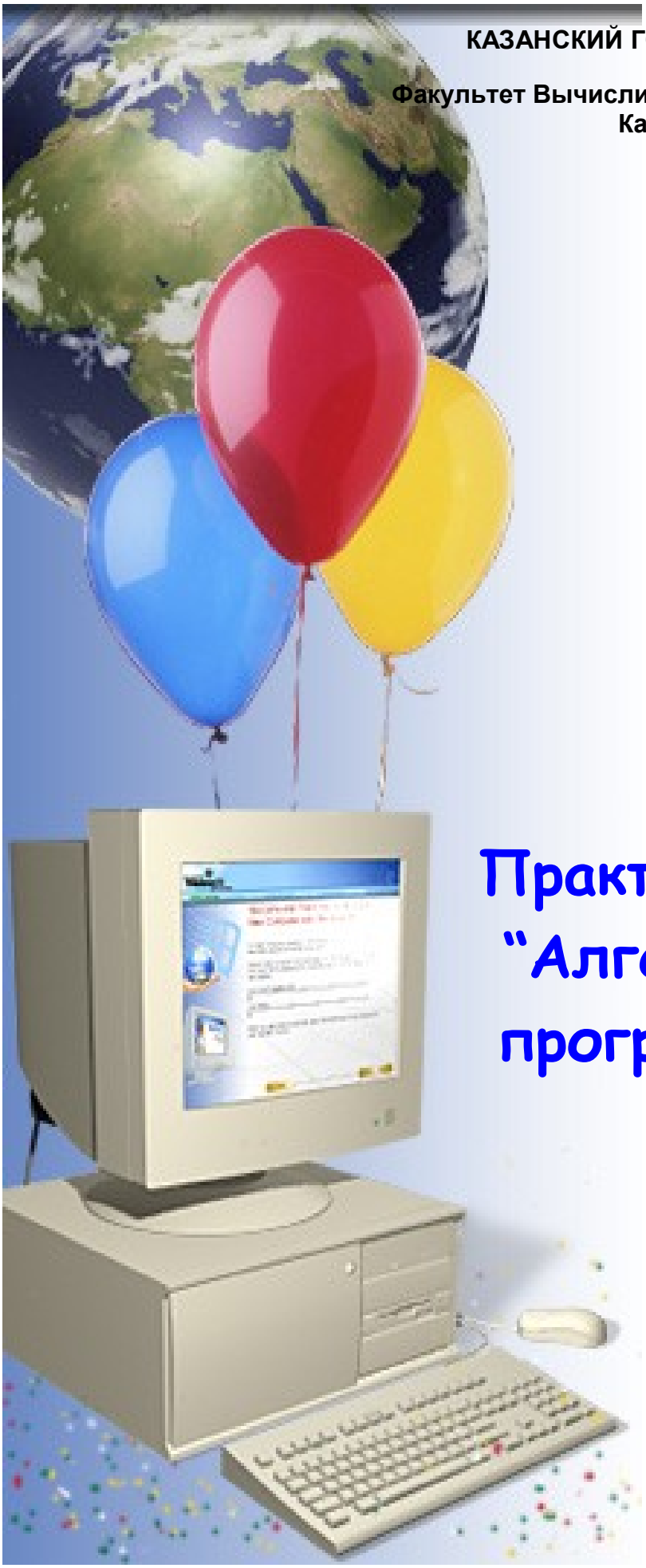
КАЗАНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Факультет Вычислительной Математики и Кибернетики
Кафедра Экономической Кибернетики



АНДРИАНОВА А.А.,
ИСМАГИЛОВ Л.Н.,
МУХТАРОВА Т.М.

Практикум по курсу "Алгоритмизация и программирование"

Часть 2

КАЗАНЬ - 2009

УДК 004.43

ББК 32.973.26 – 018.1

Печатается по постановлению редакционно-издательского совета факультета вычислительной математики и кибернетики Казанского государственного университета

Рецензенты:

кандидат физико-математических наук, заведующий кафедрой информационных технологий Института экономики, управления и права (г. Казань) **И.А. Фукин**

старший преподаватель кафедры системного анализа и информационных технологий Казанского государственного университета **Р.Р. Тагиров**

Андрианова А.А., Исмагилов Л.Н., Мухтарова Т.М.
Практикум по курсу “Алгоритмизация и программирование”. Часть 2: Учебное пособие / А.А. Андрианова, Л.Н. Исмагилов, Т.М. Мухтарова. – Казань: Казанский государственный университет, 2009. – 133 с.

В данном учебном пособии рассмотрены принципы структурирования программ (создание пользовательских функций), применение динамических структур данных (списки, стеки, очереди, деревья, графы) и основные операции с ними, а также примеры алгоритмов решения задач с использованием таких структур. Данное пособие предназначено для ведения практических занятий на 1, 2 курсах специальностей 080116.65 «Математические методы в экономике» и 080700.62 «Бизнес-информатика», а также может использоваться студентами других специальностей.

© Казанский государственный университет, 2009
© Андрианова А.А., Исмагилов Л.Н., Мухтарова Т.М., 2009

Содержание

Введение.....4
Глава 1. Структура программы.....6
Раздел 1. Функции 6
Домашнее задание.....34
Раздел 2. Рекурсивные функции.....38
Домашнее задание.....49
Глава 2. Линейные динамические структуры данных.....52
Раздел 3. Односвязные списки.....52
Домашнее задание.....66
Домашнее задание.....75
Раздел 5. Двусвязные списки.....80
Домашнее задание.....87
Глава 3. Нелинейные структуры данных.....89
Раздел 6. Деревья.....89
Домашнее задание.....103
Раздел 7. Графы.....105
Домашнее задание.....132
Литература 134

Введение

В этом учебном пособии рассмотрены принципы структурирования программ, применение динамических структур данных и базовые алгоритмы работы с ними. Учебное пособие предназначено для проведения практических занятий по курсам «Алгоритмизация и программирование» и «Информатика и программирование» у студентов 1-2 курсов факультета ВМК специальностей 080116.65 «Математические методы в экономике» и 080700.62 «Бизнес-информатика», а также может использоваться студентами младших курсов других специальностей, начинающих изучать программирование.

Данное учебное пособие состоит из трех глав. Первая глава посвящена принципам структурного программирования. Решение сложных задач обычно заключается в составлении большого алгоритма и, соответственно, в написании длинного программного кода его реализации. Для более эффективной разработки осуществляется разбиение этого алгоритма на небольшие и относительно независимые составные части, каждая из которых решает менее сложную отдельную подзадачу. Каждая из этих составных частей оформляется в виде отдельной подпрограммы, которая называется функцией. Глава 1 состоит из двух разделов: «Функции» и «Рекурсивные функции».

Вторая глава посвящена использованию линейных динамических структур данных. Их главная особенность – нефиксированный размер данных, в отличие от массивов и матриц, при создании которых явно указывается их размер. Глава состоит из трех разделов, каждый из которых посвящен отдельным видам линейных динамических структур: односвязным и двусвязным спискам, стекам и очередям. Для рассмотренных структур данных описываются основные операции работы с ними, и приводится ряд приложений их использования.

Третья глава посвящена построению и использованию нелинейных структур данных на примере деревьев и графов. Рассматриваются принципы создания, хранения и работы с ними. Глава состоит из двух разделов – «Деревья» и «Графы».

В каждом разделе имеется теоретическое описание приемов работы с рассматриваемыми конструкциями, подробно разбираются несколько задач, а также приводится список задач для самостоятельного решения. При разборе каждой задачи приводится ее постановка, обсуждение используемого алгоритма решения, а также программа с комментариями, написанная на языке программирования C++. Программы отлажены с помощью оболочки проекти-

рования Microsoft Visual Studio 2005.

Глава 1. Структура программы

Раздел 1. Функции

В программировании функция – это самостоятельная единица программы, которая решает конкретную задачу. Использование функций при создании программ создает программисту ряд удобств. Во-первых, это избавляет от повторного программирования. Если какие-либо действия надо выполнить в программе несколько раз, создаем соответствующую функцию, а затем будем ее вызывать. Во-вторых, мы можем использовать одну и ту же функцию в нескольких программах. В-третьих, применение функций облегчает понимание и отладку программы.

Функция – это преобразователь некоторых данных, которые по отношению к функции являются входными, в результат – выходные данные функции.

Для определения функции в языке C++ используется следующая конструкция, которая состоит из заголовка и тела:

```
// Заголовок функции
ТипВозврЗначения ИмяФункции (СписокФормПараметров)
// Тело функции
{
      . . .
      return Результат;
}
```

где ТипВозврЗначения (тип возвращаемого значения) – тип данных результата работы функции, СписокФормПараметров (список формальных параметров) – список имен объектов с указанием их типов, которые представляют собой входные данные функции:

```
СписокФормПараметров::=Тип1 имяОб1, Тип2 имяОб2, … ТипN имяОбN
```

Формальный параметр определяет локальную переменную функции, которая будет хранить значение какого-либо входного объекта.

Тело функции содержит программную реализацию решения конкретной задачи. Результат работы функции возвращается с помощью оператора `return`.

Обращение к функции происходит посредством ее вызова. При вызове на место формальных параметров подставляются фактические (аргументы). Типы аргументов должны совпадать с типами данных соответствующих формальных параметров или, в некоторых случаях, приводиться к ним.

Например, функция нахождения минимума из двух чисел выглядит следующим образом:

```
// функция определения минимального из двух чисел
double MinElement(double a, double b)
{
      if (a < b)  return a;
      else return b;
}


void main(void)
{
      int a=3, b=6;
      double x=4.6, y=8.1;
      // функция вызывается со значениями фактических
      // параметров x и y
      // результат работы функции записывается в переменную c
      double c=MinElement(x,y);
      . . .
      // функция вызывается со значениями фактических
      // параметров a и b, приведенных к типу double
      // результат работы функции записывается в переменную c
      c=MinElement((double)a,(double)b);
      . . .
      // функция вызывается со значениями фактических
      // параметров a и y,
      // целый параметр a приводится к типу double
      // результат работы функции записывается в переменную c
      c=MinElement((double)a,y);
      . . .
}
```

Если функция в программе вызывается до ее определения, то в этом случае ее необходимо предварительно объявить. Объявлением (прототипом) функции является конструкция вида

```
ТипВозврЗначения ИмяФункции (СписокТиповФормПараметров)
```

где СписокТиповФормПараметров (список типов формальных парамет-

ров) – список типов объектов, представляющих собой входные данные функции:

```
СписокТиповФормПараметров::=Тип1, Тип2, … ТипN
```

В список типов формальных параметров можно также добавить и имена объектов. Тогда конструкция объявления совпадает с заголовком самой функции.

Иногда возникают ситуации, когда необходимо изменить значение фактических параметров внутри функции. Поэтому в некоторых языках существует способ передачи параметров по ссылке на переменную, т. е. в функцию передается не само значение переменной-параметра, а адрес памяти, по которому находится эта переменная. Тогда работа внутри функции происходит непосредственно с этой областью памяти. В разных языках программирования параметр-ссылка обозначается по-разному. В языке C++ для обозначения ссылки используется знак '&', в C# – `ref`, в Pascal – `var`.

Например, требуется написать функцию, которая меняет местами значения двух переменных. Приведем два варианта функции:

1) параметры передаются по значению

```
void Change1 (int a, int b)
{
      int t=a; a=b; b=t;
}
```

2) параметры передаются по ссылке

```
void Change2 (int& a, int& b)
{
      int t=a; a=b; b=t;
}
```

При вызове первый вариант функции не решает требуемую задачу.

```
void main(void)
{
      int x=10, y=40;
      Change1(x,y);      // результат - x=10, y=40
      Change2(x,y);      // результат - x=40, y=10;
      . . .
}
```

Параметрами функции (входными и выходными данными) могут являться массивы и матрицы. Напомним, что массив – это набор переменных одина-

кового типа, которые расположены в непрерывной области памяти друг за другом. Обращение к этим переменным осуществляется с помощью указания имени массива и порядкового номера переменной в нем. В языке C++ имя массива является указателем, содержащим адрес его первого элемента. На рис.1.1 изображен массив *a*, содержащий *n* элементов, имеющих порядковые номера от 0 до *n*-1.

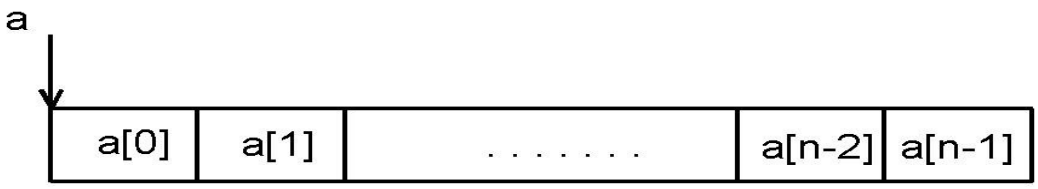


Рис.1.1. Схема расположения элементов массива в памяти.

Таким образом, при передаче массива в функцию в качестве параметров передаются имя массива и количество элементов в нем.

Пусть требуется написать функцию, определяющую, является ли массив симметричным. Прототип этой функции выглядит следующим образом:

```
bool Symmetry(int * a, int n);
```

Функция возвращает значение `true`, если массив симметричный, `false` – в противном случае.

Приведем также определение этой функции:

```
// определение функции проверки симметричности массива
bool Symmetry(int * a, int n)
{
      int i;
      for(i=0; i<n/2; i++)
          if(a[i] != a[n-1-i])
              return false; // нашлась несимметричная пара
      return true;
}
```

В случае параметров-матриц также следует передать имя матрицы и ее размер, который задается двумя параметрами – количеством строк и столбцов (в случае квадратной матрицы можно указать только один размер).

Организовать хранение матрицы в памяти можно так, как представлено на рис.1.2. При таком способе хранения матрица представляет собой набор

массивов, каждый из которых является строкой матрицы. Адреса всех строк матрицы образуют отдельный массив указателей, имя которого и является именем матрицы. Размер этого массива соответствует количеству строк матрицы.

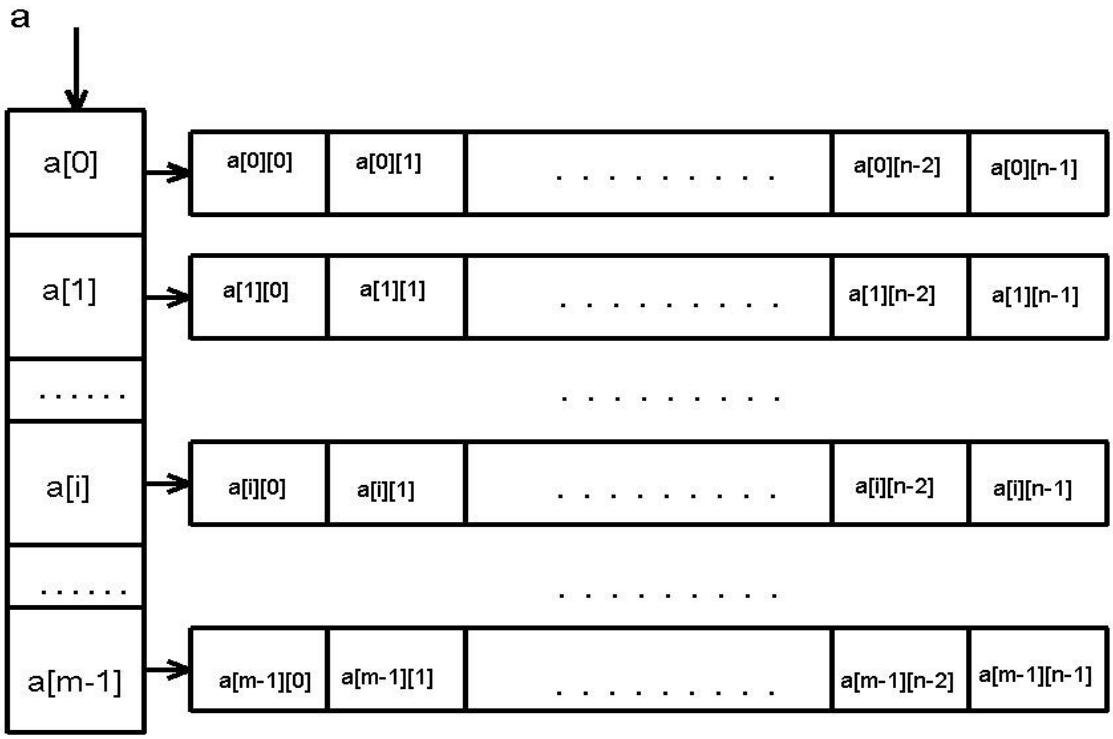


Рис.1.2. Схема расположения элементов матрицы в памяти

На рис.1.2 изображена схема хранения матрицы *a* размера *m* x *n*. Здесь *a* – имя матрицы, указывающее на первый элемент массива адресов строк, *a*[*i*] (*i* = 0,…, *m*-1) – указатель на первый элемент *i*-ой строки матрицы.

Приведем программный код выделения памяти под матрицу, соответствующий данной схеме.

```
// выделение памяти под матрицу a размера m x n
// с элементами типа float
float** a=new float*[m];    // выделение памяти под массив
                            // указателей на первые
                            // элементы строк
if(a==NULL)
{
     printf("Нет требуемого объема памяти.\n");
     return;
}

// выделение памяти под каждую строку матрицы
for(i=0;i<m;i++)
```

```
{
     a[i]=new float[n];
     if(a[i]==NULL)
     {
          printf("Нет требуемого объема памяти.\n");
          // освобождение уже выделенной памяти
          for(j=0; j<i; j++)
               delete [] a[j];
          delete [] a;
          return;
     }
}
```

Рассмотрим пример передачи матрицы в функцию в качестве параметра.

Пусть требуется проверить, является ли квадратная матрица симметричной относительно главной диагонали. Прототип этой функции имеет следующий вид:

```
bool Symmetry(int ** a, int n);
```

Функция возвращает значение `true`, если матрица симметрична, `false` – в противном случае.

Приведем также определение этой функции:

```
// определение функции проверки симметричности матрицы
bool Symmetry(int ** a, int n)
{
     int i,j;
     for(i=0; i<n-1; i++)
          for(j=i+1; j<n;j++)
               if(a[i][j] != a[j][i])
                    return false;
     return true;
}
```

Если функция должна менять адрес расположения массива, то указатель на его первый элемент должен передаваться по ссылке, т.е. с указанием символа '&'.

Далее приводится описание процесса решения некоторых задач с применением пользовательских функций.

Задача 1. Дано четное число $n>2$. Проверить, выполняется ли для этого числа гипотеза Гольдбаха: каждое четное число $n>2$ может быть представлено

в виде суммы двух простых чисел.

Напишем отдельную функцию, определяющую, является ли число простым. Напомним, что простыми являются натуральные числа, большие или равные двум, делящиеся только на единицу и на само себя. Если число простое, функция `PrimeNumber` возвращает 1, иначе – 0. Само число передается в функцию как параметр.

В главной программе проверяем на «простоту» все возможные слагаемые. Если найдем пару простых чисел, составляющих в сумме заданное число, значит, гипотеза Гольдбаха подтвердилась, и выведем соответствующее сообщение. Если же таких чисел нет, то мы опровергли гипотезу.

Заметим, что проверку гипотезы можно также оформить в виде отдельной функции.

```
# include <stdio.h>
# include <math.h>

// прототип функции проверки, является ли число k простым
int PrimeNumber(int k);

void main(void)
{
      int n,i;
      printf("Введите число:");
      while(true)
      {
           scanf("%d",&n);
           if(n%2==1 || n<=2)
                printf("Введите четное положительное число,
                          большее двух");
           else break;
      }
      // проверка гипотезы Гольдбаха
      int f=0;
      for(i=2;i<=n/2; i++)
           if(PrimeNumber(i))
                if(PrimeNumber(n-i))
                {
                     f=1;
                     printf("%d=%d+%d\n",n,i,n-i);
                }
      if(f==0)
           printf("Гипотеза не верна\n");
}
```

```
// определение функции проверки,
// является ли число k простым
int PrimeNumber(int k)
{
      if(k<2) return 0;
      int p=1,i;
      for(i=2;i<=sqrt(k);i++)
           if(k%i==0)
           {
                p=0;
                break;
           }
      return p;
}
```

Задача 2 . Определить, принадлежит ли точка треугольнику.

Для решения этой задачи используем следующее свойство: если точка принадлежит треугольнику, то сумма площадей треугольников, образованных этой точкой и вершинами треугольника, совпадает с площадью исходного треугольника.

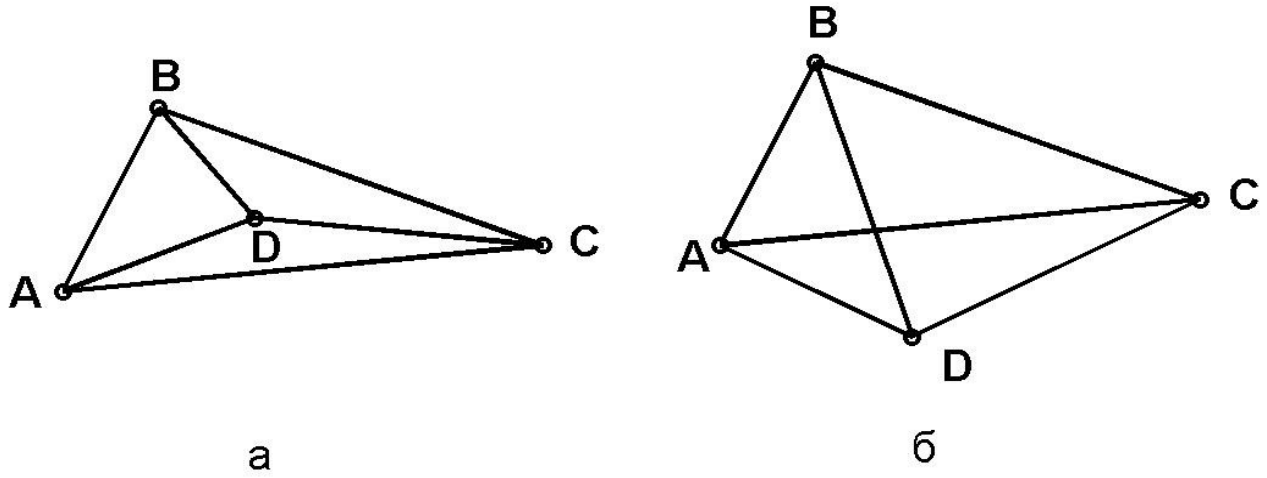


Рис. 1.3. Два случая расположения точки D относительно треугольника ABC.

На рис. 1.3 представлен треугольник ABC и случаи, когда точка D принадлежит ему (рис. 1.3 **а**) и не принадлежит (рис. 1.3 **б**). В случае (**а**) площадь треугольника ABC равна сумме площадей треугольников ABD, BDC и ADC, а в случае (**б**) она меньше суммы площадей треугольников ABD, BDC и ADC. Поэтому решение задачи сводится к вычислению площадей треугольников.

Площадь треугольника вычисляем по формуле Герона. Ее реализует функция `AreaTriangle`. Передаваемые в функцию значения соответствуют

координатам трех вершин треугольника. Возвращаемое значение – величина площади.

В главной программе вызываем функцию `AreaTriangle` четыре раза.

При вычислении площади используется стандартная функция вычисления квадратного корня. Она не дает точного значения. В случае, когда точка принадлежит треугольнику, сумма площадей внутренних треугольников может незначительно отличаться от площади исходного треугольника. Поэтому сравнение на равенство площадей происходит с точностью ε.

```
# include <stdio.h>
# include <math.h>

// прототип функции вычисления площади треугольника
float AreaTriangle(float x1, float y1,
                   float x2, float y2,
                   float x3, float y3);

void main(void)
{
      printf("Введите координаты вершин треугольника:");
      float x1,x2,x3,y1,y2,y3,x,y,eps;
      while(true)
      {
           scanf("%f",&x1);
           scanf("%f",&y1);
           scanf("%f",&x2);
           scanf("%f",&y2);
           scanf("%f",&x3);
           scanf("%f",&y3);
           if(!((x1!=x2 || y1!=y2) && (x1!=x3 || y1!=y3) &&
                                (x2!=x3 || y2!=y3)))
                printf("Неверно заданы координаты вершин
                         треугольника");
           else break;
      }
      printf("Введите координаты точки:");
      scanf("%f",&x);scanf("%f",&y);
      eps=0.0001;

      // вычисление площадей треугольников
      float s=AreaTriangle(x1,y1,x2,y2,x3,y3);
      float s1=AreaTriangle(x,y,x2,y2,x3,y3);
      float s2=AreaTriangle(x1,y1,x,y,x3,y3);
      float s3=AreaTriangle(x1,y1,x2,y2,x,y);
```

```
        if(s1+s2+s3-s < eps)
            printf("Точка принадлежит треугольнику");
        else
            printf("Точка не принадлежит треугольнику");
}

// опредение функции вычисления площади треугольника
// по формуле Герона
float AreaTriangle(float x1, float y1,
float x2, float y2,
float x3, float y3)
{
        float a=sqrt((x2-x1)*(x2-x1)+(y2-y1)*(y2-y1));
        float b=sqrt((x3-x1)*(x3-x1)+(y3-y1)*(y3-y1));
        float c=sqrt((x2-x3)*(x2-x3)+(y2-y3)*(y2-y3));
        float p=(a+b+c)/2;
        float s=sqrt(p*(p-a)*(p-b)*(p-c));
        return s;
}
```

Задача 3. Дана квадратная матрица A. Вычислить A^p (p >1 – целое число).

Для решения этой задачи создаем две функции: основную (`MatrixPower`) и вспомогательную (`MatrixProduct`). Вспомогательная функция будет перемножать две квадратные матрицы. Ее параметрами будут две перемножаемые матрицы, матрица-результат и размерность.

Для возведения матрицы A в степень p в функции `MatrixPower` (p-1) раз вызывается вспомогательная функция `MatrixProduct`.

```
# include <stdio.h>

// прототип функции вычисления произведения
// двух квадратных матриц
void MatrixProduct(int n, float**a, float**b, float**c);

// прототип функции возведения матрицы в степень
bool MatrixPower(int n, float**a, int p, float** c);

void main(void)
{
        int n,p,i,j;
        while(true)
        {
            printf("Введите порядок матрицы:");
```

```
            scanf("%d",&n);
            if(n<1)
                printf("Введите положительный порядок
                        матрицы.\n");
            else break;
        }
        float** a=new float*[n];
        if(a==NULL)
        {
            printf("Нет требуемого объема памяти.\n");
            return;
        }
        for(i=0;i<n;i++)
        {
            a[i]=new float[n];
            if(a[i]==NULL)
            {
                printf("Нет требуемого объема памяти.\n");
                for(j=0; j<i; j++)
                    delete [] a[j];
                delete [] a;
                return;
            }
        }
        float** c=new float*[n];
        if(c==NULL)
        {
            printf("Нет требуемого объема памяти.\n");
            return;
        }
        for(i=0;i<n;i++)
        {
            c[i]=new float[n];
            if(c[i]==NULL)
            {
                printf("Нет требуемого объема памяти.\n");
                for(j=0; j<i; j++)
                    delete [] c[j];
                delete [] c;
                return;
            }
        }
        printf("Введите элементы матрицы A:");
        for(i=0;i<n;i++)
            for(j=0;j<n;j++)
```

```
                scanf("%f",&a[i][j]);
        while(true)
        {
            printf("Введите степень p:");
            scanf("%d", &p);
            if(p<=0)
                printf("Введите положительную степень.\n");
            else break;
        }
        // вызов функции возведения матрицы a в степень p
        // результат записывается в матрицу c
        bool f=MatrixPower(n,a,p,c);
        if(f==false)
        {
            printf("Произошла системная ошибка.\n");
            return;
        }
        printf("Полученная матрица:\n");
        for(i=0;i<n;i++)
        {
            for(j=0;j<n;j++)
                printf("%f\t",c[i][j]);
            printf("\n");
        }
        for(i=0;i<n;i++)
        {
            delete [] a[i];
            delete [] c[i];
        }
        delete [] a;
        delete [] c;
}

// определение функции возведения матрицы в степень
bool MatrixPower(int n, float**a, int p, float** c)
{
        int i,j,k;
        float** b; // вспомогательная матрица, в которой
        // запоминается результат возведения матрицы
        // a в промежуточную степень, меньшую p
        b=new float*[n];
        if(b==NULL)
            return false;
        for(i=0;i<n;i++)
        {
```

```
            b[i]=new float[n];
            if(b[i]==NULL)
            {
                for(j=0; j<i; j++)
                    delete [] b[j];
                delete [] b;
                return false;
            }
        }
        // вначале матрица b совпадает с матрицей a
        // в первой степени
        for(i=0;i<n;i++)
            for(j=0;j<n;j++)
                b[i][j]=a[i][j];
        // возведение в степень матрицы a
        for(k=0;k<p-1;k++)
        {
            //вызов функции перемножения двух матриц
            MatrixProduct(n,a,b,c);
            // сохранение результата перемножения в матрице b
            for(i=0;i<n;i++)
                for(j=0;j<n;j++)
                    b[i][j]=c[i][j];
        }
        for(i=0;i<n;i++)
            delete [] b[i];
        delete [] b;
        return true;
    }

    // определение функции перемножения двух квадратных матриц
    void MatrixProduct(int n, float**a,float**b,float**c)
    {
        int i,j,k;
        for(i=0;i<n;i++)
            for(j=0;j<n;j++)
            {
                c[i][j]=0;
                for(k=0;k<n;k++)
                    c[i][j]=c[i][j]+a[i][k]*b[k][j];
            }
    }
```

Задача 4. Дана прямоугольная матрица. Написать функцию удаления из этой матрицы строк, содержащих только нулевые элементы.

Рассмотрим первый вариант решения этой задачи, когда создается новая

матрица для хранения результата. Программа содержит три функции:

1) функция `IsNullRow` проверяет, состоит ли строка только из нулевых элементов. Ее параметрами являются размер строки матрицы (количество столбцов) и указатель на проверяемую строку. Функция возвращает значение `true`, если строка является «нулевой», `false` – в противном случае;

2) функция `CountNullRows` осуществляет подсчет «нулевых» строк матрицы. В нее передается исходная матрица (ее имя и размеры). Функция возвращает целое число – количество «нулевых» строк;

3) функция `RemoveNullRowMatrix` формирует новую матрицу, не включая в нее «нулевые» строки исходной матрицы. Ее параметрами являются размеры исходной матрицы и указатель на нее. Для определения размера и адреса расположения новой матрицы в функцию в качестве параметров передаются еще и две переменные-ссылки: указатель на новую матрицу (`float**& b`) и количество строк в ней (`int& m1`).

```
# include <stdio.h>

// прототип функции удаления строк только
// с нулевыми элементами
void RemoveNullRowMatrix(int m, int n, float** a,
                            float**& b, int& m1);

// прототип функции определения, состоит ли строка матрицы
// только из нулевых элементов
bool IsNullRow(int n, float* a);

// прототип функции подсчета количества
// нулевых строк матрицы
int CountNullRows(int m, int n, float** a);

void main(void)
{
      int m,n,      // размеры исходной матрицы
          m1,       // количество строк матрицы-результата
          i,j;
      float**res;   // указатель на матрицу-результат
      while(true)
      {
          printf("Введите размеры матрицы:");
          scanf("%d",&m);
          scanf("%d",&n);
          if(n<1 || m<1)
              printf("Введите положительные размеры
```

```
                    матрицы.\n");
        else break;
    }
    // выделение памяти под исходную матрицу
    float** a=new float*[m];
    if(a==NULL)
    {
        printf("Нет требуемого объема памяти.\n");
        return;
    }
    for(i=0;i<m;i++)
    {
        a[i]=new float[n];
        if(a[i]==NULL)
        {
            printf("Нет требуемого объема памяти.\n");
            for(j=0; j<i; j++)
                delete [] a[j];
            delete [] a;
            return;
        }
    }
    printf("Введите элементы матрицы:");
    for(i=0;i<m;i++)
        for(j=0;j<n;j++)
            scanf("%f",&a[i][j]);

    // вызов функции формирования матрицы
    // без «нулевых» строк
    RemoveNullRowMatrix(m,n,a,res,m1);
    // проверка существования матрицы-результата
    if(res!=NULL)
    {
        // существуют «ненулевые» строки
        printf("Полученная матрица:\n");
        for(i=0;i<m1;i++)
        {
            for(j=0;j<n;j++)
                printf("%f\t",res[i][j]);
            printf("\n");
        }
        for(i=0;i<m1;i++)
            delete [] res[i];
        delete [] res;
    }
```

```
        else
            // все строки исходной матрицы были «нулевыми»
            if(m1==0)
                printf("Все строки матрицы удалены");
        for(i=0;i<n;i++)
            delete [] a[i];
        delete [] a;
}

// определение функции проверки, состоит ли строка матрицы
// только из нулевых элементов
bool IsNullRow(int n, float* a)
{
        int i;
        for(i=0;i<n;i++)
            if(a[i]!=0)
                return false;
        return true;
}

// определение функции подсчета количества
// нулевых строк матрицы
int CountNullRows(int m, int n, float** a)
{
        int i,k=0;
        for(i=0;i<m;i++)
            if(IsNullRow(n,a[i]))
                k++;
        return k;
}

// определение функции удаления строк только
// с нулевыми элементами
void RemoveNullRowMatrix(int m, int n, float** a,
                         float**& b, int& m1)
{
        int i,j,t;
        // m1 - количество «ненулевых» строк матрицы
        m1=m-CountNullRows(m,n,a);
        if(m1==0)
        {
            // все строки исходной матрицы «нулевые»
            b=NULL;
            return;
        }
```

```
        // выделение памяти под матрицу-результат
        b=new float*[m1];
        if(b==NULL)
        {
            printf("Нет требуемого объема памяти.\n");
            return;
        }
        for(i=0;i<m1;i++)
        {
            b[i]=new float[n];
            if(b[i]==NULL)
            {
                printf("Нет требуемого объема памяти.\n");
                for(j=0; j<i; j++)
                    delete [] b[j];
                delete [] b;
                return;
            }
        }

        t=0;     // номер строки исходной матрицы
        for(i=0;i<m1;i++)
        {
            // находим номер первой «ненулевой» строки
            // исходной матрицы, начиная с t-ой
            while(IsNullRow(n,a[t])) t++;
            // найденная «ненулевая» строка копируется на
            // i-ое место в матрицу-результат
            for(j=0;j<n;j++)
                b[i][j]=a[t][j];
            // переход к следующей строке исходной матрицы
            t++;
        }
    }
```

Рассмотрим второй вариант решения этой задачи – исходная матрица преобразуется посредством удаления из нее нулевых строк. Данный вариант решения возможен и эффективен только в случае использования схемы хранения матрицы, изображенной на рис.1.2. Преобразование матрицы происходит путем переприсвоения указателей на строки. Когда встречатся «нулевая» строка, освобождается занимаемая ею память, а указателю на эту строку в дальнейшем будет присвоен адрес начала первой всретившейся после нее «ненулевой» строки, если такая существует.

Этот вариант функции `RemoveNullRowMatrix` в качестве параметров

получает адрес исходной матрицы и ее размеры, которые передаются по ссылке, поскольку их значения могут измениться.

```
// определение функции удаления строк только
// с нулевыми элементами
void RemoveNullRowMatrix(int& m, int& n, float** a)
{
      int i,j,t;
      // определение количества «ненулевых» строк
      int m1=m-CountNullRows(m,n,a);
      if(m1==0)
      {
           // если все строки матрицы «нулевые»,
           // уничтожаем матрицу
           for(i=0;i<m;i++)
                delete [] a[i];
           delete [] a;
           m=0;
           n=0;
           return;
      }

      // вводим переменную для хранения
      // количества удаленных «нулевых» строк
      t=0;
      for(i=0;i<m;i++)
           if(IsNullRow(n,a[i]))
           {
                t++;
                delete [] a[i];
           }
           else
                a[i-t]=a[i];
      m=m1;
}
```

Задача 5. Написать функцию сложения двух многочленов.

Многочлен степени n можно представить в виде массива размерности (n+1). Элемент массива с индексом i хранит коэффициент слагаемого, степень которого равна i.

Правила суммирования многочленов можно сформулировать следующим образом:

1) если степени суммируемых многочленов не равны, степень многочле-

на-суммы совпадает с наибольшей степенью многочленов-слагаемых, а коэффициенты вычисляются как сумма коэффициентов при соответствующих степенях;

2) если степени суммируемых многочленов равны, степень многочлена-суммы вычисляется следующим образом. Начиная с наибольших степеней, осуществляется поиск первой ненулевой суммы коэффициентов многочленов-слагаемых при соответствующих степенях. Степень данных слагаемых и будет степенью многочлена-результата.

В функцию сложения передаем массивы коэффициентов и переменные, хранящие степени исходных многочленов. Последний параметр возвращает массив коэффициентов многочлена-суммы, а его степень передается как возвращаемое значение функции.

```
# include <stdio.h>

// прототип функции сложения двух многочленов
int SumPolynom (float*p1, int n1, float* p2, int n2,
                float*& p);

void main(void)
{
      int m,n,i;
      while(true)
      {
           printf("Введите степень многочлена A:");
           scanf("%d",&m);
           printf("Введите степень многочлена B:");
           scanf("%d",&n);

           if(m<=0 || n<=0)
                printf("Введите положительные
                         степени многочленов.\n");
           else break;
      }
      float* a=new float[m+1];
      float* b=new float[n+1];
      if(a==NULL || b==NULL)
      {
           printf("Нет требуемого объема памяти.\n");
           return;
      }

      printf("Введите коэффициенты многочлена A:");
      for(i=m;i>=0;i--)
```

```
            scanf("%f",&a[i]);
        printf("Введите коэффициенты многочлена B:");
        for(i=n;i>=0;i--)
            scanf("%f",&b[i]);
        float* c;

        int k = SumPolynom(a,m,b,n,c);
        for(i=k; i>0; i--)
            printf("%f*x^%d+",c[i],i);
        printf("%f\n",c[i]);

        delete [] a; delete [] b; delete [] c;
}

// опредение функции сложения двух многочленов
int SumPolynom(float*p1, int n1, float*p2,int n2,float*& p)
{
        int i,j;
        if(n1>n2)
        {
            p=new float[n1+1];
            for(i=0;i<n2+1;i++)
                p[i]=p1[i]+p2[i];
            for(i=n2+1;i<n1+1;i++)
                p[i]=p1[i];
            return n1;
        }
        else if(n1<n2)
            {
                p=new float[n2+1];
                for(i=0;i<n1+1;i++)
                    p[i]=p1[i]+p2[i];
                for(i=n1+1;i<n2+1;i++)
                    p[i]=p2[i];
                return n2;
            }
            else // случай, когда n1=n2
            {
                for(j=n1;j>=0; j--)
                    if(p1[j]+p2[j]!=0)
                        break;
                p=new float[j+1];
                for(i=0;i<j+1;i++)
                    p[i]=p1[i]+p2[i];
                return j;
```

```
            }
}
```

Приведем еще один вариант решения задачи с использованием пользовательского типа данных, определяющего многочлен. Такие типы данных вводятся в программу для описания множества объектов, определяемых одинаковым набором атрибутов. Например, пользовательский тип данных для описания множества многочленов может содержать два атрибута – адрес массива коэффициентов многочлена и его степень.

В большинстве языков программирования существует специальная конструкция, позволяющая определить пользовательский тип данных, в котором набор переменных различных типов хранится как единое целое. Эту конструкцию называют структурой (C++, C#, Java) или записью (Pascal). Определим структуру для хранения многочлена следующим образом:

```
struct Polynom
{
      float* a;// массив коэффициентов многочлена
      int n;   // степерь многочлена
};
```

Такое представление многочленов позволяет передавать в функцию и возвращать из функции объекты структурного типа `Polynom`, при этом осуществляется передача всех его атрибутов. При обращении к атрибутам (полям) структуры в языке C++ существуют две операции '. ' и '->'. Операция '.' применяется к переменной структурного типа, а операция '->' – к указателю на структуру. Например,

```
Polynom p1;         // создание переменной структурного типа
p1.n=10;            // задание поля n многочлена p1
p1.a=new float[p1.n+1]; // выделение памяти под массив
                        // коэффициентов многочлена p1
// выделение памяти под хранение многочлена и
// сохранение адреса в переменной-указателе p2
Polynom * p2=new Polynom;
p2->n=10;
p2->a=new float[p2->n+1];
```

Данный вариант решения задачи суммирования многочленов содержит несколько функций:

1) функция `CreatePolynom` предназначена для создания объекта структуры типа `Polynom`, т.е. выделения памяти под массив коэффициентов

многочлена. Степень многочлена передается в функцию в качестве параметра;

2) функция `ScanPolynom` предназначена для ввода коэффициентов многочлена с клавиатуры. Объект типа `Polynom`, коэффициенты которого требуется ввести, передается в функцию в качестве параметра;

3) функция `PrintPolynom` предназначена для вывода формулы многочлена на экран. Объект типа `Polynom` передается в функцию в качестве параметра;

4) функция `SumPolynom` осуществляет суммирование двух многочленов. Параметрами функции являются два объекта типа `Polynom`, которые являются операндами суммирования. Функция возвращает объект типа `Polynom`, хранящий результат суммирования.

```
# include <stdio.h>

// прототип функции выделения памяти под многочлен
Polynom CreatePolynom(int n);

// прототип функции ввода коэффициентов
// полинома с клавиатуры
void ScanPolynom(Polynom p);

// прототип функции вывода многочлена на экран
void PrintPolynom(Polynom p);

// прототип функции сложения двух многочленов
Polynom SumPolynom(Polynom p1, Polynom p2);

void main(void)
{
      int m,n,i;
      while(true)
      {
           printf("Введите степень многочлена A:");
           scanf("%d",&m);
           printf("Введите степень многочлена B:");
           scanf("%d",&n);
           if(m<=0 || n<=0)
                printf("Введите положительные
                         степени многочленов.\n");
           else break;
      }
```

```
        // выделение памяти под многочлены-операнды
        Polynom p1=CreatePolynom(m);
        Polynom p2=CreatePolynom(n);
        if(p1.a==NULL || p2.a==NULL)
        {
            printf("Нет требуемого объема памяти.\n");
            return;
        }

        printf("Введите коэффициенты многочлена A:");
        ScanPolynom(p1);
        printf("Введите коэффициенты многочлена B:");
        ScanPolynom(p2);

        Polynom p = SumPolynom(p1,p2);
        PrintPolynom(p);
        delete [] p1.a; delete [] p2.a; delete [] p.a;
}

// определение функции выделения памяти под многочлен
Polynom CreatePolynom(int n)
{
        Polynom p; // создание объекта для хранения многочлена
        p.n=n;    // определение степени многочлена
        p.a=new float[n+1];    // выделение памяти под массив
                               // коэффициентов
        if(p.a==NULL)
        {
            // памяти недостаточно
            p.n=0;
        }
        return p;
}

 // определение функции ввода коэффициентов
 // многочлена с клавиатуры
void ScanPolynom(Polynom p)
{
        int i;
        for(i=p.n;i>=0;i--)
            scanf("%f",&p.a[i]);
}

// определение функции вывода многочлена на экран
void PrintPolynom(Polynom p)
```

```
{
      int i;
      for(i=p.n; i>0; i--)
           printf("%f*x^%d+",p.a[i],i);
      printf("%f\n",p.a[i]);
}

// опредение функции сложения двух многочленов
// определение функции вычисления суммы двух многочленов
Polynom SumPolynom(Polynom p1, Polynom p2)
{
      int i,j;
      if(p1.n>p2.n)
      {
           // степень многочлена p1 больше, чем у p2
           Polynom p=CreatePolynom(p1.n);
           for(i=0;i<p2.n+1;i++)
                p.a[i]=p1.a[i]+p2.a[i];
           for(i=p2.n+1;i<p1.n+1;i++)
                p.a[i]=p1.a[i];
           return p;
      }
      else if(p1.n<p2.n)
           {
                // степень многочлена p2 больше, чем у p1
                Polynom p=CreatePolynom(p2.n);
                for(i=0;i<p1.n+1;i++)
                     p.a[i]=p1.a[i]+p2.a[i];
                for(i=p1.n+1;i<p2.n+1;i++)
                     p.a[i]=p2.a[i];
                return p;
           }
           else
           {
                // степени многочленов-операндов совпадают
                for(j=p1.n;j>=0; j--)
                     if(p1.a[j]+p2.a[j]!=0)
                          break;
                Polynom p=CreatePolynom(j);
                for(i=0;i<j+1;i++)
                     p.a[i]=p1.a[i]+p2.a[i];
                return p;
           }
}
```

Задача 6. Найти один из корней уравнения вида $f(x) = 0$ методом хорд на отрезке $[a, b]$ с точностью ε (точность по функционалу). Если корней на отрезке не существует, выдать сообщение об этом. Для применения метода хорд необходимо, чтобы:

1) функция являлась непрерывной,

2) значения функции $f(x)$ на концах отрезка имели противоположные знаки.

Для решения задачи создаем две функции: функцию вычисления значения $f(x)$ и функцию нахождения корня уравнения $f(x) = 0$ методом хорд.

Согласно методу хорд требуется построить отрезок, соединяющий точки $(a, f(a))$ и $(b, f(b))$, и найти точку пересечения этого отрезка с осью абсцисс. Обозначим точку пересечения через x. Далее сужается отрезок поиска корня путем замены одного из концов на значение x. В точку x переносится тот конец отрезка, значение функции в котором совпадает по знаку со значением $f(x)$.

На рис. 1.4 изображены варианты нахождения точки пересечения построенной хорды с осью абсцисс. На рис. 1.4 **а** приведено изменение левого конца отрезка (на следующей итерации $a = x$), на рис. 1.4 **б** – правого ($b = x$).

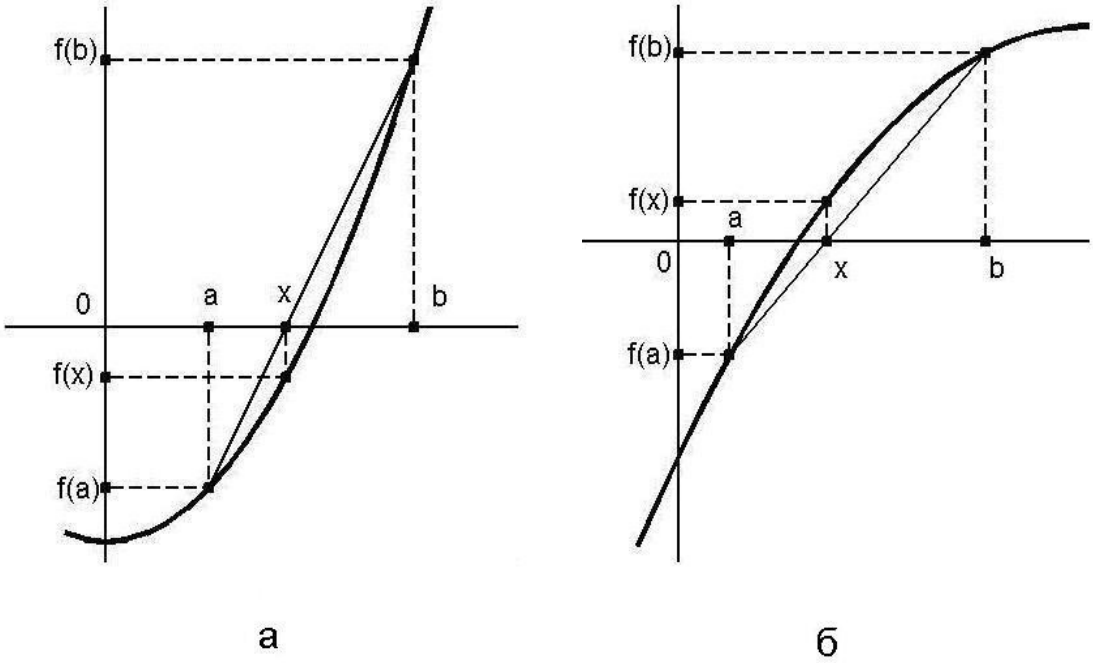


Рис. 1.4. Демонстрация одной итерации метода хорд.

```
# include <stdio.h>
# include <math.h>

// определение функции вычисления значения f(x) = x² -4
// может быть другая функция
```

```
float f(float x)
{
      return (x*x-4);
}

// прототип функции решения уравнения методом хорд
int RootEquation(float a, float b,float eps, float& x);

void main(void)
{
      printf("Введите концы отрезка a, b:");
      float a,b,eps;
      scanf("%f",&a);
      scanf("%f",&b);
      if(a>b)
      {
           float z=a;
           a=b;
           b=z;
      }
      eps=0.0001;
      float x;

      int f=RootEquation(a,b,eps,x);
      if(f==0)
           printf("Корней нет.\n");
      else
           printf("x=%f\n",x);
}

// определение функции решения уравнения методом хорд
int RootEquation(float a, float b,float eps, float& x)
{
      // вычисление значений f(x) на концах отрезка
      float fa = f(a), fb = f(b);
      // a=b
     if(a==b && fa==0)
      {
           x=a; return 1;// корень найден
      }
      if(fa*fb<0)
      {
           x=a;
           // условие выхода из цикла - значение функции в
           // одном из концов отрезка близко к нулю
```

```
            while((fa>eps || fa<-eps) && (fb>eps || fb<-eps))
            {
                // вычисление точки пересечения хорды
                // с осью абсцисс
                x=(a*fb-b*fa)/(fb-fa);
                // изменение отрезка поиска корня
                if(fa*f(x)>0)
                    a=x;
                else
                    b=x;
            }
            return 1; // корень найден
        }
        return 0; // корень не найден
}
```

Задача 7. Дана символьная строка, состоящая из строчных букв латинского алфавита и пробелов. Набор символов, не разделенных пробелами, образует слово. Требуется написать функцию шифрования символьной строки по следующему алгоритму:

1) определяется количество букв в слове минимальной длины. Обозначим это количество через *D*;

2) каждый символ строки за исключением пробела, заменяется символом, расположенным в алфавите через *D* символов после. Например, при *D*=3, символ 'a' заменяется символом 'd', символ 'b' заменяется символом 'e' и т.д. Для тех символов, после которых не существует *D* символов до конца алфавита, поиск замены продолжается, начиная с буквы 'a'. Например, при *D*=3, символ 'x' заменяется символом 'a', символ 'y' заменяется символом 'b' и символ 'z' заменяется символом 'c'.

Поскольку количество букв в слове минимальной длины может быть больше размера алфавита, смещение для нахождения символа-замены будет вычисляться по формуле `shift=D%26`.

```
# include <stdio.h>

// прототип функции вычисления длины символьной строки
int Length(char* str);

// прототип функции вычисления количества букв в слове
// минимальной длины
int MinLength(char* str);

// прототип функции шифрования символьной строки
```

```
char* EncodeString(char* str);

void main(void)
{
      char str[100];
      gets(str);
      char* res=EncodeString(str);
      puts(res);
      delete [] res;
}

// определение функции вычисления длины символьной строки
int Length(char* str)
{
      int i=0;
      // считаем символы, пока не встретился
      // символ конца строки '\0'
      while(str[i]!='\0')
           i++;
      return i;
}

// определение функции вычисления количества букв
// в слове минимальной длины
int MinLength(char* str)
{
      int min=0,        // длина слова минимальной длины
      i=0,k;
      while(str[i]!='\0')
      {
           // пропускаем пробелы
           while(str[i]==' ') i++;
           k=0;          // длина текущего слова
           // проход по символам текущего слова
           while(str[i]!=' ' && str[i]!='\0')
           {
                k++; i++;
           }
           // если встретилось первое слово (min=0)
           // или встретилось слово с меньшей длиной,
           // меняем значение переменной min
           if(min==0 || min>k) min=k;
      }
      return min;
}
```

```
// определение функции шифрования символьной строки
char* EncodeString(char* str)
{
      int i, ch;
      // определение длины исходной символьной строки
      int len=Length(str);
      // выделение памяти под строку-результат
      char* res=new char[len+1];
      // определение размера слова минимальной длины
      int D=MinLength(str);
      // определение смещения при шифровании
      int shift=D%26;
      // цикл замены символов
      for(i=0;i<len;i++)
      {
          if(str[i]!=' ')
          {
              // если символ не является пробелом,
              // вычисляем код символа-замены
              ch=str[i]+shift;
              if(ch>(int)'z')
                  // нахождение символа-замены,
                  // начиная с символа 'a'
                  res[i]='a'+(ch-'z'-1);
              else
                  res[i]=ch;
          }
          else res[i]=' ';
      }
      res[len]='\0';
      return res;
}
```

Домашнее задание

1. Дано натуральное число, меньшее 1 000 000. Написать функцию его перевода в римскую запись (результат представить в виде строки).

2. Даны координаты четырех точек на плоскости. Написать функцию

определения, являются ли эти точки вершинами параллелограмма.

3. Написать функцию вычисления площади выпуклого многоугольника. Координаты вершин многоугольника заданы в виде двух массивов – массива координат по оси абсцисс и массива координат по оси ординат. Вершины в массиве перечислены в порядке обхода по часовой стрелке.

4. Написать функцию вычисления радиуса описанной окружности выпуклого многоугольника, если она существует. Координаты вершин многоугольника заданы в виде двух массивов – массива координат по оси абсцисс и массива координат по оси ординат. Вершины в массиве перечислены в порядке обхода по часовой стрелке.

5. Даны координаты N точек на плоскости в виде двух массивов – массива координат по оси абсцисс и массива координат по оси ординат. Написать функцию нахождения трех точек из набора, которые образуют треугольник максимальной площади.

6. Даны координаты N точек на плоскости в виде двух массивов – массива координат по оси абсцисс и массива координат по оси ординат. Написать функцию, определяющую существуют ли в наборе четыре точки, образующие трапецию.

7. Дана прямоугольная матрица. Написать функцию нахождения седловой точки матрицы. Седловой точкой называется пара индексов (i, j) такая, что a_{ij} – минимальный элемент из i-ой строки и максимальный элемент из j-ого столбца.

8. Решить задачу 7 с использованием структуры `Matrix`, которую необходимо определить.

9. Написать функцию вычисления значения многочлена в заданной точке. Задачу решить с использованием структуры `Polynom`.

10. Написать функцию перемножения двух многочленов. Задачу решить с использованием структуры `Polynom`.

11. Написать функцию дифференцирования многочлена. Задачу решить с

использованием структуры `Polynom`.

12. Написать функцию сортировки массива методом вставок.

Метод вставок вкратце можно описать так. Пусть дан массив $a[0]\ldots a[n]$. Каждый элемент $a[i]$ (i изменяется от 1 до n) вставляется на подходящее место в уже упорядоченную совокупность $a[0]\ldots a[i\text{-}1]$. Это место определяется при обратном проходе по массиву от (i-1)-го элемента до элемента с меньшим значением, чем $a[i]$, или до начала массива.

13. Написать функцию решения системы линейных уравнений третьего порядка методом Крамера.

14. Написать функцию решения системы линейных уравнений n-ого порядка ($3 < n < 10$, n – целое число) методом Гаусса.

15. Написать функцию вычисления обратной матрицы с помощью метода Гаусса.

16. Написать функцию вычисления корня уравнения $f(x) = 0$ на отрезке $[a, b]$ с точностью $\varepsilon > 0$ методом деления отрезка пополам.

Полагаем, что отрезок задан так, что $f(a)f(b) < 0$, а функция $f(x)$ непрерывна на нем. В этом случае гарантируется, что корень на отрезке $[a, b]$ будет существовать (поскольку на границах отрезка функция принимает разные знаки, то она обязательно пересечет ось абсцисс, т.е. обязательно найдется x из $[a, b]$ такое, что $f(x) = 0$).

Метод деления отрезка пополам заключается в следующем:

1) вычислить $f(a)$ и $f(b)$;
2) вычислить середину отрезка $c = (a + b)/2$ и $f(c)$;
3) если $f(c) = 0$, то c – корень. Иначе,
4) если $f(a)f(c) < 0$, то $b = c$, иначе $a = c$;
5) если $b - a > \varepsilon$, то переход к п. 1. Иначе решение получено (любая точка из полученного отрезка, например, точка $(a + b)/2$).

17. Написать функцию вычисления с заданной точностью корня уравнения $f(x) = 0$ на отрезке $[a, b]$ методом касательных, где $f(x)$ – некоторая выпуклая функция. Полагаем, что отрезок задан так, что $f(a)f(b) < 0$.

Метод касательных заключается в следующем:

1) вычислить x_1 и x_2 – точки пересечения касательных к графику функции $f(x)$, построенных в точках a и b, с осью абсцисс. Выбрать

ту точку, которая принадлежит отрезку $[a, b]$. Запомнить ее в x;

2) вычислить $f(x)$;

3) если $f(a)f(x) < 0$, то $b = x$, иначе $a = x$.

4) если $|f(a)| < \varepsilon$, то решением является точка a; если $|f(b)| < \varepsilon$, то решением является точка b; в противном случае осуществляется переход к п. 1.

18. Написать функцию выравнивания символьной строки до заданного количества символов (если исходная строка короче, равномерно добавить между словами пробелы, в противном случае удалить лишние символы).

19. Дана символьная строка. Написать функцию определения количества вхождений в нее заданного слова.

20. Дана символьная строка. Написать функцию нахождения в ней симметричного слова максимальной длины.

21. Дана символьная строка. Написать функцию нахождения в ней слова максимальной длины, состоящего только из гласных букв.

22. Дана символьная строка. Написать функцию удаления из нее всех слов, имеющих заданную длину.

23. Дана зашифрованная символьная строка по алгоритму, приведенному в задаче 7. Написать функцию расшифровки данной символьной строки.

Раздел 2. Рекурсивные функции

Рекурсивные функции – это функции, в теле которых содержится вызов самих себя. Обычно их используют в ситуациях, когда легко свести исходную задачу к задаче того же вида, но с другими исходными данным, например, уменьшение размерности задачи, переход в новую точку и пр.

Каждый новый вызов рекурсивной функции предполагает сохранение состояния всех переменных функции в специальной области памяти, называемой стеком. После выполнения рекурсивной функции, управление передается оператору, следующему за ее вызовом. При этом состояние всех переменных восстанавливается из стека. Поэтому применение таких функций считается очень затратным с точки зрения памяти и не подходит для решения задач большой размерности. Тем не менее, существует множество задач, в которых рекурсивные алгоритмы более просты и дают компактные программы.

Следует помнить, что любая рекурсивная функция должна иметь обязательное условие выхода из рекурсии. При выполнении этого условия управление передается в место предыдущей рекурсивной итерации, откуда произошел рекурсивный вызов или в вызывающую функцию.

Далее приводится описание процесса решения некоторых задач с применением рекурсивных функций.

Задача 1 . Вычислить $n!$.

Когда рекурсия заключается в сведении к задаче меньшей размерности, алгоритм аналогичен обратному ходу метода математической индукции.

Очевидно, что $n!=n*(n-1)!$ Эта формула и определяет основное рекуррентное соотношение – $(n-1)!$ вычисляется с помощью вызова функции с фактическим параметром, принимающим значение $n-1$. Условие выхода из рекурсии – это сведение задачи к тривиальному случаю – $1!=1$, $0!=1$.

```
# include <stdio.h>

// прототип функции вычисления факториала
long int Factorial(int n);

void main(void)
{
      int n;
```

```
        printf("Введите n:");
        while(true)
        {
            scanf("%d",&n);
            if(n<0)
                printf("Введите положительное число.\n");
            else break;
        }

        long int k=Factorial(n);// первый вызов функции
        if(k>0)
            printf("%d!=%d\n",n,k);
}


// определение функции вычисления факториала
long int Factorial(int n)
{
        // для отрицательных чисел факториал не определен
        if(n<0) return -1;
        // условие выхода из рекурсии
        if(n==1 || n==0)
            return 1;
        return n*Factorial(n-1);// рекурсивный вызов
}
```

На примере решения этой задачи рассмотрим, каким образом выполняется рекурсивный вызов функции. Допустим, пользователь ввел значение переменной *n*, равное 4.

Главная функция содержит вызов функции `Factorial(4)`. Для выполнения вызова функции в оперативной памяти выделяется некоторая область, необходимая для запоминания адреса возврата, создания копий параметров и копий локальных переменных.

Свободная область памяти в момент первого вызова `Factorial(4)` на рис. 2.1 изображена в виде пустого прямоугольника. Внутри прямоугольника показано значение параметра *n*.

Вычисление факториала в функции `Factorial(4)` начинается с проверки условия выхода из рекурсии (`n==1` или `n==0`). Так как оно не выполняется, происходит второй вызов `Factorial(3)`, который является первым рекурсивным вызовом функции. Для второго вызова также выделяется некоторое количество памяти (рис. 2.2).

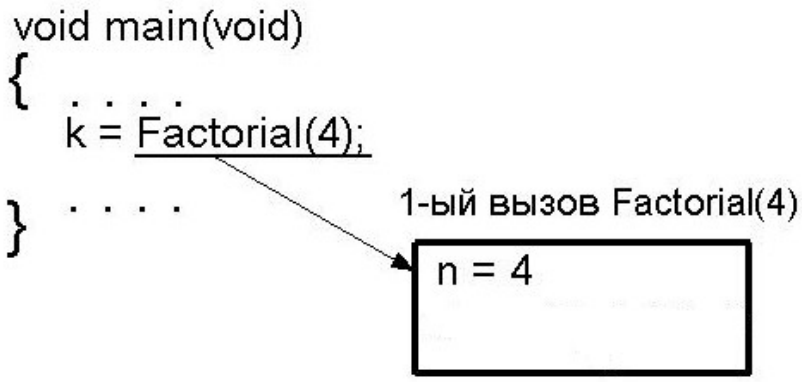


Рис. 2.1. Демонстрация первого вызова функции `Factorial(4)`.

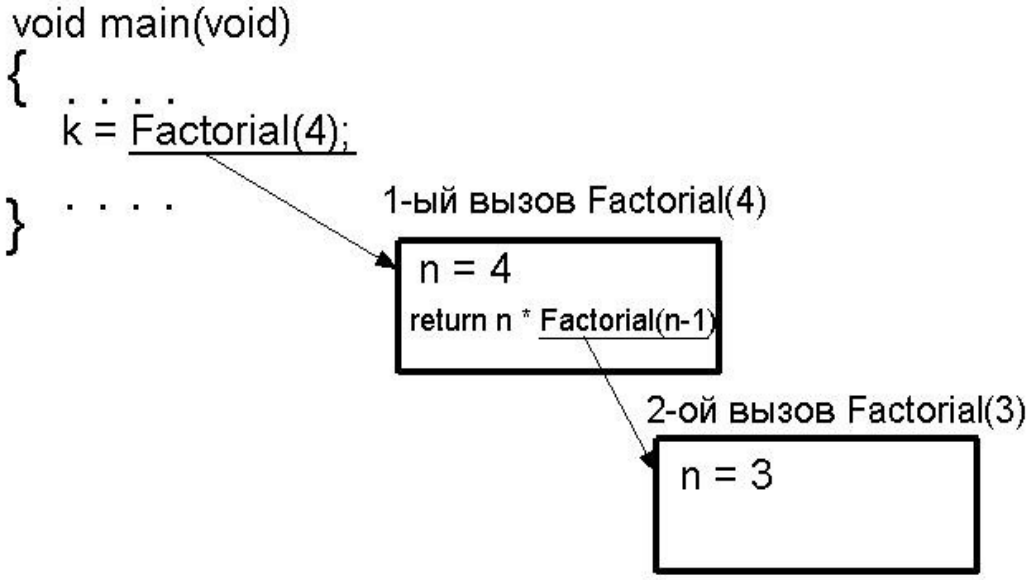


Рис. 2.2. Демонстрация первого рекурсивного вызова функции `Factorial(3)`.

Далее процесс повторяется, до тех пор, пока на четвертом вызове `Factorial(1)` не будет выполнено условие выхода из рекурсии (рис. 2.3). Поэтому после четвертого вызова происходит возврат в вызывающую функцию (т. е. в третий экземпляр `Factorial(2)`, рис. 2.4).

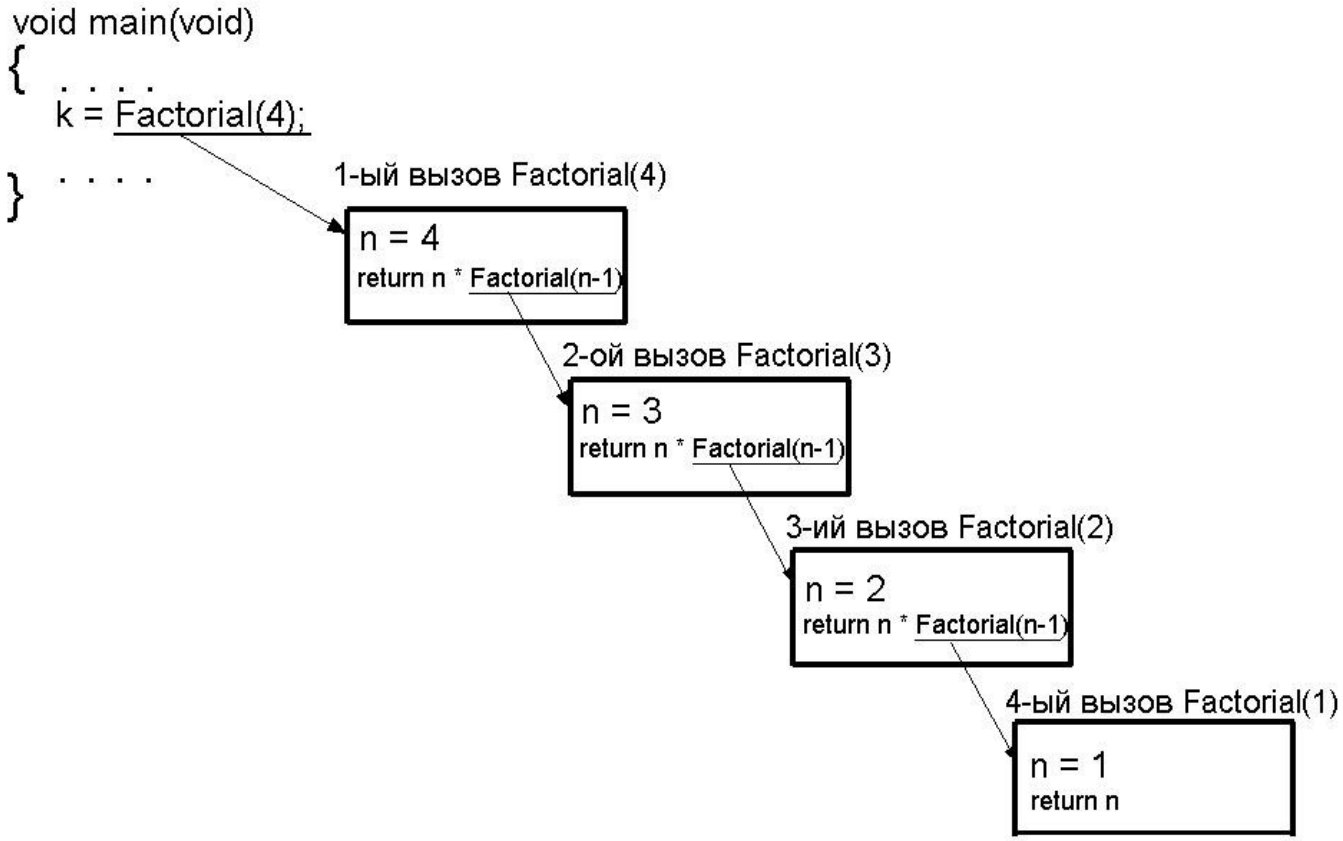


Рис. 2.3. Демонстрация рекурсивного вызова функции `Factorial(1)`.

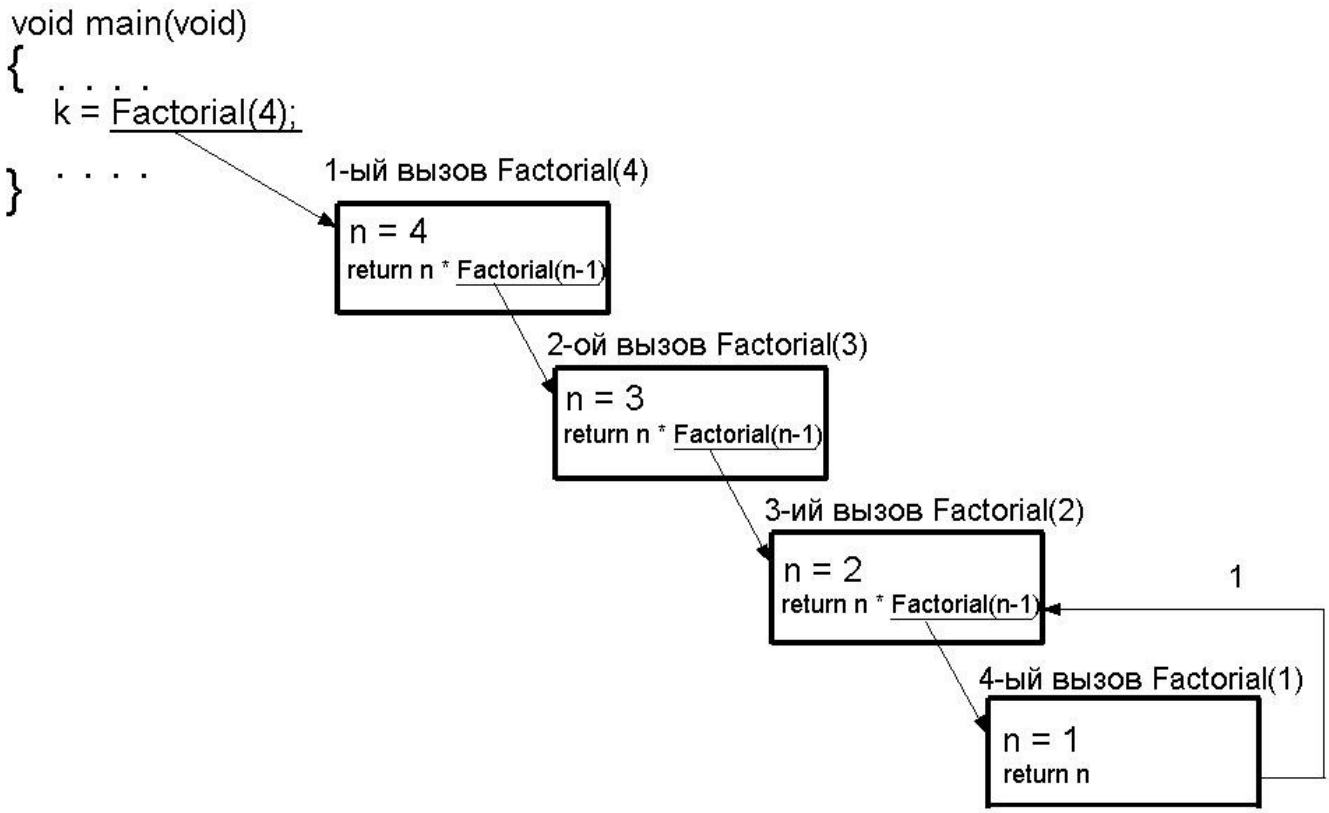


Рис. 2.4. Возврат из рекурсивного вызова функции `Factorial(1)`.

Третий экземпляр `Factorial(2)` завершается, но перед завершением возвращает значение 2*1 (*n**результат 4-го вызова) и т. д. Первый экземпляр функции перед завершением работы возвратит значение 4*результат 2-го вызова (рис. 2.5).

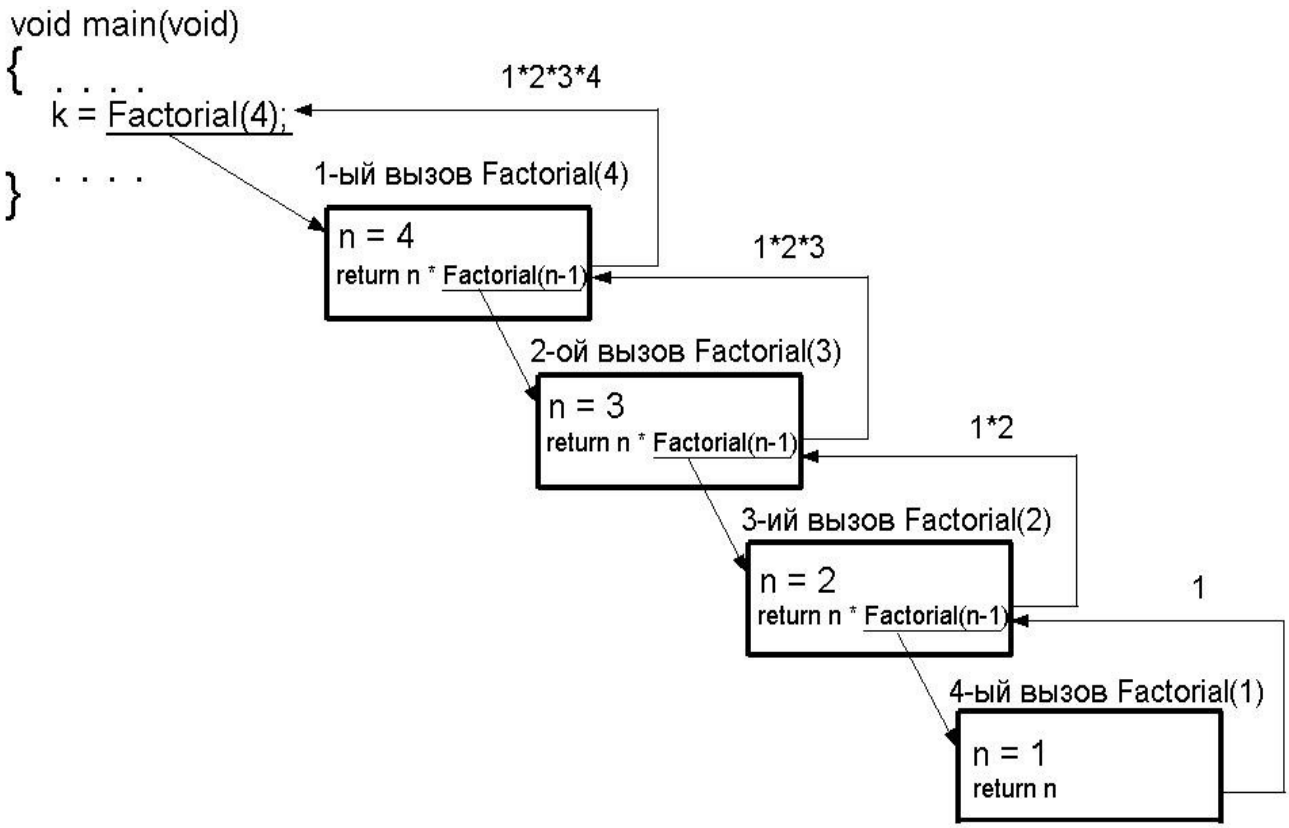


Рис. 2.5. Воврат из вызова функции `Factorial(4)`.

Задача 2. Вычислить *n*-ое число Фибоначчи. Известно, что F_1=1, F_2=1, $F_i=F_{i-1} + F_{i-2}$, т.е. каждый член последовательности чисел Фибоначчи, начиная с третьего, равен сумме двух предыдущих.

```
# include <stdio.h>

// прототип функции вычисления n-го числа Фибоначчи
int Fibonachi(int n);

void main(void)
{
      int n;
      printf("Введите n:");
      while(true)
      {
```

```
            scanf("%d",&n);
            if(n<=0)
                printf("Введите положительное число.\n");
            else break;
        }

        int k=Fibonachi(n);// вызов функции
        printf("F[%d]=%d\n",n,k);
}


// определение функции вычисления n-го числа Фибоначчи
int Fibonachi(int n)
{
        // числа Фибоначчи вычисляются только для n > 0
        if(n <= 0) return -1;

        // условие выхода из рекурсии
        if(n==2 || n==1)
            return 1;
        else
            // рекурсивный вызов функции
            return Fibonachi(n-2)+Fibonachi(n-1);
}
```

Это пример задачи, в которой рекурсивный алгоритм очевиден, но его использование приводит к большим вычислительным расходам и излишним затратам памяти. В теле функции `Fibonachi` несколько раз осуществляется рекурсивный вызов. Каждый вызов требует выделения памяти. Например, для вычисления F_{10} необходимо вызвать функцию `Fibonachi` для вычисления F_9 и F_8. В свою очередь, для вычисления F_9 снова понадобится вызвать функцию `Fibonachi` для вычисления F_8 и т.д. согласно приведенной ниже схеме:

$$F_{10}=F_9+F_8$$

$$F_9=F_8+F_7 \qquad\qquad F_8=F_7+F_6$$

$$F_8=F_7+F_6 \qquad F_7=F_6+F_5 \qquad F_7=F_6+F_5 \qquad F_6=F_5+F_4$$

$$\cdot \quad \cdot \quad \cdot \quad \cdot$$

Из схемы видно, что для некоторых аргументов приходится осуществлять повторные вызовы функции `Fibonachi`, т. е. происходят лишние вычисления.

Задача 3 . Задача о Ханойской башне.

Согласно легенде в одном из буддийских монастырей монахи уже тысячу

лет занимаются перекладыванием колец. Они располагают тремя стержнями, на которых надеты кольца разных размеров, образующих пирамиды. В начальном состоянии 64 кольца были надеты на первую пирамиду и упорядочены по размеру.

Монахи должны переложить все кольца с первой пирамиды на вторую, выполняя единственное условие – кольцо нельзя положить на кольцо меньшего размера. При перекладывании можно использовать все три пирамиды.

Монахи перекладывают одно кольцо за одну секунду. Как только они закончат свою работу, наступит конец света.

Количество перекладываний в зависимости от количества колец вычисляется по формуле $2^n - 1$. Для 64-х колец это 18 446 744 073 709 551 615 перекладываний, и, если принять скорость «одно перекладывание в секунду», получится около 584 942 417 355 лет, то есть апокалипсис наступит нескоро.

Сформулируем условие задачи. Дано три стержня *a*, *b*, *c*. На одном нанизана пирамида из *n* колец разного радиуса. Требуется перенести всю пирамиду на другой стержень, используя в качестве вспомогательного стержня третий.

Если требуется перенести пирамиду, состоящую из одного кольца, то печатается сообщение о переносе этого кольца на нужный стержень.

На рис. 2.6 показана ситуация переноса пирамиды из *n* колец ($n>1$) (рис. 2.6 **а** – исходное состояние). Для такого переноса требуется сначала перенести пирамиду размера n-1 на вспомогательный стержень (рис. 2.6 **б**). Далее перенести оставшееся кольцо на нужный стержень (рис. 2.6 **в**), и затем перенести пирамиду из n-1 кольца со вспомогательного стержня на нужный (рис. 2.6 **г**). Этот алгоритм реализаван в функции `HanoiTower`, которая имеет следующие параметры: первый параметр – номер стержня-источника, второй параметр – номер стержня назначения, третий параметр – вспомогательный стержень, четвертый параметр – количество колец в пирамиде.

```
# include <stdio.h>

// прототип функции решения задачи о Ханойской башне
void HanoiTower(int a, int b, int c, int n);

void main(void)
{
      int n;
      int a,b,c;
      printf("Введите количество колец:");
      scanf("%d",&n);
```

```
        printf("Введите номер стержня-источника (1,2 или 3):");
        scanf("%d",&a);
        printf("Введите номер стержня-приемника (1,2 или 3):");
        scanf("%d",&b);
        c=6-a-b;
        if(n<=0)
        {
            printf("Введите положительное количество
                      колец.\n");
            return;
        }
        if(a==b || a<=0 || a>3 || b<=0 || b>3)
        {
            printf("Неправильно введены номера стержней.\n");
            printf("Стержни имеют номера 1, 2 или 3, и номера
            стержней источника и приемника не совпадают.\n");
            return;
        }
        HanoiTower(a,b,c,n);// вызов функции
        printf("\n");
}
```

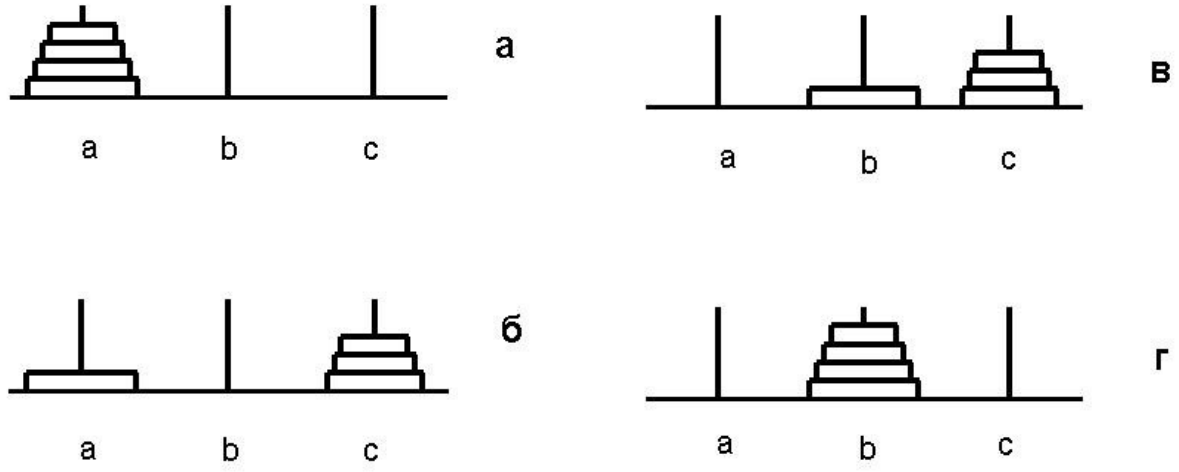


Рис. 2.6. Процесс переноса пирамиды со стержня a на стержень b.

```
// определение функции решения задачи о Ханойской башне
void HanoiTower(int a, int b, int c, int n)
{
        if(n==1) // перенос одного кольца
            printf("%d->%d\t",a,b);

        else
        {
            // рекурсивный вызов функции
```

```
            // перенос пирамиды из n-1 кольца
            // со стержня a на стержень c
            HanoiTower(a,c,b,n-1);
            // перенос одного кольца
            printf("%d->%d\t",a,b);
            // еще один рекурсивный вызов функции
            // перенос пирамиды из n-1 кольца
            // со стержня c на стержень b
            HanoiTower(c,b,a,n-1);
      }
}
```

Задача 4 . Лабиринт задан в виде прямоугольной матрицы, в которой используются следующие обозначения: 0 – возможен проход, 1 – стена. Даны координаты позиции путника в лабиринте. Требуется определить, сможет ли путник выйти из лабиринта. Считается, что путник вышел из лабиринта, если он находится на первой или последней строке матрицы лабиринта, или на первом или последнем столбце. Путник может передвигаться по лабиринту, перемещаясь только по горизонтали или по вертикали.

Алгоритм решения данной задачи связан с использованием рекурсивных функций. Находясь в некоторой ячейке, путник может двигаться в четырех направлениях, включая то, откуда путник попал в эту ячейку. Осуществляется перебор всех этих направлений. Если по направлению можно сдвинуться (соседняя ячейка по направлению содержит 0), то делаем шаг по этому направлению. Если ни по одному направлению сдвинуться нельзя (все соседние ячейки содержат 1, кроме той, откуда путник попал в текущую ячейку, или все пути из соседних ячеек привели в тупик), то осуществляется обратный ход в ту ячейку, откуда путник попал в текущую. Если в результате работы алгоритма, будет осуществлен возврат в начальную ячейку, делается вывод о том, что путник не может выйти из лабиринта.

```
# include <stdio.h>

// прототип функции поиска выхода из лабиринта
int Labirint(int** a, int m,int n,int x,int y,
               int xprev, int yprev);



void main(void)
{
      int m,n,x,y,i,j;
```

```
printf("Введите размеры матрицы лабиринта:");
while(true)
{
     scanf("%d",&m);
     scanf("%d",&n);
     if(n<=0 || m<=0)
          printf("Введите положительные
                    размеры\n");
          else break;
}
int** a=new int*[m];
if(a==NULL)
{
     printf("Нет достаточного объема памяти.\n");
     return;
}
for(i=0;i<m;i++)
{
     a[i]=new int[n];
     if(a[i]==NULL)
     {
          printf("Нет достаточного объема памяти.\n");
          for(j=0; j<i; j++)
               delete [] a[j];
          delete [] a;
          return;
     }
}

printf("Введите матрицу, задающую лабиринт.\n");
for(i=0;i<m;i++)
     for(j=0;j<n;j++)
     {
          scanf("%d",&a[i][j]);
          if(a[i][j]!=0 && a[i][j]!=1)
          {
               printf("Введите 0 или 1");
               j--;
          }
     }


printf("Введите начальные координаты путника:");
int f=1;
while(f==1)
{
```

```
            scanf("%d",&x);
            scanf("%d",&y);
            if(x>=1 && x<=m && y>=1 && y<=n)
                if(a[x-1][y-1]==0)
                    f=0;
                else printf("Путник замурован в стену.\n
                             Введите координаты внутри
                             коридора.\n");
            else
                printf("Введите координаты внутри
                        лабиринта.\n");
        }

        x--; y--;
        f=Labirint(a,m,n,x,y,x,y);// вызов функции
        if(f==1)
            printf("Выход есть!!!\n");
        else
            printf("Выхода нет!!!\n");
        for(i=0;i<m;i++)
            delete [] a[i];
        delete [] a;
    }


    // определение функции поиска выхода из лабиринта
    int Labirint(int** a, int m,int n,int x,int y,
                int xprev, int yprev)
    {
        // отмечаем пройденную ячейку
        a[x][y]=2;
        // печать координат пройденной ячейки
        printf("%d %d\n",x,y);

        // условие выхода из рекурсии -
        // выход из лабиринта найден
        if(x==0 || y==0 || x==m-1 || y==n-1)
            return 1;

        int f=0;// значение f равно 1, если выход найден, и
                // 0, в противном случае

        // попытка перейти в ячейку, расположенную внизу
        if(a[x+1][y]==0 && !(x+1==xprev && y==yprev))
            f=Labirint(a,m,n,x+1,y,x,y); // рекурсивный вызов
```

```
                                        // функции

        // попытка перейти в ячейку, расположенную справа
        if(f==0 && a[x][y+1]==0 && !(x==xprev && y+1==yprev))
            f=Labirint(a,m,n,x,y+1,x,y); // рекурсивный вызов
                                        // функции

        // попытка перейти в ячейку, расположенную сверху
        if(f==0 && a[x-1][y]==0 && !(x-1==xprev && y==yprev))
            f=Labirint(a,m,n,x-1,y,x,y); // рекурсивный вызов
                                        // функции

        // попытка перейти в ячейку, расположенную слева
        if(f==0 && a[x][y-1]==0 && !(x==xprev && y-1==yprev))
            f=Labirint(a,m,n,x,y-1,x,y); // рекурсивный вызов
                                        // функции
        // возврат в предыдущую ячейку
        a[x][y]=0;
        return f;
}
```

Домашнее задание

1. Написать рекурсивную функцию вычисления $n!! = n*(n-2)*(n-4)\ldots$. Последний сомножитель в произведении равен 2, если n – четное число, 1 – в противном случае.

2. Написать рекурсивную функцию вычисления числа комбинаций из n по k ($C(n, k)$) с помощью следующих рекуррентных соотношений:

$$C(N, 0) = C(N,N) = 1,$$
$$C(N,K) = C(N - 1,K) + C(N - 1, K - 1), \text{ где } 0 < K < N.$$

3. Написать рекурсивную функцию вычисления наибольшего общего делителя двух положительных целых чисел m и n ($m > n$).

$$\text{НОД}(m, n) = \text{НОД}(n, m\%n), \text{ если } n \neq 0; \text{ НОД}(m, 0) = n.$$

4. Написать рекурсивную функцию вычисления суммы цифр из записи целого положительного числа, не используя операторов цикла.

5. Написать рекурсивную функцию вычисления максимального элемента в массиве целых чисел, не используя операторов цикла.

6. Написать рекурсивную функцию проверки, образуют ли элементы массива симметричную последовательность, не используя операторов цикла.

7. Написать рекурсивную функцию нахождения индекса заданного элемента k в массиве целых чисел методом бинарного поиска, не используя операторов цикла.

8. Написать рекурсивную функцию, формирующую все перестановки n элементов массива.

9. Написать рекурсивную функцию поиска корня уравнения $f(x) = 0$:

а) методом деления отрезка пополам;

б) методом хорд;

в) методом касательных.

Описание методов приводится в разделе 1 (в задаче 6 и в задачах № 16, 17 из домашнего задания).

10. Написать рекурсивную функцию быстрой сортировки массива (алгоритм Хоара).

Алгоритм заключается в следующем. Находится такой элемент массива, который разбивает массив на два подмножества. Первое подмножество составляют элементы, которые по величине меньше разбивающего элемента, второе подмножество – элементы, которые по величине не меньше его. Далее применяется этот же алгоритм разбиения для каждого из полученных подмножеств.

Элемент, разделяющий массив, может быть определен следующим образом. Вводятся два индекса (i и j) для просмотра элементов слева и справа. Сравниваются i-ый и j-ый элементы и, если обмен не требуется, то $j=j$-1 и этот процесс повторяется. После первого обмена $i=i$+1. Сравнение продолжается с увеличением i до тех пор, пока не произойдет еще один обмен. Тогда опять уменьшается j и т. д., пока не станет $i=j$.

11. Написать рекурсивную функцию вычисления определителя n-ого порядка по правилу Лапласа с разложением по произвольной строке.

12. Написать рекурсивную функцию обхода всей шахматной доски разме-

ром 8×8 конем из заданной позиции. Если такого обхода не существует, выдать сообщение.

13. Дан набор предметов, имеющих определенный вес и ценность. Требуется поместить часть этого набора в рюкзак заданной грузоподъемности так, чтобы ценность полученного набора была максимальной (задача о рюкзаке). Написать рекурсивную функцию решения этой задачи.

14. Пусть имеется n городов. Дана матрица стоимостей проезда между городами (a_{ij} – стоимость проезда из города i в город j, если пути из i-ого города в j-ый не существует, a_{ij} = -1). Требуется найти замкнутый маршрут, т. е. такой, который начинается и заканчивается в одном и том же городе, например, в городе № 1. При этом каждый город должен быть посещен ровно один раз и стоимость проезда по маршруту должна быть минимальной (задача коммивояжера). Написать рекурсивную функцию решения этой задачи.

Глава 2. Линейные динамические структуры данных

Раздел 3. Односвязные списки

Связные списки – это способ хранения данных, который применяется в тех случаях, когда заранее неизвестно количество элементов или когда размер хранимых данных постоянно меняется. Например, списки удобно использовать в приложении, имитирующем работу телефонного автоответчика. Сообщения, которые поступают на автоответчик, должны сохраняться в некоторой структуре данных. Прослушанные сообщения требуется из этой структуры удалять.

Если использовать для этих целей массив, то нужно сразу указывать его размер достаточным для хранения максимального количества возможных элементов, что не всегда получается оценить. Если выделенной памяти не хватает для хранения всех элементов, приходится производить реорганизацию массива – выделение новой памяти необходимого размера, перенос всех элементов из прежней памяти в новую и освобождение той области памяти, которую ранее занимал массив.

Главное достоинство динамических структур данных, самым простым примером которых является связный список, состоит в том, что только в момент добавления нового элемента под него выделяется память. Соответственно, в момент удаления элемента память, занятая им, освобождается. Часто элемент списка называют также узлом.

Однако, по сравнению с массивами, здесь имеется и недостаток - заранее неизвестно местоположение каждого элемента. Поэтому приходится вместе с самими данными хранить дополнительную информацию об их местоположении. Таким образом, для хранения динамических структур данных требуется больше памяти.

Поскольку связный список должен хранить вместе данные и информацию об их местоположении, при применении списков в программе используют структурный тип данных, в который входят переменные, хранящие информационные поля, и переменная, содержащая адрес памяти следующего (или предыдущего) узла списка. Для хранения адресов памяти в большинстве языков программирования существуют специальные типы данных. Например, в языке C++ для этих целей применяется тип данных, который называется указатель.

Односвязным называют список, каждый узел которого содержит адрес

следующего элемента списка. Предположим, что для реализации автоответчика необходимо хранить следующие данные: номер телефона звонившего, его имя и само сообщение. Схематично вид этого списка будет выглядеть так, как это представлено на рис. 3.1. В поле, из которого исходит стрелка, хранится адрес следующего элемента. Стрелка как раз и указывает на этот элемент.

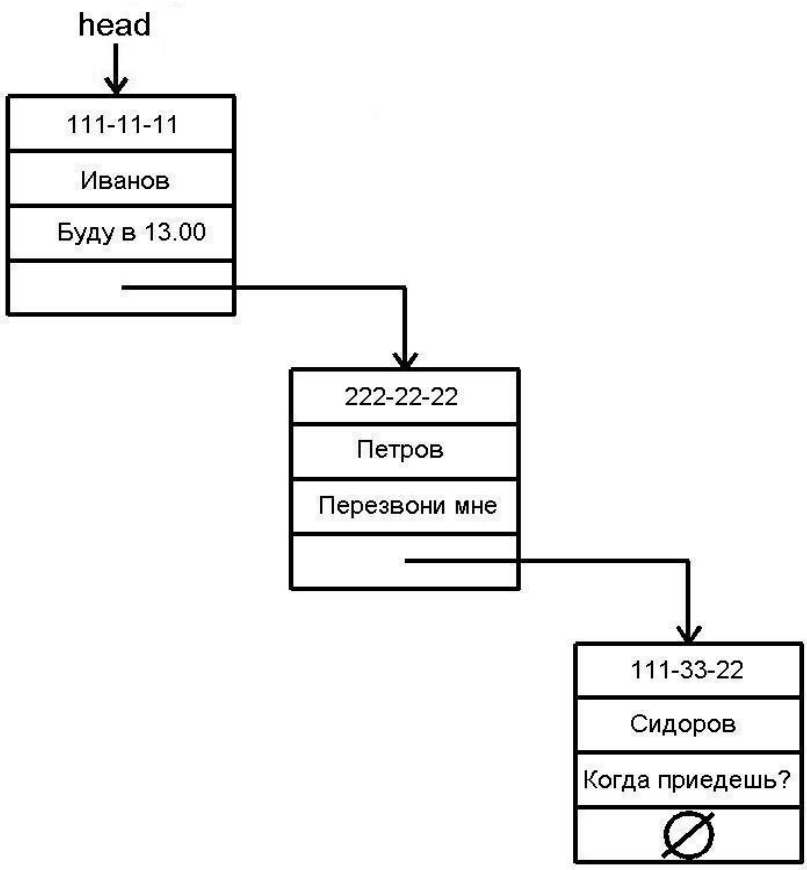


Рис 3.1. Вид списка.

Структуру для хранения элемента такого списка можно определить так:

```
struct PhoneMessage
{
      char phone[12];         // номер телефона
      char name[30];          // имя
      char message[300];      // сообщение
      PhoneMessage * next;    // адрес следующего сообщения
};
```

Чтобы правильно работать с односвязным списком, необходимо отдельно хранить адрес его первого элемента (переменная `head`). Тогда, двигаясь по цепочке, можно получить доступ к любому узлу. Признаком конца списка яв-

ляется пустая (нулевая) ссылка в поле адреса следующего элемента. Например, в языке C++ для обозначения пустой ссылки используется константа `NULL`, в C# и Java – `null`, в Pascal – `nil`.

Далее приводится описание процесса решения некоторых задач с применением пользовательских функций.

Задача 1. Пусть имеется непустой односвязный список, который содержит целые числа. Написать функцию нахождения суммы элементов списка.

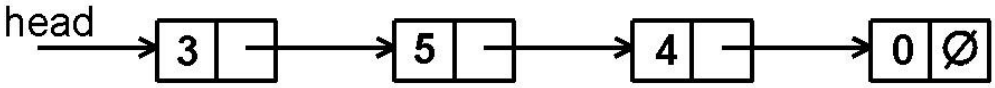


Рис.3.2. Исходный список.

```
// структура для хранения узла списка
struct List
{
      int info;     // информационное поле
      List* next;   // указатель на следующий узел списка
};
```

Чтобы найти сумму элементов списка, нужно пройти по всем его узлам, добавляя значение информационного поля (переменная `info`) к переменной, хранящей сумму. Для этого заводим переменную `current`, которая сначала хранит адрес первого элемента списка. В цикле последовательно по цепочке сдвигаем ее значение на следующий элемент (`current=current->next`). Условие окончания цикла – значение `NULL` в переменной `current`.

```
// определение функции подсчета суммы элементов списка
int SummaList(List*head)
{
      int s=0;
      // просмотр элементов списка начинается
      // с его первого элемента
      List* current=head;

      while(current!=NULL)// пока список не закончен
      {
           s+=current->info;// добавление значения текущего
```

```
                                // элемента к сумме

            current=current->next;// переход на следующий
                                  // элемент списка
        }
        return s;
    }
```

Задача 2. Пусть односвязный список содержит целые числа. Написать функцию добавления нового элемента в список в заданную позицию. Эту же функцию можно использовать для создания списка.

При решении задачи можно выделить три частных случая добавления элемента: вставка первого узла, вставка в середину и в конец списка. Если происходит вставка первого узла, требуется изменить значение переменной `head`. При вставке элемента в другую позицию требуется переместиться по списку на элемент, предшествующий месту вставки. Например, если требуется вставить элемент на пятую позицию, нужно переместиться на четвертый узел списка. Затем перенаправить адреса согласно рис. 3.4.

Пусть происходит вставка первого элемента. Сначала выделяем память под хранение первого узла и его адрес записываем в переменную `help` (рис. 3.3 **а**). Затем заносим в информационное поле нового узла значение `key`. Теперь требуется связать новый узел со всем списком. Для этого в поле `next` нового узла `help` заносим адрес прежнего первого элемента списка (`head`) (рис. 3.3 **б**). Теперь осталось только изменить начало списка, т.е. `head=help` (рис. 3.3 **в**). Заметим, что вставка пройдет корректно, даже если список был пустым. В этом случае перед вставкой `head=NULL`. После вставки `head` будет ссылаться на новый элемент, а поле `next` примет значение `NULL`.

При вставке элемента в середину (в конец) списка надо переместиться на элемент, предшествующий месту вставки. Для этого заводим переменную `current`, которая сначала хранит адрес первого элемента списка. В цикле последовательно по цепочке сдвигаем ее значение на следующий элемент (`current=current->next`). Это нужно делать до тех пор, пока не получим адрес элемента в нужной позиции (`i=pos-1`) или пока не закончится список (в этом случае `current=NULL`). Если список закончится, то вставка не производится. Если же остановка цикла произошла, когда `current` хранит адрес существующего элемента списка, выполняем следующие действия. Сначала выделяем память под хранение нового узла и его адрес записываем в переменную `help` (рис. 3.4 **а**). Затем заносим в информационное поле нового узла значение `key`. Теперь «встраиваем» новый элемент в середину списка.

Для этого сначала указываем, что следующим за новым элементом должен быть элемент `current->next` (`help->next=current->next`) (рис. 3.4 **б**) и делаем ссылку из `current` на новый элемент (`current->next=help`) (рис. 3.4 **в**).

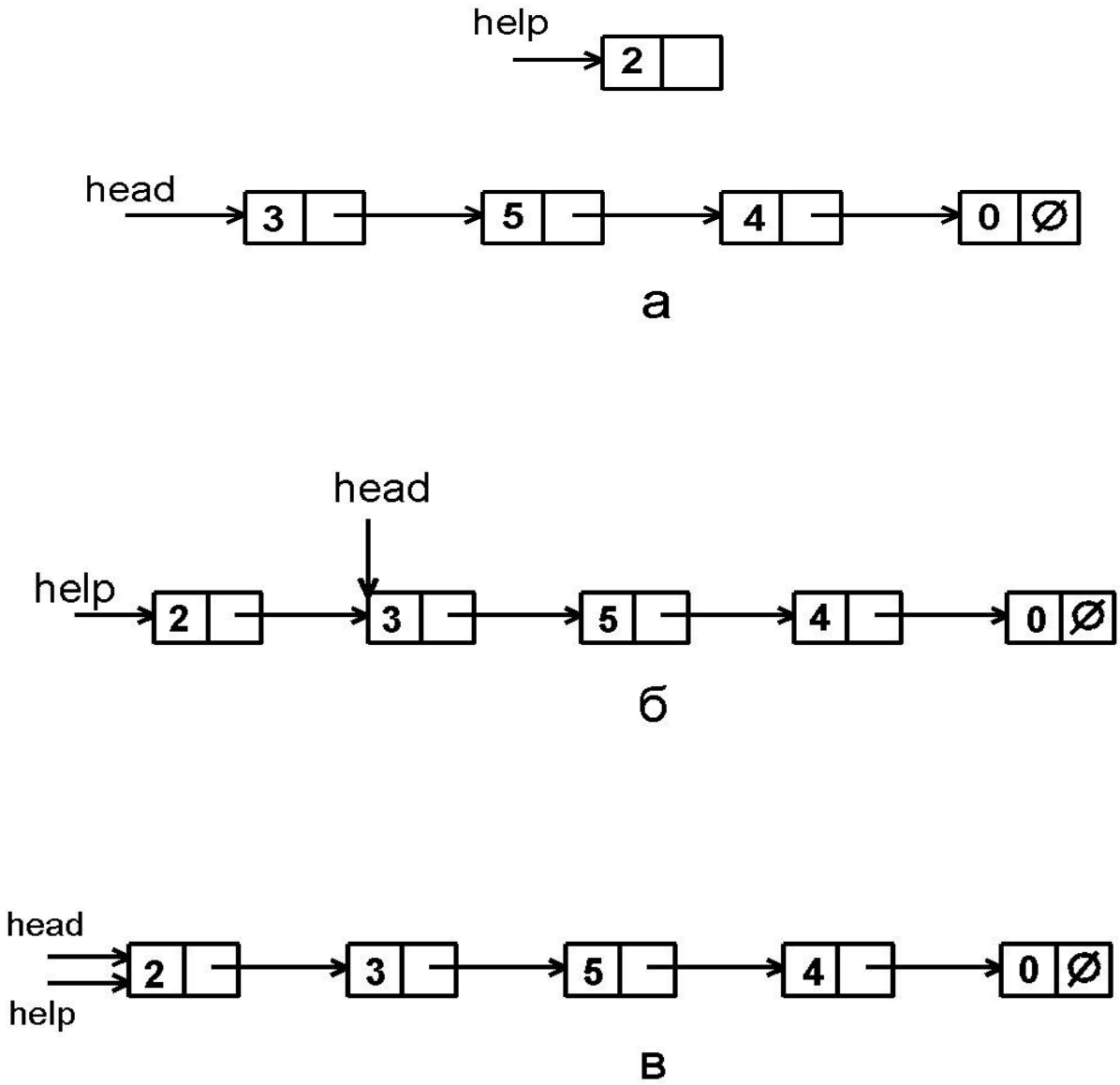


Рис. 3.3. Вставка элемента в начало списка.

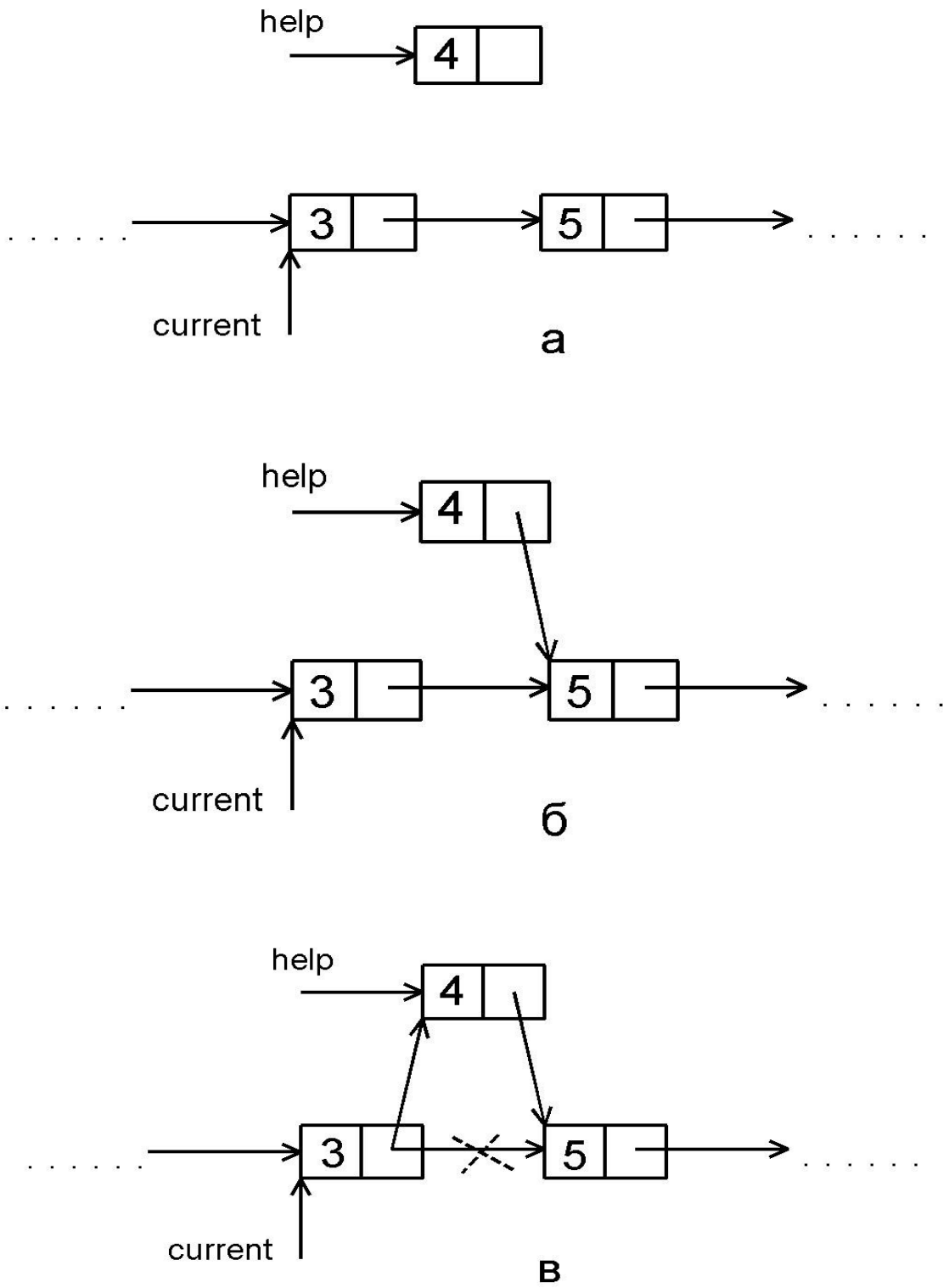


Рис 3.4. Вставка в середину списка.

Заметим, что это нужно сделать именно в этом порядке. Если поменять эти действия местами, будет потерян «хвост» списка (рис. 3.5).

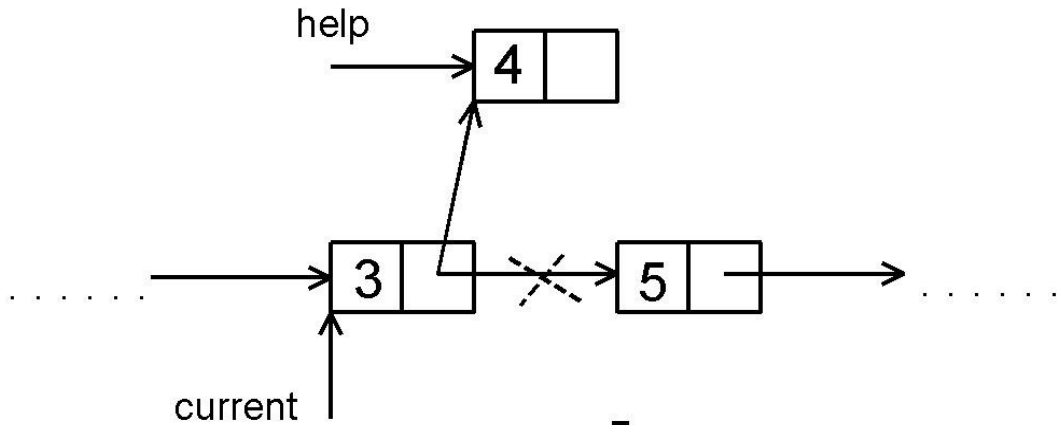


Рис 3.5. Случай потери «хвоста» списка.

```
// определение функции добавления в список
// элемента key на позицию pos
void AddList(List*& head, int key, int pos)
{
      List* current,*help;
      // добавление в начало списка
      if(pos==1)
      {
           help=new List;
           help->info=key;
           help->next=head;
           head=help;
           return;
      }

      // поиск элемента, после которого требуется
      // вставить новый элемент
      current=head;
      int i=1;
      while(i!=pos-1 && current!=NULL)
      {
           i++;
           current=current->next;
      }

      // позиция не найдена, список оказался коротким
      if(current==NULL)
      {
           printf("Некорректная позиция\n");
           return;
      }
```

```
        // вставка элемента в позицию pos
        help=new List;
        help->info=key;
        help->next=current->next;
        current->next=help;
}
```

Задача 3. Пусть односвязный список содержит целые числа. Написать функцию удаления всех элементов списка с заданным значением.

Решение задачи можно разделить на две части. Сначала удаляем все элементы с заданным значением (оно хранится в переменной `key`), находящиеся в начале списка. Это нужно для того, чтобы при удалении первого элемента изменялся заголовок списка (переменная `head`). Удаление первого элемента по шагам показано на рис. 3.6.

Если новый первый элемент будет хранить заданное значение, то процедура удаления повторяется с новым значением переменной `head`.

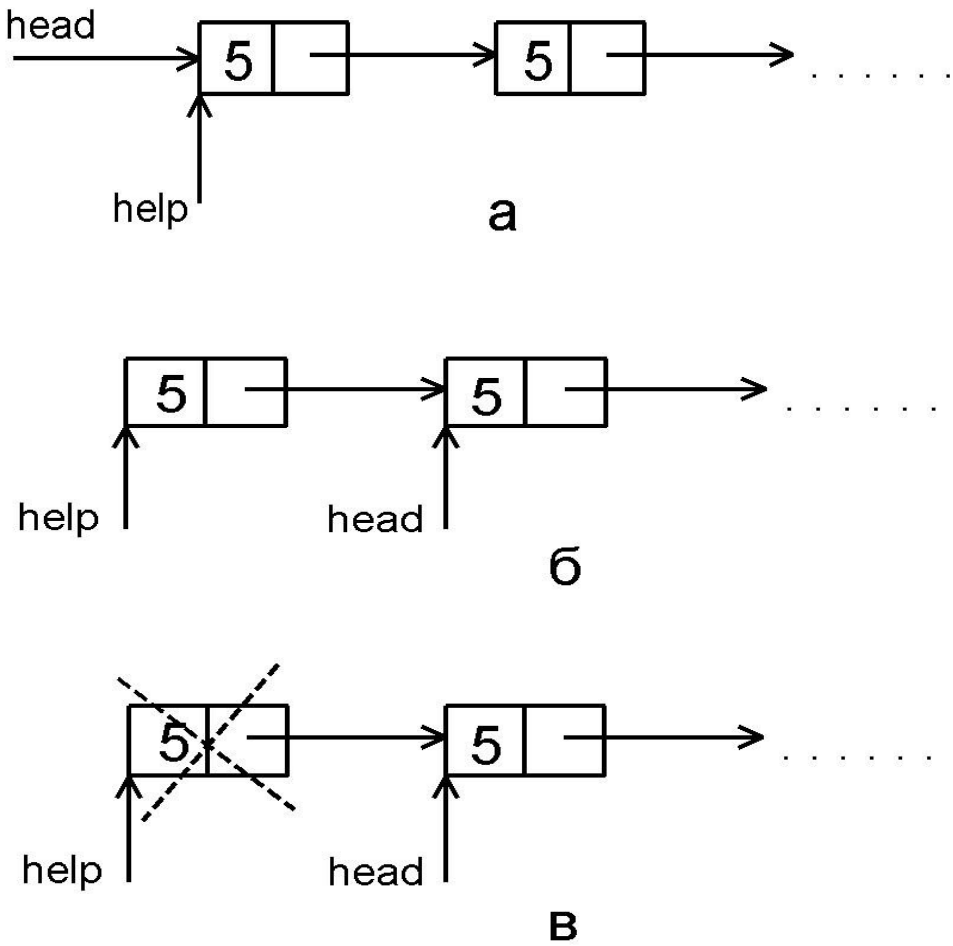


Рис. 3.6. Удаление первого элемента из начала списка.

Если после первого цикла список не окажется пустым, удаляем элементы с заданным значением из середины списка. Переменную `current` устанавливаем на начало списка. Удаляться будет следующий за `current` элемент, адрес которого будет храниться в переменной `help` (рис. 3.7 **а**). Если нужный элемент найден, то после удаления следующим за `current` должен быть элемент, идущий за `help`, иначе будет потерян весь остаток списка (`current->next=help->next`) (рис. 3.7 **б**). Теперь элемент `help` можно удалить (`delete help`) (рис. 3.7 **в**). В этом случае значение `current` не меняется, поскольку после удаления за ним может идти элемент, который, возможно, имеет заданное значение. Если же в `help` хранится значение, отличное от значения переменной `key`, сдвигаем `current` на следующий узел.

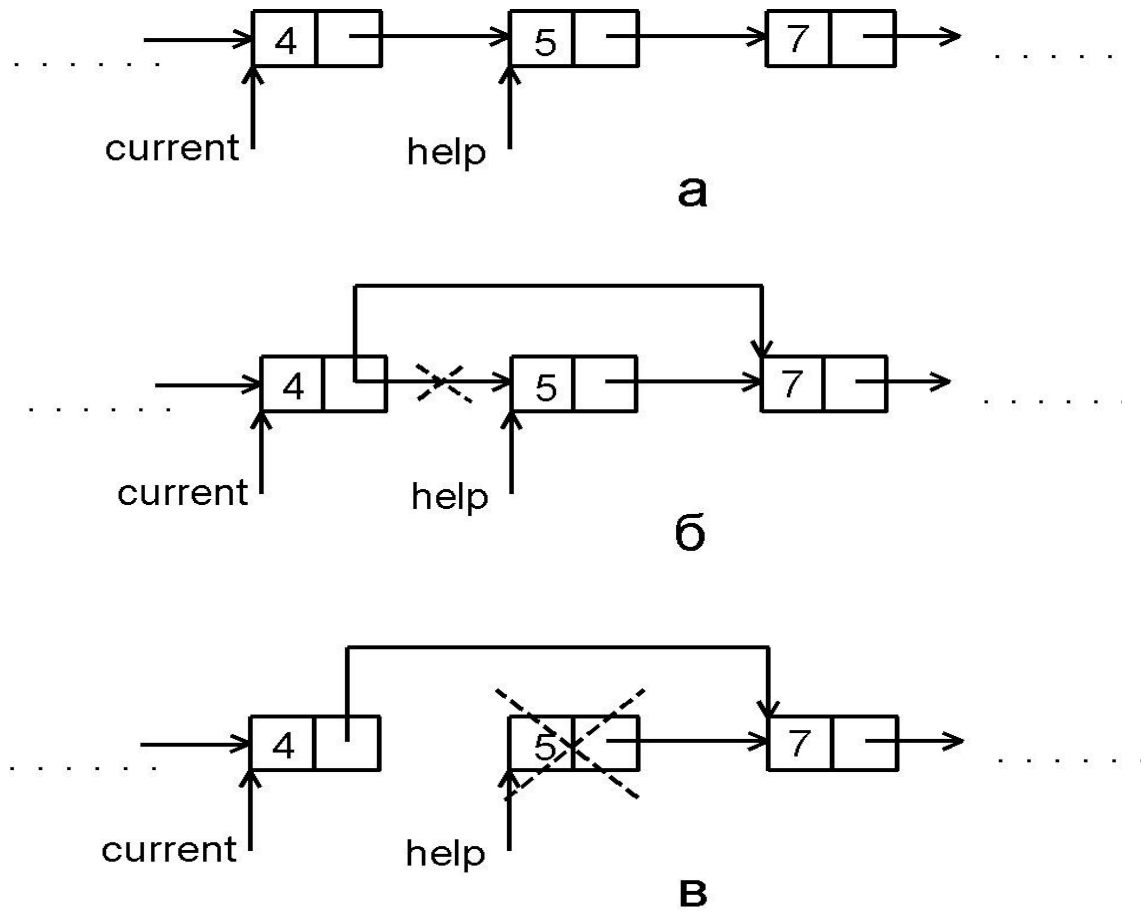


Рис 3.7 Удаление из середины списка.

```
// определение функции удаления элементов списка по ключу
void DeleteKey(List*& head, int key)
{
      List* current,*help;
```

```
        // удаление всех элементов с заданным значением,
        // расположенных в начале списка
        while(head!=NULL && head->info==key)
        {
            help=head;
            head=head->next;
            delete help;
        }

        if(head!=NULL)
        {
            current=head;
            // поиск и удаление элементов с заданным значени-
            // ем, расположенных в середине или в конце списка
            while(current->next!=NULL)
            {
                help=current->next;
                if(help->info==key)
                {
                    current->next=help->next;
                    delete help;
                }
                else
                    current=current->next;
            }
        }
    }
```

Задача 4. Дан односвязный список. Написать функцию удаления всех элементов списка.

В переменной `head` будет храниться адрес текущего элемента. Чтобы не потерять «хвост» списка, надо сохранять адрес следующего элемента в переменной `help`. Удалив текущий элемент, в переменную `head` записываем значение из `help`. Эти действия осуществляются до тех пор, пока список не станет пустым.

```
    // определение функции уничтожения списка
    void DestroyList(List* head)
    {
        List*help;
        while(head!=NULL)
        {
            help=head->next;
            delete head;
```

```
            head=help;
        }
    }
```

Задача 5. Дан односвязный список, содержащий целые числа. Написать функцию разбиения списка на два таких, что в одном будут содержаться только четные числа, в другом - нечетные.

Функция имеет три параметра: `head` – содержит начало исходного списка, `head1` – списка с нечетными элементами, `head2` – с четными. Переменные `current1`, `current2` используются для хранения текущих элементов создаваемых списков. Начальные значения `head1`, `head2`, `current1`, `current2` устанавливаются в `NULL`, поскольку результирующие списки еще не созданы.

Двигаясь по элементам исходного списка, проверяем, к какой группе относится очередной элемент. Если значение информационного поля четное, добавляем элемент в список с заголовком `head2`. Для этого, если в списке еще нет элементов, то значения переменных `head2` и `current2` становятся равными адресу этого элемента. Если же в списке уже имеются элементы, то он добавляется в конец:

```
current2->next=head;
current2=current2->next;
```

Аналогично поступаем и в том случае, когда встречается нечетный элемент, только используем переменные `head1` и `current1`.

```
// определение функции разбиения списка на два -
// в одном четные элементы, в другом - нечетные
void Decompose(List*& head, List*& head1, List*& head2)
{
      List* current1=NULL,*current2=NULL;
      head1=NULL; head2=NULL;
      // просмотр элементов исходного списка
      while(head!=NULL)
      {
            // если значение текущего элемента четное
            if(head->info%2==0)
            {
                  // добавляем элемент в список
                  // с четными значениями элементов
                  if(head2==NULL)
                        // список еще пуст
```

```
                    // формируем заголовок списка
                    head2=current2=head;
               else
               {
                    // список не пуст,
                    // добавляем в конец новый элемент
                    current2->next=head;
                    current2=current2->next;
               }
          }
          else
          {
               // добавляем элемент в список
               // с нечетными значениями элементов
               if(head1==NULL)
                    // список еще пуст
                    // формируем заголовок списка
                    head1=current1=head;
               else
               {
                    // список не пуст,
                    // добавляем в конец новый элемент
                    current1->next=head;
                    current1=current1->next;
               }
          }
          // переходим на следующий элемент
          // в исходном списке
          head=head->next;
     }
     // установка признаков конца построенных списков
     if(current1!=NULL)
          current1->next=NULL;
     if(current2!=NULL)
          current2->next=NULL;
}
```

Часто бывает удобно хранить полином (многочлен) в виде списка. Каждое слагаемое полинома задается одним элементом списка. В нем хранится значение коэффициента и показатель степени. Структура для хранения отдельного слагаемого имеет следующий вид:

```
struct Summand
{
      float a;      // коэффициент
      int n;        // степень
      Summand* next;    // указатель на следующее слагаемое
};
```

Далее приводится функция создания односвязного списка, хранящего полином, посредством ввода с клавиатуры. Оставляем ее для самостоятельного разбора.

```
// определение функции создания полинома
void CreatePolynom(Summand*& head)
{
      float koef;
      int pow;
      int m=0; // количества слагаемых в полиноме
      Summand* current,*help;
      printf("Введите полином:\n");
      printf("Введите коэффициент и степень:\n");
      // завершение цикла произойдет по нажатию клавиш Ctrl+Z
      while(scanf("%f%d",&koef,&pow)>0)
      {
           if(m==0)
           {
                // вводится первое слагаемое полинома
                head=new Summand;
                head->a=koef;
                head->n=pow;
                head->next=NULL;
                current=head;
           }
           else
           {
                // вводится очередное слагаемое полинома
                current->next=new Summand;
                current=current->next;
                current->a=koef;
                current->n=pow;
                current->next=NULL;
           }
           m++;
           printf("Введите коэффициент и степень:\n");
      }
```

```
}
```

Задача 6. Имеется односвязный список, определяющий полином. Требуется вычислить значение полинома в заданной точке.

Используя прием прохода по всем элементам списка, рассмотренный ранее (например, в задаче 1), вычисляем значение каждого слагаемого полинома и добавляем его к переменной, хранящей сумму.

```
// определение функции вычисления значения полинома
// в точке x
float CalculatePolynom(Summand* head,float x)
{
      float p_x=0;
      if(head!=NULL)
      {
           Summand* current=head;
           while(current!=NULL)
           {
                p_x+= current->a*pow(x, current->n);
                current=current->next;
           }
      }
     return p_x;
}
```

Задача 7. Имеется односвязный список, задающий полином. Требуется привести подобные слагаемые (предполагается, что полином может состоять из нескольких слагаемых с одинаковыми показателями степени).

Алгоритм решения данной задачи заключается в следующем. Берется очередной элемент списка (переменная `current`), и просматриваются все элементы, следующие за ним (переменная `help`). Если показатели степеней у двух рассматриваемых элементов совпадают, то коэффициенты складываются и помещаются в поле `info` узла, на который указывает `current`, а элемент `help` из списка удаляется.

```
// определение функции приведения подобных слагаемых
// в полиноме
void SimilarSummand(Summand*& head)
{
      Summand* current,*help,*prev;
      current=head;
      while(current!=NULL)
      {
           // поиск всех подобных слагаемых
```

```
            // для слагаемого, хранящегося в current,
            // начиная со следующего за ним;
            // адрес предыдущего слагаемого необходим
            // для последующего удаления
            prev=current;
            help=current->next;
            while(help!=NULL)
            {
                if(help->n==current->n)
                {
                    // нашли подобное слагаемое
                    // приводим его и удаляем из списка
                    current->a+=help->a;
                    prev->next=help->next;
                    delete help;
                    help=prev->next;
                }
                else
                {
                    prev=help;
                    help=help->next;
                }
            }
            current=current->next;
        }
    }
```

Домашнее задание

1. Дан односвязный список. Написать функцию печати его элементов.

2. Дан односвязный список. Написать функцию удаления элемента списка, расположенного в заданной позиции.

3. Дан односвязный список. Написать функцию создания копии имеющегося односвязного списка.

4. Дан односвязный список. Написать функцию, которая:

   1) формирует новый список, расположив элементы исходного списка в обратном порядке;

   2) располагает элементы в обратном порядке в исходном списке.

5. Дан односвязный список. Написать функцию добавления в начало списка нового элемента, значение которого равно количеству вхождений минимального элемента в список.

6. Дан односвязный список. Написать функцию добавления двух новых элементов – с минимальным значением в начало списка и с максимальным значением в конец списка.

7. Дан односвязный список. Написать функцию удаления из него всех элементов с минимальным значением.

8. Написать функцию сортировки элементов односвязного списка методом вставок.

9. Написать функцию слияния двух отсортированных односвязныйх списков.

10. Даны два односвязных списка. Написать функцию, проверяющую, содержится ли первый список во втором.

11. Даны два односвязных списка. Написать функцию, формирующую новый список, состоящий из элементов, которые содержатся в обоих исходных списках.

12. Даны два односвязных списка. Написать функцию формирования нового списка из тех элементов первого списка, которые не содержатся во втором.

13. Дан односвязный список. Написать функцию нахождения суммы элементов, расположенных между минимальным и максимальным элементами списка.

14. Даны два односвязных списка. Написать функцию формирования нового списка, состоящего из тех элементов первого списка, которые входят во второй список более одного раза.

15. Написать функцию печати формулы полинома.

16. Написать функции, реализующие операции с полиномами: сложение,

вычитание, умножение на число, умножение двух полиномов, возведение полинома в целую степень.

17. Написать приложение для работы с односвязным списком, узлы которого содержат информацию о полученных сообщениях (автоответчик, SMS и пр). Требуется реализовать возможности добавления нового сообщения, просмотра всех сообщений, поиска сообщений по заданному отправителю, поиска всех сообщений по заданной дате, удаления выбранного сообщения, удаления всех сообщений, сортировки сообщений списка по дате или по отправителю.

Раздел 4. Стеки и очереди

Стек и очередь – это такие динамические структуры данных (подобно спискам), в которых добавление и извлечения элементов происходит по четко определенным правилам. Стек работает по правилу LIFO (Last In First Out), т.е. «последним вошел, первым вышел». Очередь работает по правилу FIFO (First In First Out), т.е. «первым вошел, первым вышел». Для реализации стеков и очередей можно использовать массивы или динамические списки. Добавление нового элемента в стек осуществляется в конец массива или списка, а извлечение элемента – из конца массива или списка. Добавление элемента в очередь осуществляется в конец массива или списка, а извлечение элемента – из начала массива или стека.

Далее приводится описание процесса решения некоторых задач с применением пользовательских функций.

Задача 1. Написать функции добавления и извлечения элемента из стека. Стек реализовать в виде односвязного списка.

По аналогии с реализацией списка требуется отдельная структура для хранения одного элемента. Эта структура будет иметь следующий вид:

```
struct Node
{
      int info;     // информационное поле
      Node* next;   // указатель на следующий элемент
};
```

Для хранения стека вводится другая структура, в которой хранится адрес начала списка.

```
// структура стека
struct Stack
{
      Node* head; // указатель на вершину стека
                  //(начало списка)
};
```

Указатель на последний элемент, вошедший в стек, называется его вершиной. Заголовок (начало) списка в приведенном примере указывает на вершину стека. Поэтому добавление и извлечение элементов стека будет осуществляться в вершину стека (в начало списка).

Для добавления создается новый элемент списка (переменная p). Новый элемент добавляется в начало списка. Для этого он связывается с прежним первым элементом (p->next=s.head), и заголовок списка изменяется, указывая на новый элемент (s.head=p).

```
// определение функции добавления в стек нового элемента
void PushStack(Stack& s, int key)
{
      Node* p=new Node;
      p->next=s.head;
      p->info=key;
      s.head=p;
}
```

При извлечении элемента из стека возможны два случая: стек пуст или стек не пуст. Поэтому кроме извлекаемого значения приходится формировать еще и код возврата (в нашем примере – это возвращаемое значение функции). Код возврата будет равен -1, если стек был пуст, в противном случае он будет равен 1. Извлекаемое из стека значение будет возвращаться через параметр функции n. Вершиной стека станет следующий элемент списка, а бывшая вершина стека будет удалена.

```
// определение функции извлечения элемента из стека
int PopStack(Stack& s, int& n)
{
      // проверка стека на пустоту
      if(s.head==NULL)  return -1;
      n=s.head->info;
      Node* p=s.head;
      s.head=s.head->next;
      delete p;
      return 1;
}
```

Задача 2. Написать функции добавления и извлечения элемента из очереди. Очередь реализовать в виде односвязного списка.

Структура элемента очереди будет той же, что и в задаче 1.

```
struct Node
{
      int info;
      Node* next;
};
```

Для хранения очереди создается структура, хранящая адреса начала и конца списка. Это необходимо, так как добавление и извлечение происходит с разных сторон очереди.

```
// структура очереди
struct Queue
{
      Node* head;   // указатель на начало очереди
      Node* tail;   // указатель на конец очереди
};
```

При добавлении элемента в очередь приходится отдельно рассматривать ситуацию, когда очередь пуста. В этом случае нужно формировать ее начало и конец – они будут указывать на единственный элемент очереди.

Если же в очереди имеются элементы, то новый добавляется в конец (`q.tail->next=new Node`), и конец очереди сдвигается.

```
// определение функции добавления в очередь нового элемента
void PushQueue(Queue& q, int key)
{
      if(q.head==NULL)
      {
           q.head=q.tail=new Node;
           q.head->info=key;
           q.head->next=NULL;
           return;
      }
      q.tail->next=new Node;
      q.tail=q.tail->next;
      q.tail->info=key;
      q.tail->next=NULL;
}
```

Если необходимо извлечь элемент из очереди, надо формировать код возврата, который будет равен -1, если очередь пуста, и 1 – в противном случае. Извлекаемое из очереди значение будет возвращаться через параметр функции `n`.

Из очереди всегда извлекается первый элемент. Поэтому его значение сохраняется в параметр `n`, а затем он удаляется. Началом очереди становится следующий элемент списка. Если в очереди был один элемент (после удаления `q.head=NULL`), то одновременно обнуляется и `q.tail`.

```
// определение функции извлечения элемента из очереди
int PopQueue(Queue& q, int& n)
{
      if(q.head==NULL)
           return -1;
      n=q.head->info;
      Node* p=q.head;
      q.head=q.head->next;
      delete p;
      if(q.head==NULL)
           q.tail=NULL;
      return 1;
}
```

Очереди и стеки имеют широкое применение в различных задачах разбора математических выражений, обхода деревьев и графов.

Задача 3. Дан односвязный список. Написать функцию, определяющую, образуют ли его элементы симметричную последовательность.

Функция будет возвращать значение `true`, если список является симметричным, `false` – в противном случае.

По определению пустой список является симметричным. Поэтому, если список пуст, то возвращаем значение `true`.

Для проверки симметричности списка нужно проверить на равенство все пары элементов, равноотстоящих от середины списка. Каждая пара содержит один элемент из первой половины списка и один – из второй. Элементы первой половины списка последовательно заносятся в очередь, второй половины – в стек. Если количество элементов списка будет нечетным, то серединный элемент будет симметричен сам себе и не будет помещен ни в очередь, ни в стек.

Элементы, находящиеся в начале очереди и в вершине стека, образуют пару расположенных симметрично относительно середины списка элементов. Поэтому, извлекая элементы из стека и очереди и проверяя их на равенство, можно определить, симметричен ли заданный список.

```
// определение функции проверки симметричности
// последовательности чисел в односвязном списке
bool Symmetry(List* head)
{
      int n=0,a,b;
      if(head==NULL)
      {
           printf("Список пуст!");
```

```
        return true;
    }
    List* current;
    // подсчитываем количество элементов в списке
    for(current=head;current!=NULL; current=current->next)
        n++;

    // создание вспомогательных стека и очереди
    Stack s;
    s.head=NULL;
    Queue q;
    q.head=q.tail=NULL;

    current=head;
    // запись первой половины элементов списка в очередь
    for(int i=0;i<n/2;i++)
    {
        PushQueue(q,current->info);
        current=current->next;
    }
    // при нечетном количестве элементов в списке
    // пропускаем элемент, расположенный в середине
    if(n%2==1)
        current=current->next;

    // запись второй половины элементов списка в стек
    for(i=0;i<n/2;i++)
    {
        PushStack(s,current->info);
        current=current->next;
    }

    bool f=true;
    // одновременное извлечение значений элементов
    // из очереди и стека, и их сравнение
    for(i=0;i<n/2;i++)
    {
        PopQueue(q,a);
        PopStack(s,b);
        if(a!=b)
            f=false;
    }
    return f;
}
```

Задача 4. Дано положительное целое число в десятичной системе счисления. Перевести его в заданную систему счисления.

При переводе числа в заданную систему счисления производится последовательное получение остатков от деления числа на основание системы счисления, которые при записи в обратном порядке образуют требуемое число. Поэтому вычисленные остатки от деления заносятся в стек, а затем извлекаются из него для печати. Если основание системы счисления больше 10 (тогда для записи числа используются буквенные обозначения), на печать выводится соответствующий символ (10 – ‘A’, 11 – ‘B’ и т.д.).

```
// определение функции перевода положительного целого числа
// из десятичной системы счисления в любую заданную
// с основанием от 2 до 16
void PrintChislo(int number, int base)
{
      Stack s;
      s.head=NULL;
      int copy=number, n;

     // вычисленные остатки от деления числа
     // на основание системы счисления помещаются в стек
      while(number!=0)
      {
           PushStack(s,number%base);
           number=number/base;
      }

      // печать разрядов полученного числа
      // путем извлечения их значений из стека
      printf("Число %d в %d-ой системе счисления - ",
                                 copy,base);
      while(PopStack(s,n)==1)
      {
           if(n<10)
                printf("%d",n);
           else
                printf("%c",'A'+n-10);
      }
      printf("\n");
}
```

Домашнее задание

1. Написать функции уничтожения стека и очереди.

2. Написать функции печати стека и очереди.

3. 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.

4. 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.

5. 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.

6. Дана символьная строка с некоторым выражением, в котором могут содержаться скобки трех видов – (), {}, []. Написать функцию проверки правильности расстановки скобок в этой строке.

Правильное скобочное выражение определяется следующим образом:

1) Строка, не содержащая скобок, является правильным скобочным выражением.

2) Если строка A – правильное скобочное выражение, то (A), [A], и {A} – правильные скобочные выражения.

3) Если строки A и B – правильные скобочные выражения, то AB (конкатенация строк A и B) – правильное скобочное выражение.

Например, в выражениях [(A+B)*(A-B)+B и [(A+B]) нарушается второй пункт определения правильного скобочного выражения. В первом выражении

правильное скобочное выражение “(A+B)*(A-B)+B” некорректно заключено в [] (нет закрывающей скобки). Во втором выражении внутри [] содержится неправильное скобочное выражение “(A+B” (нет закрывающей круглой скобки).

Для решения задачи применяется следующий алгоритм синтаксического анализа, основанный на использовании стека. Выражение анализируется посимвольно слева направо, пока не будет обнаружен признак его конца или не нарушится пункт 2 определения. В анализируемом выражении рассматриваются только скобки. При нахождении открывающей скобки заносится в стек информация о ее типе. При обнаружении закрывающей скобки любого типа анализируется содержание стека. Если стек пуст, то считается, что скобки в выражении расставлены неправильно. Если стек не пуст, то из него извлекается последняя открывающая скобка и проверяется соответствие ее типа типу обнаруженной закрывающей скобки. Если в рассматриваемой паре скобок открывающая и закрывающая скобки имеют различный тип, выражение считается некорректным, в противном случае, продолжается поиск следующей скобки во входном выражении. При достижении признака конца строки вновь анализируется состояние стека. Пустой стек соответствует правильной расстановке скобок. Наличие скобок в стеке соответствует ошибке расстановки скобок.

Синтаксический анализ расстановки скобок иллюстрируется на примере следующего алгебраического выражения:

```
{ X + [ Y - ( A + B ) ] * C - [ ( D + E ) ] }
/\   /\    /\      /\ /\     /\ /\     /\/\/\
1     2     3       4  5      6  7      8 9 10
```

Под строкой алгебраического выражения указаны номера состояний стека при разборке скобок. Состояния стека приведены на следующей диаграмме состояний (рис. 4.1).

```
|     |    |     |    |_____|    |     |    |     |
|     |    |_____|    |__(__|    |_____|    |     |
|_____|    |__[__|    |__[__|    |__[__|    |_____|
|__{__|    |__{__|    |__{__|    |__{__|    |__{__|

   1          2          3          4          5
                                  '('~')'     '['~']'
|     |    |_____|    |     |    |     |    |     |
|_____|    |__(__|    |_____|    |     |    |     |
|__[__|    |__[__|    |__[__|    |_____|    |     |
|__{__|    |__{__|    |__{__|    |__{__|    |_____|

   6          7          8          9          10
                       '('~')'    '['~']'
```

Рис. 4.1. Диаграммы состояний стека.

7. 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:

а) по возрастанию номеров открывающих скобок;

б) по возрастанию номеров закрывающих скобок.

8. 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.

9. 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).

10. Дана символьная строка, содержащая правильно записанное арифметическое выражение. Написать функцию перевода выражения в постфиксную форму. Постфиксной формой выражения называется такая запись, в которой знак операции следует за операндами. Такая запись не содержит скобок. Например, выражение $a * b$ в постфиксной форме имеет вид $ab *$, $a * b+c$ имеет вид $ab * c +$, $a * (b + c)$ имеет вид $abc + *$.

Для решения задачи можно использовать следующий алгоритм. Рассматриваются поочередно все символы строки. В стек записывается открывающая скобка, и выражение далее анализируется посимвольно слева направо. Если встречается операнд (число или переменная), то он сразу помещается в строку-результат. Если встречается открывающая скобка, то она заносится в стек, а если встречается закрывающая скобка, то из стека извлекаются находящие-

ся там знаки операций до ближайшей открывающей скобки, которая также удаляется из стека. Все эти знаки в порядке их извлечения помещаются в строку-результат. Когда же встречается знак операции, то из стека извлекаются знаки операций, приоритет которых больше или равен приоритету данной операции, и они помещаются в строку-результат, после чего рассматриваемый знак переносится в стек. Когда выражение заканчивается, выполняются такие же действия, что и при встрече закрывающей скобки.

Рассмотрим алгоритм на примере простейшего выражения:

$$a + (b - c) * d$$

Таблица 1. Перевод выражения в постфиксную форму.

Символ	Действие	Состояние выходной строки	Состояние стека
a	'a' – переменная. Помещаем ее в строку-результат.	a	Пуст
+	'+' – знак операции. Помещаем его в стек (поскольку стек пуст, приоритеты можно не проверять).	a	+
(	'(' – открывающая скобка. Помещаем в стек.	a	(+
b	'b' – переменная. Помещаем ее в выходную строку.	a b	(+
-	'-' – знак операции, который имеет приоритет 2. Проверяем стек: на вершине находится символ '(', приоритет которого равен 1. Следовательно, мы должны просто поместить текущий символ '-' в стек.	a b	- (+
c	'c' – переменная. Помещаем ее в выходную строку.	a b c	- (+

Таблица 1. Перевод выражения в постфиксную форму.

Продолжение.

Символ	Действие	Состояние выходной строки	Состояние стека
)	')' – закрывающая скобка. Извлекаем из стека в выходную строку все символы, пока не встретим открывающую скобку. Затем уничтожаем обе скобки.	a b c -	+
*	'*' – знак операции, который имеет приоритет 3. Проверяем стек: на вершине находится символ '+', приоритет которого равен 2, т.е. меньший, чем приоритет '*'. Следовательно, мы должны поместить текущий символ '*' в стек.	a b c -	* +
d	'd' – переменная. Помещаем ее в выходную строку.	a b c - d	* +

Теперь вся входная строка разобрана, но в стеке еще остаются знаки операций, которые мы должны просто извлечь в строку-результат. Итак, мы получили конечный результат: «*a b c - d* * +».

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

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.

12. Решить задачу 7 при условии, что операндами выражения являются натуральные числа.

Двусвязные списки в отличие от односвязных характеризуются тем, что дают возможность просмотра элементов списка в обоих направлениях. Это достигается посредством хранения в каждом узле двух адресов: следующего и предыдущего элементов. Первый элемент имеет ссылку на предыдущий, равную `NULL`, а последний элемент имеет ссылку на следующий, равную `NULL` (рис. 5.1).

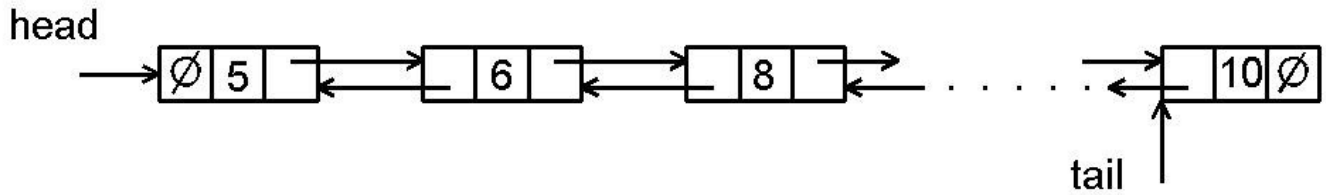


Рис. 5.1. Пример двусвязного списка

```
// элемент списка
struct Element
{
      int info;          // информационное поле
      Element* prev;     // указатель на предыдущий элемент
      Element* next;     // указатель на следующий элемент
};

// структура для двусвязного списка
struct TwoWayList
{
      Element* head;     // адрес первого элемента списка
      Element* tail;     // адрес последнего элемента списка
};
```

Далее приводится описание процесса решения некоторых задач с применением пользовательских функций.

Задача 1. Дан текстовый файл, содержащий целые числа. Написать функцию построения двусвязного списка из этих чисел.

Параметрами функции являются имя файла (`file_name`) и структура для хранения начала и конца списка (переменная `L`). Код возврата функции будет равен 1, если список создан успешно, 0 – если создан пустой список, -1 – если файл не был открыт.

Отдельно обрабатывается создание первого узла – выделение памяти и заполнение данными элемента `L.head`. Если файл пуст, то `L.head` и `L.-`

tail принимают значения NULL.

Для добавления каждого из последующих узлов используется переменная current, которая будет хранить адрес последнего созданного элемента списка. Ее начальное значение – адрес первого элемента. Далее для каждого считанного из файла числа создается новый узел (адрес запоминается в переменной help). Заполняются поля этого узла: info – число из файла, prev содержит адрес current. Элемент добавляется в конец списка. В завершении устанавливается признак конца списка (current->next=NULL) и адрес последнего элемента сохраняется в L.tail.

```
// определение функции построения двусвязного списка
// из чисел, хранящихся в файле
int CreateTwoWayList(char* file_name, TwoWayList& L)
{
      Node *current, *help;
      int k;
      // открытие файла
      FILE* f=fopen(file_name,"r");

      if(f==NULL)
      {
           // Ошибка открытия файла
           return -1;
      }

      // считывание первого числа из файла и
      // создание первого элемента списка
      if(fscanf(f,"%d",&k)>0)
      {
           L.head=new Node;
           L.head->info=k;
           L.head->prev=NULL;
      }
      else
      {
          // Файл пуст, следовательно, список тоже пуст
           L.head=L.tail=NULL;
           return 0;
      }

      current=L.head;
      // считывание еще непрочитанных чисел из файла и
      // добавление их в конец списка
      while(fscanf(f,"%d",&k)>0)
```

```
        {
            help=new Node;
            help->info=k;
            current->next=help;
            help->prev=current;
            current=current->next;
        }

        // установка признака конца двусвязного списка
        L.tail=current;
        current->next=NULL;
        fclose(f);
        return 1;
    }
```

Задача 2. Дан двусвязный список, содержащий целые числа. Определить, является ли он симметричным.

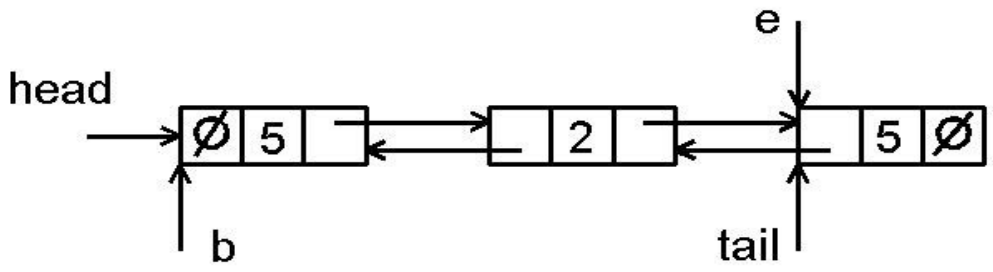
При решении задачи используются две переменные-указатели `b` и `e` для просмотра элементов с начала и с конца списка соответственно. Для этого их начальные значения устанавливаются на `L.head` и `L.tail`. Смещая их одновременно (`b` на следующий элемент, `e` – на предыдущий), проверяем значения в этих узлах на равенство. Смещение заканчивается в одном из двух случаев:

1) встретилась неравная пара значений;

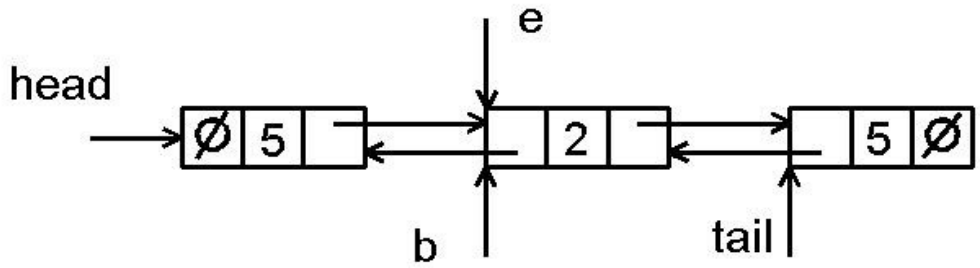
2) были просмотрены все элементы списка. При нечетном количестве узлов адреса `b` и `e` совпадут (рис. 5.2).

При четном количестве узлов адрес `b` будет указывать на первый элемент второй половины списка, а `e` – на последний элемент первой половины. В этом случае в качестве условия выхода из цикла можно использовать одно из следующих равенств: `b->prev=e` или `e->next=b` (рис. 5.3).

Если встретилась неравная пара значений, то смещение указателей прекращается, и функция возвращает значение `false`. Если просмотрены все элементы, то список является симметричным, и функция возвращает значение `true`. По определению пустой список является симметричным.

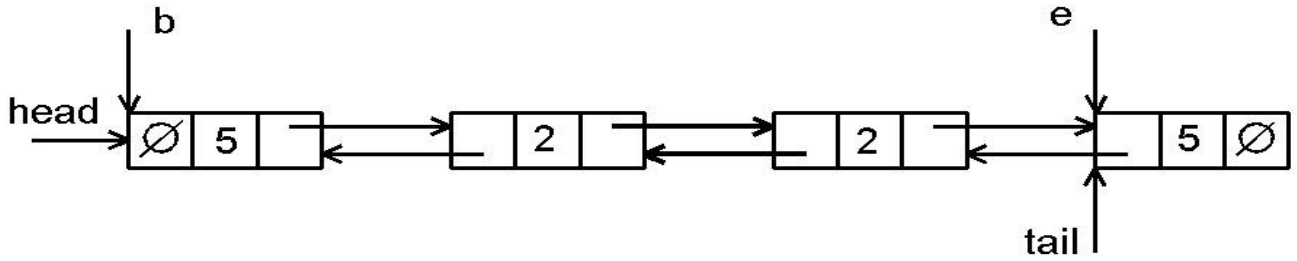


а

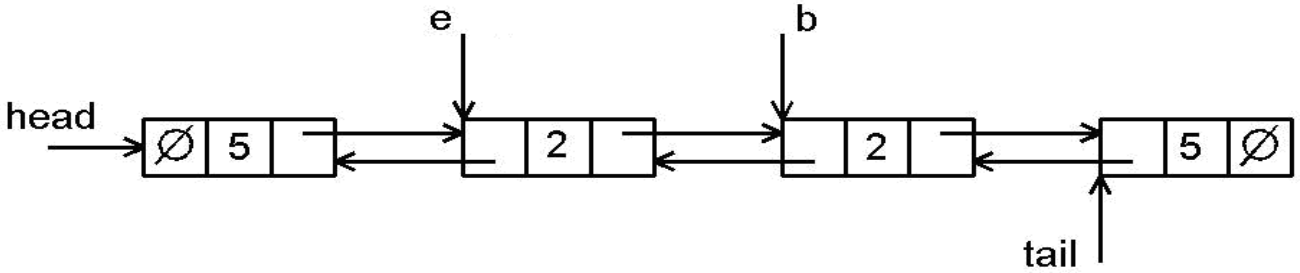


б

Рис. 5.2. Условие окончания просмотра элементов списка
при их нечетном количестве.



а



б

Рис. 5.3. Условие окончания просмотра элементов списка
при их четном количестве.

```
// определение функции проверки симметричности списка
```

```
bool Symmetry(TwoWayList& L)
{
     Node *b=L.head, *e=L.tail;
     if(b==NULL)  return false;

     // просмотр элементов с начала и с конца списка
     // до тех пор, пока значения соответствующих элементов
     // равны и не просмотрены все элементы списка
     while(b->info==e->info && b!=e && b->prev!=e)
     {
          b=b->next;
          e=e->prev;
      }

      // определение условия, по которому произошел
      // выход из цикла
      if(b->info!=e->info)
          // найдена несимметричная пара элементов
           return false;
     // список симметричен
     return true;
}
```

Задача 3. Дан двусвязный список, содержащий целые числа. Отсортировать его методом вставок.

Метод вставок применительно к массиву можно описать так. Пусть дан массив $a[0]\ldots a[n]$. Каждый элемент $a[i]$ (i изменяется от 1 до n) вставляется на подходящее место в уже упорядоченной совокупности $a[0]\ldots a[i\text{-}1]$. Это место определяется при обратном проходе по массиву от (i-1)-го элемента до элемента с меньшим значением, чем $a[i]$, или, если такого элемента не нашлось, до начала массива.

Если список пуст или состоит из одного элемента, сортировать его не нужно. Для просмотра всех элементов списка, начиная со второго, создается переменная `current`. Если `current` стоит на своем месте, т. е. предыдущий элемент имеет меньшее значение информационного поля, то `current` либо сдвигается на следующий узел, либо происходит выход из цикла, когда достигнут конец списка.

Если требуется изменить позицию элемента `current`, то выполняются следующие действия:

1) удалить `current` из списка, присоединив «хвост» списка к предыдущему элементу;

2) определить его место в уже отсортированной части списка;

3) вставить current на найденное место.

При удалении сохраняем адреса предыдущего и следующего элементов в переменных help1 и help2 соответственно. Далее соединяем эти два узла. Если следующего узла не существует, т.е. current был последним узлом, L.tail устанавливается на help1.

Для определения места вставки элемента, двигаемся от узла help1 к началу списка до тех пор, пока не встретим узел со значением, меньшим «вставляемого», или пока не достигнем начала списка. Если current нужно вставить на первое место, то связываем элементы current и help1 и меняем начало списка (L.head=current). Если же позиция вставки current не начальная, то сохраняем адрес элемента, перед которым нужно вставить current, в переменную help3, и организуем связи с учетом вставки current. Далее повторяем те же действия для следующего элемента, адрес которого был сохранен в переменной help2.

```
// определение функции сортировки списка методом вставок
void SortList(TwoWayList& L)
{
      Node* current,*help1,*help2,*help3;
      if(L.head==L.tail) return;
      current=L.head->next;
      bool f=true; // булева переменная, которая принимает
                   // значение true, если конец списка
                   // не достигнут, а иначе - false
      while(f)
      {
           if(current->prev->info > current->info)
           {
                // значение элемента current меньше,
                // чем значение последнего элемента в уже
                // упорядоченной части списка

                //извлекаем элемент current из списка
                help2=current->next;
                help1=current->prev;
                help1->next=current->next;
                // если элемент current был
                // последним в списке
                if(current==L.tail)
                {
                      // переносим конец списка
                      // на предыдущий элемент
```

```
                    L.tail=current->prev;
                    // устанавливаем признак достижения
                    // конца списка
                    f=false;
                }
                else help2->prev=help1;
                // двигаясь в обратном порядке от предыдущего
                // положения current, определяем место
                // вставки current
                while(help1!=L.head &&
                              help1->info>current->info)
                    help1=help1->prev;
                // вставка элемента current
                if(help1->info>current->info)
                {
                    // в начало списка
                    help1->prev=current;
                    current->next=help1;
                    L.head=current;
                }
                else
                {
                    // в середину упорядоченной части списка
                    help3=help1->next;
                    help3->prev=current;
                    current->next=help3;
                    current->prev=help1;
                    help1->next=current;
                }
                // продолжаем просмотр неупорядоченной части
                // списка
                current=help2;
            }
            else
            {
                // если список просмотрен и current стоит
                // на своем месте устанавливаем признак
                // достижения конца списка
                if(current==L.tail) f=false;
                // переход к следующему элементу
                current=current->next;
            }
        }
    }
```

Домашнее задание

1. Дан двусвязный список. Написать функцию печати списка в обратном порядке.

2. Дан двусвязный список. Написать функцию добавления элемента в заданную позицию.

3. Дан двусвязный список. Написать функцию удаления элемента из заданной позиции.

4. Дан двусвязный список. Написать функцию добавления в начало списка нового элемента, значение которого равно количеству вхождений минимального элемента в список.

5. Дан двусвязный список. Написать функцию добавления двух новых элементов – с минимальным значением в начало списка и с максимальным значением в конец списка.

6. Дан двусвязный список. Написать функцию удаления из него всех элементов с минимальным значением.

7. Дан двусвязный список. Написать функцию печати всех симметричных подпоследовательностей элементов списка.

8. Дан двусвязный список из n элементов (n – четное). Вычислить выражение:

$$\max\{a_1 + a_n;\ a_2 + a_{n-1};\ldots\ a_{n/2} + a_{n/2+1}\}$$

9. Дан двусвязный список из n элементов (n – четное). Вычислить выражение:

$$\max\{\min\{a_1, a_n\},\min\{a_2, a_{n-1}\},\ldots\ \min\{a_{n/2}, a_{n/2+1}\}\}$$

10. Дан двусвязный список. Написать функцию, которая:

1) формирует новый список, содержащий элементы исходного списка, располагая их в обратном порядке;

2) располагает элементы списка в обратном порядке, не создавая нового списка.

11. Написать функцию слияния двух отсортированных по неубыванию двусвязанных списков.

12. Даны два двусвязных списка. Написать функцию формирования нового двусвязного списка, состоящего из элементов, входящих одновременно в оба исходных списка.

13. Даны два двусвязных списка. Написать функцию формирования нового двусвязного списка из тех элементов первого списка, которые не содержатся во втором.

14. Дан двусвязный список. Написать функцию нахождения суммы элементов, расположенных между минимальным и максимальным элементами списка.

Глава 3. Нелинейные структуры данных

Раздел 6. Деревья

Еще одним из наиболее важных видов динамических структур данных являются деревья. Существует большое количество видов деревьев – свободные, ориентированные, упорядоченные, бинарные деревья, В-деревья. В данном разделе будут рассматриваться только бинарные деревья. Формально бинарное дерево можно представить в виде набора узлов (вершин), один из которых называется корнем дерева, а остальные узлы образуют два непересекающихся подмножества, каждое из которых также образует дерево (рис. 6.1). Узлы соединены между собой дугами. От каждого узла бинарного дерева может исходить две дуги к левому и правому узлам. При этом начальный узел называется родительским, два других узла – его потомками.

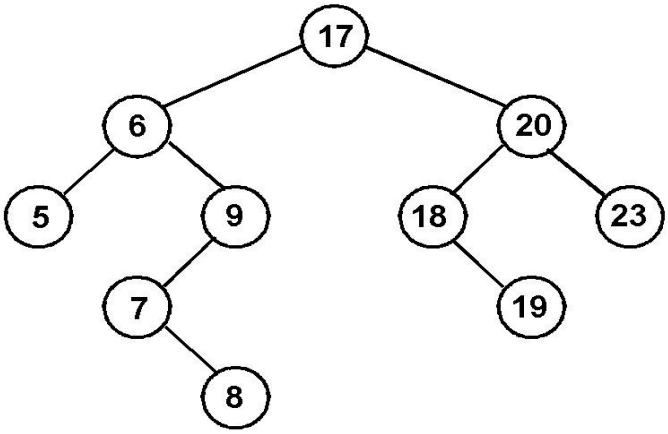


Рис. 6.1. Вид бинарного дерева.

На рис. 6.1 узел «17» является корнем дерева – узлом, у которого нет узла-родителя. Остальные узлы образуют два непересекающихся подмножества, называемых левым и правым поддеревьями. Узлы «6» и «20» являются потомками узла «17».

В свою очередь, каждое поддерево также является деревом, например, корнем левого поддерева является узел «6», который имеет двух потомков (рис. 6.2).

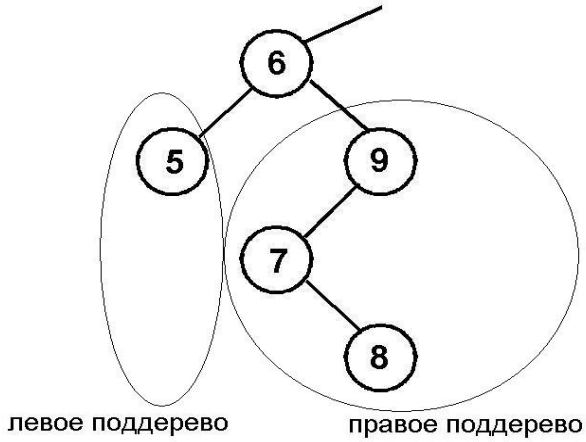


Рис. 6.2. Левое и правое поддеревья.

Узлы, расположенные на одинаковом расстоянии от корня дерева, образуют один уровень дерева. Листом дерева называют узел, не имеющий потомков. Глубина дерева – это номер максимального уровня дерева или максимальная длина пути от корня до какого-либо листа.

Например, на рис. 6.3 узел «17» образует нулевой уровень, узлы «6», «20» – первый уровень и т. д. Узлы «5», «8», «19», «23» являются листовыми, глубина дерева равна 4 (соответствует пути из узла «17» в листовой узел «8»).

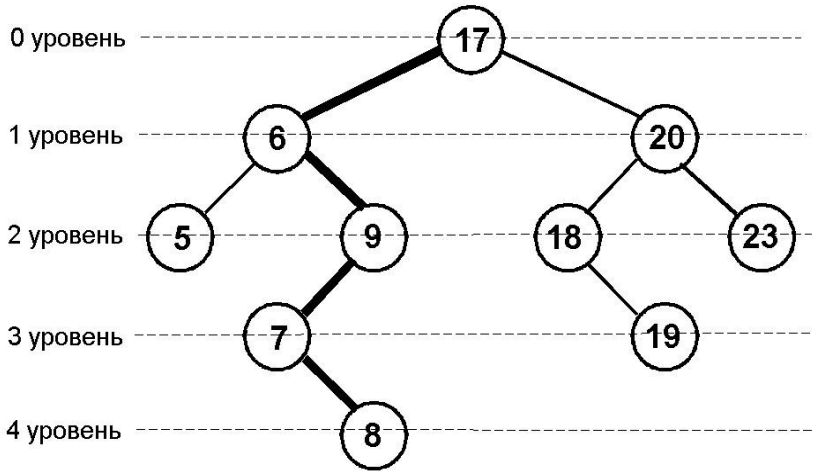


Рис. 6.3. Уровни дерева. Глубина дерева.

Вид структуры узла бинарного дерева такой:

```
// структура узла бинарного дерева
struct Tree
{
      int info;          // информационное поле
      Tree* left;        // указатель на левое поддерево
      Tree* right;       // указатель на правое поддерево
};
```

Подвидом бинарных деревьев является дерево сортировки. Это такое дерево, в котором слева от любой вершины находятся вершины с элементами, меньшими, чем в данной вершине, а справа – с большими или равными элементами. Пример дерева сортировки представлен на рис. 6.1.

Далее приводится описание процесса решения некоторых задач с применением пользовательских функций.

Задача 1. Дано дерево сортировки. Написать функцию добавления в дерево нового элемента.

При добавлении нового узла в дерево используются следующие правила:

1) если дерево было пустым, то значение помещается в корень дерева;

2) в противном случае просматриваются узлы дерева, начиная с корня, и осуществляется переход к левому поддереву, если новый элемент имеет меньшее значение, чем значение в текущей вершине. В противном случае – переход к правому поддереву.

Как только достигается вершина, от которой нет возможности передвигаться в нужном направлении, т.е. указатель на соответствующее поддерево равен `NULL`, создаем новый узел и присоединяем его к этой вершине в соответствии с определением дерева сортировки.

Рассмотрим вставку новых элементов в дерево сортировки, представленного на рис. 6.1. Например, сначала добавим элемент «12», затем «10».

Просматриваем узлы дерева, начиная с корня. Поскольку 17 больше 12, то переходим к левому потомку – узлу «6». 6 меньше 12, следовательно, переходим на правый потомок узла «6» – узел «9». 9 меньше 12, поэтому следует перейти к правому потомку, которого не существует. Таким образом, определилось место вставки нового элемента – справа от узла «9».

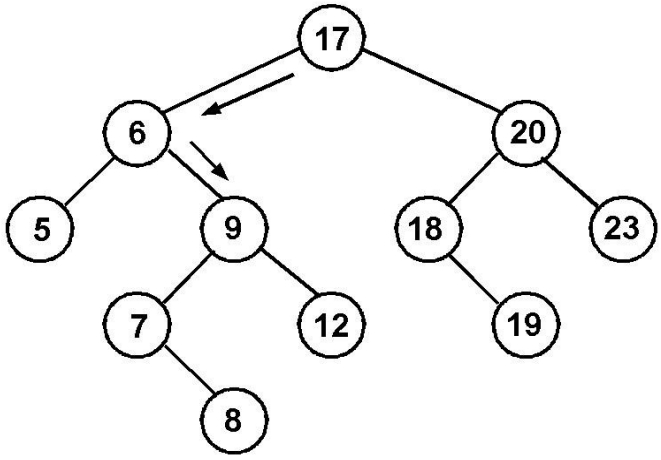


Рис. 6.4. Вставка значения 12.

При вставке узла со значением 10, аналогично переходим по узлам «17», «6», «9», «12», и поскольку 10 меньше 12, то связываем узел «10» с узлом «12» слева.

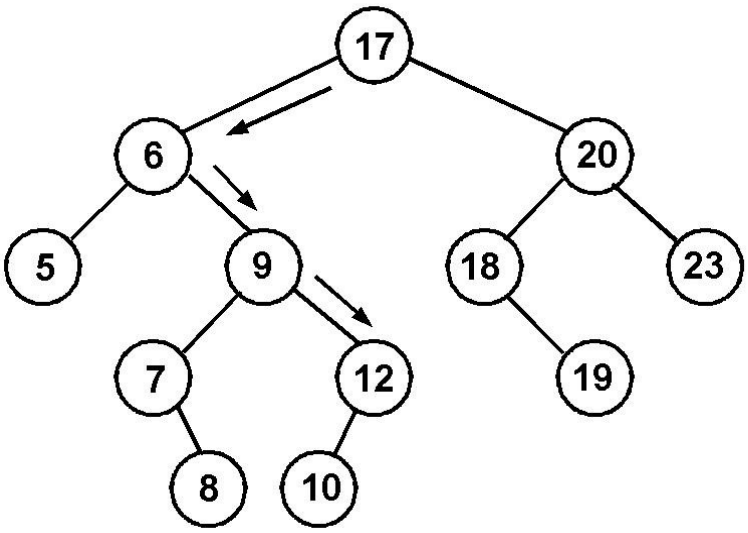


Рис. 6.5. Вставка значения 10.

Для решения задачи рассмотрим два варианта функции – рекурсивный и нерекурсивный. Разберем сначала рекурсивный вариант. Поскольку каждое поддерево по определению также является деревом, то каждый узел дерева можно обрабатывать как корень.

Параметрами функции будут – адрес корня поддерева (переменная `root`) и добавляемое значение (`key`).

Если дерево пусто, формируем корень дерева. Для этого адрес вновь созданного элемента сохраняется в переменной `root`, заполняется информационное поле, и ссылки на левое и правое поддеревья обнуляются. На этом функция завершает свою работу.

Если дерево непусто, вызываем эту же функцию для левого (со значениями аргументов `root->left` и `key`) или для правого поддеревьев (`root->right` и `key`), в зависимости от добавляемого значения.

```
// определение рекурсивной функции
// добавления в дерево нового элемента
void AddTree(Tree*& root, int key)
{
      // если дерево пусто, формируем корень дерева
      if(root==NULL)
      {
           Tree* help=new Tree;
```

```
            help->info=key;
            help->left=NULL;
            help->right=NULL;
            root=help;
            return;
        }
        // определяем, в левое или в правое поддерево узла root
        // добавляется новый элемент
        if(root->info>key)
            // рекурсивный вызов функции
            // добавления в левое поддерево
            AddTree(root->left,key);
        else
            // рекурсивный вызов функции
            // добавления в правое поддерево
            AddTree(root->right,key);
}
```

Теперь рассмотрим нерекурсивный вариант решения задачи.

В случае, когда дерево пусто, поступаем так же, как и в рекурсивном варианте.

Иначе перемещаемся по узлам дерева в соответствии с правилами, используя вспомогательную переменную `help`, до тех пор, пока не определим место вставляемого элемента. Если `help->info` больше `key`, то добавление нового узла осуществляется при условии, что `help->left = NULL`. Если же `help->info` меньше или равно `key`, то добавление нового узла осуществляется, когда `help->right=NULL`. Новый узел связывается с узлом `help` с помощью надлежащего указателя (`help->left` или `help->right`).

```
// определение нерекурсивной функции добавления
// в дерево нового элемента
void AddTree(Tree*& root, int key)
{
        Tree *help,*newNode;
        // если дерево пусто, формируем корень дерева
        if(root==NULL)
        {
            root = new Tree;
            root->info=key;
            root->left=NULL;
            root->right=NULL;
            return;
```

```
        }
        help=root;
        // двигаемся по узлам дерева, начиная с корня,
        // пока не найдем место вставки нового элемента
        while (true)
        {
            if(help->info>key)
                // в текущем узле help значение больше нового
                if(help->left==NULL)
                {
                    // если у узла help нет левого
                    // поддерева, новый узел
                    // «подцепляется» к узлу help слева
                    newNode=new Tree;
                    newNode->info=key;
                    newNode->left=NULL;
                    newNode->right=NULL;
                    help->left= newNode;
                    return;
                }
                else
                    // если существует левое поддерево,
                    // перемещаемся к левому потомку help
                    help=help->left;
            else
                // в текущем узле help значение
                // меньше или равно новому
                if(help->right==NULL)
                {
                    // если у узла help нет правого
                    // поддерева, новый узел
                    // «подцепляется» к help справа
                    newNode=new Tree;
                    newNode->info=key;
                    newNode->left=NULL;
                    newNode->right=NULL;
                    help->right= newNode;
                    return;
                }
                else
                    // если существует правое поддерево,
                    // перемещаемся к правому потомку help
                    help=help->right;
        }
    }
```

Заметим, что функцию добавления элемента в дерево удобно использовать и для построения дерева сортировки. Далее приводится пример функции создания дерева сортировки на основе данных из текстового файла.

```
// определение функции создания дерева на основе
// данных текстового файла
int CreateTree(char* FileName, Tree*& root)
{
      int k;
      // открытие файла
      FILE* f=fopen(FileName,"r");
      if(f==NULL)
      {
           //Ошибка открытия файла
           return -1;
      }

      // изначально дерево пусто
      root=NULL;
      // в цикле считываем числа из файла в переменную k
      while(fscanf(f,"%d",&k)>0)
           // добавляем элемент со значением k в дерево
           AddTree(root,k);
      fclose(f);
      return 1;
}
```

Задача 2. Дано дерево сортировки. Написать функцию печати элементов дерева в порядке неубывания.

Согласно правилам построения дерева сортировки сначала следует распечатать элементы левого поддерева, потом элемент самой вершины и, в завершении, элементы правого поддерева.

Очень просто выглядит решение этой задачи с использованием рекурсивной функции, поскольку само определение дерева является рекурсивным (каждое поддерево также является деревом).

```
// определение рекурсивной функции распечатки
// значений из узлов дерева в порядке неубывания
void PrintTree(Tree* root)
{
      if(root!=NULL)
      {
           // печатаем значения из левого поддерева узла root
           PrintTree(root->left);
```

```
            // печатаем информационное поле узла root
            printf("%d ",root->info);
            // печатаем значения из правого
            // поддерева узла root
            PrintTree(root->right);
        }
    }
```

При написании нерекурсивного варианта функции печати требуется использовать стек. Для возвращения к вершинам верхнего уровня, в нем нужно хранить адреса уже пройденных вершин.

В таблице 2 поясняется процесс печати элементов дерева, изображенного на рис. 6.1, в порядке неубывания.

Таблица 2. Печать элементов дерева.

Содержимое по адресу `temp`	Состояние стека	Выходная строка	Комментарии
17	Пуст		
6	17		Начинаем движение до крайнего левого листового узла
5	17, 6		
NULL	17, 6, 5		Левого потомка узла «5» не существует.
5	17, 6	5	Извлекаем из стека элемент 5, выводим на печать. Переходим к правому потомку узла «5».
NULL	17, 6		Правого потомка узла «5» тоже не существует. Поэтому переходим к родительскому узлу «6», извлекая его из стека.
6	17	5, 6	Переходим к правому потомку узла «6»
9	17		Начинаем движение влево.
7	17, 9		

Таблица 2. Печать элементов дерева. Продолжение.

Содержимое по адресу `temp`	Состояние стека	Выходная строка	Комментарии
NULL	17, 9, 7		Движение влево невозможно, извлекаем родительский узел из стека.
7	17, 9	5, 6, 7	Переходим к правому потомку узла «7».
8	17, 9		Начинаем движение влево. Левого потомка узла «8» не существует.
NULL	17, 9, 8		
8	17, 9	5, 6, 7, 8	Переходим к правому потомку узла «8».
NULL	17, 9		Переходим к родительскому узлу «9», извлекая его из стека.
. . . . . и т. д.			
23	пуст		Переходим к правому потомку узла «20».
NULL	23		Начинаем движение влево. Левого потомка узла «23» не существует.
23	пуст	5, 6, 7, 8, 9, 17, 18, 19, 20, 23	Извлекаем из стека элемент 23, выводим на печать. Переходим к правому потомку узла «23».
NULL	пуст		Все узлы распечатаны.

Для реализации стека, в котором содержатся адреса вершин дерева, используется структура следующего вида:

```
// структура одного элемента стека
struct Element
{
      Tree* node;   //адрес узла дерева - информационное поле
      Element* next; // указатель на следующий элемент списка
};

struct Stack
{
      Element* head;    // адрес вершины стека
};
```

Функции добавления и извлечения элемента из стека изменятся следующим образом:

```
 // определение функции добавления в стек нового элемента
 void PushStack(Stack& s, Tree* ptr)
 {
      Element* p=new Element;
      p->next=s.head;
      p->node=ptr;
      s.head=p;
}

// определение функции извлечения элемента из стека
int PopStack(Stack& s, Tree*& ptr)
{
      if(s.head==NULL)
           return -1;
      ptr=s.head->node;
      Element* p=s.head;
      s.head=s.head->next;
      delete p;
      return 1;
}
```

Создается переменная `temp`, с помощью которой осуществляется просмотр вершин дерева.

Адрес текущей вершины помещается в стек, затем осуществляется переход к левому поддереву (`temp=temp->left`), т.е. текущей вершиной становится левый потомок. И так продолжается до тех пор, пока не достигнем крайнего левого узла и обработаем его (`temp=NULL`). Далее извлекаем узел из стека, печатаем его значение и переходим к его правому потомку. Если правого потомка у узла нет, из стека извлекается очередная вершина. Если же

стек пуст, то это означает, что дерево было просмотрено полностью. В противном случае в стеке остались непросмотренные вершины, поэтому снова осуществляется попытка прохода по левой ветке, но уже для другой текущей вершины.

```
// определение нерекурсивной функции распечатки элементов
// дерева в порядке неубывания
void PrintTree(Tree* root)
{
      if(root==NULL) return;
      // создание вспомогательного стека
      Stack s; s.head=NULL;
      // текущий узел – корень дерева
      Tree* temp=root;
      // обход будет закончен, если все узлы окажутся
      // просмотренными, т.е. temp будет указывать
      // на несуществующий узел, и стек станет пустым
      while(s.head!=NULL || temp!=NULL)
      {
           // перемещаемся по крайней левой ветке,
           // исходящей из узла temp
           while(temp!=NULL)
           {
                // помещаем каждый узел ветки в стек
                PushStack(s,temp);
                temp=temp->left;
           }
           // «прошли» последний узел левой ветки,
           // осуществляем возврат, т.е. извлекаем из стека
           // предыдущую вершину и сохраняем ее в temp
           PopStack(s,temp);
           // печатаем значение информационного поля
           printf("%d ",temp->info);
           // переходим на правого потомка текущего узла temp
           temp=temp->right;
      }
}
```

Задача 3. Дана символьная строка, содержащая правильное арифметическое выражение. Операндами выражения являются числа с плавающей точкой, допустимые операции – сложение, вычитание, умножение и деление. Требуется построить дерево разбора этого выражения.

Деревом разбора выражения называется бинарное дерево, построенное по следующим правилам:

1) формула из одного числа представляется деревом из одной вершины, операндом которого является это число;

2) если выражение имеет операции, то листовые узлы (не имеющие потомков) будут хранить операнды, а остальные узлы – операции, причем левое и правое поддеревья узла-операции будут представлять собой выражение его операндов.

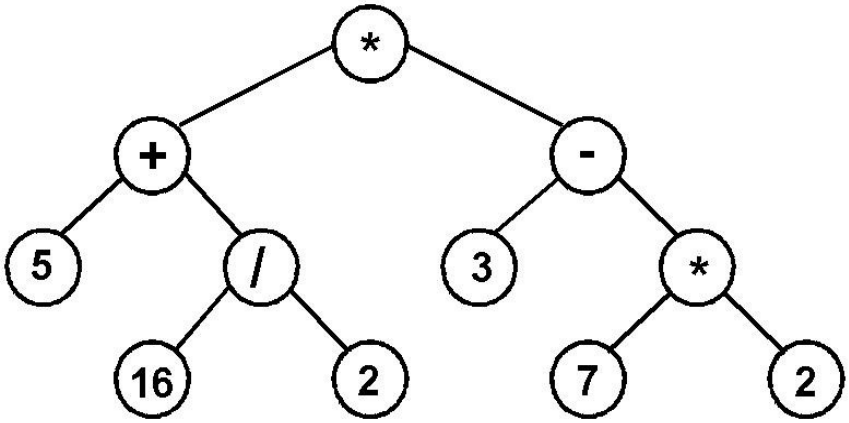


Рис. 6.6 Дерево разбора выражения (5+16/2)*(3-7*2).

Узел дерева хранит два информационных поля – символ операции и значение операнда. Если в узле дерева хранится операнд, то поле операции будет содержать символ '@'.

```
// структура для хранения узла дерева
struct Tree
{
      float operand;      // поле операнда
      char operation; // поле операции
      Tree* left;    // указатель на левый операнд операции
      Tree* right;   // указатель на правый операнд операции
};
```

При построении дерева надо учитывать приоритет операций. Поскольку умножение и деление выполняются в первую очередь, то в дереве они будут располагаться на более низких уровнях по отношению к сложению и вычитанию (если таковые имеются в формуле).

Параметрами функции являются символьная строка (`str`), содержащая выражение, признак обрабатываемых операций (`level`) и длина строки (`len`). Особо поясним значение параметра `level`. Если он равен 0, это означает, что строка содержит все возможные операции и операнды, если `level`

равно 1, тогда в рассматриваемой подстроке могут содержаться только операции умножения и деления и операнды, при `level`, равном 2 – только операнды.

Возвращаемое значение функции – адрес созданного дерева.

Выражение начинаем разбирать с конца.

Рассмотрим три возможных случая обработки строки, соответствующих трем значениям параметра `level`.

1) Когда значение аргумента `level` равно 0, осуществляется поиск первого знака '+' или '–'. Знак операции помещается в вершину дерева, и рекурсивно вызывается функция построения левого поддерева из подформулы, расположенной в строке до символа найденной операции, а затем правого поддерева, начиная со следующего символа. При этом в правой части строки не содержится операций сложения и вычитания, следовательно, значение аргумента `level=1`. Левая часть строки может содержать все допустимые операции и операнды. Для нее значение `level=0`.

Если же в строке не встретилось ни одного знака '+' или '–', то вызывается функция построения дерева-выражения со значением `level=1`.

2) Когда значение аргумента `level` равно 1, осуществляется поиск первого знака '*' или '/'. Далее происходит построение поддеревьев аналогично первому случаю. В правой части строки не содержится операций умножения и деления, поэтому значение аргумента `level=2`. Левая часть строки может содержать операции умножения и деления и операнды. Для нее значение `level=1`.

3) Когда значение аргумента `level` равно 2, создается новый узел с операндом и связывается с узлом соответствующей операции.

```
// определение функции создания дерева разбора выражения
Tree* CreateTree(char* str, int level, int len)
{
      Tree* next;
      // начинаем анализировать строку-выражение с конца
      int i=len-1;
      switch(level)
      {
           // выражение может содержать операции
           // '+', '-', '*', '/'
           case 0:
                // ищем позицию первой с конца
                // операции '+' или '-'
                while(i!=0 && str[i]!='+' && str[i]!='-') i--;
```

```
                if((str[i]=='+' || str[i]=='-') && i!=0)
                {
                    // операция '+' или '-' найдена, создаем
                    // узел дерева для этой операции
                    next=new Tree;
                    next->operation=str[i];

                    // левый операнд – выражение, находящееся
                    // слева от i-го символа, оно может
                    // содержать все виды операций (level=0)
                    // осуществляем рекурсивный вызов для
                    // построения поддерева этого выражения
                    next->left=CreateTree(str,0,i);

                    // правый операнд – выражение,
                    // находящееся справа от i-го символа,
                    // оно может содержать только
                    // операции '*' и '/' (level=1)
                    // осуществляем рекурсивный вызов для
                    // построения поддерева этого выражения
                    next->right=CreateTree(str+i+1,1,
                                           len-i-1);
                }
                else
                    // операции '+' и '-' не найдены,
                    // выражение может содержать только
                    // операции '*' и '/' (level=1)
                    // осуществляем рекурсивный вызов для
                    // построения поддерева выражения
                    next=CreateTree(str,1, len); break;

        // выражение может содержать операции '*', '/'
        case 1:
            // ищем позицию первой с конца
            // операции '*' или '/'
            while(i!=0 && str[i]!='*' && str[i]!='/') i--;
            if(str[i]=='*' || str[i]=='/')
            {
                // операция '*' или '/' найдена, создаем
                // новый узел дерева для этой операции
                next=new Tree;
                next->operation=str[i];

                // левый операнд – выражение, находящееся
                // слева от i-го символа, оно может содержать
                // операции '*' и '/' (level=1)
```

```
                // осуществляем рекурсивный вызов для
                // построения поддерева этого выражения
                next->left=CreateTree(str,1, i);

                // правый операнд - выражение, находящееся
                // справа от i-го символа, оно может
                // содержать только операнды (level=2)
                // осуществляем рекурсивный вызов для
                // построения поддерева этого выражения
                next->right=CreateTree(str+i+1,2, len-i-1);
            }
            else
                // операции '*' и '/' не найдены, выражение
                // может содержать только операнды (level=2)
                // осуществляем рекурсивный вызов для
                // построения поддерева выражения
                next=CreateTree(str,2,len); break;

        // выражение содержит операнды
        case 2:
            // создаем новый листовой узел дерева
            next=new Tree;
            next->operation='@';
            sscanf(str, "%f", &(next->operand));
            next->left=NULL;
            next->right=NULL;
        }
        // возвращаем адрес созданного узла
        // первый вызов функции вернет
        // адрес корневого узла дерева
        return next;
    }
```

Домашнее задание

1. Дано бинарное дерево. Написать функцию уничтожения дерева.

2. Дано бинарное дерево. Написать функцию вычисления глубины дерева.

3. Дано бинарное дерево. Написать функцию вычисления количества узлов на заданном уровне дерева.

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

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

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

7. Дано дерево разбора выражения. Написать функцию вычисления выражения.

8. Дано дерево разбора выражения. Распечатать постфиксную (префиксную) форму этого выражения.

Напомним, что постфиксной записью выражения называется такая запись, в которой знак операции следует за операндами. Такая запись не содержит скобок. Например, выражение $a * b$ в постфиксной форме имеет вид $ab *$, $a * b + c$ имеет вид $ab * c +$, $a * (b + c)$ имеет вид $abc + *$.

Префиксной записью выражения называется такая запись, в которой знак операции предшествует ее операндам. Такая запись также не содержит скобок. Например, выражение $a * b$ в префиксной форме имеет вид $* ab$, $a*b+c$ имеет вид $+ * abc$, $a * (b + c)$ имеет вид $* a + bc$.

Раздел 7. Графы

На практике часто приходится решать задачи, в которых моделируются схемы расположения объектов и связей между ними. Например, города – это объекты, а дороги, их соединяющие, – это связи. Для таких задач удобно использовать структуру данных, которая называется графом.

Граф – это пара множеств *V* и *E*, где *V* – непустое конечное множество точек, называемых вершинами, а *E* – множество ребер, соединяющих пары вершин.

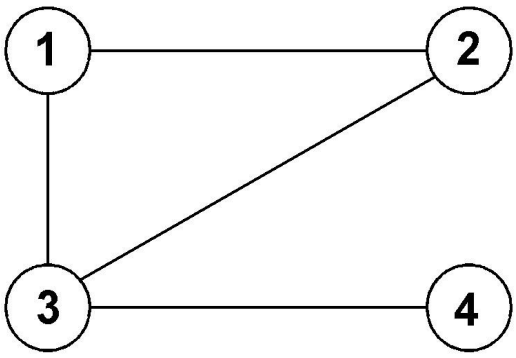


Рис. 7.1 Пример графа.

На рис. 7.1 представлен граф с множеством вершин {«1», «2», «3», «4»} и множеством ребер {<1, 2>, <1, 3>, <2, 3>, <3, 4>}.

Пусть *u* и *v* – вершины, а *e* – ребро, их соединяющее.

В этом случае вершина *u* и ребро *e* называются инцидентными (также как и вершина *v* и ребро *e*). На рис. 7.1 вершина «1» и ребро <1, 2> инцидентны друг другу. Степень вершины – это число ребер, инцидентных ей, например, вершина «1» имеет степень 2. Ребра, инцидентные одной вершине, называются смежными. Вершины, соединенные одним ребром, также называются смежными. Например, смежными являются вершины «1» и «2», смежными ребрами являются <1, 3>, <2, 3> и <3, 4>.

Граф называется полным, если любые две вершины соединены ребром. 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 несмежны в *G*. Объединение графов *G* и *G*’ образуют полный граф. Граф, изображенный на рис. 7.1, не является полным. Его дополнение изображено на рис. 7.2.

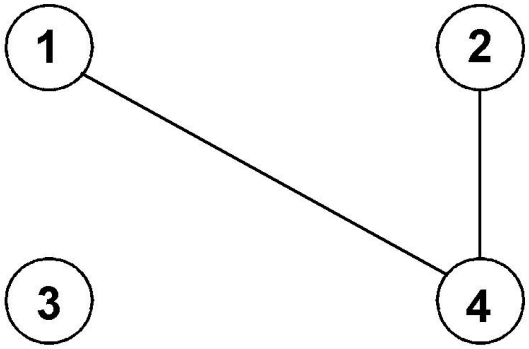


Рис. 7.2. Дополнение графа, изображенного на рис. 7.1.

Если каждому ребру графа ставится в соответствие некоторое число, то оно называется весом ребра, а граф называется взвешенным.

Путь, соединяющий вершины u и v – это последовательность вершин v_0, $v_1 \ldots v_n$, где $v_0=u$, а $v_n= v$ и для любых $i = 0 \ldots n$ вершины v_i и v_{i+1} соединены ребром. Если граф взвешенный, то длиной пути называется сумма весов его ребер. Путь называется замкнутым, если $v_0 = v_n$. Цикл – это замкнутый путь, в котором ни одно ребро не повторяется дважды. Простой цикл – это цикл, в котором ни одна вершина не повторяется дважды.

Расстояние между двумя вершинами – это длина кратчайшего пути, соединяющего эти вершины.

Деревом называется связный граф без циклов.

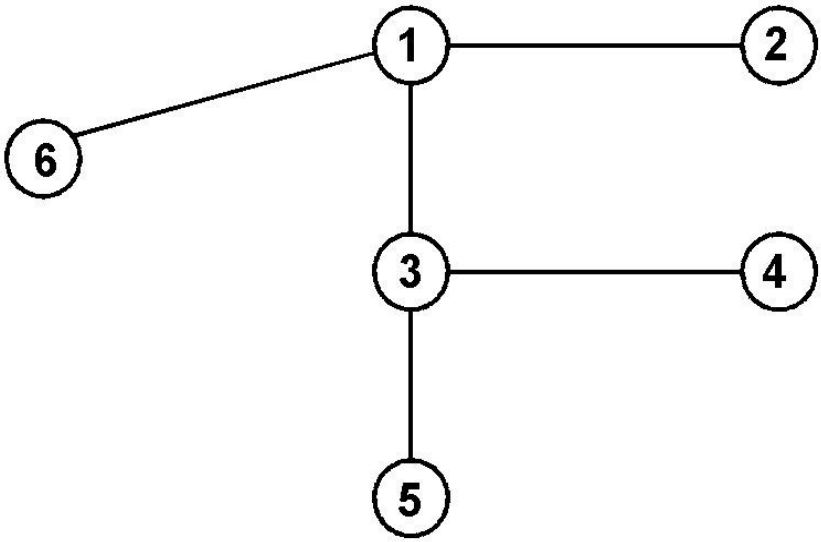


Рис. 7.3. Пример дерева – связного графа без циклов.

Те деревья, которые были рассмотрены в предыдущем разделе, являются частным случаем такого графа с выделенной корневой вершиной и направ-

ленными ребрами от родителя к потомку.

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.

Каркасом называется дерево, содержащее все вершины исходного графа и часть его ребер. Граф может иметь несколько каркасов. Каркасами для графа, представленного на рис. 7.1, являются следующие графы (деревья):

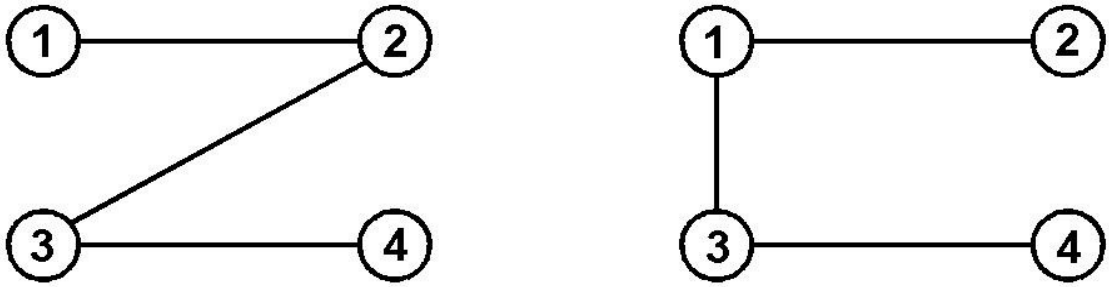


Рис. 7.4. Каркасы графа, изображенного на рис. 7.1.

Приведем часто используемые способы представления графов.

1. **Матрица инцидентности**

A(n, m)={a_{ij}},

где n – число вершин, а m – число ребер графа

$$a_{ij} = \begin{cases} 1, \text{ вершина } i \text{ инцидентна ребру } j \\ 0, \text{ вершина } i \text{ и ребро } j \text{ не инцидентны} \end{cases}$$

В каждом столбце такой матрицы содержится ровно две единицы. Равных столбцов в матрице нет.

Приведем пример матрицы инцидентности для графа, изображенного на рис. 7.1.

$$A = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} <1,2> & <1,3> & <2,3> & <3,4> \end{array} \\ \begin{pmatrix} 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix} \end{array}$$

Недостаток данного способа представления состоит в том, что требуется $m*n$ ячеек памяти, большинство из которых заполнены нулями.

2. **Матрица смежности**.

$A(n, n)=\{ a_{ij} \}$,

где n – число вершин.

$$a_{ij} = \begin{cases} 1, \text{ вершина } i \text{ смежна вершине } j \\ 0, \text{ вершины } i \text{ и } j \text{ не смежны} \end{cases}$$

В случае взвешенного графа элементу a_{ij} вместо 1 присваивается значение соответствующего веса, а вместо 0 – большое число, обозначающее бесконечность (в текстах программ большое число обозначается `MAX_INT` $= 10^6$). Эта матрица является симметричной с нулями на главной диагонали. Число единиц в строке равно степени соответствующей вершины.

Приведем пример матрицы смежности для графа, изображенного на рис. 7.1.

$$A = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 1 & 1 & 0 \\ 2 & 1 & 0 & 1 & 0 \\ 3 & 1 & 1 & 0 & 1 \\ 4 & 0 & 0 & 1 & 0 \end{array}$$

Преимуществом данного способа является легкость получения информации о существовании ребра между двумя вершинами. По сравнению с предыдущим способом требуется меньшее количество памяти для графа с большим числом ребер. Недостатком считается постоянный размер матрицы, не зависящий от числа ребер графа.

3. **Список ребер**.

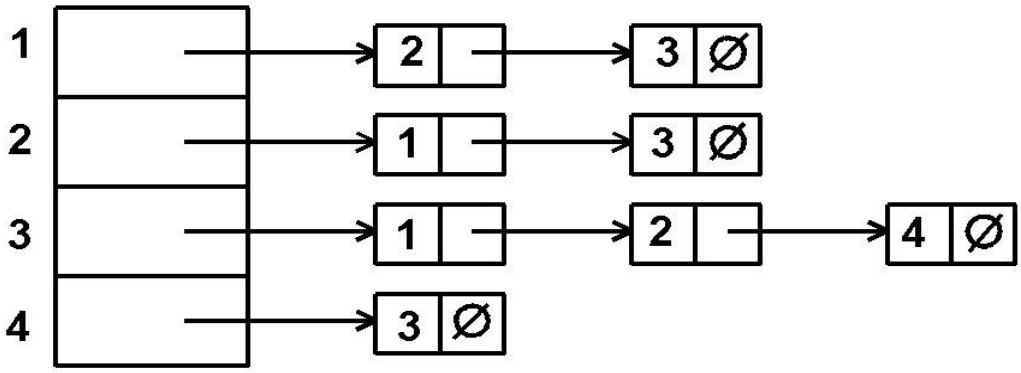
Граф представляется в виде списка (массива) элементов, хранящих информацию про одно ребро (номер начальной и конечной вершин, длина ребра и пр.)

4. **Динамический список смежности (хэш-таблица)**.

В данном случае создается массив списков по количеству вершин графа.

Каждый *i*-ый список из массива содержит номера вершин, смежных с *i*-ой вершиной.

Приведем пример динамического списка смежности для графа, изображенного на рис. 7.1:



Для каждого из представлений графа приведем описание структур данных.

```
// структура графа в виде матрицы инцидентности
struct Graph1
{
      int n;    // количество вершин графа
      int m;    // количество ребер графа
      int** a;  // матрица инцидентности графа
};

// структура графа в виде матрицы смежности
struct Graph2
{
      int n;    // количество вершин графа
      int m;    // количество ребер графа
      int** a;  // матрица смежности графа
};

// граф в виде массива ребер

// структура для хранения информации об одном ребре графа
struct Edge
{
      int begin;    // номер начальной вершины ребра
      int end;      // номер конечной вершины ребра
};
```

```
// структура графа в виде списка (массива) ребер
struct Graph3
{
      int n;        // количество вершин графа
      int m;        // количество ребер графа
      Edge * edges; // массив элементов типа Edge
};

// граф в виде списков смежности

// структура одного элемента списка,
// хранящего номер смежной вершины
struct ElementList
{
      int number;        // номер вершины
      ElementList* next;// адрес следующего элемента списка
};

// структура графа в виде списков смежности
struct Graph4
{
      int n;                // количество вершин графа
      int m;                // количество ребер графа
      ElementList** nodes; // массив, содержащий списки
                            // смежных вершин
};
```

Далее приводится описание процесса решения некоторых задач с применением пользовательских функций.

Задача 1. Дан граф в виде списка ребер. Преобразовать граф в матрицу смежности.

Сначала выделяем память под матрицу смежности. Первоначально заполняем все ее элементы нулями. Проходя по всем ребрам из массива, заполняем элементы матрицы единицами, исходя из номеров начальной и конечной вершин ребра.

```
// определение функции преобразования представления графа
// из массива дуг в матрицу смежности
void Preobr32(Graph3& g1, Graph2& g2)
{
      int i,j;
      g2.n=g1.n; g2.m=g1.m;
      // выделение памяти под матрицу смежности
```

```
        g2.a=new int*[g2.n];
        for(i=0;i<g2.n; i++)
             g2.a[i]=new int[g2.n];
        for(i=0; i<g2.n;i++)
             for(j=0; j<g2.n;j++)
                  g2.a[i][j]=0;

        // заполнение элементов матрицы единицами
        // в зависимости от номеров начальной и конечной
        // вершин каждого ребра
        for(i=0; i<g1.m; i++)
        {
             g2.a[g1.edges[i].begin-1][g1.edges[i].end-1]=1;
             g2.a[g1.edges[i].end-1][g1.edges[i].begin-1]=1;
        }
}
```

Задача 2. Дан граф в виде списка ребер. Преобразовать граф в динамический список смежности.

Сначала выделяется память под массив заголовков списков смежных вершин. В начальный момент все списки пустые. Далее для каждого ребра из массива ребер (номер начальной вершины сохраняем в переменной `begin`, номер конечной вершины – в переменной `end`) осуществляется добавление новых элементов в списки: элемента с вершиной `end` в конец списка смежности для вершины `begin` и элемента с вершиной `begin` в конец списка смежности для вершины `end`.

```
// определение функции преобразования представления графа
// из массива дуг в списки смежности
void Preobr34(Graph3& g1, Graph4& g2)
{
     int i;
     g2.n=g1.n; g2.m=g1.m;

     // создание массива списков смежных вершин
     // по количеству вершин графа
     g2.nodes=new ElementList*[g2.n];
     for(i=0;i<g2.n;i++)
          g2.nodes[i]=NULL;
     int begin,end;
     for(i=0;i<g1.m;i++)
     {
          // создание для каждого ребра двух элементов
          // в списках смежных вершин
```

```
            begin=g1.edges[i].begin;
            end=g1.edges[i].end;

            // конечная вершина ребра смежна начальной
            if(g2.nodes[begin-1]==NULL)
            {
                // создание первого элемента списка смежных
                // вершин, если список пуст
                g2.nodes[begin-1]=new ElementList;
                g2.nodes[begin-1]->next=NULL;
                g2.nodes[begin-1]->number=end;
            }
            else
            {
                // добавление нового элемента в «хвост»
                // списка смежных вершин
                ElementList* current=g2.nodes[begin-1],*help;
                while(current->next!=NULL)
                    current=current->next;
                help=new ElementList;
                help->next=NULL;
                help->number=end;
                current->next=help;
            }

            // начальная вершина ребра смежна конечной
            if(g2.nodes[end-1]==NULL)
            {
                g2.nodes[end-1]=new ElementList;
                g2.nodes[end-1]->next=NULL;
                g2.nodes[end-1]->number=begin;
            }
            else
            {
                ElementList* current=g2.nodes[end-1],*help;
                while(current->next!=NULL)
                    current=current->next;
                help=new ElementList;
                help->next=NULL;
                help->number=begin;
                current->next=help;
            }
        }
    }
```

Задача 3. Дан граф в виде матрицы смежности. Определить длины всех путей из узла А в узел В.

Разберем два варианта решения этой задачи – рекурсивный и нерекурсивный.

Идея алгоритма в обоих случаях заключается в следующем. Находясь в некоторой текущей вершине А, последовательно переходим в каждую из смежных с ней вершин. Каждую пройденную вершину помечаем как вершину, включенную в путь, и одновременно добавляем длину соответствующего ребра к длине строящегося пути. Таким образом, все вершины фактически разделены на две группы: уже включенные в путь и оставшиеся. Теперь задача сводится к нахождению всех путей из последней помеченной вершины в конечную, при этом рассматриваются только те смежные с ней вершины, которые еще не включены в путь. Если на каком-то этапе достигается конечная вершина, печатаем путь.

Если для некоторой вершины все пути рассмотрены, снимаем с нее пометку и возвращаемся в предыдущую вершину, продолжая поиск оставшихся путей из нее.

Например, пусть дан граф, изображенный на рис. 7.5.

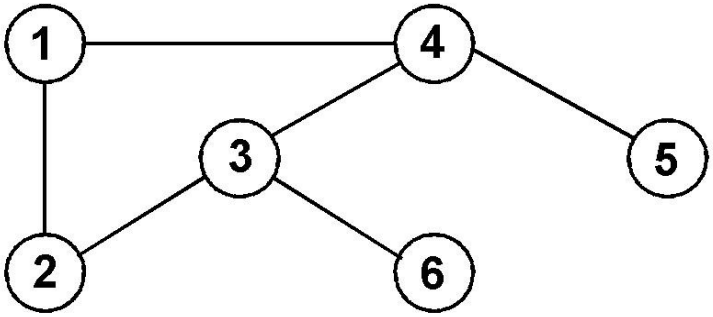


Рис. 7.5. Пример графа.

Требуется найти все пути из вершины 1 в вершину 6. Помечаем вершину 1 как включенную в путь. Для нее существуют две смежные вершины – 2 и 4. Сначала переходим в вершину 2 и помечаем ее. Теперь задача свелась к поиску всех путей из вершины 2 в вершину 6. Для текущей вершины 2 смежными являются две – 1 и 3. Однако вершина 1 уже включена в путь, поэтому ее не рассматриваем и переходим в вершину 3. Из нее сначала находится путь 3->4->5. Вершина 5 не имеет непомеченных смежных вершин, поэтому возвращаемся в вершину 4. Для нее нерассмотренных возможных путей больше нет. Поэтому возвращаемся в вершину 3, для которой существует еще одна нерассмотренная и непомеченная смежная вершина 6. Таким образом, найден путь из начальной вершины в конечную (1->2->3->6).

Аналогично, осуществляем возврат до вершины 1. После рассмотрения всех возможных путей через вершину 2, остаются пути, которые проходят через вершину 4. Из всех возможных путей (1->4->3->2, 1->4->3->6 и 1->4->5) искомым путем является 1->4->3->6. Таким образом, найдены все возможные пути из вершины 1 в вершину 6.

Функция `AllWays` реализует рекурсивный вариант этого алгоритма. Параметрами функции являются исходный граф в виде матрицы смежности `g`, номера начальной вершины `s` и конечной вершины `t`, массив пометок вершин графа `x`, номер текущей вершины в строящемся пути `k` и длина уже построенного пути `sum`. Пометка в массиве `x` представляет собой номер вершины в построенном пути, например, `x[3]=2` означает, что вершина 3 в построенном пути является второй.

```
// определение рекурсивной функции поиска всех путей
// в графе, соединяющих две заданные вершины
void AllWays(Graph2 g,int s, int t, int* x, int k, int sum)
{
    int i,j,nom;
    s--; t--;
    // указываем, что порядковый номер s-ой вершины графа
    // в строящемся пути равен k
    x[s]=k;

    // условие выхода из рекурсии -
    // проверка достижения конечной вершины пути
    if(s==t)
    {
        // цикл распечатки найденного пути
        nom=1;
        while(true)
        {
            for(i=0;i<g.n;i++)
                if(x[i]==nom)
                {
                    printf("%d",i+1);
                    break;
                }
            if(i==b) break;
            printf("->");
            nom++;
        }
        printf(" sum=%d\n",sum);
        x[s]=0;
```

```
            return;
        }

        // цикл поиска непомеченных вершин,
        // смежных s-ой вершине
        for(j=0;j<g.n;j++)
            if(x[j]==0 && g.a[s][j]>0 && g.a[s][j]!=MAX_INT)
                // переход в смежную вершину с номером j
                // рекурсивный вызов для поиска всех путей
                // из j-ой вершины в t-ую
                AllWays(g,j+1,t+1,x,k+1, sum+g.a[s][j]);

        // возврат из s-ой вершины, снятие пометки о том,
        // что s-ая вершина включена в путь
        x[s]=0;
    }
```

Для нерекурсивного варианта этой функции требуется только граф и номера начальной и конечной вершин. Массив пометок, номер вершины в пути и длина пути будут созданы внутри функции.

```
    // определение нерекурсивной функции поиска всех путей
    //в графе,  соединяющих две заданные вершины
    void AllWays(Graph2 g,int s, int t, int* x)
    {
        int v=s,      // номер текущей вершины
            k=2,      // порядковый номер вершины в пути
            jj=-1,    // номер вершины, с которой начинается
                      // поиск смежных вершин
           sum=0,     // длина построенного пути
           j,f,ii,nom;

        if(s==t) return;
        s--; t--;

        // вспомогательный стек для хранения вершин возврата
        Stack st;
        st.head=NULL;

        // пометка начальной вершины
        x[v]=1;

        while(true)
        {
            f=0;
```

```
// поиск смежных вершин с v-ой
for(j=jj+1;j<g.n;j++)
     if(x[j]==0 && g.a[v][j]>0
              && g.a[v][j]!= MAX_INT)
     {
          // найдена смежная вершина j
          x[j]=k; // вершина включается в путь,
          //добавляется длина дуги <v,j>
          sum+=g.a[v][j];
          if(j==b)
          {
               // если найдена конечная вершина пути
               // печатаем найденный путь
               nom=1;
               while(true)
               {
                    for(ii=0;ii<g.n;ii++)
                         if(x[ii]==nom)
                         {
                              printf("%d",ii+1);
                              break;
                         }
                    if(ii==t)
                    {
                         printf("sum=%d\n",sum);
                         break;
                    }
                    printf("->");
                    nom++;
               }
               // осуществляем возврат в предыдущую
               // вершину, т.е. снимаем пометку с
               // j-ой вершины и вычитаем длину
               // ребра <v,j> из длины пути
               x[j]=0;
               sum-=g.a[v][j];
          }
          else
          {
               // конечная вершина не достигнута
               // помещаем вершину v в стек,
               // запоминаем найденную смежную
               // вершину в jj
               k++;
               PushStack(st,v);
```

```
                    jj=j;
                    f=1; break;
                }
            }
        if(f==1)
        {
            // если перешли в очередную вершину пути,
            // запоминаем ее номер в v и указываем,
            // что смежные вершины будем искать,
            // начиная с нулевой вершины
            v=jj; jj=-1;
        }
        else
        {
            // не нашли смежных непомеченных вершин,
            // осуществляем возврат в предыдущую вершину
            jj=v;   // сохраняем номер текущей вершины,
                    // чтобы продолжить поиск смежных
                    // вершин со следующей за ней
            if(PopStack(st,v)!=-1)
            {
                // извлекли из стека предшествующую
                // вершину и поместили ее номер в v,
                // с прежней вершины jj снимаем пометку
                x[jj]=0;
                k--;
                sum-=g.a[v][jj];
            }
            else
                // стек оказался пустым, т.е. были
                // просмотрены все возможные варианты,
                // осуществляем выход из цикла
                break;
        }
    }
}
```

Задача 4. Дан взвешенный граф в виде матрицы смежности. Найти кратчайший путь между двумя заданными вершинами (`s` – начальная вершина, `t` – конечная вершина).

Одним из самых известных алгоритмов решения данной задачи является алгоритм Дейкстра (1959 г.).

Кратчайший путь будет представлен с помощью двух массивов `T` и `H`. `T[v]` – длина кратчайшего пути от начальной вершины к `v`, `H[v]` – вершина,

непосредственно предшествующая v на кратчайшем пути.

Для решения задачи требуется вспомогательный массив X размера n. С его помощью будут помечаться вершины, которые уже были просмотрены, поэтому начальные значения элементов массива равны нулю. Массив T первоначально должен заполняться бесконечными значениями, поскольку длина кратчайшего пути пока неизвестна.

Для начальной вершины s H[s]=0 (начальной вершине ничего не предшествует), T[s]=0 (путь s->s имеет длину 0), X[s]=1 (вершина просмотрена). Фиксируется вершина v=s.

Для зафиксированной вершины перебираем все смежные ей непросмотренные вершины (переменная u). Если длина пути из s в u через v (T[v] +g.a[v][u]) (длина кратчайшего пути до v плюс вес ребра v->u) меньше, чем прежнее значение T[u], меняем его и запоминаем в H[u] предшествующую вершину v. Далее находим минимальный элемент массива T, соответствующий непросмотренным вершинам (в переменной v запоминаем его индекс). Если все оставшиеся элементы массива T равны бесконечности, можно сделать вывод, что требуемого пути не существует (в этом случае функция возвращает значение 0). Если найденный минимум соответствует вершине t, то задача решена (в этом случае функция возвращает значение 1). В противном случае помечаем вершину v как пройденную и осуществляем перебор смежных ей вершин.

Рассмотрим процесс решения задачи поиска кратчайшего пути из вершины 1 в вершину 6 на графе, представленном на рис. 7.6 (таблица 3).

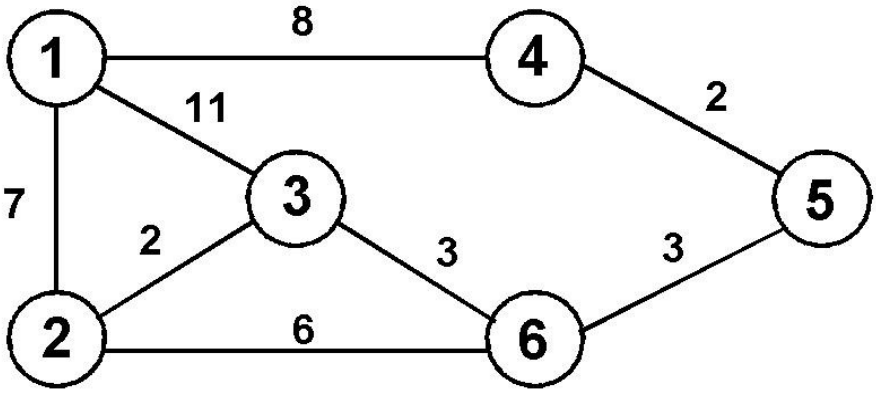


Рис. 7.6. Пример взвешенного графа.

Таблица 3. Поиск кратчайшего пути алгоритмом Дейкстра.

№	Действия с комментариями	Зафиксир. вершина	Состояние массива пометок X	Состояние массива длин T	Состояние массива путей H
1	Фиксируем начальную вершину пути	1	{1, 0, 0, 0, 0, 0}	{1, 0, 0, 0, 0, 0}	{0,∝, ∝, ∝, ∝, ∝}
2	Для вершины 1 смежными непомеченными вершинами являются вершины 2, 3, 4. Меняем для них элементы массивов T и H, т.е. T[2], H[2], T[3], H[3], T[4], H[4] (в T заносим длину ребра, в H – номер зафикс. вершины).	1	{1, 0, 0, 0, 0, 0}	{0, 7, 11, 8, ∝, ∝}	{0, 1, 1, 1, ∝, ∝}
3	Находим в массиве T минимальный элемент, среди тех, которые соответствуют непомеченным вершинам (T[2]=7). Помечаем соответсвтующую минимальному значению вершину 2 и фиксируем ее.	2	{1, 1, 0, 0, 0, 0}	{0, 7, 11, 8, ∝, ∝}	{0, 1, 1, 1, ∝, ∝}
4	Для вершины 2 смежными непомеченными вершинами являются вершины 3, 6. Меняем для них элементы массивов T и H.	2	{1, 1, 0, 0, 0, 0}	{0, 7, 9, 8, ∝, 13}	{0, 1, 2, 1, ∝, 2}

Таблица 3. Поиск кратчайшего пути алгоритмом Дейкстра. Продолжение.

№	Действия с комментариями	Зафиксир. вершина	Состояние массива пометок `X`	Состояние массива длин `T`	Состояние массива путей `H`
5	Находим в массиве `T` минимальный элемент, среди тех, которые соответствуют непомеченным вершинам (`T[4]`=8). Помечаем вершину 4 и фиксируем ее.	4	{1, 1, 0, 1, 0, 0}	{0, 7, 9, 8, ∝, 13}	{0, 1, 2, 1, ∝, 2}
6	Для вершины 4 смежной непомеченной вершиной является вершина 5. Меняем `T[5]` и `H[5]`.	4	{1, 1, 0, 1, 0, 0}	{0, 7, 9, 8, 10, 13}	{0, 1, 2, 1, 4, 2}
7	Находим в массиве `T` минимальный элемент, среди соответствующих непомеченным вершинам (`T[3]`=9). Помечаем вершину 3 и фиксируем ее.	3	{1, 1, 1, 1, 0, 0}	{0, 7, 9, 8, 10, 13}	{0, 1, 2, 1, 4, 2}
8	Для вершины 3 смежной непомеченной вершиной является вершина 6. Меняем элементы `T[6]` и `H[6]`.	3	{1, 1, 1, 1, 0, 0}	{0, 7, 9, 8, 10, 12}	{0, 1, 2, 1, 4, 3}
9	Находим в массиве `T` минимальный элемент, среди тех, которые соответствуют непомеченным вершинам (`T[5]`=10). Помечаем вершину 5 и фиксируем ее.	5	{1, 1, 1, 1, 1, 0}	{0, 7, 9, 8, 10, 12}	{0, 1, 2, 1, 4, 3}
10	Для вершины 5 смежной непомеченной вершиной является вершина 6. Ми-				

	нимальным элементом массива T среди тех, которые соответствуют непомеченным вершинам, является T[6]=12. Поскольку вершина 6 – конечная вершина, то задача решена.

Таким образом, длина кратчайшего пути из вершины 1 в вершину 6 равна 12 (T[6]).

```
// определение функции поиска кратчайшего пути
// между двумя вершинами с помощью алгоритма Дейкстра
int Dijkstra(Graph2& g, int s, int t, int* H, int* T)
{
      // создание вспомогательного массива пометок
      int* X=new int[g.n];
      int flag,i,m,u,
      v=s-1; // номер текущей вершины
      // начальная инициализация массива пометок X и массива
      // длин кратчайших путей T
      for(i=0;i<g.n;i++)
      {
           T[i]=MAX_INT;
           X[i]=0;
      }
      // инициализация элементов массивов для начальной
      // вершины s
      H[s-1]=0;
      T[s-1]=0;
      X[s-1]=1;
      while(true)
      {
           // цикл поиска непомеченных вершин,
           // смежных с s-ой вершиной
           for(u=0;u<g.n;u++)
                if(g.a[v][u]!=0 && g.a[v][u]!=MAX_INT)
                {
                     // корректировка длин кратчайших путей
                     // происходит, если путь из вершины s
                     // в вершину u, проходящий через
                     // вершину v, будет короче, чем прежний
                     if(X[u]==0 && T[u]>T[v]+g.a[v][u])
                     {
                          T[u]=T[v]+g.a[v][u];
                          H[u]=v+1;
                     }
                }
```

```
            // поиск непомеченной вершины, которой
            // соответствует минимальный кратчайший путь из
            // s-ой вершины
            m=MAX_INT; v=0;
            for(u=0;u<g.n;u++)
                if(X[u]==0 && T[u]<m)
                {
                    v=u+1;
                    m=T[u];
                }

            // если нет путей во все непомеченные вершины,
            // пути в конечную тоже не существует
            if(v==0)
            {
                delete [] X;
                return 0;
            }

            // если найденная вершина является конечной,
            // путь найден
            if(v==t)
            {
                delete [] X;
                return 1;
            }
            // конечной вершины еще не достигли, помечаем
            // вершину с минимальной длиной кратчайшего пути
            // как пройденную, и она становится текущей
            v--;
            X[v]=1;
        }
    }
```

Особо требуется пояснить, как осуществить печать найденного пути из массива `H`.

Так как элемент `H[i]` содержит вершину, предшествующую `i`-ой вершине пути, то сам путь можно получить только в обратном порядке (от вершины `t` к вершине `s`). Начиная с конечной вершины `v=t`, переходим от вершины `v` к вершине `H[v]`, пока не дойдем до начальной вершины `s` (для распечатки пути в прямом порядке можно применить стек).

```
. . .
i=t-1;
```

```
printf("%d<-",t);
while(H[i]!=0)
{
      printf("%d<-",H[i]);
      i=H[i]-1;
}
. . .
```

Рассмотрим процесс печати кратчайшего пути для нашего примера. Результатом решения задачи является массив `H[]`={0, 1, 2, 1, 4, 3}. Находясь в конечной вершине 6, получаем номер предыдущей вершины на пути `H[6]=3`. Далее вершине 3 предшествует `H[3]=2`, второй – `H[2]=1`. Таким образом, кратчайший путь из вершины 1 в вершину 6 : 1->2->3->6.

Задача 5. Дан взвешенный граф в виде матрицы смежности. Найти кратчайшие пути между всеми парами вершин.

Для решения этой задачи часто используется алгоритм Флойда (1962 г.).

Результатом работы алгоритма являются две матрицы: `H` и `T` размера `n`×`n`, где `n` – количество вершин графа. Элемент матрицы `T[i][j]` соответствует кратчайшему расстоянию между вершинами `i` и `j`. В матрице `H` хранится информация о кратчайших путях:

$$H[i][j]=\begin{cases} k, \text{ если } k \text{ – номер первой вершины, в которую необходимо перейти, двигаясь из } i \text{ в } j \\ 0, \text{ если из вершины } i \text{ в вершину } j \text{ нет пути.} \end{cases}$$

Сначала осуществляется начальная инициализация матриц `H` и `T`. Матрица `T` совпадает с матрицей смежности, т.е. фиксируются пути, состоящие из одного ребра. В матрице `H` для каждой пары смежных вершин в качестве начального значения берется номер конечной вершины ребра.

Для любой пары вершин `j` и `k` осуществляется попытка включить в путь новую вершину `i`. Это происходит при выполнении следующих условий:

1) вершина `i` не является концом рассматриваемого пути, т.е. не совпадает с `j` или с `k`;

2) существует путь из `j` в `i` и из `i` в `k`;

3) длина полученного пути через вершину `i` меньше, чем прежняя длина.

Если все эти условия выполнены, то меняется длина кратчайшего пути в матрице `T`, и первой вершиной на пути из `j` в `k` становится первая вершина на

пути из j в i.

```
// определение функции построения всех кратчайших
// путей в графе  с помощью алгоритма Флойда
void Floyd(Graph2 g, int**& H, int**& T)
{
      int i,k,j;
      // выделение памяти под матрицу длин кратчайших путей T
      // и матрицу путей H
      T=new int*[g.n];
      H=new int*[g.n];
      for(i=0;i < g.n;i++)
      {
           T[i]=new int[g.n];
           H[i]=new int[g.n];
      }
      // начальная инициализация матриц T и H согласно
      // матрице смежности графа
      for(i=0;i < g.n;i++)
           for(j=0;j< g.n;j++)
           {
                T[i][j]=g.a[i][j];
                if(g.a[i][j]==MAX_INT) H[i][j]=0;
                else H[i][j]=j+1;
           }
      for(i=0;i< g.n;i++)
      {
           for(j=0;j< g.n;j++)
                for(k=0;k< g.n;k++)
                     // если можно проехать из j-ой вершины в
                     // k-ую через i-ую более коротким путем,
                     // меняем длину T[j][k] и путь
                     if(i!=j && i!=k && T [j][i]!=MAX_INT &&
                           T[i][k]!= MAX_INT &&
                           T[j][k]>T[j][i]+T[i][k])
                     {
                          H[j][k]=H[j][i];
                          T[j][k]=T[j][i]+T[i][k];
                     }
      }
}
```

Задача 6. Дан взвешенный граф в виде матрицы смежности. Найти диаметр графа, т.е. максимальное расстояние между всевозможными парами его вершин.

Параметрами функции являются исходный граф, и две ссылки на переменные, в которые будут помещены номера двух вершин (расстояние между этими вершинами является максимальным в графе).

Решить эту задачу можно, используя алгоритм Флойда. Полученная с помощью этого алгоритма матрица `T` является матрицей расстояний. Находим максимальный элемент матрицы, значение которого и будет являться диаметром графа.

```
// определение функции поиска диаметра графа
int Diametr(Graph2 g, int &s, int &t)
{
      int ** T;
      int ** H;
      int i,j;

      // получение матрицы расстояний между вершинами графа
      Floyd(g,H,T);

      // поиск максимального расстояния
      int max=-MAX_INT;
      for(i=0;i<g.n;i++)
           for(j=0;j<g.n;j++)
                if(T[i][j]!=MAX_INT && i!=j && T[i][j]>max)
                {
                     max=T[i][j];
                     s=i+1;
                     t=j+1;
                }
      return max;
}
```

Задача 7. Дан взвешенный граф в виде матрицы смежности. Построить для него каркас минимального веса.

Для построения каркаса имеется несколько алгоритмов. Мы будем использовать алгоритм Прима (1957 г.).

Сформированный каркас будет храниться в массиве ребер. Каждому ребру соответствуют номера начальной и конечной вершин и длина. Для этого создадим структуру `Edge`:

```
struct Edge
{
      // номера начальной и конечной вершин ребра
      int begin, end;
```

```
            // длина ребра
            int len;
        };
```

В алгоритме Прима каркас формируется посредством последовательного добавления вершин. Для определения добавляемой на каждой итерации вершины используются два вспомогательных массива. Элемент массива `alpha[i]` – это ближайшая к *i*-ой вершина, уже включенная в каркас, `beta[i]` – это длина ребра, соединяющего *i*-тую вершину с вершиной `alpha[i]`, `nodes` – это массив номеров вершин, включенных в каркас.

В `nodes` включается начальная вершина. Просматривая все смежные с ней вершины (переменная `v`), запоминаем для них в массиве `alpha` начальную вершину, а в массиве `beta` – длину соответствующего ребра. Для несмежных вершин, соответственно, `alpha[v]=-1` и `beta[v]=∞`.

До тех пор, пока в каркас не будут включены все вершины, выполняем следующие действия:

1) среди вершин, не включенных в каркас, находим ближайшую к нему. Обозначим найденную вершину через `w`;

2) добавляем вершину `w` в каркас:

а) в массив `nodes` заносим вершину `w`;

б) в массив ребер заносим ребро (`alpha[w],w`);

3) пересчитываем те элементы массивов `alpha` и `beta`, которые соответствуют не включенным в каркас вершинам. Для них вершина `w` может стать ближайшей из каркаса.

Рассмотрим процесс решения задачи построения каркаса минимального веса алгоритмом Прима для графа, изображенного на рис. 7.6 (таблица 4).

Таблица 4. Построение каркаса минимального веса алгоритмом Прима.

№	Действия с комментариями	Тек. вершина	Состояние массива ребер `ostov`	Состояние массива вершин в каркасе `nodes`	Состояние массивов ближайших вершин `alpha` и длин `beta`
1	Включаем в каркас вершину 1	1	пуст	{1}	–
2	Заполняем массивы `alpha` и `beta`. Смежными с вершиной 1 являются вершины 2, 3, 4. В `beta` заносим длины всех ребер, смежных с вершиной 1, в `alpha` – номер текущей вершины 1.	1	пуст	{1}	`alpha` = {0, 1, 1, 1, -1, -1} `beta` = {0, 7, 11, 8, ∝, ∝}
3	Находим в массиве `beta` минимальный элемент среди тех, которые соответствуют вершинам, не включенным в `nodes`: `beta[2]`=7. Включаем вершину 2 в массив `nodes`, а ребро <1, 2> в каркас. Вершина 2 становится текущей.	2	{<1,2>}	{1, 2}	`alpha` = {0, 1, 1, 1, -1, -1} `beta` = {0, 7, 11, 8, ∝, ∝}
4	Изменяем массивы `alpha` и `beta`. Смежными с вершиной 2 являются вершины 3, 6. В `beta` заносим длины ребер из вершины 2 во все смежные, в `alpha` – номер текущей вершины 2.	2	{<1,2>}	{1, 2}	`alpha` = {0, 1, 2, 1, -1, 2} `beta` = {0, 7, 2, 8, ∝, 6}

Таблица 4. Построение каркаса минимального веса алгоритмом Прима. Продолжение.

№	Действия с комментариями	Тек. вершина	Состояние массива ребер ostov	Состояние массива вершин в каркасе nodes	Состояние массивов ближайших вершин alpha и длин beta
5	Находим в массиве beta минимальный элемент среди тех, которые соответствуют вершинам, не включенным в nodes: beta[3]=3. Включаем вершину 3 в массив nodes, а ребро <2, 3> в каркас. Вершина 3 становится текущей.	3	{<1,2>, <2,3>}	{1, 2, 3}	alpha = {0, 1, 2, 1, -1, 2} beta = {0, 7, 2, 8, ∝, 6}
6	Изменяем массивы alpha и beta. Смежной с вершиной 3 является вершина 6. В beta заносим длину ребра из вершины 3 в вершину 6, в alpha – номер текущей вершины 3.	3	{<1,2>, <2,3>}	{1, 2, 3}	alpha = {0, 1, 2, 1, -1, 3} beta = {0, 7, 2, 8, ∝, 3}
7	Находим в массиве beta минимальный элемент среди тех, которые соответствуют вершинам, не включенным в nodes: beta[6]=3. Включаем вершину 6 в массив nodes, а ребро <3, 6> в каркас. Вершина 6 становится текущей.	6	{<1,2>, <2,3>, <3,6>}	{1, 2, 3, 6}	alpha = {0, 1, 2, 1, -1, 3} beta = {0, 7, 2, 8, ∝, 3}

Таблица 4. Построение каркаса минимального веса алгоритмом Прима. Продолжение.

№	Действия с комментариями	Тек. вершина	Состояние массива ребер `ostov`	Состояние массива вершин в каркасе `nodes`	Состояние массивов ближайших вершин `alpha` и длин `beta`
8	Изменяем массивы `alpha` и `beta`. Смежной с вершиной 6 является вершина 5. В `beta` заносим длину ребра из вершины 6 в вершину 5, в `alpha` – номер текущей вершины 6.	6	{<1,2>, <2,3>, <3,6>}	{1, 2, 3, 6}	`alpha` = {0, 1, 2, 1, 6, 3} `beta` = {0, 7, 2, 8, 3, 3}
9	Находим в массиве `beta` минимальный элемент среди тех, которые соответствуют вершинам, не включенным в `nodes`: `beta[5]`=3. Включаем вершину 5 в массив `nodes`, а ребро <5, 6> в каркас. Вершина 5 становится текущей.	5	{<1,2>, <2,3>, <3,6>, <5, 6>}	{1, 2, 3, 6, 5}	`alpha` = {0, 1, 2, 1, 6, 3} `beta` = {0, 7, 2, 8, 3, 3}
10	Изменяем массивы `alpha` и `beta`. Смежной с вершиной 5 является вершина 4. В `beta` заносим длину ребра из вершины 5 в вершину 4, в `alpha` – номер текущей вершины 5.	5	{<1,2>, <2,3>, <3,6>, <5, 6>}	{1, 2, 3, 6, 5}	`alpha` = {0, 1, 2, 5, 6, 3} `beta` = {0, 7, 2, 2, 3, 3}

Таблица 4. Построение каркаса минимального веса алгоритмом Прима. Продолжение.

№	Действия с	Тек.	Состояние	Состояние	Состояние мас-

	комментариями	вер-шина	массива ребер ostov	массива вершин в каркасе nodes	сивов ближайших вер-шин alpha и длин beta
11	Находим в массиве beta минимальный элемент среди тех, ко-торые соответствуют вершинам, не вклю-ченным в nodes: beta[4]=2. Включа-ем вершину 4 в массив nodes, а ребро <4, 5> в каркас. Вершина 4 становится текущей.	4	{<1,2>, <2,3>, <3,6>, <5, 6>, <4, 5>}	{1, 2, 3, 6, 5, 4}	–
В каркас добавлены все вершины. Решение получено.					

```
// определение функции построения каркаса минимального веса
// для графа с помощью алгоритма Прима
void Prim(Graph2 g, Edge*& ostov)
{
      // выделение памяти под массив ребер каркаса
      ostov=new Edge[g.n-1];
      // выделение памяти под массив включенных
      // в каркас вершин
      int* nodes=new int[g.n];
      // выделение памяти под вспомогательные массивы
      // ближайших вершин из каркаса и расстояний до них
      int* alpha=new int[g.n];
      int* beta=new int[g.n];
      // включение в каркас нулевой вершины и инициализация
      // вспомогательных массивов согласно матрице смежности
      nodes[0]=0;
      int count=1, u=0, v, w, i, ii, min, flag;
      for(v=1;v<g.n;v++)
           if(g.a[0][v]!=MAX_INT)
           {
                // если v смежна нулевой вершине
                alpha[v]=0;
                beta[v]=g.a[0][v];
           }
           else
```

```
        {
            // если v не смежна нулевой вершине
            alpha[v]=-1;
            beta[v]=MAX_INT;
        }
    for(i=0;i<g.n-1;i++)
    {
        // поиск ближайшей вершины среди невключенных
        // в каркас
        min=MAX_INT;
        for(v=1;v<g.n;v++)
        {
            // проверка на включение вершины в каркас
            flag=0;
            for(int ii=0; ii<count; ii++)
                if(nodes[ii]==v)
                    flag=1;
            if(flag==0)
                if(beta[v]<min)
                {
                    w=v;
                    min=beta[v];
                }
        }
        // включение найденной вершины в каркас
        nodes[count]=w;
        count++;
        // включение ребра <w, alpha[w]> в каркас
        ostov[i].begin=alpha[w]+1;
        ostov[i].end=w+1;
        ostov[i].len=g.a[alpha[w]][w];
        // цикл корректировки значений во вспомогательных
        // массивах - вершина w может оказаться ближайшей
        // из каркаса для еще невключенных вершин
        for(v=0;v<g.n;v++)
        {
            flag=0;
            for(ii=0;ii<count; ii++)
                if(nodes[ii]==v)
                    flag=1;
            if(flag==0 && g.a[w][v]!=MAX_INT)
                if(beta[v]>g.a[w][v])
                {
                    alpha[v]=w;
                    beta[v]=g.a[v][w];
```

```
                        }
                }
        }
        delete [] nodes;
        delete [] alpha;
        delete [] beta;
}
```

Домашнее задание

1. Написать функции преобразования графа из одного вида представления в другой:

а) из матрицы смежности в списки смежности.

б) из матрицы смежности в матрицу инцидентности.

в) из списков смежности в матрицу смежности.

2. Дан граф. Написать функцию построяния 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 несмежны в *G*.

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

4. 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.

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

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

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

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

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

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

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

12. Дан граф. Написать функцию, определеющую является ли он деревом. Деревом называется связный граф без циклов.

13. Построить каркас заданного графа алгоритмом Краскала (1956). Реализовать решение в виде функции. Алгоритм заключается в следующем. Последовательно в каркас включаются ребра с наименьшей длиной так, чтобы получившаяся часть каркаса не содержала циклов. Для проверки наличия цикла можно использовать алгоритм Дейкстры: если рассматривается ребро $<a, b>$, то ищем путь в построенной части каркаса из узла a в узел b. Если путь будет найден, добавление ребра приведет к образованию цикла, т.е. это ребро в каркас не включаем. Если же цикла не будет, то ребро включается в каркас.

Литература

1. Крячков, А.В. Программирование на С и С++. Практикум [Текст] / А.В.Крячков, И.В.Сухинина, В.К.Томшин. – Москва: Горячая линия-Телеком, 2000. – 344 с.

2. Абрамов, С.А. Задачи по программированию [Текст] / С.А.Абрамов, Г.Г.Гнездилова, Е.Н.Капустина, М.И.Селюн. – Москва: Наука, 1988. – 224 с.

3. Пильщиков, В.Н. Сборник упражнений по языку Паскаль [Текст] / В.Н.Пильщиков. – Москва: Наука, 1989. – 160 с.

4. Павловская, Т.А. С/С++. Структурное программирование: Практикум [Текст] / Т.А. Павловская, Ю.А. Щупак. – СПб: Питер, 2002. – 240 с.

5. Шилдт, Г. Справочник программиста по С/С++ [Текст]: пер. с англ. / Герберт Шилдт. – Москва: Издательский дом “Вильямс”, 2001. – 448 с.

6. Новиков Ф.А. Дискретная математика для программистов: Учебник для вузов. 3-е изд. [Текст] / Ф.А.Новиков. – СПб: Питер, 2008. – 384 с.

7. Вирт, Н. Алгоритмы и структуры данных [Текст]: пер. с англ. / Никлаус Вирт. – СПб: Невский Диалект, 2008. – 352 с.