готовую последовательность, а некоторое число элементов сдвигается вправо. Данный процесс перемещения называется *просачиванием* элемента.

*Здесь и далее, во всех процедурах сортировки ключи упорядочиваются по возрастанию. На входе процедур - количество элементов массива (**n**), на выходе - упорядоченный массив элементов(**a**).*

```
Procedure Straight_Insertion(n:integer; Var a:t);
Var
i,j:word;
x:integer;
Begin
   For i:=2 To n Do
              begin
                    x:=a[i]; a[0]:=x; j:=i-1;
                       While x<a[j] Do
                  begin
                        a[j+1]:=a[j]; j:=j-1;
                  end;
                        a[j+1]:=x
               end;
End;{Straight_Insertion}
```

Процедура упорядочивает элементы массива начиная с первого. Нулевой элемент используется процедурой как вспомогательный.

В основной программе массив необходимо объявить следующим образом:

```
TYPE
      t=array[0..20] of integer;
VAR
      A:t;
      N,i:word;
```

Ввод массива:

```
FOR  i:=1 to n do
      Read(a[i]);
```

Вывод отсортированного массива:

```
FOR  i:=1 to n do
     Wrire(a[i],' ');
```

Сортировка бинарными включениями. Алгоритм сортировки прямыми включениями можно легко улучшить, пользуясь тем, что готовая последовательность a[1],...,a[i-1], в которую нужно включить новый элемент, уже упорядочена. Поэтому место включения можно найти значительно

быстрее, применив бинарный поиск, который исследует средний элемент готовой последовательности и продолжает деление пополам, пока не будет найдено место включения. Модифицированный алгоритм сортировки называется *сортировкой бинарными включениями.*

```
Procedure Binary_Insertion(n:word;Var a:t);
Var
i,j,k,r,m:word;
x:integer;
Begin
        For i:=2 To n Do
        begin
        x:=a[i]; k:=1; r:=i-1;
        While k<=r Do
        begin
        m:=(k+r) div 2;
        If x<a[m] Then r:=m-1
        Else k:=m+1
        end;
        For j:=i-1 DownTo l Do a[j+1]:=a[j];
        a[k]:=x
        end;
End; {Binary_Insertion}
```

Сортировка выбором

Прямой выбор. Этот метод основан на следующем правиле: выбираем элемент с наименьшим ключом. Он меняется местами с первым элементом. Эти операции затем повторяются с оставшимися *n*-1 элементами, затем с *n*-2 элементами, пока не останется только один элемент – *наибольший*. Этот метод называемый *сортировкой простым выбором,* в некотором смысле противоположен сортировке простыми включениями; при сортировке простым выбором рассматриваются *все* элементы *входного массива* для нахождения элемента с наименьшим ключом, и этот *один* очередной элемент отправляется в *готовую последовательность*.

```
        Procedure Straight_Selection(n:word;Var a:t);
    Var
      i,j,k:word;
      x:integer;
  Begin
    For i:=1 To n-1 Do
     begin
        x:=a[i]; k:=i;
        For j:=i+1 To n Do
          If x>a[j] Then
        begin
         k:=j; x:=a[j];
```

```
      end;
    a[k]:=a[i]; a[i]:=x;
   end
End;{Straight_Selection}
```

Порядок выполнения работы

1. Изучить теоретические сведения по теме: ”Алгоритмы сортировок включением и выбором”
2. Разработать программу для реализации рассмотренных в данной работе методов сортировок. Предоставить пользователю возможность выбора метода сортировки.
3. Показать работающую программу преподавателю.
4. Ответить на контрольные вопросы.

Варианты заданий

1. Разработать процедуру сортировки массива целых чисел методом прямого включения. Правильность сортировки проверить путем подсчета контрольной суммы и числа серий в массиве. Предусмотреть подсчет количества пересылок и сравнений (M и C), сравнить их с теоретическими оценками.
2. Разработать процедуру сортировки массива целых чисел методом Шелла. Правильность сортировки проверить путем подсчета контрольной суммы и числа серий в массиве. Предусмотреть подсчет количества пересылок и сравнений (M и C), сравнить их с теоретическими оценками.
3. Сравнить метод прямого включения и метод Шелла по количеству сравнений и пересылок для различных типов массивов. Разработать процедуру построения графика зависимости величин M и C от размера массива.
4. Сравнить метод прямого включения и метод Шелла с ранее расмотренными методами сортировки по количеству сравнений и пересылок для различных типов массивов. Разработать процедуру построения графика зависимости величин M и C от размера массива
5. Экспериментально определить подходящую последовательность шагов для метода Шелла.
6. Опытным путем определить константы в выражениях оценки для количества сравнений и пересылок в методе Шелла.

Контрольные вопросы

1. Понятие сортировки. Виды сортировок.
2. Сортировки включением. Описание алгоритмов методов сортировки прямыми и бинарными включениями.
3. Сортировки включением. Описание алгоритмов методов сортировки прямыми и бинарными включениями.
4. Сортировка выбором. Описание алгоритма.
5. Фрагменты программ для реализации данных методов сортировок.

Лабораторная работа № 9
Реализация алгоритмов обменных сортировок

Цель работы: формирование знаний и умений по изучению методов внутренних сортировок. Приобретение навыков реализации алгоритмов сортировки.

Краткие теоретические сведения

Существует три группы методов внутренней сортировки (сортировка включением, сортировка выбором, обменная сортировка). В данной лабораторной работе рассмотрены методы обменных сортировок.

Сортировка прямого обмена (пузырьковая)

Метод, в котором обмен двух элементов является основной характеристикой процесса. Алгоритм сортировки простым обменом основан на принципе сравнения и *обмена пары соседних элементов* до тех пор, пока не будут рассортированы все элементы. Если мы будем рассматривать массив, расположенный вертикально, а не горизонтально и представим себе элементы пузырьками в резервуаре с водой, обладающими “весами”, соответствующими их ключам, то каждый проход по массиву приводит к “всплыванию” пузырька на соответствующий его весу уровень. Этот метод широко известен как сортировка *методом пузырька.*

Начнем последовательность обменов с элементами a[N]. В каждом проходе местами меняются соседние элементы. Самый легкий элемент поднимается за один проход, самый тяжелый опускается вниз за один проход. Если самый большой элемент был первым, то выполняется N проходов. Критерием окончания является отсутствие обменов при очередном проходе.

*Здесь и далее, во всех процедурах сортировки ключи упорядочиваются по возрастанию. На входе процедур - количество элементов массива (**n**), на выходе - упорядоченный массив элементов(**a**).*

```
Procedure Bubble_Sort(n:word;Var a:t);
      Var
      i,j:word;
      x:integer;
  Begin
    For i:=2 To n Do
      begin
        For j:=n DownTo i Do
        If a[j-1]>a[j] Then
         begin
            x:=a[j-1]; a[j-1]:=a[j]; a[j]:=x
         end
      end
  End;{Bubble_Sort}
```

Шейкерная сортировка

Улучшение алгоритма – это запоминать, производится ли на данном проходе какой-либо обмен. Если нет, то это означает, что алгоритм может закончить работу. Этот процесс улучшения можно продолжить, если запомнить не только сам факт обмена, но и место (индекс) последнего обмена. Все пары соседних элементов с индексами, меньшими этого индекса k, уже расположены в нужном порядке. Поэтому следующие проходы можно заканчивать на этом индексе, вместо того чтобы двигаться до установленной заранее нижней границы i.

Однако внимательный программист заметит здесь странную асимметрию: один неправильно расположенный "пузырек" в "тяжелом" конце рассортированного массива всплывет на место за один проход, а неправильно расположенный элемент в "легком" конце будет опускаться на правильное место только на один шаг на каждом проходе. Улучшение: менять направление следующих один за другим проходов. Полученный в результате алгоритм называют *шейкерной сортировкой*.

Итак, для того, чтобы тяжелые элементы сразу попадали вниз пузырьковую сортировку выполняют так, чтобы направление прохода было снизу вверх, следующий проход – сверху вниз и так далее.

```
Procedure Shaker_Sort(n:word;Var a:t);
    Var
      j,k,l,r:word;
      x:integer;
  Begin
    l:=2; r:=n; k:=n;
    Repeat
      For j:=r DownTo l Do
        If a[j-1]>a[j] Then
        begin
          x:=a[j-1]; a[j-1]:=a[j]; a[j]:=x; k:=j;
        end;
      l:=k+1;
      For j:=l To r Do
        If a[j-1]>a[j] Then
        begin
          x:=a[j-1]; a[j-1]:=a[j]; a[j]:=x; k:=j;
        end;
      r:=k-1;
    Until l>r
  End;{Shaker_Sort}
```

Обменная сортировка разделением

Этот метод называется еще *быстрой сортировкой*. Быстрая сортировка основана на том факте, что для достижения наибольшей эффективности

желательно производить обмены элементов на больших расстояниях. Реализуется она на основе следующего алгоритма.

Выбирается любой произвольный элемент массива, далее массив просматривается слева направо до тех пор пока не будет найден элемент больший выбранного; а затем просмотрим его справа налево, пока не найдем элемент меньший выбранного. Найденные элементы поменяем местами. Затем продолжим процесс “просмотра с обменом”, пока два просмотра не встретятся где-то в середине массива. В результате массив разделится на две части: левую – с ключами меньшими выбранного элемента; и правую – с большими ключами. Описанный алгоритм применяется к обоим этим частям, в результате чего последовательность разбивается на 4 части. Алгоритм применяется к каждой четвертинке и т.д. Разделение заканчивается, когда в каждой части остается 1 элемент.

Здесь в процедуре Quick_Sort вызывается процедура Sort1, которая реализует алгоритм разделения и обмена для одной части массива.

```
Procedure Quick_Sort(n:word;Var a:massiv);
    Procedure Sort1(l,r:word);
      Var
        w,x:integer;
        i,j:word;
    Begin
      i:=l; j:=r;
      x:=a[(l+r) div 2];
      Repeat
        While a[i]<x Do i:=i+1;
        While a[j]>x Do j:=j-1;
        If i<=j Then
          begin
            w:=a[i]; a[i]:=a[j]; a[j]:=w;
            i:=i+1; j:=j-1
          end
      Until i>j;
      If l<j Then Sort1(l,j);
      If i<r Then Sort1(i,r);
    End;{Sort1}
  BEGIN
    Sort1(1,n);
  END;{Quick_Sort}
```

Пирамидальная сортировка

Пирамида определяется как последовательность ключей h_l, h_{l+1}, ..., h_r такая, что $h_i <= h_{2i}$, $h_i <= h_{2i+1}$ для всякого i=l,...,r/2.

Предположим, что дана пирамида с элементами h_{l+1}, ..., h_r для некоторых значений l и r и нужно добавить новый элемент x для того, чтобы

сформировать расширенную пирамиду h_l, ..., h_r. Новый элемент x сначала помещается в вершину дерева, а затем “просеивается” по пути, на котором находятся меньшие по сравнению с ним элементы, которые одновременно поднимаются вверх; таким образом, формируется новая пирамида.

Здесь в процедуре Heap_Sort вызывается процедура Sift, которая реализует алгоритм формирования пирамиды.

```
Procedure Heap_Sort(n:word;Var a:t);
  Var
    l,r:word;x:integer;
  Procedure Sift;
    Label 13;
    Var i,j:word;
  Begin
    i:=l; j:=2*i; x:=a[i];
    While j<=r Do
    begin
      If j<r Then
        If a[j]<a[j+1] Then j:=j+1;
      If x>=a[j] Then Goto 13;
      a[i]:=a[j]; i:=j; j:=2*i;
    end;
    13: a[i]:=x
  End;{Sift}
BEGIN
  l:=(n div 2)+1; r:=n;
  While l>1 Do
  begin
    l:=l-1; Sift
  end;
  While r>1 Do
  begin
    x:=a[1]; a[1]:=a[r]; a[r]:=x;
    r:=r-1; Sift
  end
END;{Heap_Sort}
```

Порядок выполнения работы

1. Изучить теоретические сведения по теме: ”Алгоритмы обменных сортировок”.
2. Разработать программу для реализации рассмотренных в данной работе методов сортировок. Предоставить пользователю возможность выбора метода сортировки.
3. Показать работающую программу преподавателю.
4. Ответить на контрольные вопросы.

Варианты заданий

1. Разработать процедуру сортировки массива целых чисел методом пирамидальной сортировки. Правильность сортировки проверить путем подсчета контрольной суммы и числа серий в массиве. Предусмотреть подсчет количества пересылок и сравнений (М и С), сравнить их с теоретическими оценками.

2. Разработать процедуру сортировки массива целых чисел методом Хоара . Правильность сортировки проверить путем подсчета контрольной суммы и числа серий в массиве. Предусмотреть подсчет количества пересылок и сравнений (М и С), сравнить их с теоретическими оценками.

3. Сравнить метод пирамидальной сортировки и метод Хоара по количеству сравнений и пересылок для различных типов массивов. Разработать процедуру построения графика зависимости М+С от размера массива n для случайного массива.

4. Сравнить трудоемкости метода Хоара и метода прямого выбора для различных типов массивов. Разработать процедуру построения графика зависимости величин М и С от размера массива

5. Экспериментально определить устойчивость и зависимость от начальной отсортированности массива пирамидальной сортировки и метода Хоара.

6. Опытным путем определить константы в выражениях оценки для количества сравнений и пересылок в методе Хоара.

7. Реализовать метода Хоара с меньшим количеством рекурсивных вызовов. Определить необходимое количество рекурсивных вызовов.

8. Запрограммировать метод Хоара без использования рекурсивных вызовов.

Контрольные вопросы

1. Пузырьковая сортировка. Описание алгоритма и фрагмент программы.
2. Шейкерная сортировка. Описание алгоритма и фрагмент программы.
3. Пирамидальная сортировка.
4. Быстрая сортировка.

Лабораторная работа № 10
Реализация алгоритмов внешних сортировок

Цель работы: формирование знаний и умений по изучению методов внешних сортировок. Приобретение навыков реализации алгоритмов сортировки.

Краткие теоретические сведения

Внешняя сортировка применяется при сортировке последовательных файлов, которые называются л*ентами*. Последовательные файлы имеют

такую особенность, что для того, чтобы получить доступ к элементу, находящемуся внутри последовательности нужно просмотреть все элементы, расположенные до него. Т.о. в каждый текущий момент мы имеем доступ только к 1 элементу последовательного файла.

Метод простого слияния

Пусть в последовательном файле (ленте) имеется следующая информация:

31 42 09 29 37 01 12 06

Во внешних сортировках каждый проход состоит из *фаз*. В методе простого слияния существует 2 фазы: *разделения и слияния*. Для рассмотренного метода нужны 3 ленты. Исходная последовательность (A) разделяется на 2 ленты (B и C):

A: 31 42 09 29 37 01 12 06 –исходный файл

B: 31 42 09 29

C: 37 01 12 06

Затем выполняется *фаза слияния*: анализируются доступные на лентах B и C элементы и посылаются на ленту A в виде упорядоченных пар:

A: 31 37 01 42 09 12 06 29.

Теперь содержимое ленты A состоит из упорядоченных пар. Затем снова разделяется наполовину:

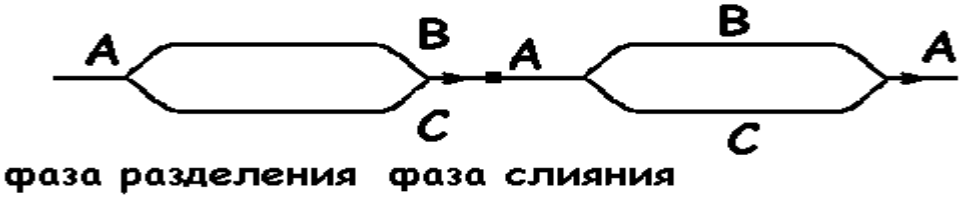
B: 31 37 01 42

C: 09 12 06 29.

Последовательность сливается в упорядоченные четверки.

A: 09 12 31 37 01 06 29 42.

Схематично метод простого слияния выглядит так:



Естественное слияние

Предыдущий метод не учитывает того факта, что в исходной последовательности уже могут быть упорядоченные подпоследовательности. Они называются *сериями*. Это последовательности вида:

A[L],A[L+1],…A[k],

такая, что:

A[L] …A[k],

A[L-1]>A[L],

A[k+1]<A[k].

Метод работает так: сначала первая серия посылается на ленту B, вторая- на ленту C, третья- на ленту B, четвертая на C и т.д. Первая серия с ленты B сливается с первой серией ленты C. В результате на ленте A куда сливаются эти серии появится 1-ая *увеличенная* серия.

Затем вторая серия ленты В сливается со второй серией ленты С и т.д. После выполнения всех вариантных слияний количество серий на ленте А окажется в двое меньше, чем в исходной последовательности. Затем снова разделение серий, снова слияние и т.д.

Многопутевое слияние

В этом методе исходная лента разделяется не на две, а на m лент (m>2). Схематично метод простого слияния выглядит так (рис1.):

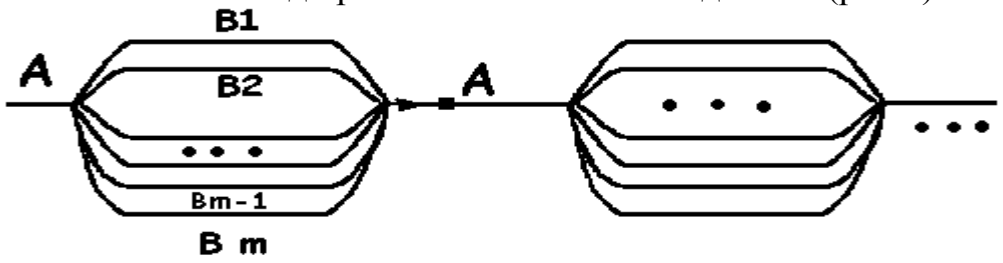


Рисунок 1 – Многопутевое слияние

На рисунке 1- B1,B2, …, Bm соответственно 1,2, .., m –тая *дополнительные ленты*.

Алгоритм метода работает так: первая серия на ленте А при выполнении фазы разделения помещается на ленту В1, вторая серия-на ленту В2 и т.д., m- тая серия на ленту Bm. M+1 серия на ленту В1, M+2- на ленту В2 и т.д.

После выполнения фазы разделения происходит слияние *всех первых серий* каждой дополнительной ленты. Для организации слияния в ОП может быть организован массив из m элементов.

После выполнения первой фазы разделения на каждой из дополнительных лент будет размещено примерно]N/m[серий. После выполнения второй фазы разделения число серий на каждой ленте]N/m[/m.

Методы внешней сортировки могут быть использованы для упорядочивания массивов, хранимых в ОП. При этом некоторые наиболее эффективные из них сравнимы или даже превосходят по эффективности самую быстродействующую сортировку – пирамидальную.

Порядок выполнения работы

1. Изучить теоретические сведения по теме: ”Алгоритмы внешних сортировок”
2. Показать выполненное задание преподавателю.
3. Ответить на контрольные вопросы.

Варианты заданий

1.Разработать процедуру сортировки последовательности целых чисел методом прямого слияния. Правильность сортировки проверить путем подсчета контрольной суммы и числа серий в последовательности. Предусмотреть подсчет количества пересылок и сравнений (М и С), сравнить их с теоретически-ми оценками.

2. Разработать процедуру цифровой сортировки последовательности целых чисел Правильность сортировки проверить путем подсчета контрольной суммы и числа серий в последовательности. Предусмотреть подсчет количества пересылок и сравнений (M и C), сравнить их с теоретическими оценками.

3. Сравнить метод прямого слияния и метод цифровой сортировки по количеству сравнений и пересылок для различных последовательностей. Разработать процедуру построения графика зависимости M+C от длины последовательности n.

4. Сравнить трудоемкости методов сортировки последовательностей длины n и методов быстрой сортировки массивов длины n. результаты оформить в виде таблицы.

5. Экспериментально определить устойчивость и зависимость от начальной отсортированности последовательности методов сортировки последовательностей.

6. Опытным путем определить константы в выражениях оценки для количества сравнений и пересылок в методе прямого слияния и методе цифровой сортировки.

7. Определить длину чисел (количество цифр в записи числа) в последовательности, при которой цифровая сортировка работает медленнее, чем метод Хоара для массива той же длины.

Контрольные вопросы

1. Понятие внешней сортировки. Виды сортировок.
2. Метод простого слияния. Описание алгоритма.
3. Естественное слияние. Описание алгоритма.
4. Многопутевое слияние.Описание алгоритма.

Лабораторная работа № 11.
Поиск в таблице значений

Цель работы: 1)ознакомиться с методами решения задач поиска, а именно поиска в таблице, в соответствии с данным заданием; 2)получить навыки в программировании задач поиска элементов.

Краткие теоретические сведения

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;
```

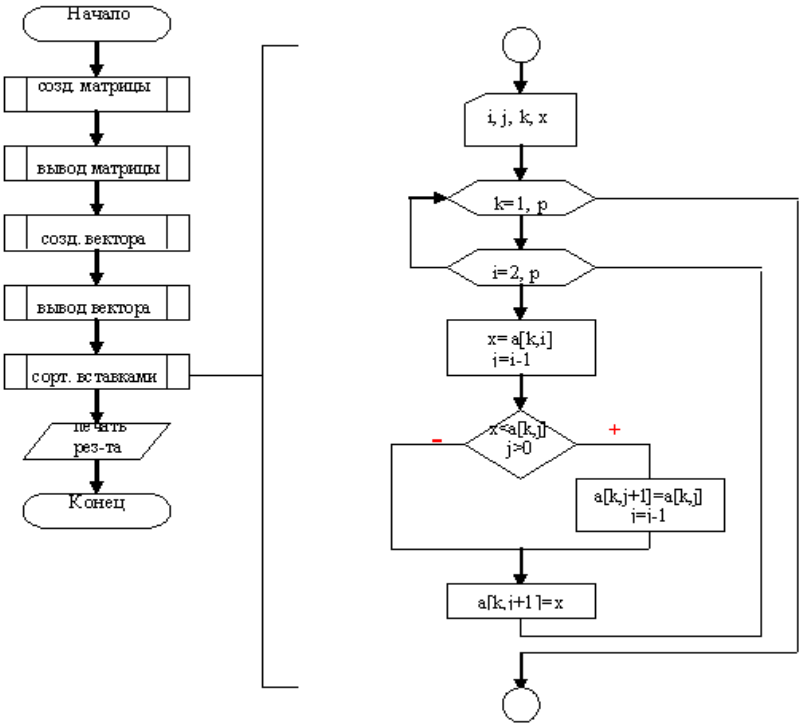
```
                    if T[m,i]<x[i] then L:=m+1 else R:=m
end;
if R<N then begin
        i:=0;
         while (T[R,i]=x[i]) and (x[i]<>00h) do i:=i+1
end;
{(R<N) and (T[R,i]=x[i]) фиксирует совпадение}.
```

Но при достаточно большом значении N можно использовать алгоритм линейного поиска и алгоритм последовательного сравнения строк.

Пример выполнения

Задание: осуществить последовательный поиск совпадений в таблице, используя сортировку вставкой.

Блок-схема алгоритма



Текст программы:

```
uses crt;
type
   data=integer;
   matrix=array[1..15,1..15] of integer;
   vector=array[1..15] of integer;
var
  a:matrix;
  x:vector;
  n,i,j,k:word;
  l,m,r:integer;
  found:boolean;
```

```
    procedure create_matrix(var a:matrix;p:word);
    var
      i,j:word;
begin
    writeln('Матрица создана:');
    for i:=1 to p do
    for j:=1 to p do a[i,j]:=2*(9+random(100));
end;

procedure vivod_matrix(var a:matrix;p:word);
var
  i,j:word;
begin
    for i:=1 to p do
    begin
        for j:=1 to p do write(a[i,j]:6);
        writeln;
        end;
end;
procedure create_vector(var x:vector;p:word);
var
  i:word;
begin
    writeln('Вектор создан:');
    for i:=1 to p do x[i]:=random(100);
end;
procedure vivod_vector(var x:vector;p:word);
var
  i:word;
begin
    for i:=1 to p do write(x[i]:6);
    writeln;
end;
procedure insert(var a:matrix;p:word);
var
  i,j,k:integer;
  x:data;
begin
    writeln('Сортировка вставкой:');
    for k:=1 to p do
    for i:=2 to p do
    begin
        x:=a[k,i];
        j:=i-1;
```

```
            while (x<a[k,j]) and (j>0) do
            begin
                a[k,j+1]:=a[k,j];
                j:=j-1;
            end;
            a[k,j+1]:=x;
        end;
    end;
    begin
        clrscr;
        write('Введите число элементов массива: n=');
        readln(n);
        create_matrix(a,n);
        vivod_matrix(a,n);
        writeln('Упорядоченная матрица в строках:');
        insert(a,n);
        vivod_matrix(a,n);
        create_vector(x,n);
        vivod_vector(x,n);
        writeln('Совпадения элементов матрицы и вектора');
        writeln('-------------------------------------');
        for i:=1 to n do
        begin
            for j:=1 to n do
            for k:=1 to n do if a[i,j]=x[k] then writeln(x[k],': совпадение в ',i,' – й
строке матрицы');
        end;
        writeln('Нажмите любую клавишу');
        readkey;
    end.
```

Варианты заданий

1. Разработать алгоритм и программу последовательного поиска значений в одномерном массиве
2. Разработать алгоритм и программу бинарного поиска значений в одномерном массиве
3. Разработать алгоритм и программу последовательного поиска значений в таблице данных
4. Разработать алгоритм и программу бинарного поиска значений в таблице данных
5. Разработать алгоритм и программу поиска положительных элементов в каждой строке матрицы
6. Разработать алгоритм и программу поиска положительных элементов в каждом столбце матрицы
7. Разработать алгоритм и программу поиска элементов, кратных 2, в

каждой строке матрицы 8.Разработать алгоритм и программу поиска положительных элементов в главной диагонали матрицы

9. Разработать алгоритм и программу поиска положительных элементов в нижней треугольной матрице

10. Разработать алгоритм и программу поиска наименьшего элемента в матрице

Контрольные вопросы

1. Какие задачи решаются при помощи поиска в структурах данных?
2. В чем заключается смысл алгоритма линейного поиска?
3. В чем заключается смысл алгоритма поиска делением пополам (двоичного поиска)?
4. Какие особенности поиска в таблице данных?
5. Как осуществляется поиск текстовой информации? Как реализуется алгоритм прямого поиска строки?

Лабораторная работа № 12.
Бинарные деревья

Цель работы: 1) ознакомиться со способами представления деревьев и методами их прохождения; 2) получить практические навыки в программировании задач с использованием деревьев.

Краткие теоретические сведения

В повседневной жизни мы очень часто встречаем примеры деревьев. Например, люди часто используют генеалогическое дерево для изображения структуры своего рода.

Второй пример – это структура большой организации; использование древообразной структуры для представления ее "иерархического строения" ныне широко используется во многих компьютерных задачах.

Третий пример – это грамматическое дерево; изначально оно служило для грамматического анализа компьютерных программ, а ныне широко используется и для грамматического анализа литературного языка.

Мы начнем изучение деревьев с того, что определим их как некий абстрактный объект и введем всю связанную с ними терминологию.

Самыми простыми из деревьев считаются бинарные деревья.

Бинарное дерево – это конечное множество элементов, которое либо пусто, либо содержит один элемент, называемый корнем дерева, а остальные элементы множества делятся на два непересекающихся подмножества, каждое из которых само является бинарным деревом.

Эти подмножества называются левым и правым поддеревьями исходного дерева.

Каждый элемент бинарного дерева называется узлом дерева.

Два узла являются братьями, если они сыновья одного и того же отца.

Если каждый узел бинарного дерева, не являющийся листом, имеет непустые правые и левые поддеревья, то дерево называется строго бинарным деревом.

Свойство. Строго бинарное дерево с n листами всегда содержит 2n – 1 узлов.

Уровень узла в бинарном дереве может быть определен следующим образом. Корень дерева имеет уровень 0, и уровень любого другого узла дерева имеет уровень на 1 больше уровня своего отца.

Глубина бинарного дерева – это максимальный уровень листа дерева, что равно длине самого длинного пути от корня к листу дерева. Полное бинарное дерево уровня n – это дерево, в котором каждый узел уровня n является листом и каждый узел уровня меньше n имеет непустые левое и правое поддеревья.

Почти полное бинарное дерево – это бинарное дерево, для которого существует неотрицательное целое k, такое, что

1) каждый лист в дереве имеет уровень k или k+1;
2) если узел дерева имеет правого потомка уровня k+1, тогда все его левые потомки, являющиеся листами, также имеют уровень k+1.

Узлы почти полного бинарного дерева могут быть пронумерованы так, что корню назначается номер 1, левому сыну - удвоенный номер его отца, а правому – удвоенный номер отца плюс единица (намного проще это понять, если представить эти числа в двоичной системе).

Есть еще одна разновидность бинарных деревьев, которая называется «упорядоченные бинарные деревья».

Упорядоченные бинарные деревья – это деревья, в которых для каждого узла X выполняется правило: в левом поддереве - ключи, меньшие X, в правом поддереве – большие или равные X. Структура бинарного дерева построена из узлов. Узел дерева содержит поле данных и два поля с указателями.

Поля указателей называются левым указателем и правым указателем, поскольку они указывают на левое и правое поддерево соответственно. Значение nil является признаком пустого дерева (рисунок).

Над бинарным деревом есть операция его прохождение, т.е. нужно обойти все дерево, отметив каждый узел один раз.

Существует три способа обхода бинарного дерева.

В прямом порядке:

а) попасть в корень;

б) пройти в прямом порядке левое поддерево;

в) пройти в прямом порядке правое поддерево

В симметричном порядке:

а) пройти в симметричном порядке левое поддерево;

б) попасть в корень;

в) пройти в симметричном порядке правое поддерево.

В обратном порядке:

а) пройти в обратном порядке левое поддерево;

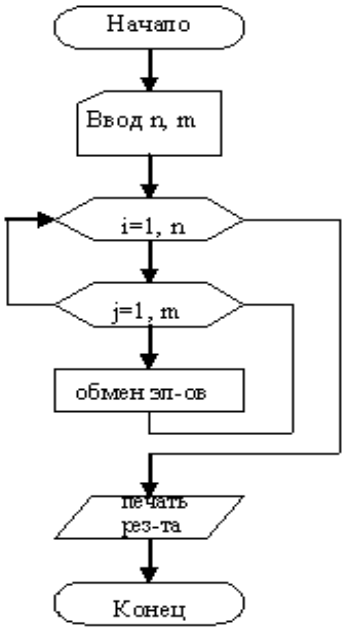
б) пройти в обратном порядке правое поддерево;

в) попасть в корень.

Пример выполнения

Задание: разработать программу, реализующую прямое прохождение (известное также как прохождение в глубину) бинарного дерева

Блок-схема алгоритма



Текст программы:

```
uses
   crt;
const
    nvertex=100;
    nadj=1000;
var
  f:text;
  n,nt:integer;
  adj:array [1..nadj] of integer;
  fst:array [1..nvertex] of integer;
  nbr:array [1..nvertex] of integer;
  vtx:array [1..nvertex] of integer;
  mark:array [1..nvertex] of integer;
  t:array [1..nvertex] of integer;
  b:array [1..nvertex] of integer;

procedure init(var y:boolean);
var
  i,j,m:integer;
```

```
begin
    for i:=1 to n do
    for j:=1 to nbr[i] do
    begin
        y:=false;
        for m:=1 to n do if adj[fst[i]+j]=vtx[m] then
                    begin
                        y:=true;
                        adj[fst[i]+j]:=m;
                        break;
                        end;
        if not y then exit;
        end;
end;

procedure deep(x,u:integer;var c:integer);
var
  i,v:integer;
begin
    c:=c+1;
    mark[x]:=c;
    for i:=1 to nbr[x] do
    begin
        v:=adj[fst[x]+i];
        if mark[v]=0 then
        begin
            nt:=nt+2;
            t[nt-1]:=x;
            t[nt]:=v;
            b[nt div 2]:=1;
            deep(v,x,c);
            end;
            else if (mark[v]<mark[x]) and (v<>u) then
                begin
                    nt:=nt+2;
                    t[nt-1]:=x;
                    t[nt]:=v;
                    b[nt div 2]:=0;
                    end;
    end;
end;

procedure way;
var
```

```
  v,c:integer;
begin
    nt:=0;
    c:=0;
    for v:=1 to n do mark[v]:=0;
    for v:=1 to n do if mark[v]=0 then deep(v,0,c);
end;

procedure main;
var
  i,j:integer;
  y:boolean;
begin
    writeln('Reading...');
    assign(f,'in.txt');
    reset(f);
    read(f,n);
    fst[1]:=0;
    for i:=1 to n do
    begin
        read(f,vtx[i]);
        read(f,nbr[i]);
        for j:=1 to nbr[i] do read(f,adj[fst[i]+j]);
        fst[i+1]:=fst[i]+nbr[i];
        end;
    close(f);
    writeln('Solve...');
    assign(f,'out.txt');
    rewrite(f);
    init(y);
    if not y then
    begin
        writeln(f,'BAD SEED! Hi from hell');
        writeln('BAD SEED! Hi from hell');
        close(f);
        exit;
        end;
    way;
    for i:=1 to nt div 2 do write(f,vtx[t[2*i-1]]:3);
    writeln(f);
    for i:=1 to nt div 2 do write(f,vtx[t[2*i]]:3);
    writeln(f);
    for i:=1 to nt div 2 do write(f,b[i]:3);
    writeln(f);
```

```
    close(f);
    writeln('End.');
    writeln('Press any key.');
end;
begin
    clrscr;
    main;
    readkey;
end.
```

Варианты заданий

Задание 1

1. Написать процедуру включения нового элемента в случайное дерево поиска.
2. Определить необходимое количество операций для включения элемента в дерево. Сравнить эту величину с высотой дерева.
3. Написать процедуру исключения элемента с заданным ключом из случайного дерева поиска.
4. Определить необходимое количество операций для исключения элемента из дерева. Сравнить эту величину с высотой дерева.
5. Построить случайное дерево поиска для данных, которые поступают в случайном порядке. Найти высоту построенного дерева и сравнить с теоретическими оценками.
6. Проверить, является ли построенное случайно дерево деревом поиска с помощью логической функции.
7. Сделать обход слева направо для построенного случайного дерева. Подтвердить, что оно является деревом поиска.
8. Написать процедуру графического изображения СДП.
9. Построить случайное дерево поиска для данных, которые поступают в возрастающем (убывающем) порядке. Найти высоту построенного дерева и сравнить с теоретическими оценками.
10. Построить случайное дерево поиска и ИСДП для одних и тех же данных. Определить высоты построенных деревьев и найти отношение высот. Сравнить полученную величину с теоретической оценкой.

Задание 2

1. Написать программы вычисления высоты дерева с использованием представлений деревьев.
2. Написать программу, выводящую списки узлов дерева, при обходе этого дерева в прямом порядке.
3. Написать программы обхода двоичных деревьев в прямом порядке.
4. Написать программу поиска элементов, кратных 2, в бинарном дереве.
5. Написать программу поиска положительных элементов в бинарном дереве.

6. Написать программу, выводящую списки узлов дерева, при обходе этого дерева в внутреннем порядке.

7. Написать программы обхода двоичных деревьев в обратном порядке.

8. Написать программы обхода двоичных деревьев в внутреннем порядке.

9. Написать программу, выводящую списки узлов дерева, при обходе этого дерева в обратном порядке.

10. При прохождении дерева в порядке уровней, в список узлов сначала заносится корень дерева, затем все узлы глубины 1, далее все узлы глубины 2 и т.д. Уз¬лы одной глубины заносятся в список узлов в порядке слева направо. Напи¬сать программу обхода деревьев в порядке уровней.

Контрольные вопросы

1. Каким образом можно представить тип данных Дерево?

2. Какие особенности у типа данных Бинарное дерево? Где они применяются?

3. Как можно представить бинарное дерево?

4. Какие существуют способы прохождения бинарных деревьев? Где они применяются?

5. Какие существуют алгоритмы, использующие тип данных Дерево?

6. В чем состоит смысл алгоритма сортировки с прохождением бинарного дерева?

7. В чем состоит смысл алгоритма сортировки методом турнира с выбыванием?

8. Каким образом бинарные деревья применяются для сжатия информации?

9. Как представляются выражения с помощью деревьев?

10. Как используется тип данных Сильноветвящееся дерево? Каково применение сильноветвящихся деревьев?

11. Каким образом можно представить тип данных Граф?

12. Какие существуют алгоритмы, использующие тип данных Граф?

ПРАКТИЧЕСКИЕ РАБОТЫ ПО КУРСУ «Алгоритмы и структуры данных»

Приводится список задач для разбора на практических занятиях.

Занятия 1-4 (8 часов). Работа с файлами.

1. Дан текстовый файл, в котором записана последовательность целых чисел. Записать в новый файл только четные из них.

2. Сгенерировать набор целых чисел, записать их в бинарный файл. Не считывая, данные в массив, найти из этого файла максимальный элемент.

3. Для файла, сгенерированного в задании 2, найти количество чисел, которые являются простыми.

4. Для файла, сгенерированного в задании 2, проверить, образуют ли числа в нем возрастающую последовательность.

5. Для файла, сгенерированного в задании 2, проверить, образуют ли они симметричную последовательность.

6. Дан текстовый файл, в котором записана последовательность целых чисел. Создать новый текстовый файл, в котором числа идут в обратном порядке.

7. Дан бинарный файл (например, сгенерированный в задании 2), в котором записана последовательность целых чисел. Создать новый файл, в котором числа идут в обратном порядке.

8. Дан текстовый файл, в котором записана последовательность слов. Создать новый текстовый файл, в котором слова идут в обратном порядке.

9. Записать бинарный файл с массивом целых чисел. Создать новый текстовый файл, в котором числа будут отсортированы в порядке возрастания.

10. Дан текстовый файл, содержащий информацию о городах мира (название города, название страны, численность населения, площадь). Выполнить следующие задания:

- Найти самый населенный город в Германии;
- Найти город, который имеет наибольшую плотность населения;
- Распечатать названия всех стран, которые имеют города с населением более 1000000 человек
- Найти количество городов, которые расположены во Франции.

11. В текстовом файле содержится текст на английском языке, заканчивающийся точкой (другие символы "." в тексте отсутствуют).Требуется написать программу, которая будет определять и выводить на экран английскую букву, встречающуюся в этом тексте чаще всего, и количество таких букв. Строчные и прописные буквы при этом считаются неразличимыми. Если искомых букв несколько, то программа должна выводить на экран первую из них по алфавиту.

12. В текстовом файле содержатся фамилии и имена студентов. Известно, что общее количество студентов не превосходит 100. В первой строке вводится количество зарегистрированных студентов (N). Далее следуют N строк, имеющих следующий формат: <Фамилия> <Имя>. Требуется написать программу, которая формирует и печатает уникальный логин для каждого студента по следующему правилу: если фамилия встречается первый раз, то логин – это данная фамилия, если фамилия встречается второй раз, то логин – это фамилия, в конец которой приписывается число 2 и т.д.

13. Текстовый файл содержит произвольные алфавитно-цифровые символы. Ввод этих символов заканчивается точкой. Требуется написать программу, которая будет печатать последовательность строчных английских букв ('a' 'b'... 'z') из входной последовательностии частоты их повторения. Печать должна происходить в алфавитном порядке.

Занятия 5-8 (8 часов). Алгоритмы поиска и сортировки (по книге Никлаус Вирт «Алгоритмы и структуры данных»).

1. Дан текстовый файл, в котором записана отсортированная последовательность целых чисел. Написать программу поиска в нем местоположения заданного пользователем числа, используя последовательный, бинарный и интерполяционный поиск. Подсчитать

количество сравнений, которые выполняют эти алгоритмы. Сравнить их эффективность по количеству сравнений.

2. Дан текстовый файл, в котором записана отсортированная последовательность целых чисел. Написать программу поиска в нем местоположения заданной пользователем последовательности, используя алгоритм последовательного поиска, алгоритм Кнута-Морриса-Прата и алгоритм Боуера-Мура). Подсчитать количество сравнений, которые

выполняют эти алгоритмы. Сравнить их эффективность по количеству сравнений.

3. Дан текстовый файл, в котором записана последовательность целых чисел. Написать программу сортировки массива с помощью алгоритмов внутренней сортировки (сортировка вставками, метод пузырька, сортировка выбором, сортировка Шелла, пирамидальная сортировка и быстрая сортировка). Подсчитать количество сравнений, которые выполняют эти алгоритмы. Сравнить их эффективность по количеству сравнений.

4. Дан текстовый файл, в котором записана последовательность целых чисел. Написать программу сортировки массива с помощью алгоритмов внешней сортировки (простое слияние, естественное слияние, многофазная сортировка).

Занятия 9-12 (8 часов). Использование классов-коллекций (ArrayList, HashTable, Stack, Queue) (динамические структуры данных).

1. Дан текстовый файл, в котором записана последовательность целых чисел. Сформировать из них два списка – в одном только четные элементы, в другом – нечетные.

2. Использование списка для представления полинома. Написать функции:

- Ввода полинома;
- Печати полинома;
- Вычисления полинома в заданной точке;
- Получения суммы двух полиномов;
- Получения произведения двух полиномов;
- Получения производной полинома;
- Получения первообразной полинома.

3. Использование списка для представления разреженной матрицы (матрицы с большим количеством нулевых элементов). Написать функции:

- Ввода матрицы;

- Печати матрицы;
- Суммирования двух матриц;
- Умножения двух матриц;
- Транспонирования матрицы.

4. Написать программу «Предметный указатель». Каждый компонент указателя содержит слово и номера страниц, на которых это слово встречается. Количество номеров страниц, относящихся к одному слову может быть любым. Предусмотреть возможность формирования указателя с клавиатуры и из файла, печати предметного указателя, сохранения в файл, вывода номеров страниц для заданного слова, добавления и удаления элемента из указателя.

5. Написать программу «Адресная книжка». Каждая запись в книжке содержит имя адресата, дату рождения и список номеров телефона (домашнего, мобильного и пр.). Предусмотреть возможность формирования адресной книжки с клавиатуры и из файла, печати адресной книжки, поиска записи по какому-либо признаку (фамилии, дате рождения или номеру телефона), добавления и удаления записей, сохранения в файл.

6. Написать программу «Каталог библиотеки». Каждая запись каталога содержит информацию о книге – название, автор, количество экземпляров, количество экземпляров «на руках». Предусмотреть возможность формирования каталога с клавиатуры и из файла, печати каталога, сохранения в файл, поиска книги по какому-либо признаку (например, автору или названию), добавления книг в библиотеку, удаления книг из нее, фиксации получения или возврата книги читателем.

7. Дан односвязный список, содержащий целые числа. Написать функцию, которая за один проход по списку распечатывает эти числа следующим образом: сначала все четные числа в обратном порядке их следования в списке, затем все нечетные числа в прямом порядке их следования. Для решения задачи использовать стек и очередь.

8. Дан односвязный список, содержащий целые числа. Написать функцию, которая за один проход по списку распечатывает эти числа таким образом, что сначала распечатаются все отрицательные числа, затем – все положительные. Элементы, равные нулю, печататься не должны. Для решения задачи использовать очередь.

9. Получить двоичное представление заданного целого числа, используя стек.

10. Лабиринт задан в виде прямоугольной матрицы, в которой используются следующие обозначения: 0 – возможен проход, 1 – стена. Даны координаты позиция путника в лабиринте. С помощью стека распечатать путь, по которому путник может выйти из лабиринта, если выход существует. Считается, что путник вышел из лабиринта, если он находится на первой или последней строке матрицы, или на первом или последнем столбце. Путник может двигаться только по горизонтали или по вертикали.

11. Дана символьная строка, которая содержит правильное скобочное

выражение. Для каждой пары скобок (открывающей и соответствующей ей закрывающей) распечатать номера их позиций в строке, упорядочив пары:

- по возрастанию номеров открывающих скобок;
- по возрастанию номеров закрывающих скобок.

12. Дана символьная строка, содержащая правильно записанное математическое выражение следующего вида:

<формула>::=<цифра>|M(<формула>,<формула>)|m(<формула>,<формула>)

M – операция вычисления max из двух выражений, m - операция вычисления min из двух выражений. Написать функцию вычисления значения этого выражения.

13.Дана символьная строка, содержащая правильно записанное логическое выражение следующего вида:

<формула> ::= T | F | And(<формула> , <формула>) |Or(<формула> , <формула>)

Not (<формула>)And – операция логического И, Or –операция логического ИЛИ, Not -операция логического НЕ.

Написать функцию вычисления этого выражения (функция должна возвращать true, если значение выражения равно T, false – в противном случае).

14. Дана символьная строка, содержащая постфиксную форму правильно записанного арифметического выражения, операндами которого являются цифры. Написать функцию вычисления этого выражения.

15. Можно использовать следующий алгоритм. Выражение просматривается слева направо. Если встречается цифра, то она заносится в стек. Если встречается знак операции, то из стека извлекаются два операнда, над ними выполняется операция и ее результат записывается в стек. Когда выражение заканчивается, в стеке остается одно число – значение выражения.

Занятия 13-14 (4 часов). Алгоритмы на графах.

1. Дан граф в виде матрицы смежности. Определить длины всех путей из узла A в узел B.

2. Дан взвешенный граф в виде матрицы смежности. Найти кратчайший путь между двумя заданными вершинами (s – начальная вершина, t – конечная вершина).

3. Дан взвешенный граф в виде матрицы смежности. Найти кратчайшие пути между всеми парами вершин.

4. Дан взвешенный граф в виде матрицы смежности. Найти диаметр графа, т.е. максимальное расстояние между всевозможными парами его вершин.

5. Дан взвешенный граф в виде матрицы смежности. Построить для него каркас минимального веса.

6. Дан граф. Написать функцию построения его дополнения.

Дополнением графа G называется граф G’ с тем же множеством вершин, что и в G, причем две вершины в G’ смежны тогда и только тогда, когда они не смежны в G.

7. Дан граф. Написать функцию определяющую, является ли он полным. Граф называется полным, если любые две его вершины соединены ребром.

8. Дан граф. Написать функцию поиска всех источников (стоков) графа. Источник графа – это вершина, из которой достижимы все остальные вершины (из этой вершины можно построить путь в любую вершину графа). Сток графа – это вершина, достижимая из всех других вершин.

9. Дан граф. Написать функцию поиска всех вершин графа, недостижимых из заданной вершины.

10. Даны граф и номера трех его вершин. Написать функцию поиска всех путей из первой вершины во вторую, которые проходят через третью вершину.

11. Даны граф и номера двух его вершин. Написать функцию поиска всех путей из первой вершины во вторую, длина которых больше заданной величины.

12. Дан граф. Написать функцию, которая находит путь максимальной длины, исходящий из заданной вершины.

13. Дан граф. Написать функцию поиска пути максимальной длины в графе.

14. Дан граф. Написать функцию поиска медианы графа, т.е. такой его вершины, что сумма расстояний от нее до остальных вершин минимальна.

15. Дан граф. Написать функцию, определяющую является ли он связным. Связным называется граф, из каждой вершины которого можно построить путь в любую другую его вершину.

16. Дан граф. Написать функцию, определяющую является ли он деревом. Деревом называется связный граф без циклов.

Занятия 15-16 (4 часов). Алгоритмы работы с деревьями сортировки.

1. Дан текстовый файл, содержащий последовательность целых чисел. Написать функцию создания на его основании дерева сортировки.

2. Дано дерево сортировки. Написать функцию печати элементов дерева (рекурсивный и не рекурсивный варианты).

3. Дано дерево сортировки. Написать функцию вычисления глубины дерева.

4. Дано дерево сортировки. Написать функцию вычисления количества узлов на заданном уровне дерева.

5. Дано дерево сортировки. Написать функцию подсчета количества повторяющихся элементов в узлах дерева.

6. Дано дерево сортировки. Написать функцию нахождения максимального четного значения в дереве.

7. Даны два бинарых дерева. Определить, подобны ли они. Два дерева являются подобными, если либо оба они пусты, либо оба не пусты и при этом их левые и правые поддеревья являются подобными, т.е. деревья имеют одинаковую структуру.

8. Дано дерево сортировки. Написать функцию добавления в дерево нового элемента.

9. Дано дерево сортировки. Написать функцию удаления из дерева заданного элемента.

ПРИМЕРНЫЕ ТЕСТОВЫЕ ВОПРОСЫ

Вопрос 1. Какие из указанных ниже структур данных относятся к встроенным:

1) списки;
2) целый тип;
3) дерево;
4) стек.

Вопрос 2. Какая из ниже перечисленных структур данных отличается наличием вершины:

1) дерево;
2) множество;
3) стек;
4) массив.

Вопрос 3. Описание

```
Var
  i, j : integer;
  x : real;
  s: string;
```

объявляет переменные. Переменная s будет является переменной типа:

1) целый;
2) действительный;
3) строка;
4) массив.

Вопрос 4. Упорядоченная совокупность элементов некоторого типа, адресуемых при помощи одного или нескольких индексов, называется:

1) массив;
2) дерево;
3) стек;
4) список.

Вопрос 5. Структура данных, объединяющая элементы данных разных типов, называется:

1) массив;
2) дерево;
3) стек;
4) запись.

Вопрос 6. Какие существуют типы указателей:

1) переменные;
2) типизированные;
3) динамические;
4) статические.

Вопрос 7. Структура данных, состоящая из элементов, содержащих такое число указателей, которое позволяет организовать их одновременно в виде нескольких различных списков;

1) мегасписок;
2) n-список;
3) мультисписок;
4) дублирующий список.

Вопрос 8. Структуру данных стек можно организовать с помощью:

1) массивов;
2) деревьев;
3) записей;
4) графов.

Вопрос 9. Частным случаем графа является:

1) стек;
2) очередь;
3) дерево;
4) матрица.

Вопрос 10. В бинарном дереве из каждой вершины выходит:

1) произвольное количество дуг;
2) не более двух дуг;
3) не более трех дуг;
4) четное количество дуг.

Вопрос 11. Какой метод поиска применяется только к отсортированным массивам:

1) линейный поиск;
2) КМП-поиск;
3) двоичный поиск;
4) алгоритм Боуера и Мура.

Вопрос 12. Пузырьковая сортировка относится к:

1) сортировке выбором;
2) обменной сортировке;
3) сортировке вставкой;
4) усовершенствованной сортировке.

Вопрос 13. К усовершенствованным алгоритмам сортировки относится:

1) сортировка Шелла;
2) пузырьковая сортировка;
3) сортировка выбором;
4) сортировка вставкой.

Вопрос 14. Хеширование данных используется для:

1) поиска элементов в таблице;
2) заполнения таблицы значениями;
3) предварительного упорядочивания элементов в таблице;
4) обмена элементов в таблице.

Вопрос 15. К методам ускорения доступа к данным относится:
1) метод разрешения коллизий;
2) метод вставки;
3) рехеширование;
4) метод выбора.

Вопрос 16. Способ организации вычислительного процесса, при котором подпрограмма (процедура или функция) в ходе выполнения составляющих её операторов обращается сама к себе:
1) итерация;
2) рекурсия;
3) цикл;
3) условие.

Вопрос 17. Какая из ниже перечисленных структур данных отличается наличием «хвоста» и «головы»:
1) дерево;
2) множество;
3) стек;
4) очередь.

Вопрос 18. Структура данных, в которой отсутствует возможность индексирования отдельных элементов, называется:
1) массив;
2) запись;
3) стек;
4) множество.

Вопрос 19. Структуры данных, в процессе обработки которых изменяются не только значения переменных, но и сама их организация, называются:
1) статические структуры;
2) матрицы;
3) динамические структуры;
4) файлы.

Вопрос 20. Описание данных, представленное следующим образом

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

представляет собой:

1) очередь;
2) линейный список;
3) массив;
4) стек.

Вопрос 21. Структуру данных очередь можно организовать с помощью:
1) указателей;
2) деревьев;
3) записей;
4) графов.

Вопрос 22. Структуру данных дерево можно организовать с помощью:
1) деревьев;
2) записей;
3) курсоров;
4) графов.

Вопрос 23. Алгоритм поиска, при котором осуществляется сдвиг слова не на один символ на каждом шаге, а на некоторое переменное количество символов, называется:
1) алгоритм Боуера и Мура;
2) алгоритм прямого поиска;
3) алгоритм бинарного поиска;
4) алгоритм Кнута, Мориса и Пратта.

Вопрос 24. Какой метод поиска, основывается на необычном соображении – сравнение символов начинается с конца слова, а не с начала:
1) линейный поиск;
2) КМП-поиск;
3) двоичный поиск;
4) алгоритм Боуера и Мура.

Вопрос 25. Сортировка Хоара относится к:
1) сортировке выбором;
2) обменной сортировке;
3) сортировке вставкой;
4) усовершенствованной сортировке.

Вопрос 26. К усовершенствованным алгоритмам сортировки относится:
1) пузырьковая сортировка;
2) быстрая сортировка;
3) сортировка выбором;
4) сортировка вставкой.

Вопрос 27. К методам разрешения коллизий относится:
1) поиск элементов в таблице;
2) метод цепочек;
3) метод умножения;
4) метод свертки.

Вопрос 28. К методам ускорения доступа к данным относится:
1) метод вставки;

2) метод хеширования;
3) рехеширование;
4) метод выбора.

Вопрос 29. Какая из ниже перечисленных структур данных отличается наличием корня:

1) дерево;
2) множество;
3) стек;
4) массив.

Вопрос 30. Структура данных, объединяющая элементы данных одного типа, называется:

1) массив;
2) дерево;
3) стек;
4) запись.

Вопрос 31. Структуру данных список можно организовать с помощью:

1) стеков;
2) деревьев;
3) массивов;
4) графов.

Вопрос 32. Какой метод поиска применяется к последовательностям данных:

1) линейный поиск;
2) КМП-поиск;
3) двоичный поиск;
4) алгоритм Боуера и Мура.

Вопрос 33. Дерево, у которого ветви каждого узла упорядочены:

1) бинарное дерево;
2) сильноветвящееся дерево;
3) дерево поиска;
4) связанное дерево.

Вопрос 34. Арность дерева – это:

1) количество узлов;
2) число ветвей, выходящих из узла;
3) число ветвей;
4) количество поддеревьев.

Вопрос 35. Упорядоченные n-арные деревья, где n>2, называется:

1) бинарным деревом;
2) сильноветвящимся деревом;
3) деревом поиска;
4) связанным деревом.

Вопрос 36. Сортировка Шелла относится к:

1) сортировке выбором;
2) обменной сортировке;
3) сортировке вставкой;

4) внешней сортировке.

Вопрос 37. К алгоритмам внешней сортировки относится:

1) сортировка Шелла;
2) пузырьковая сортировка;
3) простое слияние;
4) сортировка вставкой.

Вопрос 38. Сортировка, применяемая к данным, которые хранятся во внешней памяти компьютера:

1) сортировка выбором;
2) обменная сортировка;
3) сортировка вставкой;
4) внешняя сортировка.

Вопрос 39. Сортировка, применяемая к данным, которые хранятся в оперативной памяти компьютера:

1) сортировка выбором;
2) внутренняя сортировка;
3) сортировка вставкой;
4) внешняя сортировка.

Вопрос 40. Сортировка простым выбором относится к:

1) внешней сортировке;
2) пузырьковой сортировке;
3) внутренней сортировке;
4) сортировке Шелла.

Вопрос 41. Метод разрешения коллизий относится к:

1) методу вставки;
2) рехешированию;
3) ускорению доступа к данным;
4) методу выбора.

Вопрос 42. Квадратичное опробование относится к методу:

1) свёртки;
2) разрешения коллизий;
3) деления;
4) цепочек.

СПИСОК РЕКОМЕНДУЕМОЙ ЛИТЕРАТУРЫ

1. Абрамов, С.А. Лекции о сложности алгоритмов [Электронный ресурс]: учебное пособие/ Абрамов С.А.— Электрон. текстовые данные.– М.: МЦНМО, 2009. – Режим доступа: http://www.iprbookshop.ru/3959. – ЭБС «IPRbooks», по паролю.

2. Вирт, Н. Алгоритмы и структуры данных [Электронный ресурс]: учебное пособие/ Вирт Н.– Электрон. текстовые данные.– М.: ДМК Пресс, 2010. – 272 с. – Режим доступа: http://www.iprbookshop.ru/7965.— ЭБС «IPRbooks», по паролю.

3. Вирт, Н. Алгоритмы и структуры данных: пер. с англ. [Текст]: учеб. пособие/ Вирт Н. – СПб.: Невский Диалект , 2008.- 352 с.

4. Воеводин, В.В. Вычислительная математика и структура алгоритмов [Электронный ресурс]: учебное пособие/ Воеводин В.В.— Электрон. текстовые данные. – М.: Московский государственный университет имени М.В. Ломоносова, 2010. – 168 с. – Режим доступа: http://www.iprbookshop.ru/13042.– ЭБС «IPRbooks», по паролю.

5. Игошин, В.И. Математическая логика и теория алгоритмов [Текст]: учеб. пособие для вузов / В.И. Игошин – М.: Академия, 2008.- 448 с.

7. Комлева, Н.В. Структуры и алгоритмы компьютерной обработки данных [Электронный ресурс]: учебное пособие/ Комлева Н.В.– Электрон. текстовые данные. – М.: Евразийский открытый институт, Московский государственный университет экономики, статистики и информатики, 2004.— 140 с.– Режим доступа: http://www.iprbookshop.ru/10898. – ЭБС «IPRbooks», по паролю.

6. Маньшин, М.Е. Математическая логика и теория алгоритмов [Электронный ресурс]: учебное пособие/ Маньшин, М.Е.- М.: Волгоградский институт бизнеса,Вузовское образование, 2013. – 106 с.– Режим доступа: http://www.iprbookshop.ru/11334.- ЭБС «IPRbooks», по паролю.

7. Никлаус, Вирт Алгоритмы и структуры данных. Новая версия для Оберона [Электронный ресурс]/ Никлаус Вирт – Электрон. текстовые данные.— М.: ДМК Пресс, 2010. – 272 с. – Режим доступа: http://www.iprbookshop.ru/7965.– ЭБС «IPRbooks», по паролю.

8. Сундукова, Т.О. Структуры и алгоритмы компьютерной обработки данных [Электронный ресурс]/ Сундукова Т.О., Ваныкина Г.В.— Электрон. текстовые данные. – М.: Интернет-Университет Информационных Технологий (ИНТУИТ), 2011. – 475 с.– Режим доступа: http://www.iprbookshop.ru/16736.— ЭБС «IPRbooks», по паролю

Дополнительная литература

1. Ахо А. Структуры и алгоритмы / А. Ахо, Д. Хопкрофт - М.: Вильямс, 2001.

2. Балюкевич, Э.Л. Математическая логика и теория алгоритмов [Электронный ресурс]: учебное пособие/ Балюкевич Э.Л.- М.: Евразийский

открытый институт, 2009.– 230 с.- Режим доступа: http://www.iprbookshop.ru/10863.- ЭБС «IPRbooks», по паролю

3. Бентли Дж. Жемчужины программирования, 2-е изд. / Дж.Бентли – СПб.: Питер, 2002.

4. Вирт Н. Алгоритмы и структуры данных / Н.Вирт - М.: Мир, 1989.

5. Вирт Н. Алгоритмы + структуры данных = программы / Н.Вирт - М.: Мир, 1985.

6. Вирт Н. Алгоритмы и структуры данных / Н.Вирт – СПб.: Невский диалект, 2001.

7. Дмитриева М.В. Турбо Паскаль и Турбо Си: построение и обработка структур данных: учебное пособие / М.В. Дмитриева, А.А. Кубенский – СПб.: Изд-во С.-Петербургского университета, 1995.

8. Кнут Д. Искусство программирования. т.3. Сортировка и поиск: пер. с англ. / Д.Кнут – М.: Вильямс, 2000.

9. Кортмен Т. Алгоритмы: построение и анализ / Т. Кортмен, Ч. Лейзеррсон, Р. Ривест – М: МЦНМО, 2000.

10. Лэнгсам Й. Структуры данных для персональных ЭВМ / Й. Лэнгсам, М. Огенстайн – М.: Мир, 1989.

11. Лэнгсам Й. Структуры данных для персональных ЭВМ / Й. Лэнгсам, М. Огенстайн, А. Тененбау – М.: Мир, 1989.

12. Матьяш В.А. Структуры и алгоритмы обработки данных / В.А. Матьяш, В.А. Путилов, В.В. Фильчаков, С.В. Щекин – Апатиты: КФ ПетрГУ, 2000.

13. Павлов Л.А. Структуры и алгоритмы обработки данных в ЭВМ. Алгоритмы на графах: консп. лекций / Л.А. Павлов – Чуваш. ун-т: Чебоксары, 2001.

14. В.Н. Пичугин Р.В. Фёдоров Структуры и алгоритмы компьютерной обработки данных Учебное пособие / Чуваш. ун-т: Чебоксары, 2008.

15. Сибуя М. Алгоритмы обработки данных / М. Сибуя, Т. Ямамото – М.: Мир, 1986.

16. Ускова О.Ф. Программирование алгоритмов обработки данных / О.Ф. Ускова, Н.В. Огаркова, И.Е. Воронина, М.В. Мельников – СПб.: БХВ-Петербург, 2003.

ОСНОВНЫЕ ПОНЯТИЯ И ОПРЕДЕЛЕНИЯ (ГЛОССАРИЙ)

В дальнейшем часто придётся использовать термины, связанные с процессом разработки и функционирования программ, поэтому здесь приведён краткий «словарик», чтобы уже не возвращаться к проблеме определений. Для составления этого «словаря» использованы в основном материалы сайта «Глоссарий. Ру» (http://glossary.ru).

Алгоритм – точное предписание исполнителю совершить определённую последовательность действий для достижения поставленной цели за конечное число шагов.

Данные – сведения: полученные путём измерения, наблюдения, логических или арифметических операций; и представленные в форме, пригодной для постоянного хранения, передачи и (автоматизированной) обработки.

Тип данных – характеристика набора данных, которая определяет: диапазон возможных значений данных из набора; допустимые операции, которые можно выполнять над этими значениями; способ хранения этих значений в памяти.

Различают:

- простые типы данных: целые, действительные числа, символы, строки, логические величины;
- составные типы данных: массивы, файлы и др.

Программа – согласно ГОСТ 19781-90 – данные, предназначенные для управления конкретными компонентами системы обработки информации в целях реализации определённого алгоритма.

Алгоритмический язык (язык программирования)

Язык программирования – искусственный (формальный) язык, предназначенный для записи алгоритмов. Язык программирования задаётся своим описанием и реализуется в виде специальной программы: компилятора или интерпретатора.

Транслятор – в широком смысле – программа, преобразующая текст, написанный на одном языке, в текст на другом языке.

Транслятор – в узком смысле – программа, преобразующая: программу, написанную на одном (входном) языке в программу, представленную на другом (выходном) языке.

Транслятор языка программирования – программа, преобразующая исходный текст программы на языке программирования в машинный язык вычислительной системы, на которой эта программ должна выполняться.

Интерпретатор – транслятор, способный параллельно переводить и выполнять программу, написанную на алгоритмическом языке высокого уровня.

Компилятор – программа, преобразующая текст, написанный на алгоритмическом языке, в программу, состоящую из машинных команд. Компилятор создаёт законченный вариант программы на машинном языке.

Константа – в программировании – элемент данных, который занимает место в памяти, имеет имя и определённый тип, причём его значение никогда не меняется.

Переменная – в языках программирования – именованная часть памяти, в которую могут помещаться разные значения переменной. Причём в каждый момент времени переменная имеет единственное значение (или единственный набор значений). В процессе выполнения программы значение переменной может изменяться.

Тип переменных определяется типом данных, которые они представляют.

Подпрограмма – самостоятельная часть программы, которая разрабатывается независимо от других частей и затем вызывается по имени.

Функция – подпрограмма, которая на основе некоторых данных (аргументов функции) вычисляет значение некоторой переменной («функция возвращает значение»).

Объект – понятие объектно-ориентированного программирования, программный модуль, объединяющий в единое целое данные и программы, манипулирующие данными. Объект характеризуется свойствами, которые являются параметрами объекта и методами, которые позволяют воздействовать на объект и его свойства.

Метод – действие в виде процедуры, которое выполняет объект (иногда говорят – выполняется над объектом).

Идентификатор – символическое имя объекта, переменной или подпрограммы, которые однозначно идентифицируют их в программе.

Выражение – конструкция на языке программирования, предназначенная для выполнения вычислений. Выражение состоит из операндов, объединённых знаками операций. Различают арифметические, логические и символьные выражения.

Операнд – константа, переменная, функция, выражение или другой объект языка программирования, над которым производятся операции.

Арифметическая операция – вычислительная операция над числами.

Во многих языках программирования определены двуместные арифметические операции: сложения, вычитания, умножения, деления, деления нацело, вычисление остатка от деления.

Логическая операция – операция над логическими («булевскими») операндами, принимающими значения «Истина» или «Ложь». Наиболее распространёнными являются следующие операции:

- многоместное логическое сложение;
- многоместное логическое умножение;
- одноместное логическое отрицание.

(«Многоместная» операция означает, что в ней может два и более операндов, а в «одноместной» или «унарной» операции участвует только один операнд).

Операция отношения производит сравнение двух величин. Результат операции отношения является «булевской» переменной, принимающей значение «Истина» (True или логическая 1) или «Ложь» (False или логический 0).

Массив (массив данных) – совокупность, как правило, однотипных данных, каждое из которых идентифицируется с именем массива и индексом (индексами).

В зависимости от количества индексов массивы бывают одномерные (линейные), двухмерные и т.д.

Индекс – номер (или номера, если массив данных многомерный), добавляемый к имени массива, чтобы идентифицировать каждый элемент данного массива.

Например, a[1,3] означает, что определён элемент двухмерного массива a с индексом 1,3 (строка – 1, столбец – 3).

Присваивание – операция записи значения в переменную. В каждом языке программирования определён оператор присваивания. Если в переменную записывается новое значение, старое стирается.

Цикл (циклические вычисления) означают многократное выполнение одних и тех же операций. В зависимости от задачи различаются циклы с переменной (со счётчиком, с известным количеством повторений) и циклы с условием (цикл повторяется, пока не выполнится условие завершение цикла).

Зацикливание – для циклов с условием – ситуация, при которой условие завершения цикла никогда не выполняется.

ЭРКЕНОВА Мадина Умаровна

АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ

Учебно-методическое пособие для
обучающихся II курса по специальности
09.03.04 «Программная инженерия»

Корректор Чагова О.Х.
Редактор Чагова О.Х.

Сдано в набор 14.05.2025 г.
Формат 60x84/16
Бумага офсетная.
Печать офсетная.
Усл. печ. л. 7,9
Заказ № 5103
Тираж 100 экз.

Оригинал-макет подготовлен
в Библиотечно-издательском центре СКГА
369000, г. Черкесск, ул. Ставропольская, 36