Министерство образования Республики Беларусь

Учреждение образования

«Белорусский государственный университет
информатики и радиоэлектроники»

Кафедра «Вычислительные методы и программирование»

Основы программирования в среде *C++ Builder*

Лабораторный практикум по курсу
«Основы алгоритмизации и программирования»
для студентов 1 – 2-го курсов БГУИР

В 2-х частях

Часть 2

Минск БГУИР 2009

УДК 681.3.061 (075.8)
ББК 32.973.26-018.1 я73
О – 75

Рецензенты:

профессор кафедры систем управления Военной Академии Республики Беларусь, доктор технических наук, профессор В.А. Куренев,

заведующий кафедрой вычислительной техники Белорусского государственного аграрного технического университета, кандидат технических наук, доцент Ю.Н.Силкович

Авторы:

В. Л. Бусько, А. Г. Корбит, Т. М. Кривоносова, А. А. Навроцкий, Д.Л. Шилин

О-75 **Основы программирования в среде *C++ Builder*:** лаб. практикум по курсу «Основы алгоритмизации и программирования» для студ. 1 – 2-го курсов БГУИР. В 2 ч. Ч. 2 / Бусько В. Л. [и др.] . – Минск : БГУИР, 2009. – 61 с.: ил.

ISBN (ч. 2)

Приведены краткие теоретические сведения по алгоритмам обработки динамических структур данных (линейные и нелинейные списки), алгоритмам сортировки и поиска, а также некоторым методам приближенных вычислений; примеры их реализации на языке *C++* в среде *Builder*, 9 лабораторных работ и индивидуальные задания к ним.

УДК 681.3.061 (075.8)
ББК 32.973.26-018.1 я73

ISBN (ч. 2)
ISBN 985–444–583–6

СОДЕРЖАНИЕ

Лабораторная работа №1. Рекурсивные функции

Цель работы: изучить способы реализации алгоритмов с использованием рекурсии.

1.1. Краткие теоретические сведения

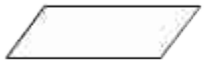
Рекурсия – это способ организации вычислительного процесса, при котором функция в ходе выполнения входящих в нее операторов обращается сама к себе. Классическим примером является вычисление факториала *n*! (*n* > 0):

```
double Faktorial_R (int n)  {
        if (n < 2) return 1;                    // Условие окончания рекурсии
        else
                return n* Faktorial_R (n–1);    // Рекурсивное обращение к функции
}
```

При выполнении правильно организованной рекурсивной функции осуществляется последовательный переход от текущего уровня организации алгоритма к нижнему уровню, в котором будет получено *нерекурсивное* решение задачи (в приведенном примере при *n* < 2), т.е. не требующее дальнейшего обращения к функции.

При описании алгоритмов используем следующие стандартные фигуры блок-схем:


- начало функции (заголовок) и конец функции (выход и возвращение в точку вызова);


- ввод-вывод данных;


- вычислительный процесс (операция присваивания);


- блок сравнения (операторы *if*, *switch-case*);


- заголовок оператора цикла;


- обращение к функции пользователя.

1.2. Пример выполнения задания

Написать программу вычисления факториала ***положительного*** числа ***n***, содержащую функции пользователя с рекурсией и без рекурсии.

1.2.1. Реализация задания в оконном приложении

Вид формы и полученные результаты представлены на рис. 1.1. Компонента *Edit*1 используется для ввода *n*, а компоненты *Edit*2 и *Edit*3 – для вывода результатов.

Листинг программы может иметь следующий вид:

Блок-схема функции-обработчика *Button1Click* представлена на рис. 1.2.

```
        . . .
    double Faktorial(int);
    double Faktorial_R(int);
//-------------------- Кнопка START ----------------------------------------
void __fastcall TForm1::Button1Click(TObject *Sender)
{
        int n = StrToInt(Edit1->Text);
        switch(RadioGroup1->ItemIndex) {
            case 0:
                Edit2->Text = FloatToStrF(Faktorial_R(n), ffFixed, 8, 1);
            break;
            case 1:
                Edit3->Text = FloatToStrF(Faktorial(n), ffFixed, 8, 1);
            break;
        }
}
//------------------ Функция без рекурсии --------------------------------------
double Faktorial(int n)
{
      double f = 1;
          for (int i = 1; i <= n; i++) f  *=  i;
          return f;
}
//------------------- Рекурсивная функция ---------------------------------------
double Faktorial_R(int n)
{
        if (n < 2) return 1;
        else
          return n*Faktorial_R(n-1);
}
```

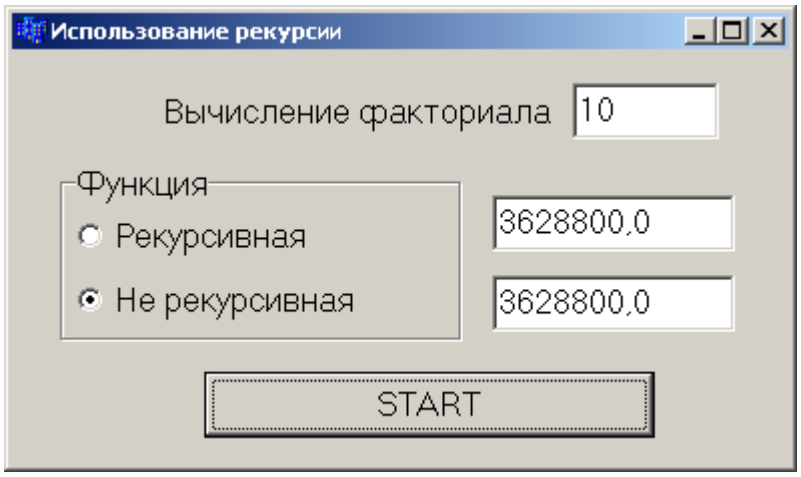


Рис. 1.1

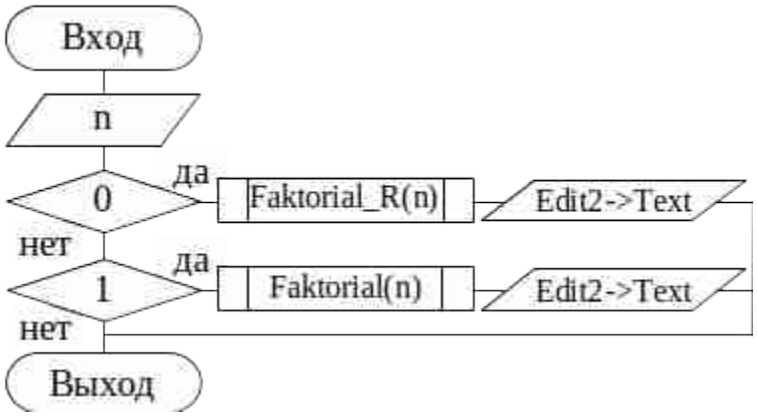


Рис. 1.2

Блок-схемы функций пользователя *Faktorial_R* и *Faktorial* представлены на рис. 1.3.

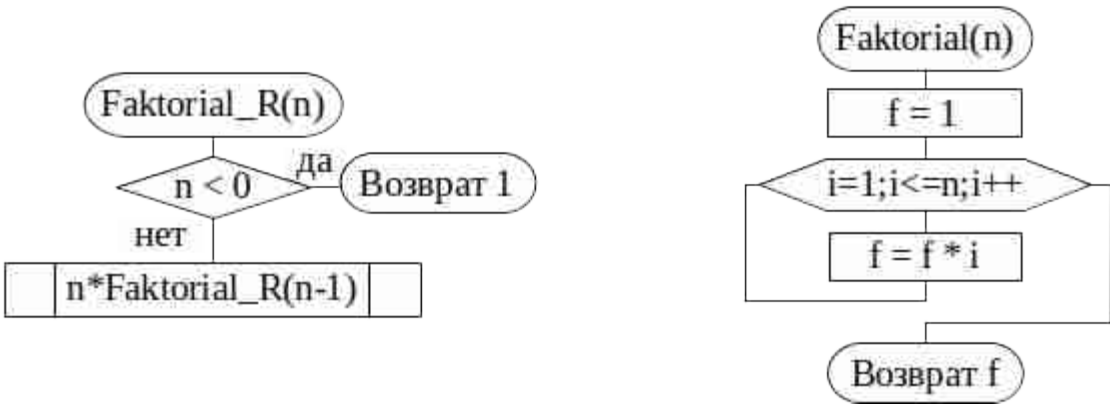


Рис. 1.3

1.2.2. Реализация задания в консольном приложении

Тексты функций пользователя смотрите в предыдущем примере, а листинг основной функции может иметь следующий вид:

```
…
#include <iostream.h>
#include <iomanip.h>                    // Для использования setprecision(n)
…
double Faktorial(int);
double Faktorial_R(int);
void  main(void)
{
        int n, kod;
        while(true) {                   // Бесконечный цикл
                cout << "\n Recurs - 0\n Simple - 1\n Else - Exit\t";
                cin >> kod;
                if (kod <0 || kod > 1) return;
```

```
            cout << "\tInput n " ;
            cin >> n;
            switch(kod) {
                  case 0:
               cout << setprecision(10) << "\tRecurs = " << Faktorial_R(n) << endl;
          break;
          case 1:
               cout << setprecision(10) << "\tSimple = " << Faktorial(n) << endl;
          break;
          }
     }
}
```

Результаты выполнения программы представлены на рис. 1.4:

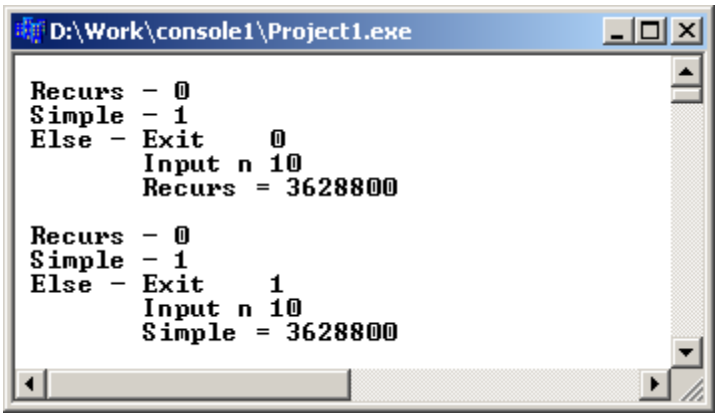


Рис. 1.4

1.3. Индивидуальные задания

Составить алгоритм в виде блок-схемы, написать и отладить поставленную задачу с использованием рекурсивной и обычной функций. Сравнить полученные результаты.

1. Для заданного целого десятичного числа N получить его представление в p-ичной системе счисления ($p < 10$).

2. В упорядоченном массиве целых чисел a_i ($i = 1, ..., n$) найти номер находящегося в массиве элемента c, используя метод двоичного поиска.

3. Найти наибольший общий делитель чисел M и N, используя теорему Эйлера: если M делится на N, то *НОД* (N,M) = N, иначе *НОД* (N,M) = (M%N, N).

4. Числа Фибоначчи определяются следующим образом: $Fb(0) = 0$; $Fb(1) = 1$; $Fb(n) = Fb(n–1) + Fb(n–2)$. Определить $Fb(n)$.

5. Найти значение функции Аккермана $A(m, n)$, которая определяется для всех неотрицательных целых аргументов m и n следующим образом:

$A(0, n) = n + 1$;
$A(m, 0) = A(m–1, 1)$; при $m > 0$;
$A(m, n) = A(m–1, A(m, n–1))$; при $m > 0$ и $n > 0$.

6. Найти методом деления отрезка пополам минимум функции $f(x) = 7\sin^2(x)$ на отрезке [2, 6] с заданной точностью ε (например 0,01).

7. Вычислить значение $x = \sqrt{a}$, используя рекуррентную формулу $x_n = \frac{1}{2}\left(x_{n-1} + \frac{a}{x_{n-1}}\right)$, в качестве начального значения использовать $x_0 = 0{,}5 \cdot (1 + a)$.

8. Найти максимальный элемент в массиве a_i (i=1, ⋯, n), используя очевидное соотношение $max(a_1, \cdots, a_n) = max[max(a_1, \cdots, a_{n-1}), a_n]$.

9. Вычислить значение $y(n) = \sqrt{1 + \sqrt{2 + \ldots + \sqrt{n}}}$.

10. Найти максимальный элемент в массиве a_i (i=1, ⋯, n), используя соотношение (деления пополам) $max(a_1, \cdots, a_n) = max[max(a_1, \cdots, a_{n/2}), max(a_{n/2+1}, \cdots, a_n)]$.

11. Вычислить значение $y(n) = \cfrac{1}{n + \cfrac{1}{(n-1) + \cfrac{1}{(n-2) + \cfrac{1}{\cdots \\ \cdots + \cfrac{1}{1 + \frac{1}{2}}}}}}$.

12. Вычислить произведение четного количества n ($n \geq 2$) сомножителей следующего вида:

$$y = \left(\frac{2}{1} \cdot \frac{2}{3}\right) \cdot \left(\frac{4}{3} \cdot \frac{4}{5}\right) \cdot \left(\frac{6}{5} \cdot \frac{6}{7}\right) \cdot \cdots .$$

13. Вычислить $y = x^n$ по следующему правилу: $y = (x^{n/2})^2$, если n четное и $y = x \cdot y^{n-1}$, если n нечетное.

14. Вычислить значение $C_n^k = \frac{n!}{k!(n-k)!}$ (значение 0! = 1).

15. Вычислить $y(n) = \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{\cdots}}}}$, n задает число ступеней.

16. В заданном массиве заменить все числа, граничащие с цифрой «1», нулями.

Лабораторная работа №2. Динамическая структура СТЕК

Цель работы: изучить алгоритмы работы с динамическими структурами данных в виде стека.

2.1. Краткие теоретические сведения

Стек – структура типа ***LIFO*** (*Last In, First Out*) – последним вошел, первым выйдет. Элементы в стек можно добавлять или извлекать только через его вершину. Программно стек реализуется в виде однонаправленного списка с одной точкой входа – вершиной стека.

Максимальное число элементов стека не ограничивается, т.е. по мере добавления в стек нового элемента память под него должна запрашиваться, а при удалении – освобождаться. Таким образом, стек – динамическая структура данных, состоящая из ***переменного*** числа элементов.

Для работы со стеком введем следующую структуру (вместо приведенного типа *Stack* может быть любой другой идентификатор):

```
struct Stack {
        int info;       // Информационная часть элемента, например int
        Stack *next;    // Адресная часть – указатель на следующий элемент
} *begin;               // Указатель вершины стека
```

При работе со стеком обычно выполняются следующие операции*:

– формирование стека (добавление элемента в стек);

– обработка элементов стека (просмотр, поиск, удаление);

– освобождение памяти, занятой стеком.

Приведем примеры основных функций для работы со стеком, взяв для простоты в качестве информационной части целые числа, т.е. объявленную выше структуру типа *Stack*.

Функция формирования элемента стека

Простейший вид функции (*push*), в которую в качестве параметров передаются указатель на вершину и введенная информация, а измененное значение вершины возвращается в точку вызова оператором *return*:

```
Stack* InStack(Stack *p, int in) {
   Stack *t = new Stack;            // Захватываем память для элемента
       t -> info = in;              // Формируем информационную часть
       t -> next = p;               // Формируем адресную часть
   return t;
}
```

Обращение к этой функции для добавления нового элемента «*a*» в стек, вершиной которого является указатель *begin*: begin = InStack(begin, a);

---

* Стандартными операциями при работе со стеками являются – добавление элемента в стек *push* (), извлечение – *pop* () и чтение информации из вершины без извлечения – *peek*().

Алгоритм просмотра стека (без извлечения его элементов, т.е. без сдвига вершины)

1. Устанавливаем текущий указатель на начало списка: *t = begin*;
2. Начинаем цикл, работающий до тех пор, пока указатель *t* не равен *NULL* (признак окончания списка).
3. Выводим информационную часть текущего элемента *t -> info* на экран.
4. Текущий указатель переставляем на следующий элемент, адрес которого находится в поле *next* текущего элемента: *t = t -> next*;
5. Конец цикла.

Функция, реализующая рассмотренный алгоритм:

```
void View(Stack *p) {
        Stack *t = p;
        while( t != NULL) {
// Вывод на экран информационной части, например, cout <<  t -> info << endl;
                t = t -> Next;
        }
}
```

Обращение к этой функции: View(begin);

Блок-схема функции *View* представлена на рис. 2.1.

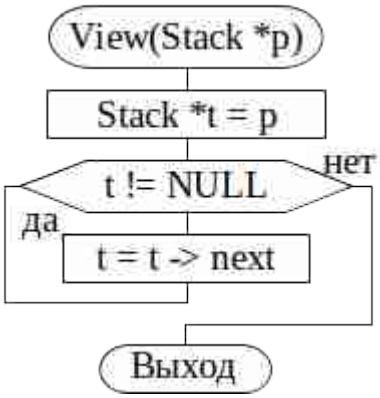


Рис. 2.1

Функция получения информации из вершины стека с извлечением:

```
Stack* OutStack(Stack* p, int *out)  {
        Stack *t = p;                   // Устанавливаем указатель t на вершину p
        *out = p -> info;
        p = p -> next;                  // Переставляем вершину p на следующий
        delete t;                       // Удаляем бывшую вершину t
        return p;                       // Возвращаем новую вершину p
}
```

Обращение к этой функции: begin = OutStack(begin, &a); информацией является переданное по адресу значение «*a*».

Функция освобождения памяти, занятой стеком:

```
void Del_All(Stack **p)   {
        Stack *t;
        while( *p != NULL) {
                t = *p;
                *p = (*p) -> Next;
                delete t;
        }
}
```

Обращение к этой функции: Del_All(&begin); после ее выполнения указатель на вершину *begin* будет равен *NULL*.

Сортировка однонаправленных списков

Для ускорения поиска информации в списке обычно при выводе данных список упорядочивают (сортируют) по ключу.

Проще всего использовать метод сортировки, основанный на перестановке местами двух соседних элементов, если это необходимо. Существует два способа перестановки элементов: обмен адресами и обмен информацией.

1. Первый способ – перестановка адресов двух соседних элементов, следующих за элементом с известным указателем. Первый элемент стека в этом случае ***не сортируется***. Для того чтобы и первый элемент оказался отсортированным, следует перед обращением к функции сортировки добавить один (любой) элемент в стек, а после сортировки – удалить его.

Функция сортировки для этого случая имеет вид

```
void Sort_p(Stack **p) {
        Stack *t = NULL, *t1, *r;
        if ((*p) -> next -> next == NULL) return;
        do {
                for (t1=*p; t1-> next->next  != t; t1=t1-> next)
                        if (t1->next->info  >  t1-> next-> next-> info){
                                r = t1->next->next;
                                t1 -> next -> next = r -> next;
                                r-> next =t1-> next;
                                t1-> next = r;
                        }
                t= t1-> next;
        } while ((*p)-> next -> next != t);
}
```

Обращение к этой функции Sort_p(&begin);

2. Второй способ – обмен информацией между текущим и следующим элементами. Функция сортировки для этого случая будет иметь вид

```
void Sort_info(Stack *p) {
        Stack *t = NULL, *t1;
        int r;
        do {
                for (t1=p; t1 -> next != t; t1 = t1-> next)
```

```
                    if (t1-> info > t1-> next -> info)  {
                        r = t1-> info;
                        t1-> info = t1-> next -> info;
                        t1-> next -> info = r;
                    }
                t = t1;
            } while (p -> next != t);
        }
```

2.2. Пример выполнения задания

Написать программу, содержащую основные функции обработки однонаправленного списка, организованного в виде стека, информационная часть которого представляет собой случайные целые числа от 0 до 20.

2.2.1. Реализация задания в оконном приложении

Вид формы и полученные результаты представлены на рис. 2.2.

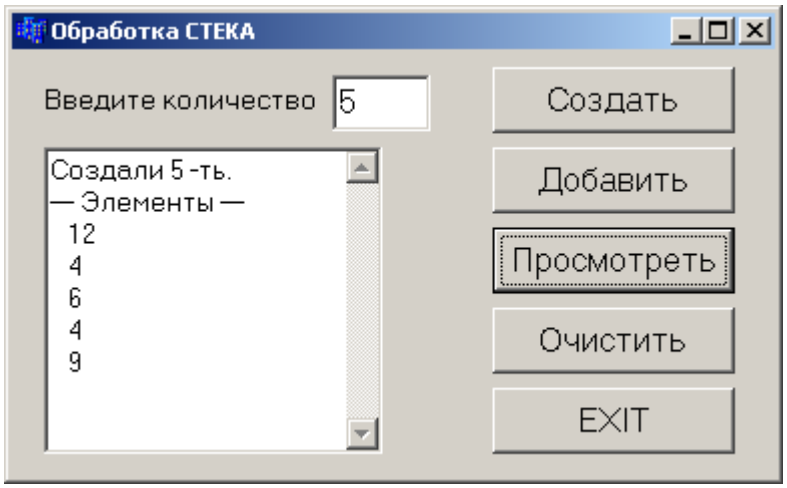


Рис. 2.2

Приведем только тексты функций, соответствующих указанным кнопкам:

```
        . . .
    struct Stack {                          // Декларация структурного типа
        int info;
        Stack * next;
    } *begin, *t;
//---------------- Декларации прототипов функций пользователя --------------------
    Stack* InStack(Stack*, int);
    void View(Stack*);
    void Del_All(Stack **);
//---------------- Текст функции-обработчика кнопки Создать ----------------------
        int i, in, n = StrToInt(Edit1->Text);
        if(begin != NULL){
            ShowMessage("Освободите память!");
            return;
        }
```

```
        for(i = 1; i <= n; i++){
            in = random(20);
            begin = InStack(begin, in);
        }
        Memo1->Lines->Add("Создали " + IntToStr(n) + " -ть.");
//------------------ Текст функции-обработчика кнопки Добавить ---------------------
        int i, in, n = StrToInt(Edit1->Text);
        for(i = 1; i <= n; i++){
            in = random(20);
            begin = InStack(begin, in);
        }
        Memo1->Lines->Add("Добавили " + IntToStr(n) + " -ть.");
//------------------ Текст функции-обработчика кнопки Просмотреть ---------------
        if(!begin){
            ShowMessage("Стек Пуст!");
            return;
        }
        Memo1->Lines->Add("--- Элементы ---");
        View(begin);
//------------------- Текст функции-обработчика кнопки Очистить -------------------
        if (begin != NULL) Del_All(&begin);
        ShowMessage("Память освобождена!");
//------------------- Текст функции-обработчика кнопки EXIT -------------------------
        if(begin != NULL) Del_All(&begin);
        Close();
//---------------------- Функция добавления элемента в Стек ----------------------------
    Stack* InStack(Stack *p, int in) {
        Stack *t = new Stack;
        t -> info = in;
        t -> next = p;
        return t;
    }
//-------------------------- Функция просмотра Стека-------------------------------------
    void View(Stack *p) {
        Stack *t = p;
        while( t != NULL) {
            Form1->Memo1->Lines->Add("   " + IntToStr( t->info));
// В консольном приложении будет, например,     cout << "   " << t->info << endl;
            t = t -> next;
        }
    }
//-------------------------- Функция освобождения памяти ------------------------------
    void Del_All(Stack **p) {
        while(*p != NULL) {
            t = *p;
            *p = (*p) -> next;
            delete t;
        }
    }
```

2.2.2. Реализация задания в консольном приложении

Декларацию шаблона структуры, декларации прототипов функций пользователя и их тексты смотрите в предыдущем примере, а листинг основной функции может иметь следующий вид:

```
        …
void main()
{
        int i, in, n, kod;
        while(true){
          cout << "\n\tCreat - 1.\n\tAdd - 2.\n\tView - 3.\n\tDel - 4.\n\tEXIT – 0.  :  " ;
          cin >> kod;
          switch(kod) {
            case 1: case 2:
                        if(kod == 1 && begin != NULL){
// Если создаем новый стек, должны освободить память, занятую предыдущим
                                cout << "Clear Memory!" << endl;
                                break;
                        }
              cout << "Input kol = ";         cin >> n;
              for(i = 1; i <= n; i++)  {
                    in = random(20);
                    begin = InStack(begin, in);
              }
              if (kod == 1) cout << "Create " << n << endl;
              else cout << "Add " << n << endl;
            break;
            case 3:         if(!begin){
                                cout << "Stack Pyst!" << endl;
                                break;
                            }
              cout << "--- Stack ---" << endl;
              View(begin);
            break;
            case 4:
              Del_All(&begin);
              cout<<"Memory Free!"<<endl;
            break;
            case 0:
              if(begin != NULL)
                        Del_All(&begin);
              return;           // Выход – EXIT
        }
    }
}
```

Полученные результаты представлены на рис. 2.3.

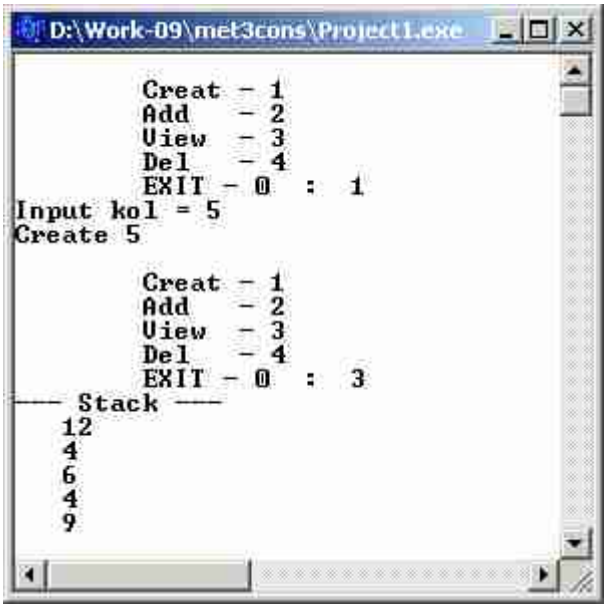


Рис. 2.3

2.3. Индивидуальные задания

Написать программу по созданию, добавлению, просмотру и решению поставленной задачи (в рассмотренных примерах это действие отсутствует) для однонаправленного линейного списка типа *СТЕК*. Реализовать сортировку стека двумя рассмотренными выше методами.

Решение поставленной задачи представить в виде блок-схемы.

Во всех заданиях создать список из положительных и отрицательных случайных целых чисел.

1. Разделить созданный список на два: в первом – положительные числа, во втором – отрицательные.

2. Удалить из созданного списка элементы с четными числами.

3. Удалить из созданного списка отрицательные элементы.

4. В созданном списке поменять местами крайние элементы.

5. Из созданного списка удалить элементы, заканчивающиеся на цифру 5.

6. В созданном списке поменять местами элементы, содержащие максимальное и минимальное значения.

7. Перенести из созданного списка в новый список все элементы, находящиеся между вершиной и максимальным элементом.

8. Перенести из созданного списка в новый список все элементы, находящиеся между вершиной и элементом с минимальным значением.

9. В созданном списке определить количество и удалить все элементы, находящиеся между минимальным и максимальным элементами.

10. В созданном списке определить количество элементов, имеющих значения, меньше среднего значения от всех элементов, и удалить эти элементы.

11. В созданном списке вычислить среднее арифметическое и заменить им первый элемент.

12. Созданный список разделить на два: в первый поместить четные, а во второй – нечетные числа.

13. В созданном списке определить максимальное значение и удалить его.

14. Из созданного списка удалить каждый второй элемент.

15. Из созданного списка удалить каждый нечетный элемент.

16. В созданном списке вычислить среднее арифметическое и заменить им все четные значения элементов.

Лабораторная работа №3. Динамическая структура ОЧЕРЕДЬ

Цель работы: изучить возможности работы со списками, организованными в виде очереди.

3.1. Краткие теоретические сведения

Очередь – линейный список, в котором извлечение данных происходит из начала, а добавление – в конец, т.е. это структура, организованная по принципу ***FIFO*** (*First In, First Out*) – первым вошел, первым выйдет.

При работе с очередью используют ***два указателя*** – на первый элемент (начало – ***begin***) и на последний (конец – ***end***). Очереди организуются в виде односвязных или двухсвязных списков, в зависимости от количества связей (указателей) в адресной части элемента структуры.

Односвязные списки

Шаблон элемента структуры, информационной частью которого является целое число (аналогично стеку), будет иметь следующий вид

```
struct Spis1 {
        int info;
        Spis1 *next;
} *begin, *end;                          // Указатели на начало и на конец
```

Основные операции с очередью следующие:
– формирование очереди;
– обработка очереди (просмотр, поиск, удаление);
– освобождение занятой памяти.

Формирование очереди состоит из двух этапов: создание первого элемента и добавление нового элемента в конец очереди.

Функция формирования очереди из данных объявленного выше типа *Spis*1 с добавлением новых элементов в конец может иметь следующий вид (*b* – начало очереди, *e* – конец):

```
void Create(Spis1 **b, Spis1 **e, int in)  {   // in – переданная информация
        Spis1 *t = new Spis1;
                t -> info = in;              // Формирование информационной части
                t -> next = NULL;            // Формирование адресной части
                if(*b == NULL)               // Формирование первого элемента
                        *b = *e = t;
                else {                       // Добавление элемента в конец
                        (*e) -> next = t;
                        *e = t;
                }
}
```

Обращение к данной функции Create(&begin, &end, in);

Алгоритмы просмотра и освобождения памяти выполняются аналогично рассмотренным ранее для Стека (см. лаб. работу № 2).

Двухсвязные списки

Двухсвязным (двунаправленным) является список, в адресную часть которого кроме указателя на следующий элемент включен и указатель на предыдущий.

Зададим структуру, в которой адресная часть состоит из указателей на предыдущий (*prev*) и следующий (*next*) элементы:

```
struct Spis2    {
                int info;
                Spis2  *prev, *next;
} *begin, *end;
```

Формирование двунаправленного списка проводится в два этапа – формирование первого элемента и добавление нового, причем добавление может выполняться как в начало (*begin*), так и в конец (*end*) списка.

Создание первого элемента

1. Захватываем память под элемент: Spis2 *t = new Spis2;
2. Формируем информационную часть (t -> info = StrToInt(Edit1->Text));
3. Формируем адресные части: t -> next = t -> prev = NULL;
4. Устанавливаем указатели начала и конца списка на первый элемент:
   begin = end = t;

Добавление элемента

Добавить в список новый элемент можно как в начало, так и в конец. Захват памяти и формирование информационной части выполняются аналогично предыдущему алгоритму (пп. 1 – 2).

Если элемент добавляется в начало списка, то выполняется следующая последовательность действий:

```
t -> prev = NULL;       // Предыдущего нет
t -> next = begin;      // Связываем новый элемент с первым
begin -> prev = t;      // Изменяем адрес prev бывшего первого
begin = t;              // Переставляем указатель begin на новый
```

В конец элемент добавляется следующим образом:

```
t -> next = NULL;       // Следующего нет
t -> prev = end;        // Связываем новый с последним
end -> next = t;        // Изменяем адрес next бывшего последнего
end = t;                // Изменяем указатель end
```

Просмотр списка

Просмотр списка можно выполнять с начала, или с конца списка. Просмотр с начала выполняется так же, как для однонаправленного списка (в функции *View*() лаб. работы № 2 необходимо изменить структурный тип).

Просмотр списка с конца

1. Устанавливаем текущий указатель на конец списка: *t = end*;

2. Начало цикла, работающего до тех пор, пока *t* != *NULL*.

3. Информационную часть текущего элемента *t -> info* выводим на экран.

4. Переставляем текущий указатель на предыдущий элемент, адрес которого находится в поле *prev* текущего элемента: *t = t -> prev*;

5. Конец цикла.

Алгоритм удаления элемента в списке по ключу

Удалить из списка элемент, информационная часть (ключ) которого совпадает со значением, введенным с клавиатуры.

Решение данной задачи проводим в два этапа – поиск и удаление.

Первый этап – поиск

Алгоритм поиска аналогичен просмотру списка с начала. Введем дополнительный указатель и присвоим ему значение *NULL*: Spis2 *key = NULL. Введем с клавиатуры искомое значение *i_p* (ключ поиска).

1. Установим текущий указатель на начало списка: t = begin;

2. Начало цикла (выполнять пока t != NULL).

3. Сравниваем информационную часть текущего элемента с искомым, если они совпадают (t -> info = i_p), то запоминаем адрес найденного элемента: key = t; (если ключ поиска *уникален*, то завершаем поиск – *break*).

4. Переставляем текущий указатель на следующий элемент: t = t -> next;

5. Конец цикла.

Выполняем контроль, если key = NULL , т.е. искомый элемент не найден, то сообщаем о неудаче и этап удаления не выполняем (*return*).

Второй этап – удаление

Если элемент найден (key != NULL), то выполняем удаление элемента из списка в зависимости от его местонахождения.

1. Если удаляемый элемент находится в начале списка, т.е. *key* равен *begin*, то первым элементом списка становится второй элемент:

а) указатель начала списка переставляем на следующий (второй) элемент:

begin = begin -> next;

б) адресу *prev* присваиваем значение *NULL*, т.е. теперь предыдущего нет

begin -> prev = NULL;

2. Если удаляемый элемент в конце списка, т.е. *key* равен *end*, то последним элементом в списке должен стать предпоследний:

а) указатель конца списка переставляем на предыдущий элемент:

end = end -> prev;

б) обнуляем адрес *next* нового последнего элемента

end -> next = NULL;

3. Если удаляемый элемент находится в середине списка, нужно обеспечить связь предыдущего и следующего элементов (см. рис. 3.1):

а) от *k*-го элемента с адресом *key* обратимся к предыдущему (*k*–1)-му элементу, адрес которого key->prev, и в его поле *next* [(key->prev)->next] запишем адрес (*k*+1)-го элемента, значение которого key->next:

(key -> prev) -> next = key -> next;

б) аналогично в поле *prev* (*k*+1)-го элемента с адресом key->next запишем адрес (*k*–1)-го элемента:

(key -> next) -> prev = key -> prev;

4. Освобождаем память, занятую удаленным элементом.

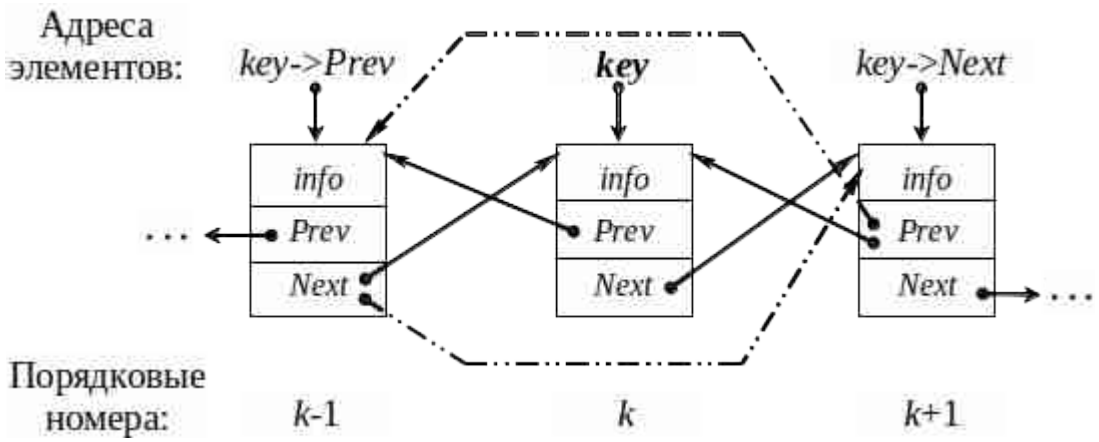


Рис. 3.1

Алгоритм освобождения памяти, занятой двунаправленным списком, аналогичен рассмотренному алгоритму для стека (см. лаб. работу № 2).

3.2. Пример выполнения задания

Написать программу, содержащую основные функции обработки двунаправленного списка, информационная часть которого представляет собой целые числа.

3.2.1. Реализация задания в оконном приложении

Вид формы и полученные результаты представлены на рис. 3.2.

Приведем только тексты функций-обработчиков соответствующих кнопок:

```
        . . .
struct Spis2 {
       int info;
       Spis2 *next, *prev;
 } *begin, *end, *t;
//---------------------------------------------------------------------------------------------------------------
void Create_Spis2(Spis2**, Spis2**, int);
void Add_Spis2(int, Spis2**, Spis2**, int);
void View_Spis2(int, Spis2*);
void Del_All(Spis2**);
```

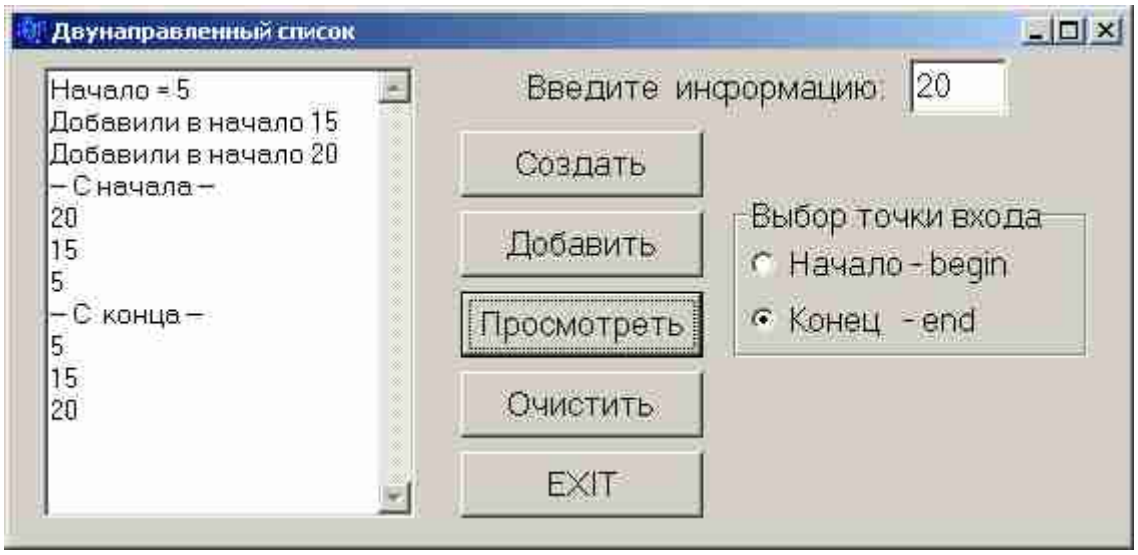


Рис. 3.2

```
//------------------- Текст функции-обработчика кнопки Создать ----------------------
        int i, in = StrToInt(Edit1->Text);
        if(begin != NULL){
                ShowMessage("Освободите Память!");
                return;
        }
        Create_Spis2(&begin, &end, in);
        Memo1->Lines->Add("Начало = " + IntToStr(begin -> info));
//------------------- Текст функции-обработчика кнопки Добавить ---------------------
        int i, in = StrToInt(Edit1->Text), kod = RadioGroup1->ItemIndex;
        String Str[2] = {"в начало ", "в конец "};
        Add_Spis2(kod, &begin, &end, in);
        if(kod == 0) t = begin;
        else t = end;
        Memo1->Lines->Add("Добавили " + Str[kod] + IntToStr(t -> info));
//------------------- Текст функции-обработчика кнопки Просмотреть -------------
        if(!begin){
                ShowMessage("Список Пуст!");
                return;
        }
        if(RadioGroup1->ItemIndex == 0) {
                t = begin;
                Form1->Memo1->Lines->Add("-- С начала --");
        }
        else {
                t = end;
                Form1->Memo1->Lines->Add("--- С  конца --");
        }
        View_Spis2(RadioGroup1->ItemIndex, t);
//--------------------- Текст функции-обработчика кнопки Очистить ----------------
        Del_All(&begin);
        ShowMessage("Память освобождена!");
```

```
//---------------------- Текст функции-обработчика кнопки EXIT ----------------------
        if(begin != NULL) Del_All(&begin);
        Close();
//------------------- Создание первого элемента ----------------------------------------
    void Create_Spis2(Spis2 **b, Spis2 **e, int in)  {
        t = new Spis2;
        t -> info = in;
        t -> next = t -> prev = NULL;
        *b = *e = t;
    }
//------------------- Добавление элемента в список -------------------------------------
    void Add_Spis2(int kod, Spis2 **b, Spis2 **e, int in)  {
        t = new Spis2;
        t -> info = in;
        if(kod == 0){
            t -> prev = NULL;
            t -> next = *b;
            (*b) -> prev = t;
            *b = t;
        }
        else {
            t -> next = NULL;
            t -> prev = *e;
            (*e) -> next = t;
            *e = t;
        }
    }
//---------------------- Просмотр элементов списка ----------------------------------------
    void View_Spis2(int kod, Spis2 *t)  {
        while(t != NULL) {
            Form1->Memo1->Lines->Add(t->info);
// В консольном приложении:        cout << t->info << endl;
            if(kod == 0) t = t->next;
            else  t = t->prev;
        }
    }
```

3.2.2. Реализация задания в консольном приложении

Декларацию шаблона структуры, декларации прототипов функций пользователя и их тексты смотрите в предыдущем примере, а основная функция может иметь следующий вид:

```
void main()
{
    int i, in, n, kod, kod1;
    char Str[2][10] = {"Begin ", "End "};
    while(true){
        cout << "\n\tCreat - 1\n\tAdd - 2\n\tView - 3\n\tDel - 4\n\tEXIT - 0  :  " ;
        cin >> kod;
        switch(kod) {
            case 1:     if(begin != NULL){
                            cout << "Clear Memory!" << endl;
```

```
                                        break;
                                }
                cout << "Begin Info = ";     cin >> in;
                Create_Spis2(&begin, &end, in);
                cout << "Creat Begin = " << begin -> info << endl;
    break;
    case 2:
                cout << "Info = ";        cin >> in;
                cout << "Add Begin - 0, Add End - 1 :  ";      cin >> kod1;
                Add_Spis2(kod1, &begin, &end, in);
                if(kod1 == 0) t = begin;
                else t = end;
                cout << "Add to " << Str[kod1] << "  " << t -> info << endl;
    break;
    case 3:     if(!begin){
                        cout << "Stack Pyst!" << endl;
                        break;
                }
                cout<<"View Begin-0,View End-1:";
                cin >> kod1;
                if(kod1 == 0) {
                        t = begin;
                        cout <<"-- Begin --" << endl;
                }
                else {
                        t = end;
                        cout <<"--- End --" << endl;
                }
                View_Spis2(kod1, t);
    break;
    case 4:
            Del_All(&begin);
            cout <<"Memory Free!"<< endl;
            break;
    case 0:  if(begin != NULL)
                Del_All(&begin);
             return;
    }
  }
}
```

Полученные результаты представлены на рис. 3.3.

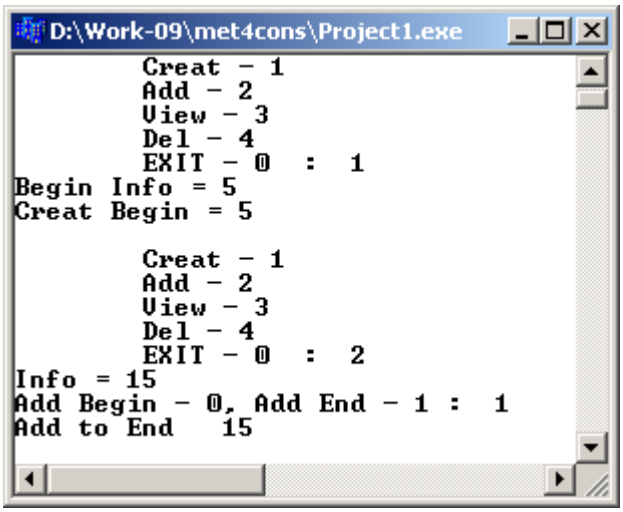


Рис. 3.3

3.3. Индивидуальные задания

Написать программу по созданию, добавлению (в начало, в конец), просмотру (с начала, с конца) и решению поставленной в лаб. работе № 2 задачи для двунаправленных линейных списков.

Лабораторная работа №4. Обратная польская запись

***Цель работы**:* изучить правила формирования постфиксной записи арифметических выражений с использованием стека.

4.1. Краткие теоретические сведения

Одной из задач при разработке трансляторов является задача расшифровки арифметических выражений.

Выражение *a+b* записано в ***инфиксной*** форме, ***+ab*** – в ***префиксной***, ***ab+*** – в ***постфиксной*** формах. В наиболее распространенной инфиксной форме для указания последовательности выполнения операций необходимо расставлять скобки. Польский математик Я. Лукашевич использовал тот факт, что при записи постфиксной формы скобки не нужны, а последовательность операндов и операций удобна для расшифровки.

Постфиксная запись выражений получила название ***обратной польской записи* (*ОПЗ*)**. Например, в ОПЗ выражение

$$r = (a + b) * (c + d) - e;$$

выглядит следующим образом:

$$r = ab + cd + * e - .$$

Алгоритм преобразования выражения из инфиксной формы в форму ОПЗ был предложен Э. Дейкстрой. При его реализации вводится понятие стекового приоритета операций.

Рассмотрим алгоритм получения ОПЗ из исходной строки символов, в которой записано выражение в ***инфиксной форме***.

1. Символы-операнды переписываются в выходную строку, в которой формируется постфиксная форма выражения.
2. Открывающая скобка записывается в стек.
3. Очередная операция выталкивает в выходную строку все операции из стека с большим или равным приоритетом.
4. Закрывающая скобка выталкивает все операции из стека до ближайшей открывающей скобки в выходную строку, открывающая скобка удаляется из стека, а закрывающая – игнорируется.
5. Если после просмотра последнего символа исходной строки в стеке остались операции, то все они выталкиваются в выходную строку.

4.2. Пример выполнения задания

Написать программу расшифровки и вычисления арифметических выражений с использованием стека.

Вид формы и полученные результаты представлены на рис. 4.1.

Приведем только тексты используемых функций-обработчиков и созданных функций пользователя (тексты функций *InStack* и *OutStack* взять в лаб. работе № 2, заменив тип *int* на *char*):

```
        . . .
struct Stack {
        char info;
        Stack *next;
} *begin;
int Prior (char);
Stack* InStack( Stack*,char);
Stack* OutStack( Stack*,char*);
double Rezult(String);
double mas[201];                                // Массив для вычисления
        Set  <char, 0, 255> znak;               // Множество символов-знаков
        int Kol = 8;
//---------------------- Текст функции-обработчика FormCreate -------------------------
          Edit1->Text = "a+b*(c-d)/e";              Edit2->Text = "";
          char a = 'a';
          StringGrid1->Cells[0][0] ="Имя"; StringGrid1->Cells[1][0] ="Знач.";
          for(int i = 1; i<Kol; i++) {
               StringGrid1->Cells[0][i] = a++;       StringGrid1->Cells[1][i] = i;
          }
//---------------------- Текст функции-обработчика кнопки Перевести ----------------
        Stack *t;
        begin = NULL;                           // Стек операций пуст
        char ss, a;
        String InStr, OutStr;                   // Входная и выходная строки
        OutStr = "";  Edit2->Text = "";
        InStr = Edit1->Text;
        znak << '*' << '/' << '+' << '-' << '^';
        int len = InStr.Length(), k;
        for (k = 1; k <= len; k++) {
                ss = InStr[k];
// Открывающую скобку записываем в стек
                if ( ss == '(' ) begin = InStack(begin, ss);
                if ( ss == ')' ) {
// Выталкиваем из стека все знаки операций до открывающей скобки
                    while ( (begin -> info) != '(' ) {
                        begin = OutStack(begin,&a);    // Считываем элемент из стека
                        OutStr += a;                   // Записываем в строку
                    }
                    begin = OutStack(begin,&a);        // Удаляем из стека скобку «(»
                }
// Букву (операнд) заносим в выходную строку
                if (ss >=  'a' && ss <= 'z' )  OutStr += ss;
/* Если знак операции, то переписываем из стека в выходную строку все опера-
ции с большим или равным приоритетом */
                if (znak.Contains(ss)) {
                    while ( begin != NULL && Prior (begin -> info) >= Prior (ss) ) {
                          begin = OutStack(begin, &a);
                          OutStr += a;
                    }
                    begin = InStack(begin, ss);
                }
```

```
        }
// Если стек не пуст, переписываем все операции в выходную строку
        while ( begin != NULL){
                begin = OutStack(begin, &a);
                OutStr += a;
         }
          Edit2->Text = OutStr;            // Выводим полученную строку
}
//---------------------- Текст функции-обработчика кнопки Посчитать ----------
        char ch;
        String OutStr = Edit2->Text;
        for (int i=1; i<Kol; i++) {
              ch = StringGrid1->Cells[0][i][1];
              mas[int(ch)]=StrToFloat(StringGrid1->Cells[1][i]);
        }
        Edit3->Text=FloatToStr(Rezult(OutStr));
//------------------ Функция реализации приоритета операций --------------------------
int Prior ( char a ){
        switch ( a ) {
                case '^':                   return 4;
                case '*':  case '/':        return 3;
                case '-':  case '+':        return 2;
                case '(':                   return 1;
        }
        return 0;
}
//--------------------- Расчет арифметического выражения -------------------------------
double Rezult(String Str) {
          char ch, ch1, ch2;
          double op1, op2, rez;
          znak << '*' << '/' << '+' << '-' << '^';
          char chr = 'z'+1;
          for (int i=1; i <= Str.Length(); i++){
                ch=Str[i];
                if (! znak.Contains(ch)) begin = InStack(begin, ch);
                else {
                        begin = OutStack(begin,&ch1);
                        begin = OutStack(begin,&ch2);
                        op1 = mas[int (ch1)];
                        op2 = mas[int (ch2)];
                        switch (ch){
                                case '+' :      rez=op2+op1;            break;
                                case '-' :      rez=op2-op1;            break;
                                case '*' :      rez=op2*op1;            break;
                                case '/' :      rez=op2/op1;            break;
                                case '^' :      rez=pow(op2,op1);       break;
                        }
                mas[int (chr)] = rez;
                begin = InStack(begin,chr);
                chr++;
                }
          }
          return rez;
```

}

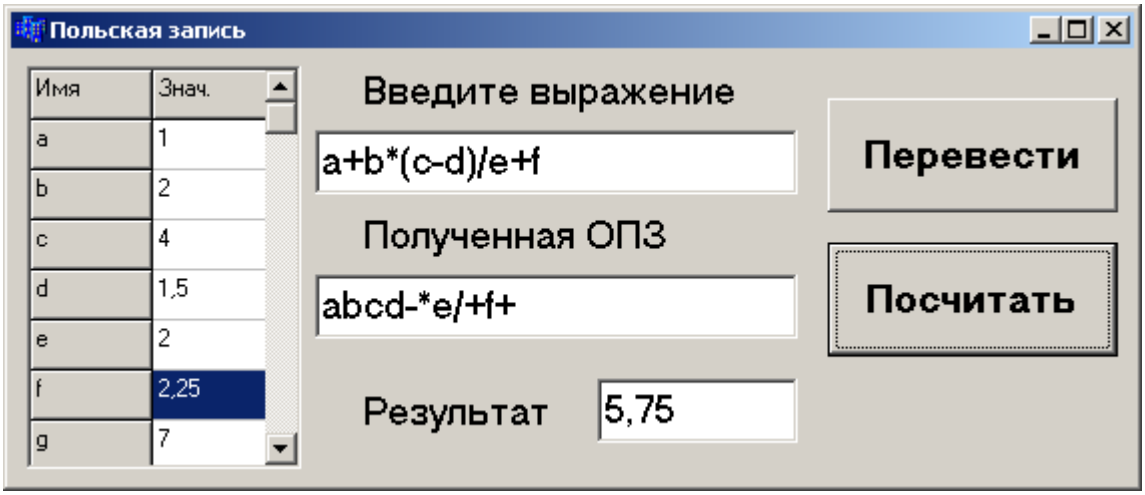


Рис. 4.1

4.3. Индивидуальные задания

Написать программу формирования ОПЗ и расчета полученного выражения. Разработать удобный интерфейс ввода исходных данных и вывода результатов. Работу программы проверить на конкретном примере (табл. 4.1).

Таблица 4.1

Выражение	*a*	*b*	*c*	*d*	*e*	Результат
1. *a*/(*b*– *c*)*(*d*+*e*)	8.6	2.4	5.1	0.3	7.9	– 26.12
2. (*a*+*b*)*(*c*– *d*)/*e*	7.4	3.6	2.8	9.5	0.9	– 81.89
3. *a*– (*b*+*c***d*)/*e*	3.1	5.4	0.2	9.6	7.8	2.16
4. *a*/*b*– ((*c*+*d*)***e*)	1.2	0.7	9.3	6.5	8.4	– 131.006
5. *a**(*b*– *c*+*d*)/*e*	9.7	8.2	3.6	4.1	0.5	168.78
6. (*a*+*b*)*(*c*– *d*)/*e*	0.8	4.1	7.9	6.2	3.5	2.38
7. *a**(*b*– *c*)/(*d*+*e*)	1.6	4.9	5.7	0.8	2.3	– 0.413
8. *a*/(*b**(*c*+*d*))– *e*	8.5	0.3	2.4	7.9	1.6	1.151
9. (*a*+(*b*/*c*– *d*))***e*	5.6	7.4	8.9	3.1	0.2	0.666
10. *a**(*b*+*c*)/(*d*– *e*)	0.4	2.3	6.7	5.8	9.1	– 1.091
11. *a*– (*b*/*c**(*d*+*e*))	5.6	3.2	0.9	1.7	4.8	– 17.51
12. (*a*– *b*)/(*c*+*d*)***e*	0.3	6.7	8.4	9.5	1.2	– 0.429
13. *a*/(*b*+*c*– *d***e*)	7.6	4.8	3.5	9.1	0.2	1.173
14. *a**(*b*– *c*)/(*d*+*e*)	0.5	6.1	8.9	2.4	7.3	– 0.144
15. (*a*+*b***c*)/(*d*– *e*)	9.1	0.6	2.4	3.7	8.5	– 2.196

16. $a - b/(c*(d - e))$	1.4	9.5	0.8	6.3	7.2	14.594

Лабораторная работа №5. Нелинейные списки

Цель работы: изучить алгоритмы обработки данных с использованием нелинейных структур в виде дерева.

5.1. Краткие теоретические сведения

Представление динамических данных в виде древовидных структур оказывается довольно удобным и эффективным для решения задач быстрого поиска информации.

Дерево состоит из элементов, называемых ***узлами*** (вершинами), которые соединены между собой направленными дугами (рис. 5.1). В случае $X \rightarrow Y$ вершина X называется ***предком*** (*родителем*), а Y – ***потомком*** (*сыном*, *дочерью*).

Дерево имеет единственный узел, не имеющий предков (ссылок на этот узел), который называется ***корнем***. Любой другой узел имеет ровно одного предка, т.е. на каждый узел дерева имеется ровно одна ссылка. Узел, не имеющий сыновей, называется ***листом*** (например узел *Y*).

Внутренний узел – это узел, не являющийся ни листом, ни корнем. ***Порядок узла*** равен количеству его узлов-сыновей. ***Степень дерева*** – максимальный порядок его узлов. ***Высота (глубина) узла*** равна числу его предков плюс один. ***Высота дерева*** – это наибольшая высота его узлов.

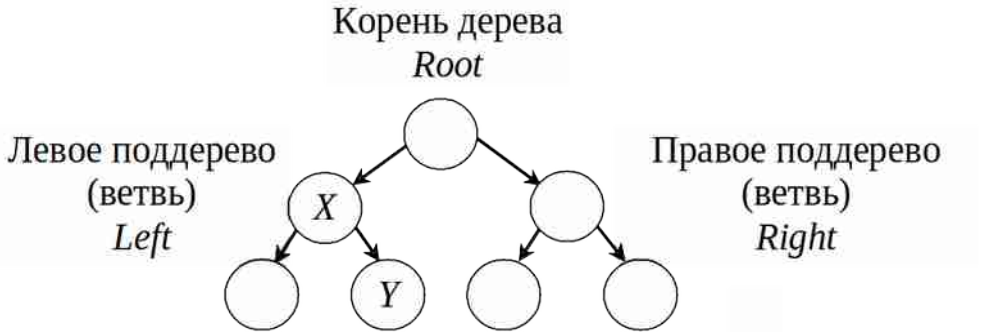


Рис. 5.1

Бинарное дерево поиска

Наиболее часто для работы со списками используются бинарные (имеющие степень 2) деревья (рис. 5.1).

В дереве поиска ключи расположены таким образом, что значения ключа у левого сына имеет значение меньшее, чем значение предка, а правого сына – большее.

Сбалансированными, или *AVL*-деревьями, называются деревья, для каждого узла которых высóты его поддеревьев различаются не более чем на 1.

Дерево по своей организации является рекурсивной структурой данных, поскольку каждое его поддерево также является деревом. В связи с этим действия с такими структурами чаще всего описываются с помощью рекурсивных алгоритмов.

При работе с бинарным деревом простейшего вида, т.е. ключами которого являются целые числа (*уникальные*, т.е. не повторяются), необходимо использовать структуру следующего вида:

```
struct Tree {
        int info;
        Tree *left, *right;
} *root;                                // root  - указатель корня
```

В общем случае при работе с деревьями необходимо уметь:

– сформировать дерево (добавить новый элемент);

– обойти все элементы дерева (например, для просмотра или выполнения некоторой операции);

– выполнить поиск элемента с указанным значением в узле;

– удалить заданный элемент.

Формирование дерева поиска состоит из двух этапов: создание корня, являющегося листом, и добавление нового элемента (листа) в найденное место. Для этого используется функция формирования листа:

```
Tree* List(int inf) {
        Tree *t = new Tree;             // Захват памяти
        t -> info = inf;                // Формирование информационной части
        t -> left = t -> right = NULL;  // Формирование адресных частей
        return t;                       // Возврат созданного указателя
}
```

1. Первоначально (root = NULL) создаем корень (***первый лист дерева***):

```
root = List (StrToInt(Edit1->Text));
```

2. Иначе (*root* != *NULL*) добавляем информацию (*key*) в нужное место:

```
void Add_List(Tree *root, int key)      {
        Tree *prev, *t;                 // prev – указатель предка нового листа
         bool find = true;
                t = root;
                while ( t && find) {
                        prev = t;
                        if( key == t->info) {
                                find = false;         // Ключ должен быть уникален
                                ShowMessage("Dublucate Key!");
                        }
                        else
                                if ( key < t -> info ) t = t -> left;
                                else   t = t -> right;
                }
                if (find) {                                 // Нашли нужное место
```

```
                t = List(key);                          // Создаем новый лист
                if ( key < prev -> info ) prev -> left = t;
                else    prev -> right = t;
        }
    }
```

Функция просмотра элементов дерева

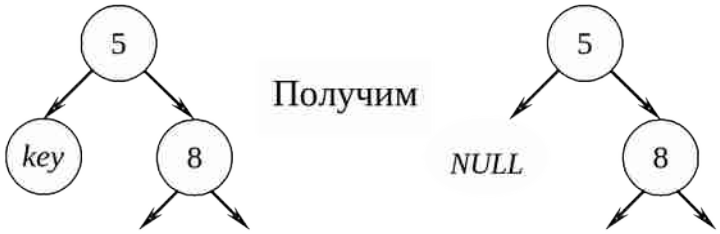
```
void View_Tree(Tree *p, int level ) {
    String str;
    if ( p )  {
        View_Tree (p -> right , level+1);               // Правое поддерево
        for ( int i=0; i<level; i++) str = str + "    ";
            Form1->Memo1->Lines->Add(str + IntToStr(p->info));
        View_Tree(p -> left , level+1);                 // Левое поддерево
      }
    }
```

Обращение к функции *View* будет иметь вид View(root, 0);

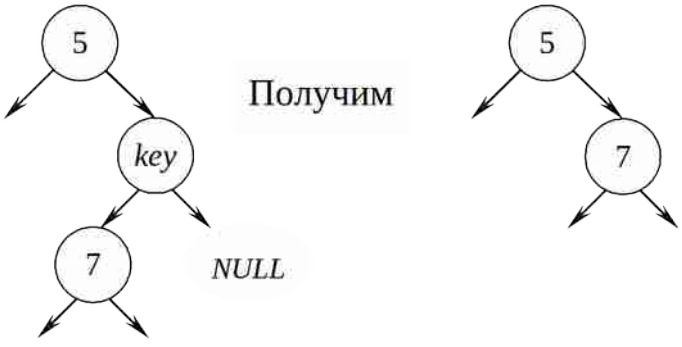
Вторым параметром функции является переменная, определяющая, на каком уровне (*level*) находится узел (у корня уровень «0»). Строка *str* используется для получения пробелов, необходимых для вывода значения на соответствующем уровне.

Удаление узла с заданным ключом из дерева поиска, сохраняя его свойства, выполняется в зависимости от того, сколько сыновей (потомков) имеет удаляемый узел.

1. Удаляемый узел является листом – просто удаляем ссылку на него. Приведем пример схемы удаления листа с ключом *key*:



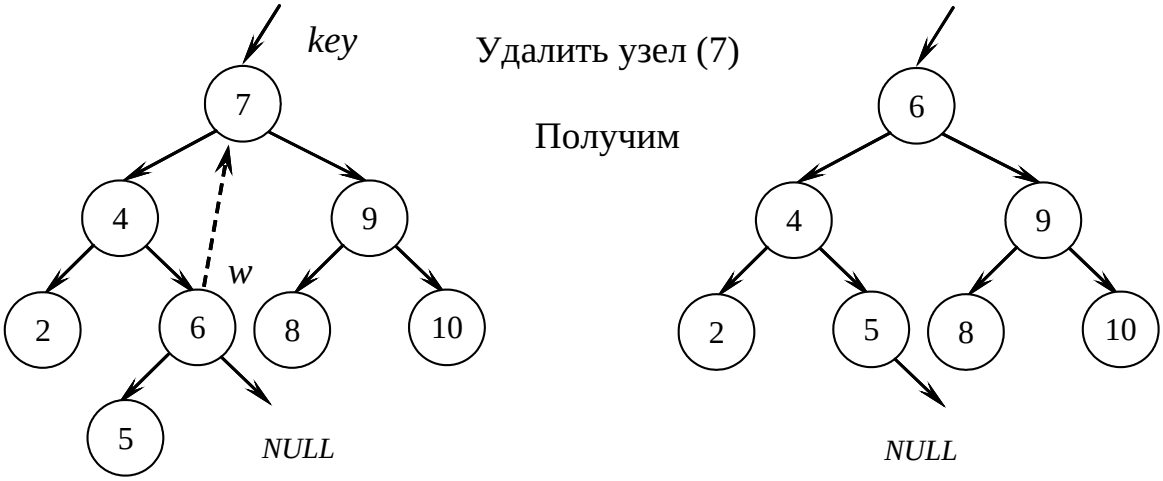
2. Удаляемый узел имеет только одного потомка, т.е. из удаляемого узла выходит ровно одна ветвь. Пример схемы удаления узла *key*, имеющего одного сына:



3. Удаление узла, имеющего двух потомков, значительно сложнее рассмотренных выше. Если *key* – удаляемый узел, то его следует заменить

узлом *w*, который содержит либо наибольший ключ (самый правый, у которого указатель *Right* равен *NULL*) в левом поддереве, либо наименьший ключ (самый левый, у которого указатель *Left* равен *NULL*) в правом поддереве.

Используя первое условие, находим узел *w*, который является самым правым узлом поддерева *key*, у него имеется только левый потомок:



Функция удаления узла по заданному ключу key может иметь вид

```
Tree* Del_Info(Tree *root, int key)  {
        Tree *Del, *Prev_Del, *R, *Prev_R;
// Del, Prev_Del – удаляемый узел и его предыдущий (предок);
// R, Prev_R – элемент, на который заменяется удаляемый, и его предок;
        Del = root;
        Prev_Del = NULL;
//------------ Поиск удаляемого элемента и его предка по ключу key ------------------
while (Del != NULL && Del -> info != key) {
        Prev_Del = Del;
        if (Del->info > key)  Del = Del->left;
                else Del = Del->right;
        }
        if (Del == NULL)  {                             // Элемент не найден
                ShowMessage ( "NOT Key!");
                return root;
        }
//-------------------------- Поиск элемента R для замены ---------------------------------
        if (Del -> right == NULL) R = Del->left;
        else
        if (Del -> left == NULL) R = Del->right;
        else {
//------------------- Ищем самый правый узел в левом поддереве ----------------------
                Prev_R = Del;
                R = Del->left;
                while (R->right != NULL) {
                        Prev_R = R;
                        R = R->right;
```

```
            }
//---------------- Формируем связи элемента R и его предка Prev_R ------------------
            if( Prev_R == Del)  R->right = Del->right;
            else {
                  R->right = Del->right;
                  Prev_R->right = R->left;
                  R->left = Prev_R;
            }
       }
      if (Del== root) root = R;                  // Удаляя корень, заменяем его на R
      else
//--------------- Поддерево R присоединяем к предку удаляемого узла ----------------
      if (Del->info < Prev_Del->info)
            Prev_Del->left = R;                        // На левую ветвь
      else  Prev_Del->right = R;                       // На правую ветвь
      delete Del;
      return root;
}
```

***Поиск узла с минимальным (максимальным) ключом**:*

```
Tree* Min_Key(Tree *p) {                    // Tree* Max_Key(Tree *p)
    while (p->left != NULL) p = p->left;    // p=p->right;
    return p;
  }
```

Тогда для получения минимального ключа p_min -> info:

```
Tree *p_min = Min_Key(root);
```

Построение сбалансированного дерева поиска для заданного (созданного) массива ключей «***a***» можно осуществить, если этот массив предварительно отсортирован в порядке возрастания ключа, с помощью следующей рекурсивной процедуры (при обращении $n = 0$, k – размер массива):

```
void Make_Blns(Tree **p, int n, int k, int *a) {
      if (n == k) { *p = NULL;
            return;
      }
      else {
            int m=(n+k)/2;
            *p = new Tree;
            (*p)->info = a[m];
            Make_Blns( &(*p)->left, n, m, a);
            Make_Blns( &(*p)->right, m+1, k, a);
      }
}
```

Функция освобождения памяти, занятой деревом

```
void Del_Tree(Tree *t) {
```

```
        if ( t != NULL)  {
        Del_Tree ( t -> left);                  // На левую ветвь
        Del_Tree ( t -> right);                 // На правую ветвь
        delete t;
        }
}
```

5.2. Пример выполнения задания

В качестве примера рассмотрим проект (для последовательно введенных ключей 10 (корень), 25, 20, 6, 21, 8, 1, 30), который создает дерево, отображает его в *Memo*, удаляет элемент по ключу и удаляет дерево. Панель диалога будет иметь вид, представленный на рис. 5.2.

Как и в предыдущих примерах, приведем только тексты функций-обработчиков соответствующих кнопок, а тексты функций пользователя рассмотрены выше:

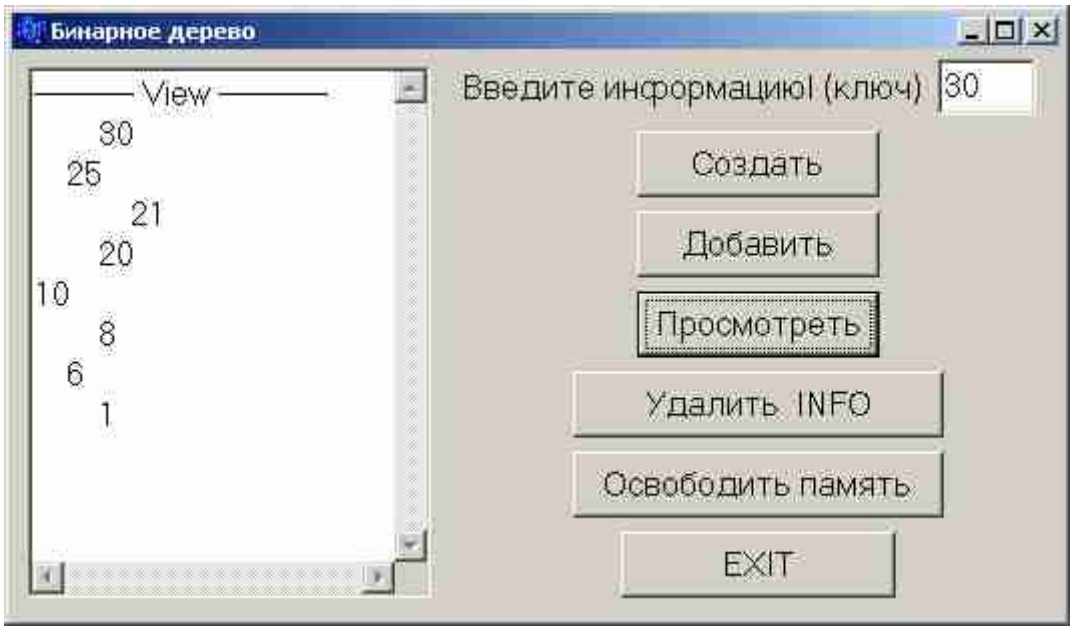


Рис. 5.2

```
//---------------------------- Шаблон структуры ---------------------------------------------
struct Tree {
                int info;
                Tree *left, *right;
}*root;                                                         // Корень
//----------------- Декларации прототипов функций работы с деревом ----------------
void Add_List(Tree*, int);
void View_Tree (Tree*, int);
Tree* Del_Info(Tree*, int);
```

```
void Del_Tree(Tree*);
Tree* List(int);
```
//---------- Текст функции-обработчика кнопки ***Создать*** ----------
```
     if(root != NULL) Del_Tree(root);
     root = List (StrToInt(Edit1->Text));
```
//---------- Текст функции-обработчика кнопки ***Просмотреть*** ----------
```
    if( root == NULL ) ShowMessage(" Create TREE !");
    else {
        Memo1->Lines->Add("---------- View ----------");
        View_Tree(root, 0);
    }
```
//---------- Текст функции-обработчика кнопки ***Добавить*** ----------
```
     if(root == NULL) root = List (StrToInt(Edit1->Text));
     else Add_List (root, StrToInt(Edit1->Text));
```
//---------- Текст функции-обработчика кнопки ***Удалить INFO*** ----------
```
              int b = StrToInt(Form1->Edit1->Text);
           root = Del_Info(root, b);
```
//---------- Текст функции-обработчика кнопки ***ОЧИСТИТЬ*** ----------
```
       Del_Tree(root);
                ShowMessage(" Tree Delete!");
       root = NULL;
```
//---------- Текст функции-обработчика кнопки ***EXIT*** ----------
```
    if(root!=NULL){
      Del_Tree(root);
      ShowMessage(" Tree Delete!");
    }
    Close();
```

5.3. Индивидуальные задания

Разработать проект для работы с деревом поиска, содержащий следующие обработчики, которые должны:

– ввести информацию (желательно, используя *StringGrid*), состоящую из целочисленного ключа и строки текста (например, номер паспорта и ФИО);

– записать информацию в дерево поиска;

– сбалансировать дерево поиска;

– добавить в дерево поиска новую запись;

– по заданному ключу найти информацию и отобразить ее;

– удалить из дерева поиска информацию с заданным ключом;

– распечатать информацию прямым, обратным обходом и в порядке возрастания ключа;

– решить одну из следующих задач.

Решение поставленной задачи оформить в виде блок-схемы.

1. Поменять местами информацию, содержащую максимальный и минимальный ключи.

2. Подсчитать число листьев в дереве. (Лист – это узел, из которого нет ссылок на другие узлы дерева).

3. Удалить из дерева ветвь, с вершиной, имеющей заданный ключ.

4. Определить максимальную глубину дерева, т.е. число узлов в самом длинном пути от корня дерева до листьев.

5. Определить число узлов на каждом уровне дерева.

6. Удалить из левой ветви дерева узел с максимальным значением ключа и все связанные с ним узлы.

7. Определить количество символов во всех строках дерева.

8. Определить число листьев на каждом уровне дерева.

9. Определить число узлов в дереве, в которых есть только один сын.

10. Определить число узлов в дереве, у которых есть две дочери.

11. Определить количество записей в дереве начинающихся с определенной буквы (например «*а*»).

12. Найти среднее значение всех ключей дерева и найти строку, имеющую ближайший к этому значению ключ.

13. Между максимальным и минимальным значениями ключей найти запись с ключом со значением, ближайшим к среднему значению.

14. Определить количество записей в левой ветви дерева.

15. Определить количество записей в правой ветви дерева.

16. Определить число листьев в левой ветви дерева.

Лабораторная работа №6. Алгоритмы поиска корней уравнений

Цель работы: изучить алгоритмы поиска корней нелинейных алгебраических уравнений с заданной точностью.

6.1. Краткие теоретические сведения

Одним из наиболее часто используемых вычислительных методов является ***метод итераций*** и различные его модификации.

Итерационные методы основаны на построении сходящейся к точному решению x^* бесконечной рекуррентной последовательности $x_0, x_1, \ldots, x_k \to x^*$ при $k \to \infty$.

Последовательность называется ***рекуррентной*** порядка m, если каждый следующий ее член выражается через m предыдущих по некоторому правилу:

$$x_k = \varphi(x_{k-1}, x_{k-2}, \ldots, x_{k-m}). \qquad (6.1)$$

Такой метод называется ***m-шаговым***. Для его реализации требуется задать m первых членов $\{x_0, x_1, \ldots, x_{m-1}\}$, называемых ***начальным приближением***. Зная начальное приближение, по формуле (6.1) последовательно находят $x_m, x_{m+1}, \ldots, x_k, \ldots$. Процесс получения следующего k-го члена через предыдущие называется k-й ***итерацией***. Итерации выполняются до тех пор, пока очередной член x_k не будет удовлетворять заданной точности, т.е. пока не выполнится условие $| x_k - x_{k-1} | < \varepsilon$, где ε – некоторая заданная малая величина. В качестве искомого решения берут последний член последовательности x_k, при котором выполнилось указанное неравенство.

Чтобы использовать итерационный метод, исходную задачу преобразуют к виду, разрешенному относительно x:

$$x = \varphi(x). \qquad (6.2)$$

При этом точное решение исходной задачи x^* является и решением (6.2).

Используем выражение (6.2) в качестве рекуррентной формулы ($m = 1$):

$$x_k = \varphi(x_{k-1}).$$

Далее, задав одно x_0 (начальное приближение), последовательно находим $x_1, x_2, \ldots, x_k$. Если полученная таким образом последовательность сходится к некоторому конечному пределу, то этот предел совпадает с точным решением x^*.

Математической моделью многих физических процессов является функциональная зависимость $y = f(x)$. Поэтому задачи исследования различных свойств функции $f(x)$ часто возникают в инженерных расчетах. Одной из таких задач является нахождение корней этого уравнения на заданном отрезке $[a, b]$, т.е. таких значений x, при которых $\boldsymbol{f(x) = 0}$.

На рис. 6.1 представлены три наиболее часто встречающиеся ситуации:

а) ***кратный*** корень: $f'(x_1^*) = 0, \quad f(\alpha_1) \cdot f(\beta_1) > 0;$

б) ***простой*** корень: $f'(x_2^*) \neq 0, \quad f(\alpha_2) \cdot f(\beta_2) < 0;$

в) ***вырожденный*** корень: $f'(x_3^*)$ не существует, $f(\alpha_3) \cdot f(\beta_3) > 0$.

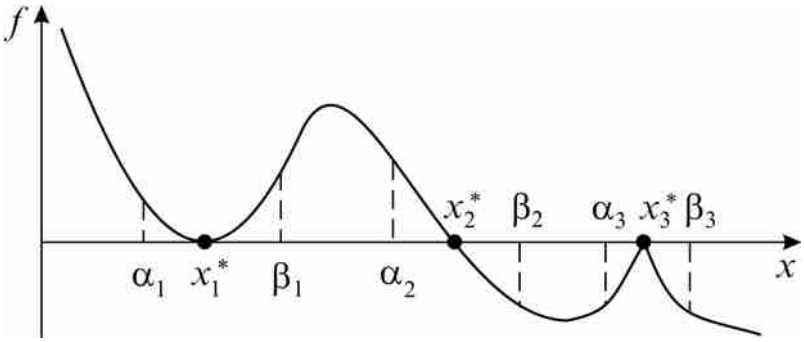


Рис. 6.1

Значения корней x_1^* и x_3^* (назовем их ***особенными***) совпадают с точкой экстремума функции, и для их нахождения можно использовать либо методы поиска минимума функции, либо алгоритм поиска интервала, на котором находится «особенный» корень.

Обычно для нахождения корней уравнения применяются численные методы, и этот поиск осуществляется в два этапа.

1. Приближенное определение местоположения – этап отделения корней (нахождение грубых корней).

2. Вычисление выбранного корня с заданной точностью ε.

Первая задача чаще всего решается графическим методом: на заданном отрезке [*a*, *b*] вычисляется таблица значений функции с некоторым шагом *h*, строится ее график, и определяются интервалы (α_i, β_i) – в дальнейшем [*a*, *b*] длиной *h*, на которых находятся корни.

Общий алгоритм поиска особенных корней

1. Начало цикла для *x*, изменяющегося от *a* до *b* с шагом *h* ($h \le \varepsilon$).

2. Если $f(x) < \varepsilon$, то *xn* – начало отрезка, на котором вероятно существует особенный корень уравнения $f(x)$ – продолжаем цикл.

3. Если $f(x) > \varepsilon$, то *xk* – конец искомого отрезка.

4. Выводим на экран полученный интервал.

5. Конец цикла.

Общий алгоритм поиска простых корней

1. Начало цикла для *x*, изменяющегося от *a* до *b* с шагом *h*.

2. Если $f(x) \cdot f(x + h) < 0$, то на отрезке [*x*, *x* + *h*] существует простой корень уравнения $f(x)$.

3. Обращаемся к созданной функции, реализующей выбранный алгоритм, параметрами которой являются: интервал [*x*, *x* + *h*], на котором существует корень, вид функции $f(x)$, заданная точность ε.

4. Выводим на экран полученное значение.

5. Конец цикла.

Поиск простого корня с заданной точностью ε осуществляется одним из итерационных методов. В соответствии с общей методикой на найденном

интервале выбирается m начальных значений (приближений) x_0, x_1, …, x_{m-1}, после чего последовательно выполняются вычисления до тех пор, пока не выполнится неравенство $|x1 - x0| < \varepsilon$ ($x0$ – значение, найденное на предыдущем шаге, $x1$ – найденное значение). Значение $x1$, при котором $|x1 - x0| > \varepsilon$, принимается в качестве приближенного значения корня.

Рассмотрим основные итерационные методы поиска корней.

Метод простой итерации

Функцию $f(x)$ записывают в виде разрешенном, относительно x:

$$x = \varphi(x). \qquad (6.1)$$

Переход от записи исходного уравнения к эквивалентной записи (6.1) можно сделать многими способами, например, положив

$$\varphi(x) = x + \psi(x) \cdot f(x), \qquad (6.2)$$

где $\psi(x)$ – произвольная, непрерывная, знакопостоянная функция (часто достаточно выбрать $\psi = const$).

Члены рекуррентной последовательности в методе простой итерации вычисляются по правилу

$$x_k = \varphi(x_{k-1}); \ k = 1,2, \ldots .$$

Метод является одношаговым, т.к. для начала вычислений достаточно знать одно начальное приближение $x_0 = a$, или $x_0 = b$, или $x_0 = (a + b)/2$.

Метод Ньютона (метод касательных)

Этот метод является модификацией метода простой итерации и часто называется методом касательных. Если $f(x)$ дважды непрерывно дифференцируемая функция и имеет непрерывную производную. Выбрав в (6.2) $\psi(x) = 1/f'(x)$, получаем эквивалентное уравнение $x = x - f(x)/ f'(x) = \varphi(x)$, тогда формула для метода Ньютона имеет вид

$$x_k = x_{k-1} - f(x_{k-1}) / f'(x_{k-1}) = \varphi(x_{k-1}), \qquad k = 1,2,\ldots .$$

Очевидно, что этот метод одношаговый ($m = 1$) и для начала вычислений требуется задать одно начальное приближение.

Метод секущих

Данный метод является модификацией метода Ньютона, позволяющей избавиться от явного вычисления производной путем ее замены приближенной формулой:

$$x_k = x_{k-1} - f(x_{k-1}) \cdot q / [\, f(x_{k-1}) - f(x_{k-1} - q)] = \varphi(x_{k-1}), \quad k = 1,2,\ldots .$$

Здесь q – некоторый малый параметр метода, который подбирается из условия наиболее точного вычисления производной ($q = h$).

Метод Вегстейна

Этот метод является модификацией предыдущего метода секущих. В нем при расчете приближенного значения производной используется вместо точки $x_{k-1} - q$ ранее полученная точка x_{k-2}. Расчетная формула метода Вегстейна

$$x_k = x_{k-1} - f(x_{k-1}) \cdot (x_{k-1} - x_{k-2}) / [\, f(x_{k-1}) - f(x_{k-2})] = \varphi(x_{k-1}, x_{k-2}), \qquad k = 1,2,\ldots .$$

Метод является двухшаговым ($m = 2$), и для начала вычислений требуется задать 2 начальных приближения $x_0 = a$, $x_1 = b$.

Функция, реализующая данный метод, может иметь вид

```
double Metod1(type_f f,double x0,double x1,double eps) {
        double y0,y1,x2,de;
          y0=f(x0);        y1=f(x1);
          do  {
                  x2=x1-y1*(x1-x0)/(y1-y0);
                  de=fabs(x1-x2);
                  x0=x1;       x1=x2;       y0=y1;       y1=f(x2);
          }  while (de>eps);
        return x2;    // Возвращаем значение, для которого достигнута точность
}
```

Метод деления отрезка пополам

Все вышеописанные методы могут работать, если функция $f(x)$ является непрерывной и дифференцируемой вблизи искомого корня. В противном случае они не гарантируют получение решения.

Для разрывных функций, а также, если не требуется быстрая сходимость, для нахождения простого корня на интервале $[a, b]$ применяют надежный метод деления отрезка пополам.

Идея метода: в качестве начального приближения выбираются границы интервала, на котором находится простой корень $x_0 = a$, $x_1 = b$; далее находится его середина $x_2 = (x_0 + x_1)/2$. Очередная точка выбирается как середина того из смежных с x_2 интервалов $[x_0, x_2]$ или $[x_2, x_1]$, на котором находится корень.

Функция, реализующая данный метод приведена в примере, а блок-схема приведена на рис 6.1.

6.2. Пример выполнения задания

Написать программу поиска ***простых*** корней функции $f(x) = 4x - 7\sin x$ на отрезке $[a, b]$ с шагом h и точностью ε методом деления отрезка пополам.

Вид формы и полученные результаты представлены на рис. 6.2.

Текст программы *Unit*1.*cpp* может иметь следующий вид:

```
typedef double (*type_f)(double);
double fun(double);
double Metod_Del_2(type_f,double,double,double);
//---------------------- Текст функции-обработчика кнопки Расчет ----------------------
        double a, b, x, eps, h, y, r;
        int nom=0, iter;
        a = StrToFloat(Edit1->Text);        b = StrToFloat(Edit2->Text);
        eps = StrToFloat(Edit3->Text);
        h = StrToFloat(Edit4->Text);
        Memo1->Lines->Add(" Функция 4*x - 7*sin(x)");
        Chart1->Series[0]->Clear();
      for(x = a-h; x< b+h; x+=h)
```

```
            Chart1->Series[0]->AddXY(x,fun(x));
      Memo1->Lines->Add("------ Корни ------");
      for(x = a; x<=b; x+=h){
        if(fun(x)*fun(x+h)<0){
            nom++;
            y = Metod_Del_2(fun,x,x+h,eps);
            Memo1->Lines->Add(IntToStr(nom)+"-й = "+FloatToStrF(y,ffFixed,8,6));
        }
      }
      if(nom==0) Memo1->Lines->Add("На отрезке корней НЕТ!");
//------------------------ Метод деления отрезка пополам -------------------------------
 double Metod_Del_2(type_f f,double x0,double x1,double eps) {
   double x2,y0,y2;
   y0=f(x0);
   do  {
        x2=(x0+x1)/2;       y2=f(x2);
        if(y0*y2 > 0) {
                x0 = x2;       y0 = y2;
        }
        else x1 = x2;
   } while (fabs(x1-x0)>eps);
   return (x0+x1)/2;
  }
//------------------------------ Заданная функция f(x) -----------------------------------
  double fun(double x) {
        return 4*x - 7*sin(x);
  }
```

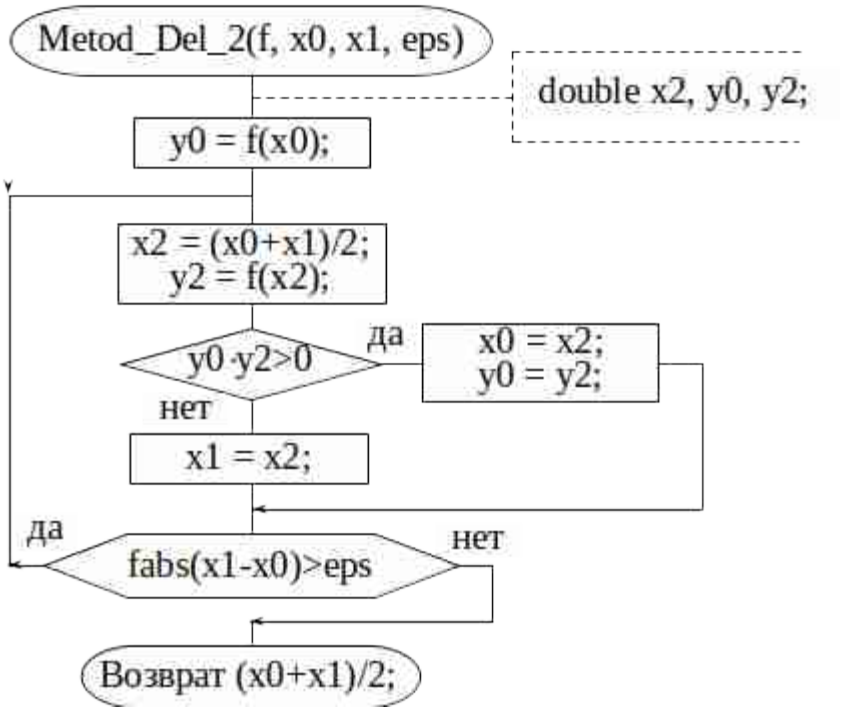


Рис. 6.1

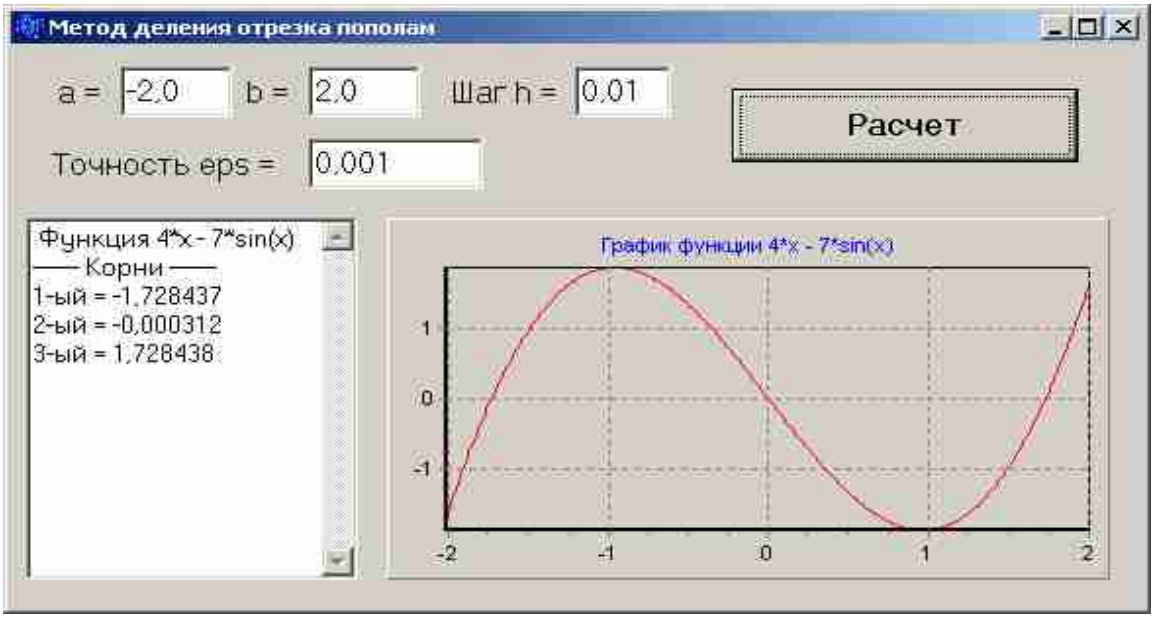


Рис. 6.2

6.3. Индивидуальные задания

Написать и отладить программу поиска всех корней функции $f(x)$ на отрезке $[a, b]$ в соответствии с вариантом (табл. 6.1). Метод нахождения корня оформить в виде отдельной функции, алгоритм которой описать блок-схемой.

Таблица 6.1

Вид функции $f(x)$	a	b	Заданный метод
1. $4x - 7\sin(x)$	–2	2	Метод простой итерации
2. $x^2 - 10\sin^2(x) + 2$	–1	3	Метод Ньютона
3. $\ln(x) - 5\cos(x)$	1	8	Метод секущих
4. $e^x / x^3 - \sin^3(x) - 2$	4	7	Метод Вегстейна
5. $\sqrt{x} - \cos^2(x) - 2$	4	8	Метод секущих
6. $\ln(x) - 5\sin^2(x)$	2	6	Метод простой итерации
7. $x - 5\sin^2(x) - 5$	3	9	Метод секущих
8. $\sin^2(x) - x/5 - 1$	–4	0	Метод секущих
9. $x^3 + 10x^2 - 50$	–12	5	Метод Вегстейна
10. $x^3 - 5x^2 + 12$	–2	5	Метод Ньютона
11. $x^3 + 6x^2 - 0.02e^x - 14$	–6	2	Метод Ньютона
12. $x^2 + 5\cos(x) - 3$	–4	2	Метод Ньютона
13. $\sin^2(x) - 3\cos(x)$	–7	3	Метод секущих
14. $x^3 - 50\cos(x)$	–4	3	Метод Вегстейна
15. $x^2 - 10\sin^2(x) + 2$	–1	3	Метод секущих
16. $0.1x^3 + x^2 - 10\sin(x) - 8$	–4	4	Метод Ньютона

Примечание. Все функции на указанном интервале имеют три корня.

Лабораторная работа №7. Аппроксимация функций

Цель работы: изучить алгоритмы аппроксимации функций; освоить методику построения и использования алгебраических интерполяционных многочленов Лагранжа и Ньютона

7.1. Краткие теоретические сведения

Понятие аппроксимации

Одной из наиболее часто встречающихся задач является установление характера зависимости между различными величинами, что позволяет по значению одной величины определить значение другой. Математической моделью зависимости одной величины от другой является понятие функции $y = f(x)$.

В практике расчетов, связанных с обработкой экспериментальных данных, вычислением $f(x)$, разработкой вычислительных методов, встречаются следующие ситуации:

- установить вид функции $y = f(x)$, если известны только некоторые значения, заданные таблицей $\{(x_i, y_i), i = 1, \ldots, m\}$;
- упростить вычисление известной функции $f(x)$ или ее характеристик (производной, максимума и т.п.), если $f(x)$ имеет слишком сложный вид.

Ответы на эти вопросы даются теорией аппроксимации функций, основная задача которой состоит в нахождении функции $y = \varphi(x)$, близкой (т.е. аппроксимирующей) к исходной функции.

Основной подход к решению этой задачи заключается в том, что аппроксимирующая функция $\varphi(x)$ выбирается зависящей от нескольких свободных параметров $c = (c_1, c_2, \ldots, c_n)$, т.е. $y = \varphi(x) = \varphi(x, c_1, \ldots, c_n) = \varphi(x, c)$, значения которых подбираются из условия близости $f(x)$ и $\varphi(x)$.

В зависимости от способа подбора параметров вектора c получают различные ***методы аппроксимации***.

Наиболее простой является ***линейная аппроксимация,*** при которой выбирают функцию $\varphi(x, c)$, линейно зависящую от параметров c, т.е. в виде обобщенного многочлена:

$$\varphi(x, \vec{c}) = c_1\varphi_1(x) + \ldots + c_n\varphi_n(x) = \sum_{k=1}^{n} c_k\varphi_k(x). \qquad (7.1)$$

Здесь $\{\varphi_1(x), \ldots, \varphi_n(x)\}$ – известная система линейно независимых функций, в качестве которых могут быть выбраны любые элементарные функции или их комбинации. Важно, чтобы эта система была ***полной***, т.е. обеспечивающей аппроксимацию $f(x)$ многочленом (8.1) с заданной точностью при $n \to \infty$.

При интерполяции обычно используется система линейно независимых функций $\{\varphi_k(x) = x^{k-1}\}$. Для среднеквадратичной аппроксимации удобнее в качестве $\varphi_k(x)$ брать ортогональные на интервале [-1, 1] многочлены *Лежандра*:

$\{\varphi_1(x) = 1;\ \varphi_2(x) = x;\ \varphi_{k+1}(x) = [(2k + 1)x\varphi_k(x) - k\varphi_{k-1}(x)];\ k = 2,3,\ldots, n\};$

$$\int_{-1}^{1} \varphi_k(x) \cdot \varphi_l(x)dx = 0;\quad k \neq l.$$

Интерполяция является одним из способов аппроксимации функций. Суть ее состоит в следующем. В области значений x, представляющей некоторый интервал $[a, b]$, где функции f и φ должны быть близки, выбирают упорядоченную систему точек (узлов) $x_1 < x_2 < \ldots < x_n$ (обозначим $x = (x_1, \ldots, x_n)$), число которых равно количеству искомых параметров $c_1, c_2, \ldots, c_n$. Далее параметры c подбирают такими, чтобы функция $\varphi(x, c)$ совпадала с $f(x)$ в этих узлах, для чего решают полученную систему из n алгебраических уравнений.

В случае линейной аппроксимации (7.1) система для нахождения коэффициентов c линейна и имеет следующий вид:

$$\sum_{k=1}^{n} c_k \varphi_k(x_i) = y_i;\quad i = 1, 2, \ldots, n;\quad y_i = f(x_i). \qquad (7.2)$$

Для большинства практически важных приложений при интерполяции наиболее удобны обычные алгебраические многочлены.

Интерполяционным многочленом называют алгебраический многочлен степени $n - 1$, совпадающий с аппроксимируемой функцией в выбранных n точках.

Общий вид алгебраического многочлена

$$\varphi(x, \vec{c}) = P_{n-1}(x) = c_1 + c_2 x + c_3 x^2 + \ldots + c_n x^{n-1} = \sum_{k=1}^{n} a_k x^{k-1}. \qquad (7.3)$$

Наиболее часто в приложениях используют интерполяционные многочлены в форме Лагранжа и Ньютона, т.к. многочлены в этой форме прямо записаны через значения таблицы $\{(x_i, y_i),\ i = 1, \ldots, n\}$.

Интерполяционный многочлен Ньютона

$$N_{n-1}(x_T) = y_1 + \sum_{k=1}^{n-1} (x_T - x_1)(x_T - x_2)\ldots(x_T - x_k)\Delta_1^k, \qquad (7.4)$$

где x_T – текущая точка, в которой надо вычислить значение многочлена, Δ_1^k – разделенные разности порядка k, которые вычисляются по следующим рекуррентным формулам:

$$\Delta_i^1 = \frac{y_i - y_{i+1}}{x_i - x_{i+1}},\qquad i = 1\ldots n;$$

$$\Delta_i^2 = \frac{\Delta_i^1 - \Delta_{i+1}^1}{x_i - x_{i+2}},\qquad i = 1\ldots(n - 2);$$

. . .

$$\Delta_i^k = \frac{\Delta_i^{k-1} - \Delta_{i+1}^{k-1}}{x_i - x_{i+k}}, \quad i = 1 \ldots (n - k).$$

Схема расчета многочлена Ньютона представлена на рис. 7.1.

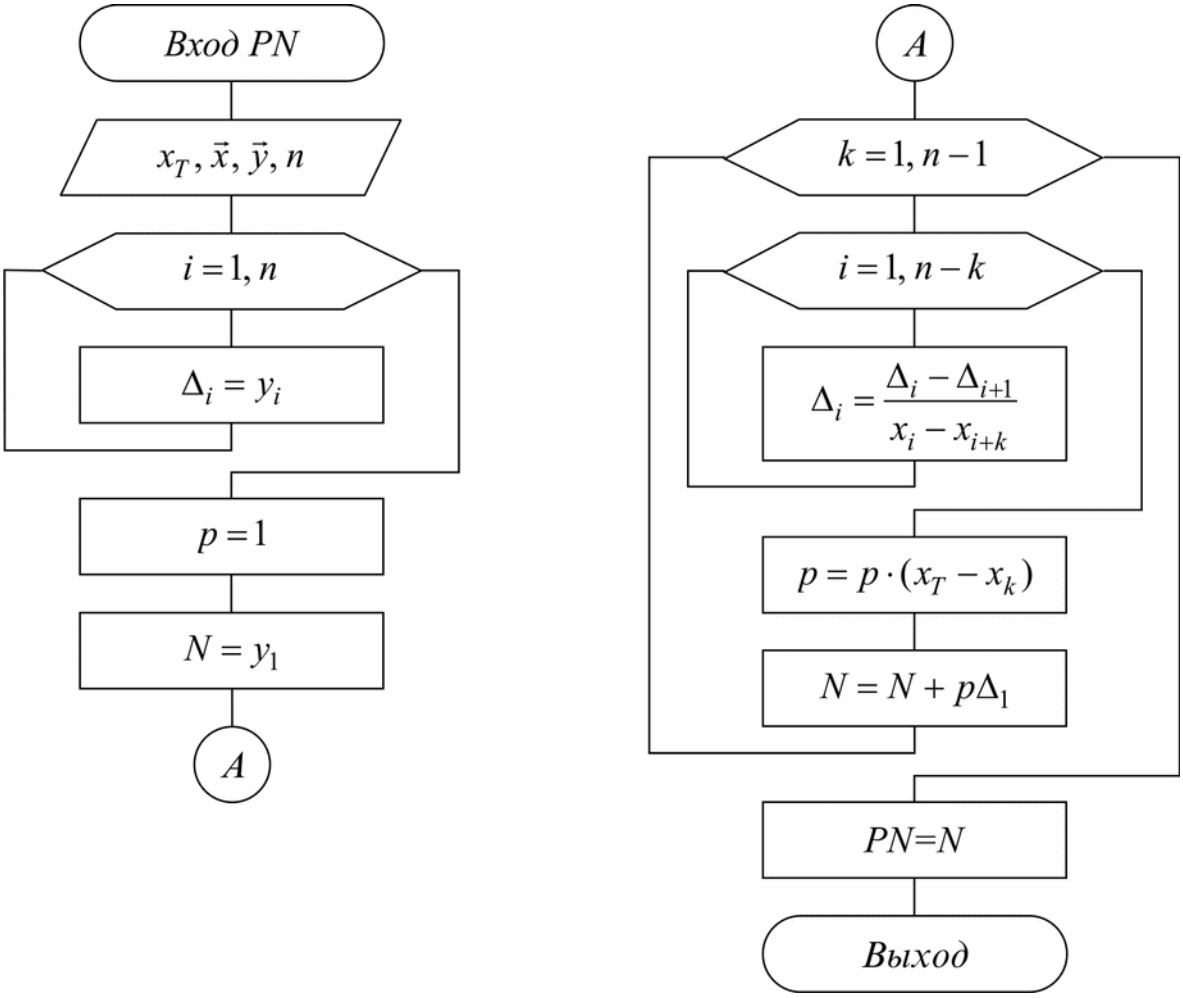


Рис. 7.1

Линейная и квадратичная интерполяции

Вычисления по интерполяционной формуле (7.4) для $n > 3$ используют редко. Обычно при интерполяции по заданной таблице из $m > 3$ точек применяют квадратичную $n = 3$ или линейную $n = 2$ интерполяцию. В этом случае для приближенного вычисления значения функции f в точке x находят в таблице ближайший к этой точке i-узел из общей таблицы, строят интерполяционный многочлен Ньютона первой или второй степени по формулам

$$N_1(x_T) = y_{i-1} + (x_T - x_{i-1}) \frac{y_i - y_{i-1}}{x_i - x_{i-1}}, \quad x_{i-1} \le x_T \le x_i; \qquad (7.5)$$

$$N_2(x_T) = N_1(x_T) + (x_T - x_{i-1})(x_T - x_i)\frac{\left(\frac{y_{i-1} - y_i}{x_{i-1} - x_i}\right) - \left(\frac{y_i - y_{i+1}}{x_i - x_{i+1}}\right)}{x_{i-1} - x_{i+1}}; \quad x_{i-1} \le x_T \le x_{i+1}$$

и за значение $f(x)$ принимают $N_1(x)$ (*линейная интерполяция*) или $N_2(x)$ (*квадратичная интерполяция*).

Интерполяционный многочлен Лагранжа

$$L_{n-1}(x_T) = \sum_{k=1}^{n} y_k e_k(x_T); \quad e_k(x_T) = \prod_{\substack{i=1 \\ i \neq k}}^{n} \frac{x_T - x_i}{(x_k - x_i)}. \qquad (7.6)$$

Многочлены $e_k^{n-1}(x_j)$ выбраны так, что во всех узлах, кроме k-го, они обращаются в ноль, в k-м узле они равны единице:

$$e_k^{n-1}(x_j) = \begin{cases} 1, & j = k; \\ 0, & j \neq k. \end{cases}$$

Очевидно, что $L_{n-1}(x_i) = e_i(x_i) y_i = 1 \cdot y_i = y_i,\ i = 1,\ldots,n$.

Функция, реализующая вычисления с помощью многочлена Лагранжа, рассмотрена в примере.

7.2. Пример выполнения задания

Составить алгоритм, по которому написать и отладить программу аппроксимации функции $f(x) = x^3 - 5x^2$ на интервале [-2, 5] многочленом Лагранжа, m – количество точек, в которых известна функция, n – количество рассчитываемых значений.

Вид формы и полученные результаты представлены на рис. 7.2. Тексты функций-обработчиков и функции пользователя будут иметь следующий вид:

```
double fun(double);
double Mn_Lagr(double*, double, int);
//------------------ Текст функции-обработчика кнопки Вычислить ------------------
    double x,h,h1, a, b, *mas_x, *mas_y_t;
    int i,n,m;
    a = StrToFloat(Edit1->Text);          b = StrToFloat(Edit2->Text);
    m = StrToInt(Edit3->Text);            n = StrToInt(Edit4->Text);
    h = (b-a)/(m-1);                      h1 = (b-a)/(n-1);
    mas_x = new double[m+1];              mas_y_t = new double[n+1];
    for(x=a, i=0; i<m; i++){
        mas_x[i] = x;
        x+=h;
    }
    Memo1->Lines->Add("---- Многочлен Лагранжа ---");
    Memo1->Lines->Add("Получили " + IntToStr(n) + " значений:");
    for(x=a, i=0; i<n; i++, x+=h1) {
                mas_y_t[i] = Mn_Lagr(mas_x,x,m);
                Memo1->Lines->Add(" x = "+FloatToStrF(x,ffFixed,8,2)
```

```
                +"  f*(x) = "+FloatToStrF(mas_y_t[i],ffFixed,8,4));
        }
//---------------------------- Очистка Графиков ----------------------------------------
        Chart1->Series[0]->Clear();        Chart1->Series[1]->Clear();
```

Рис. 7.2

```
//---------------------------- Вывод Графиков ----------------------------------------
        for(x=a-0.1; x<b+0.1; x+=0.01)
                Chart1->Series[0]->AddXY(x,fun(x));
        for(x=a,i=0; i<n; i++,x+=h1)
                Chart1->Series[1]->AddXY(x,mas_y_t[i]);
        delete []mas_x;
        delete []mas_y_t;
}
//---------------------------- Исходная функция f(x) ----------------------------------
double fun(double x) {
        return pow(x,3) - 5 * x*x;
}
//---------------------------- Многочлен Лагранжа -------------------------------------
double Mn_Lagr(double *x, double xt, int kol)  {
        int i, k;
        double e, p=0;
        for(k=0; k<kol; k++) {
            e=1.;
            for (i=0;i<kol;i++)
                  if (i!=k)    e *= ((xt-x[i])/(x[k]-x[i]));
```

```
            p += e*fun(x[k]);
        }
 return p;
}
```

7.3. Индивидуальные задания

Написать и отладить программу (задания из табл. 7.1) аппроксимации функции $f(x)$ на интервале $[a, b]$ заданным методом, m – количество точек, в которых известна функция (размер таблицы). Вид функции задан для получения значений таблицы (x_i, y_i), $i=1,2,\ldots,m$ и проверки качества аппроксимации.

Решение задачи оформить отдельной функцией, алгоритм которой описать в виде блок-схемы.

Таблица 7.1

Функция $f(x)$	a	b	m	Вид аппроксимации
1. $4x - 7\sin(x)$	–2	3	11	Многочлен Лагранжа
2. $x^2 - 10\sin^2(x)$	0	3	4	Многочлен Ньютона
3. $\ln(x) - 5\cos(x)$	1	8	4	Линейная интерполяция
4. $e^x / x^3 - \sin^3(x)$	4	7	4	Квадратичная интерполяция
5. $\sqrt{x} - \cos^2(x)$	5	8	4	Многочлен Лагранжа
6. $\ln(x) - 5\sin^2(x)$	3	6	11	Многочлен Ньютона
7. $x - 5\sin^2(x)$	1	4	11	Линейная интерполяция
8. $\sin^2(x) - x/5$	0	4	11	Квадратичная интерполяция
9. $x^3 + 10x^2$	–8	2	5	Многочлен Лагранжа
10. $x^3 - 5x^2$	–2	5	5	Многочлен Ньютона
11. $x^3 + 6x^2 - 0.02e^x$	–5	3	5	Линейная интерполяция
12. $x^2 + 5\cos(x)$	–1	4	5	Квадратичная интерполяция
13. $\sin^2(x) - 3\cos(x)$	1	7	11	Многочлен Лагранжа
14. $x^3 - 50\cos(x)$	–2	5	11	Многочлен Ньютона
15. $\ln(x) - 5\cos(x)$	1	8	4	Линейная интерполяция
16. $0.1x^3 + x^2 - 10\sin(x)$	–4	2	11	Квадратичная интерполяция

Лабораторная работа №8. Алгоритмы вычисления интегралов

Цель работы: изучить алгоритмы нахождения значений интегралов.

8.1. Краткие теоретические сведения

Формулы для вычисления интеграла $U = \int_a^b f(x)dx$ получают следующим образом. Область интегрирования [*a*, *b*] разбивают на малые отрезки, тогда значение интеграла по всей области равно сумме интегралов на этих отрезках.

Выбирают на каждом отрезке [x_i, x_{i+1}] 1–5 узлов и строят интерполяционный многочлен соответствующего порядка. Вычисляют интеграл от этого многочлена, и в результате получают формулу численного интегрирования через значения подынтегральной функции в выбранной системе точек. Такие выражения называют ***квадратурными формулами.***

Рассмотрим наиболее часто используемые квадратурные формулы для равных отрезков длиной $h = (b - a)/m$; $x_i = a + (i - 1) \cdot h$; $i = 1, 2, \ldots, m$; где m – количество разбиений отрезка интегрирования.

Формула средних

Формула средних получается, если на каждом *i*-м отрезке взять один центральный узел $x_{i+1/2} = (x_i + x_{i+1})/2$, соответствующий середине отрезка. Функция на каждом отрезке аппроксимируется многочленом нулевой степени (константой) $P_0(x) = y_{i+1/2} = f(x_{i+1/2})$. Заменяя площадь криволинейной фигуры площадью прямоугольника высотой $y_{i+1/2}$ и основанием h, получим формулу средних (рис. 8.1):

$$\int_a^b f(x)dx \approx \sum_{i=1}^{m} \int_{x_i}^{x_{i+1}} P_0(x)dx = h\sum_{i=1}^{m} y_{i+1/2} = \Phi_{CP}. \qquad (8.1)$$

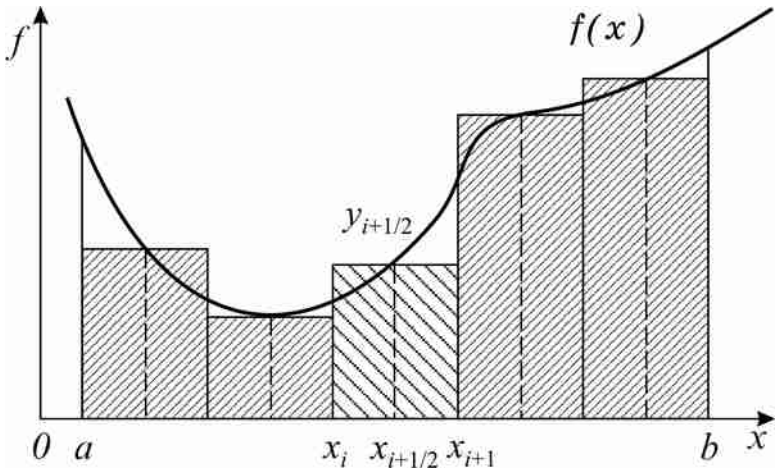


Рис. 8.1

Формула трапеций

Формула трапеций получается при аппроксимации функции $f(x)$ на каждом отрезке $[x_i, x_{i+1}]$ интерполяционным многочленом первого порядка, т.е. прямой, проходящей через точки (x_i, y_i), (x_{i+1}, y_{i+1}). Площадь криволинейной фигуры заменяется площадью трапеции с основаниями y_i, y_{i+1} и высотой h (рис. 8.2):

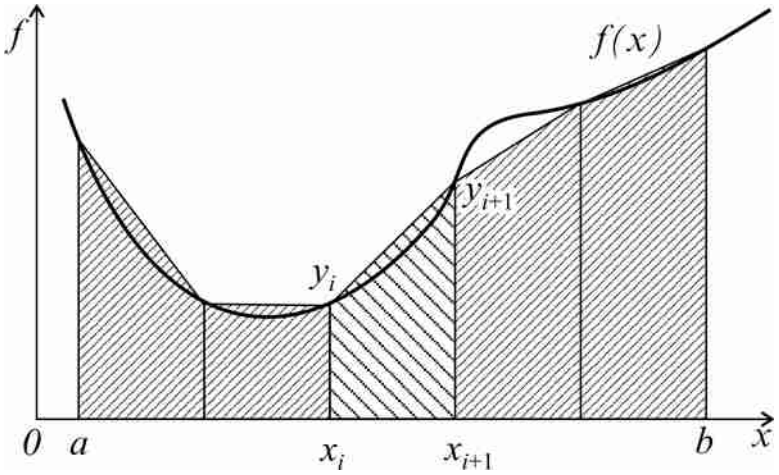


Рис. 8.2

$$\int_a^b f(x)dx \approx \sum_{i=1}^{m} \int_{x_i}^{x_{i+1}} P_1(x)dx = h\sum_{i=1}^{m} \frac{y_i + y_{i+1}}{2} = h\left[\frac{y_1 + y_{m+1}}{2} + \sum_{i=2}^{m} y_i\right] = \Phi_{TP}. \quad (8.2)$$

Формула Симпсона

Формула Симпсона получается при аппроксимации функции $f(x)$ на каждом отрезке $[x_i, x_{i+1}]$ интерполяционным многочленом второго порядка (параболой) с узлами x_i, $x_{i+1/2}$, x_{i+1}. После интегрирования параболы получаем (рис. 8.3)

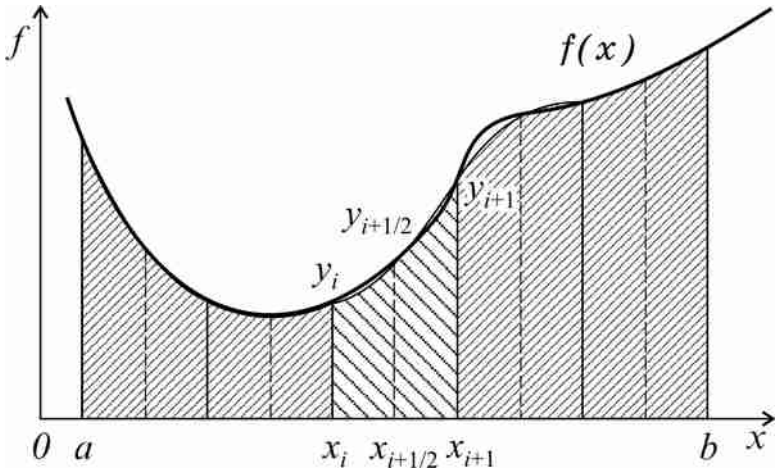


Рис. 8.3

$$\int_a^b f(x)dx \approx \sum_{i=1}^{m} \int_{x_i}^{x_{i+1}} P_2(x)dx = \frac{h}{6}\sum_{i=1}^{m}(y_i + 4y_{i+0.5} + y_{i+1}) = \Phi_{СИМ}\ . \qquad (8.3)$$

После приведения подобных членов получаем более удобный для программирования вид:

$$\Phi_{СИМ} = \frac{h}{3} \cdot \left[\frac{y_1 + 4y_{1+0.5} + y_{m+1}}{2} + \sum_{i=2}^{m}(2y_{i+0.5} + y_i) \right].$$

Схема с автоматическим выбором шага по заданной точности

***Метод* 1**. Одним из вариантов вычисления интеграла с заданной точностью является следующий.

1. Задают первоначальное число интервалов разбиения m и вычисляют приближенное значение интеграла $S1$ выбранным методом.

2. Число интервалов удваивают $m = 2m$.

3. Вычисляют значение интеграла $S2$.

4. Если $|S1 - S2| \geq \varepsilon$ (ε – заданная погрешность), то $S1 = S2$, расчет повторяют – переход к пункту 2.

5. Если $|S1 - S2| < \varepsilon$, т.е. заданная точность достигнута, выполняют вывод результатов: $S2$ – найденное значение интеграла с заданной точностью ε, m – количество интервалов.

***Метод* 2**. Анализ формул (8.1), (8.2) и (8.3) показывает, что точное значение интеграла находится между значениями $\Phi_{СР}$ и $\Phi_{ТР}$, при этом имеет место соотношение

$$\Phi_{СИ} = (\Phi_{ТР} - 2 \cdot \Phi_{СР})/3.$$

Это соотношение часто используется для контроля погрешности вычислений. Расчет начинается с $m = 2$ и производится по двум методам, в результате получают $\Phi_{СР}$, $\Phi_{ТР}$. Если $|\Phi_{СР} - \Phi_{ТР}| \geq \varepsilon$, увеличивают $m = 2m$ и расчет повторяют.

Формулы Гаусса

При построении предыдущих формул в качестве узлов интерполяционного многочлена выбирались середины и (или) концы интервала разбиения. При этом оказывается, что увеличение количества узлов не всегда приводит к уменьшению погрешности, т.е. за счет удачного расположения узлов можно значительно увеличить точность.

Суть методов Гаусса порядка n состоит в таком расположении n узлов интерполяционного многочлена на отрезке $[x_i, x_{i+1}]$, при котором достигается минимум погрешности *квадратурной формулы*. Анализ показывает, что узлами, удовлетворяющими такому условию, являются нули ортогонального многочлена Лежандра степени n (см. подразд. 7.1).

Для $n = 1$ один узел должен быть выбран в центре отрезка, следовательно, метод средних является методом Гаусса 1-го порядка.

Для *n* = 2 узлы должны быть выбраны следующим образом:

$$x_i^{1,2} = x_{i+1/2} \mp \frac{h}{2} \cdot 0.5773502692,$$

и соответствующая формула ***Гаусса 2-го*** порядка имеет вид

$$\int_a^b f(x)dx \approx \frac{h}{2}\sum_{i=1}^{n}[f(x_i^1) + f(x_i^2)].$$

Для *n* = 3 узлы выбираются следующим образом:

$$x_i^0 = x_{i+1/2}, \quad x_i^{1,2} = x_i^0 \mp \frac{h}{2} \cdot 0.7745966692,$$

и соответствующая формула ***Гаусса 3-го*** порядка имеет вид

$$\int_a^b f(x)dx \approx \frac{h}{2}\sum_{i=1}^{n}(\frac{5}{9} f(x_i^1) + \frac{8}{9} f(x_i^0) + \frac{5}{9} f(x_i^2)).$$

8.2. Пример выполнения задания

Написать и отладить программу вычисления значения интеграла от функции *f*(*x*) = 4*x* - 7sin*x* на интервале [-2, 3] методом Симпсона с выбором: по заданному количеству разбиений *n* и заданной точности ε. Панель диалога будет иметь вид, представленный на рис. 8.4.

Как и в предыдущих примерах, приведем только тексты функций-обработчиков соответствующих кнопок:

```
  . . .
typedef double (*type_f)(double);
double fun(double);
double Simps(type_f, double, double, int);
```

//---------- Текст функции-обработчика кнопки ***РАСЧЕТ*** ----------

```
    double a, b, x, eps, h, Int1, Int2, pogr;
    int n, n1;
    a = StrToFloat(Edit1->Text);       b = StrToFloat(Edit2->Text);
    eps = StrToFloat(Edit3->Text);     n = StrToInt(Edit4->Text);
    h = (b - a)/100;                          // Шаг вывода исходной функции
    Chart1->Series[0]->Clear();
    for(x = a-h; x< b+h; x+=h)
            Chart1->Series[0]->AddXY(x,fun(x));
    switch(RadioGroup2->ItemIndex)  {
      case 0:
            Memo1->Lines->Add("Расчет по разбиению на n = " + IntToStr(n));
            Int1 = Simps(fun,a,b,n);
      break;
      case 1:
            n1=2;
            Memo1->Lines->Add("Расчет по точности eps");
            Int1 = Simps(fun,a,b,n1);
```

```
                do {
                        n1*=2;
                        Int2 = Simps(fun,a,b,n1);
                        pogr = fabs(Int2-Int1);
                        Int1 = Int2;
                } while(pogr > eps);
                Memo1->Lines->Add("При n = " +IntToStr(n1));
        break;
    }
    Memo1->Lines->Add("Значение интеграла = " + FloatToStrF(Int1,ffFixed,8,6));
//------------------------------ Метод Симпсона ---------------------------------------------
double Simps(type_f f, double a, double b, int n) {
        double s=0,h,x;
        h=(b-a)/n;
        x=a;
        for(int i=1; i<=n; i++)  {
                s+=f(x)+4*f(x+h/2)+f(x+h);
                x+=h;
        }
        return s*h/6;
}
//------------------------------ Подынтегральная функция f(x) -----------------------------
double fun(double x) {
        return 4*x - 7*sin(x);              // На интервале [-2, 3] значение 5,983
}
```

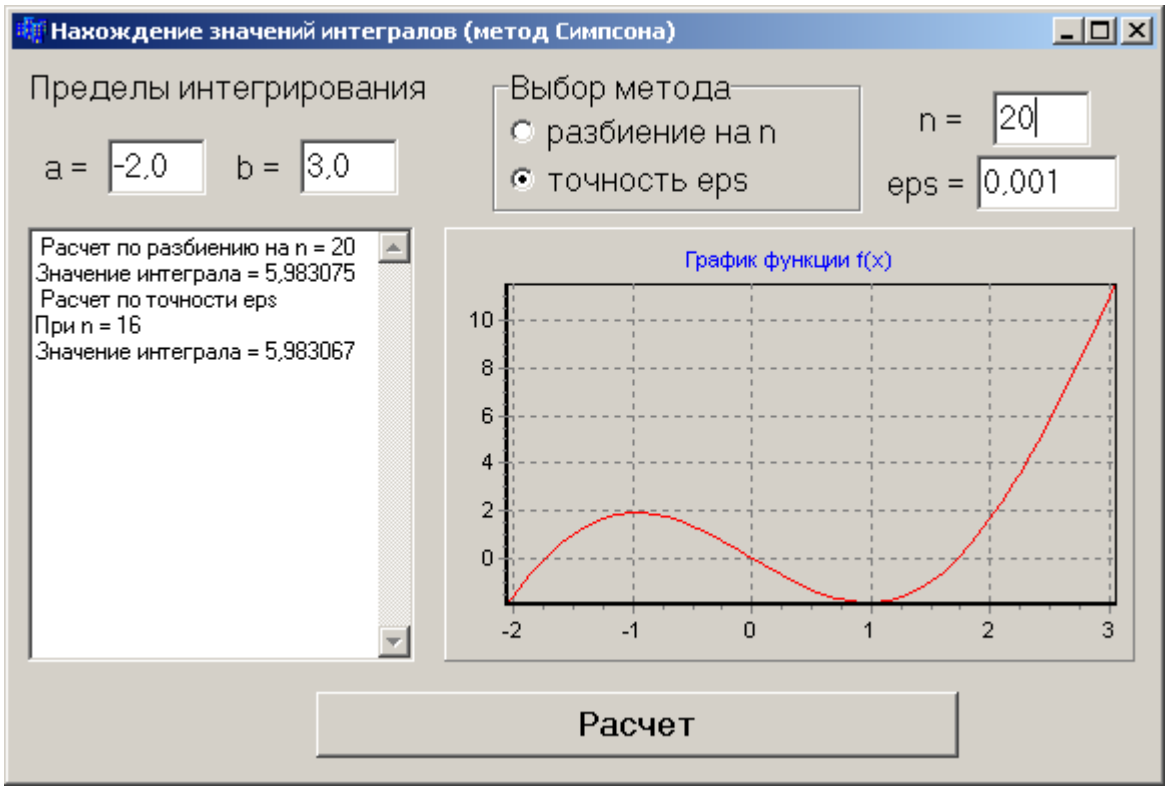


Рис. 8.4

8.3. Индивидуальные задания

Написать и отладить программу вычисления интеграла указанным методом двумя способами – по заданному количеству разбиений *n* и заданной точности ε (метод 1) (задания табл. 8.1).

Реализацию указанного метода оформить отдельной функцией, алгоритм которой описать в виде блок-схемы.

Таблица 8.1

Функция *f*(*x*)	*a*	*b*	Метод интегрирования	Значение интеграла
1. $4x - 7\sin(x)$	–2	3	Средних	5.983
2. $x^2 - 10\sin^2(x)$	0	3	Трапеций	–6.699
3. $\ln(x) - 5\cos(x)$	1	8	Симпсона	8.896
4. $e^x / x^3 - \sin^3(x)$	4	7	Автомат-метод 2	6.118
5. $\sqrt{x} - \cos^2(x)$	5	8	Гаусса 2	6.067
6. $\ln(x) - 5\sin^2(x)$	3	6	Гаусса 3	–3.367
7. $x - 5\sin^2(x)$	1	4	Средних	0.100
8. $\sin^2(x) - x/5$	0	4	Трапеций	0.153
9. $x^3 + 10x^2$	–8	2	Симпсона	713.3
10. $x^3 - 5x^2$	–2	5	Автомат-метод 2	–69.42
11. $x^3 + 6x^2 - 0.02e^x$	–5	3	Гаусса 2	167.6
12. $x^2 + 5\cos(x)$	–1	4	Гаусса 3	22.09
13. $\sin^2(x) - 3\cos(x)$	1	7	Средних	3.533
14. $x^3 - 50\cos(x)$	–2	5	Автомат-метод 2	154.73
15. $0.1x^3 + x^2 - 10\sin(x)$	–4	2	Симпсона	20.375
16. $\sin^2(x) - x/5$	0	4	Трапеций	0.153

Лабораторная работа №9. Алгоритмы поиска и сортировки в массивах

Цель работы: изучить способы сортировки и поиска в массивах структур и файлах.

9.1. Краткие теоретические сведения

При обработке баз данных часто применяются массивы структур. Обычно база данных накапливается и хранится на диске в файле. К ней часто приходится обращаться, обновлять, перегруппировывать. Работа с базой может быть организована двумя способами.

1. Внесение изменений и поиск осуществляется прямо на диске, используя специфическую технику работы со структурами в файлах. При этом временные затраты на обработку данных (поиск, сортировку) значительно возрастают, но нет ограничений на использование оперативной памяти.

2. Считывание всей базы (или необходимой ее части) в массив структур. При этом обработка производится в оперативной памяти, что значительно увеличивает скорость, однако требует больших затрат памяти.

Наиболее частыми операциями при работе с базами данных являются «поиск» и «сортировка». При этом алгоритмы решения этих задач существенно зависят от того, организованы структуры в массивы или размещены на диске.

Обычно элемент данных (структура) содержит некое ключевое поле (*ключ*), по которому его можно найти. Ключом может служить любое поле структуры, например, фамилия, номер телефона или адрес. Основное требование к ключу в задачах поиска состоит в том, чтобы операция проверки на равенство была корректной, поэтому при поиске данных по ключу, имеющему вещественное значение, следует указывать не его конкретное значение, а интервал, в который это значение попадает.

9.1.1. Алгоритмы поиска

Предположим, что у нас имеется следующая структура:

```
struct Ttype {
        type key;           // Ключевое поле типа type
                . . .       // Описание других полей структуры
} *a;                       // Указатель для динамического массива структур
```

Задача поиска требуемого элемента в массиве структур *a* (размер *n* – задается при выполнении программы) заключается в нахождении индекса *i_key*, удовлетворяющего условию *a*[*i_key*].*key* = *f_key*, *key* – интересующее нас поле структуры данных, *f_key* – искомое значение того же типа что и *key*. После нахождения индекса *i_key* обеспечивается доступ ко всем другим полям найденной структуры *a*[*i_key*].

Линейный поиск используется, когда нет никакой дополнительной информации о разыскиваемых данных, и представляет собой последовательный перебор всех элементов массива. Если поле поиска является уникальным, то поиск выполняется до обнаружения требуемого ключа или до конца, если ключ не обнаружен. Если же поле поиска не уникальное, приходится перебирать все данные до конца массива:

```
int i_key = 0, kod = 0;
for (i = 1; i < n; i++)
        if (a[i].key == f_key) {
                kod = 1;
// Обработка найденного элемента, например, вывод
                i_key = i;
                                // break;      – если поле поиска уникальное
        }
if(kod == 0)                    // Вывод сообщения, что элемент не найден
```

Функция поиска *всех* элементов целочисленного динамического массива ***a*** размера ***n***, равных значению ***x***, может иметь следующий вид:

```
void Poisk_Lin(int *a, int  n, int x)
{
        int i, kod = 0;
        for(i = 0; i < n; i++)
                if ( a[i] == x ) {
                        kod = 1;
                        Form1->Memo1->Lines->Add(IntToStr(a[i]));
                        // В консольном приложении  cout << a[i] << endl;
                }
        if(kod == 0) // Вывод сообщения, что элемент не найден
}
```

Поиск делением пополам используется, если данные ***упорядочены*** по возрастанию (по убыванию) ключа *key*. Алгоритм поиска осуществляется следующим образом:

– берется средний элемент *m*;

– если элемент массива *a*[*m*].*key* < *f_key*, то все элементы $i \le m$ исключаются из дальнейшего поиска, иначе – исключаются все элементы с индексами $i > m$.

Приведем пример, реализующий этот алгоритм:

```
int i_key = 0, j = n–1, m;
while(i_key < j) {
        m = (i_key + j)/2;
        if (a[m].key < f_key) i_key = m+1;
        else  j = m;
}
if (a[i_key].key != key) return -1;              // Элемент не найден
```

```
        else return i;
```

Проверка совпадения *a*[*m*].*k* = *f_key* в этом алгоритме внутри цикла отсутствует, т.к. тестирование показало, что в среднем выигрыш от уменьшения количества проверок превосходит потери от нескольких «лишних» вычислений до выполнения условия *i_key* = *j*.

Функция поиска *одного* значения ***x*** в целочисленном динамическом массиве ***a*** размера ***n*** может иметь следующий вид:

```
void Poisk_Del_2(int *a, int  n, int x)
{
        int i = 0, j = n-1, m;
        while(i < j) {
                m = (i+j)/2;
                if (x > a[m]) i = m+1;
                else j = m;
        }
        if (a[i] == x)  Form1->Memo1->Lines->Add(IntToStr(a[i]));
                        // В консольном приложении  cout << a[i] << endl;
        else            // Вывод сообщения, что элемент не найден
}
```

9.1.2. Алгоритмы сортировки

Под сортировкой понимается процесс перегруппировки элементов массива, приводящий к их упорядоченному расположению относительно ключа.

Цель сортировки – облегчить последующий поиск элементов. Метод сортировки называется устойчивым, если в процессе перегруппировки относительное расположение элементов с равными ключами не изменяется. Основное условие при сортировке массивов – это не вводить дополнительных массивов, т.е. все перестановки элементов должны выполняться в исходном массиве. Сортировку массивов принято называть ***внутренней***, а сортировку файлов – ***внешней***.

Методы внутренней сортировки классифицируются по времени их работы. Хорошей мерой эффективности может быть число операций сравнений ключей и число пересылок (перестановок) элементов.

Прямые методы имеют небольшой код и просто программируются, быстрые, усложненные методы требуют меньшего числа действий, но эти действия обычно более сложные, чем в прямых методах, поэтому для достаточно малых значений *n* (***n*** $\leq$ **50**) прямые методы работают быстрее. Значительное преимущество быстрых методов начинает проявляться при ***n*** $\geq$ **100**.

Среди *простых* методов наиболее популярны следующие.

1. ***Метод прямого обмена*** (пузырьковая сортировка):

```
for (i = 0; i < n–1; i++)
        for (j = i+1; j < n; j++)
                if (a[i].key > a[j].key) {          // Переставляем элементы
                        r = a[i];  a[i] = a[j];  a[j] = r;
                }
```

Функция сортировки элементов целочисленного динамического массива ***a*** размера ***n*** может иметь следующий вид:

```
void Sort_Puz(int *a, int n)
{
        int i, j, r;
        for(i = 0; i < n - 1; i++)
                for( j = i + 1; j < n; j++)
                        if (a[i] > a[j]) {                  // Переставляем элементы
                                r = a[i];   a[i] = a[j];   a[j] = r;
                        }
}
```

2. ***Метод прямого выбора***:

```
        for (i = 0; i < n–1; i++) {
                m = i;
                for (j = i+1; j < n; j++)
                        if (a[j].key < a[m].key) m = j;
                r = a[m];                                   // Переставляем элементы
                a[m] = a[i];
                a[i] = r;
        }
```

Функция сортировки элементов целочисленного динамического массива ***a*** размера ***n*** может иметь следующий вид:

```
void Sort_Vub(int *a, int n)
{
        int i_min, i, j, r;
        for(i=0; i<n-1; i++) {
                i_min = i;
                for(j = i+1; j < n; j++)
                if (a[i_min] > a[j]) i_min = j;
                if (i_min != i) {                           // Переставляем элементы
                        r = a[i_min];    a[i_min] = a[i];     a[i] = r;
                }
        }
}
```

Реже используются: 3) сортировка с помощью прямого (двоичного) включения; 4) *шейкерная* сортировка (модификация пузырьковой).

К *улучшенным* методам сортировки относятся следующие.

1. ***Метод Д. Шелла*** (1959), усовершенствование метода прямого включения.

2. Сортировка ***с помощью дерева***, метод ***HeapSort***, Д. Уильямсон (1964).

3. Сортировка ***с помощью разделения***, метод ***QuickSort***, Ч. Хоар (1962), улучшенная версия пузырьковой сортировки, являющийся на сегодняшний день самым эффективным методом.

Идея метода разделения *QuickSort* заключается в следующем. Выбирается значение ключа среднего *m*-го элемента $x = a[m].key$. Массив просматривается слева направо до тех пор, пока не будет обнаружен элемент $a[i].key > x$. Затем массив просматривается справа налево, пока не будет обнаружен элемент $a[j].key < x$. Элементы $a[i]$ и $a[j]$ меняются местами. Процесс просмотра и обмена продолжается до тех пор, пока i не станет больше j. В результате массив оказывается разбитым на левую часть $a[L]$, $0 \le L \le j$ с ключами меньше (или равными) x и правую $a[R]$, $i \le R < n$ с ключами больше (или равными) x.

Алгоритм такого разделения очень прост и эффективен:

```
i = 0; j = n – 1; x = a[(L + R)/2].key;
while (i <= j) {
        while (a[i].key < x)  i++;
        while (a[j].key > x)  j--;
        if (i <= j) {
                r = a[i];                          // Переставляем элементы
                a[i] = a[j];
                a[j] = r;
                 i++;            j--;
        }
}
```

Чтобы отсортировать массив, остается применять алгоритм разделения к левой и правой частям, затем к частям частей и так до тех пор, пока каждая из частей не будет состоять из одного единственного элемента. Алгоритм получается итерационным, на каждом этапе которого стоят две задачи по разделению. К решению одной из них можно приступить сразу, для другой следует запомнить начальные условия (номер разделения, границы) и отложить ее решение до момента окончания сортировки выбранной половины.

Рекурсивная функция сортировки элементов целочисленного динамического массива ***a*** размера ***n*** может иметь следующий вид (*begin* – первый элемент массива, *end* – последний элемент массива):

```
void Quick_Sort(int *a, int begin, int end)
{
      int left, right, x;
      left = begin;
      right = end;
      x = a[(left+right)/2];
      do {
```

```
            while(a[left] <x )  left++;
            while(x < a[right]) right--;
            if(left <= right){
                  x = a[left];
                  a[left] = a[right];
                  a[right] = x;
                  left++;
                  right--;
            }
      } while(left<=right);
      if(begin < right) Quick_Sort(a, begin, right);
      if(left < end) Quick_Sort(a, left, end);
 }
```

Обращение к этой функции Quick_Sort(a, 0, n-1);

Сравнение методов сортировок показывает, что при $n > 100$ наихудшим является метод пузырька, метод *QuickSort* в 2-3 раза лучше, чем *HeapSort*, и в 3-7 раз, чем метод Шелла.

9.2. Индивидуальные задания

Написать программу обработки файла данных, состоящих из структур, в которой реализованы следующие функции: стандартная обработка файла (создание, просмотр, добавление); линейный поиск в файле; сортировка массива (файла) методами прямого выбора и *QuickSort*; двоичный поиск в отсортированном массиве.

1. В магазине формируется список лиц, записавшихся на покупку товара. Вид списка: номер, ФИО, домашний адрес, дата учета. Удалить из списка все повторные записи, проверяя ФИО и адрес. Ключ: дата постановки на учет.

2. Список товаров на складе включает: наименование товара, количество единиц товара, цену единицы и дату поступления товара на склад. Вывести в алфавитном порядке список товаров, хранящихся больше месяца, стоимость которых превышает 100 000 р. Ключ: наименование товара.

3. Для получения места в общежитии формируется список: ФИО студента, группа, средний балл, доход на каждого члена семьи. Общежитие в первую очередь предоставляется тем, у кого доход меньше двух минимальных зарплат, затем остальным в порядке уменьшения среднего балла. Вывести список очередности. Ключ: доход на каждого члена семьи.

4. В справочной автовокзала имеется расписание движения автобусов. Для каждого рейса указаны его номер, пункт назначения, время отправления и прибытия. Вывести информацию о рейсах, которыми можно воспользоваться для прибытия в пункт назначения раньше заданного времени. Ключ: время прибытия.

5. На междугородной АТС информация о разговорах содержит дату разговора, код и название города, время разговора, тариф, номер телефона в этом

городе и номер телефона абонента. Вывести по каждому городу общее время разговоров с ним и сумму. Ключ: общее время разговоров.

6. Информация о сотрудниках фирмы включает: ФИО, табельный номер, количество проработанных часов за месяц, почасовой тариф. Рабочее время свыше 144 ч считается сверхурочным и оплачивается в двойном размере. Вывести размер заработной платы каждого сотрудника фирмы за вычетом подоходного налога (12 % от суммы заработка). Ключ: размер заработной платы.

7. Информация об участниках спортивных соревнований содержит: ФИО игрока, игровой номер, возраст, рост, вес, наименование страны, название команды. Вывести информацию о самой молодой команде. Ключ: возраст.

8. Для книг, хранящихся в библиотеке, задаются: номер книги, автор, название, год издания, издательство и количество страниц. Вывести список книг с фамилиями авторов в алфавитном порядке, изданных после заданного года. Ключ: автор.

9. Различные цеха завода выпускают продукцию нескольких наименований. Сведения о продукции включают: наименование, количество, номер цеха. Для заданного цеха необходимо вывести изделия по каждому наименованию в порядке убывания их количества. Ключ: количество выпущенных изделий.

10. Информация о сотрудниках предприятия содержит: ФИО, номер отдела, должность, дату начала работы. Вывести списки сотрудников по отделам в порядке убывания стажа. Ключ: дата начала работы.

11. Ведомость абитуриентов, сдавших вступительные экзамены в университет, содержит: ФИО, номер группы, адрес, оценки. Определить количество абитуриентов, проживающих в г. Минске и сдавших экзамены со средним баллом не ниже 8.5, вывести их фамилии в алфавитном порядке. Ключ: ФИО.

12. В справочной аэропорта имеется расписание вылета самолетов на следующие сутки, которое содержит: номер рейса, тип самолета, пункт назначения, время вылета. Вывести информацию для заданного пункта назначения в порядке возрастания времени вылета. Ключ: пункт назначения.

13. В кассе имеется информация о поездах на ближайшую неделю: дата выезда, пункт назначения, время отправления, число свободных мест. Необходимо зарезервировать m мест до города N на k-й день недели с временем отправления поезда не позднее t часов. Вывести время отправления или сообщение о невозможности выполнить заказ. Ключ: число свободных мест.

14. Ведомость абитуриентов, сдавших вступительные экзамены в университет, содержит: ФИО абитуриента, 4 оценки. Определить средний балл по университету и вывести список абитуриентов, средний балл которых выше среднего балла по университету в порядке убывания балла. Ключ: средний балл.

15. В ателье хранятся квитанции о сданной в ремонт аппаратуре, в которых указано: наименование группы изделий (телевизор, радиоприемник и т.п.), марка изделия, дата приемки, состояние готовности заказа (выполнен, не выполнен). Вывести информацию о состоянии заказов на текущие сутки по группам изделий. Ключ: дата приемки в ремонт.

16. Информация о сотрудниках института содержит: ФИО, факультет, кафедру, должность, объем нагрузки (часов). Вывести списки сотрудников по кафедрам в порядке убывания нагрузки. Ключ: объем нагрузки.

Литература

1. Основы программирования в среде C++ *Builder*: лаб. практикум по курсу «Основы алгоритмизации и программирования» для студ. 1 – 2-го курсов БГУИР. В 2 ч. Ч. 1 / Бусько В. Л. [и др.] . – Минск : БГУИР, 2007.

2. Основы алгоритмизации и программирования. Язык Си : учеб. пособие / М. П. Батура [и др.]. – Минск : БГУИР, 2007.

3. Синицын, А. К. Программирование алгоритмов в среде *Builder* C++: лаб. практикум по курсам «Программирование» и «Основы алгоритмизации и программирование» для студ. 1-2 курсов всех спец. БГУИР днев. и веч. форм обуч.: В 2 ч. / А. К. Синицын. – Минск : БГУИР. Ч. 1. – 2004, Ч. 2. – 2005.

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

7. Архангельский, А. Я. Программирование в C++ *Builder* 6 / А. Я. Архангельский. – М.: ЗАО «Издательство БИНОМ», 2002.

8. Демидович, Е. М. Основы алгоритмизации и программирования. Язык СИ / Е. М. Демидович. – Минск : Бестпринт, 2001.

9. Кнут, Д. Искусство программирования: т. 1–3. Основные алгоритмы / Д. Кнут. – М. : Издательский дом «Вильямс», 2004.

10. Топп, У. Структуры данных в C++: пер. с англ. / У. Топп, У. Форд – М. : ЗАО «Издательство БИНОМ», 2000.

11. Синицын, А. К. Алгоритмы вычислительной математики: учеб.-метод. пособие по курсу «Основы алгоритмизации и программирования» / А. К. Синицын, А. А. Навроцкий. – Минск : БГУИР, 2007.

12. Калиткин, Н. Н. Численные методы : учебное пособие / Н. Н. Калиткин – М. : Наука, 1978.

13. Бахвалов, Н. С. Численные методы : учебное пособие / Н. С. Бахвалов – М. : Наука, 1975.

14. Егоров, А. А. Вычислительные алгоритмы линейной алгебры : учеб. пособие / А. А. Егоров. – Минск : БГУ, 2005.

15. Волков, Е. А. Численные методы / Е. А. Волков. – М. : Наука, 1982.

16. Васильков, Ю. В. Компьютерные технологии вычислений в математическом моделировании / Ю. В. Васильков, Н. Н. Василькова. – М. : Финансы и статистика, 2001.

17. Вычислительные методы высшей математики : учеб. пособие для вузов. Т.1 / В. И. Крылов [и др.]. – Минск : Выш. шк., 1972.

18. Вычислительные методы высшей математики: учеб. пособие для вузов. Т.2 / В. И. Крылов [и др.] – М. : Наука, 1977.

Св. план 2008, поз. 46

Учебное издание

Бусько Виталий Леонидович
Корбит Анатолий Григорьевич
Кривоносова Татьяна Михайловна
Навроцкий Анатолий Александрович
Шилин Дмитрий Леонидович

Основы программирования в среде *C++ Builder*

Лабораторный практикум по курсу
«Основы алгоритмизации и программирования»
для студентов 1 – 2-го курсов БГУИР
В 2-х частях
Часть 2

Редактор **Е. Н. Батурчик**

Подписано в печать.	Формат 60×84 1/16	Бумага офсетная.
Печать ризографическая.	Гарнитура «*Times*»	Усл. печ. л.
Уч. изд. л.	Тираж 200 экз.	Заказ

Издатель и полиграфическое исполнение:
Учреждение образования
«Белорусский государственный университет информатики и радиоэлектроники»
Лицензия ЛП №156 от 05.02.2001.
Лицензия ЛП №509 от 03.08.2001.
220013, Минск, П. Бровки, 6