Белорусский государственный технологический университет

Факультет информационных технологий

Кафедра программной инженерии

 Реферат

По дисциплине «Основы алгоритмизации и программирования»

На тему «Тестирование программ»

Выполнила:

Студент(ка) 1 курса 7 группы

Подшиваленко Диана Игоревна

Проверил:

Белодед Николай Иванович

2024, Минск

**Содержание**

1. [Тестирование кода бинарного дерева с рекурсией… 3](#_1.Тестирование_кода_бинарного)
2. [Тестирование кода бинарного дерева без рекурсии … 6](#_2.Тестирование_кода_бинарного)
3. [Тестирование кода хэширования с помощью леса…9](#_3.Тестирование_кода_хэширования)
4. [Тестирование кода древовидно-кольцевой структуры … 12](#_4.Тестирование_кода_древовидно-коль)
5. [Тестирование кода алгоритма Хаффмена … 21](#_5.Тестирование_кода_алгоритма)
6. [Тестирование кода построения дерева-формулы…25](#_6._Тестирование_кода)

## 1.Тестирование кода бинарного дерева с рекурсией

#include <iostream>

#include <Windows.h>

using namespace std;

struct node { // структура узла дерева

int key; // поле ключа

int count; // поле количества встречающихся элементом с этим ключом

node\* left; // указатель на левое поддерево

node\* right; // указатель на правое поддерево

};

class TREE { // класс дерева

private: // приватная область

node\* Tree; // создаем узел для дерева

void Search(int, node\*\*); // прототип функции поиска

public: // публичная область

TREE() { // конструктор

Tree = NULL; // присваиваем указателю узла дерева NULL

}

node\*\* GetTree() { // функция получения вершины дерева

return &Tree; // возвращаем вершину по ссылке

}

void BuildTree(); // прототип функции построения дерева

void CleanTree(node\*\*); // прототип функции очистки дерева

void ObhodEnd(node\*\*);// прототип функции концевого обхода дерева

void ObhodLeft(node\*\*); // прототип функции левостороннего обхода

void ObhodBack(node\*\*); // прототип функции обратного обхода

void Vyvod(node\*\*, int); // прототип функции вывода дерева

int Height(node\*\*); // прототип функции определения высоты дерева

};

int main(){

SetConsoleOutputCP(1251);

SetConsoleCP(1251);

TREE A; // создаем объект класса TREE

A.BuildTree(); // строим дерево

cout << "\nВывод дерева\n";

A.Vyvod(A.GetTree(), 0); // выводим дерево

cout << "\nВысота дерева " << A.Height(A.GetTree()) << '\n'; // выводим высоту дерева

cout << "\nЛевосторонний обход дерева\n";

A.ObhodLeft(A.GetTree()); // вызываем функцию левостороннего обхода

cout << "\nКонцевой обход дерева\n";

A.ObhodEnd(A.GetTree()); // вызываем функцию концевого обхода

cout << "\nОбратный обход дерева\n";

A.ObhodBack(A.GetTree()); // вызываем функцию обратного обхода

A.CleanTree(A.GetTree()); // вызываем функцию очистки дерева

cout << '\n';

}

void TREE::BuildTree() { // функция построения дерева

int el; // буферная переменная

cout << "Вводите ключи вершин дерева\n";

cin >> el; // считываем текущее значение

while (el != 0) { // пока элемент не равен 0(признак окончания ввода)

Search(el, &Tree); // ищем место для данной вершины

cin >> el; // считываем следующий

}

}

void TREE::Search(int x, node\*\* p) { // функция поиска места для вершины

if (\*p == NULL) { // если дерево пустое

\*p = new node; // выделяем память под нее

(\*\*p).key = x; // вводим текущий ключ

(\*\*p).count = 1; // указываем количество таких элементов

(\*\*p).left = NULL; // присваиваем указателю на левое поддерево значение NULL

(\*\*p).right = NULL; // присваиваем указателю на правое поддерево значение NULL

}

else { // иначе

if (x < (\*\*p).key) { // если заданный ключ меньше текущего

Search(x, &((\*\*p).left)); // ищем место для него в левом поддереве

}

else { // иначе

if (x > (\*\*p).key) { // если заданный ключ больше текущего

Search(x, &((\*\*p).right)); // ищем место для него в правом поддереве

}

else { // иначе

(\*\*p).count = (\*\*p).count + 1; // увеличиваем количество таких элементов

}

}

}

}

void TREE::ObhodLeft(node\*\* w) { // функция левостороннего обхода

if (\*w != NULL) { // если дерево не пустое

cout << (\*\*w).key << ' '; // выводим текущий ключ

ObhodLeft(&((\*\*w).left)); // вызываем рекурсивно эту функцию в левом поддереве

ObhodLeft(&((\*\*w).right));// вызываем рекурсивно эту функцию в правом поддереве

}

}

void TREE::ObhodEnd(node\*\* w) {// функция концевого обхода

if (\*w != NULL) {// если дерево не пустое

ObhodEnd(&((\*\*w).left));// вызываем рекурсивно эту функцию в левом поддереве

ObhodEnd(&((\*\*w).right));// вызываем рекурсивно эту функцию в правом поддереве

cout << (\*\*w).key << ' ';// выводим текущий ключ

}

}

void TREE::ObhodBack(node\*\* w) {// функция обратного обхода

if (\*w != NULL) {// если дерево не пустое

ObhodBack(&((\*\*w).left));// вызываем рекурсивно эту функцию в левом поддереве

cout << (\*\*w).key << ' ';// выводим текущий ключ

ObhodBack(&((\*\*w).right));// вызываем рекурсивно эту функцию в правом поддереве

}

}

void TREE::CleanTree(node\*\* w) { // функция очистки дерева

if (\*w != NULL) { // если дерево не пустое

CleanTree(&((\*\*w).left));// вызываем рекурсивно эту функцию в левом поддереве

ObhodBack(&((\*\*w).right));// вызываем рекурсивно эту функцию в правом поддереве

delete\* w; // освобождаем память

}

}

void TREE::Vyvod(node\*\* w, int l) { // функция вывода дерева

int i; // буферная переменная

if (\*w != NULL) { //если дерево не пустое

Vyvod(&((\*\*w).right), l + 1);// вызываем рекурсивно эту функцию в правом поддереве

for (i = 1; i <= l; i++) { // в цикле выводим пробелы

cout << " ";

}

cout << (\*\*w).key << '\n'; // выводим текущее значение

Vyvod(&((\*\*w).left), l + 1);// вызываем рекурсивно эту функцию в левом поддереве

}

}

int TREE::Height(node\*\* w) { // функция нахождения высоты дерева

int h1 = 0, h2 = 0; // буферные переменные

if (\*w == NULL) { // если дерево пустое

return(-1); // возвращаем -1

}

//иначе

h1 = Height(&((\*\*w).left)); // вызываем рекурсивно эту функцию в левом поддереве

h2 = Height(&((\*\*w).right)); // вызываем рекурсивно эту функцию в правом поддереве

if (h1 > h2) { // если высота в левом поддереве больше, чем в правом

return(1 + h1); // возвращаем ее + 1

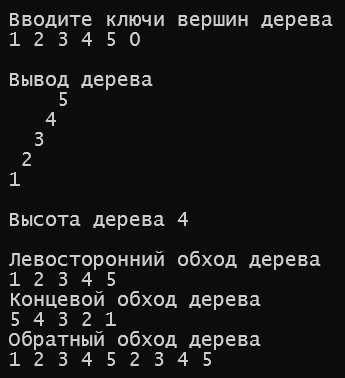
}

//иначе

return (1 + h2);// возвращаем высоту правой + 1

}

Результат выполнения:



## 2.Тестирование кода бинарного дерева без рекурсии

#include <iostream>

#include <Windows.h>

using namespace std;

struct node { // структура узла дерева

int Key;

int Count;

node\* Left;

node\* Right;

};

struct no {// Звено стека

node\* elem; // Информационное поле.

int ch; // Уровень вершины

no\* sled; // Указатель на вершину

};

class TREE{ // класс дерева

private: // приватная область

node\* Tree; // узел вершины дерева

void PushStack(no\*\*, node\*\*, int\*);// прототип функции добавления элемента

void PopStack(no\*\*, node\*\*, int\*); // прототип функции извлечения элемента

void VyvodStack(no\*\*); // вывод содержимого стека на

public: // публичная область

TREE() { // конструктор

Tree = new(node); // выделяем память

(\*Tree).Right = NULL; // указатель на правое поддерево NULL

}

node\* GetTreeRight() { // функция для возвращения узла правого поддерева

return (\*Tree).Right; // возвращаем вершину правого поддерева

}

void TreeSearch(int); // поиск вершины

void VyvodTree(node\*); // вывод дерева

};

void main()

{

SetConsoleCP(1251);

SetConsoleOutputCP(1251);

TREE A; // создаем объект класса TREE

int el; // буферная переменная

cout << "Вводите значения информационных полей вершин: " << endl;

cin >> el; // считываем текущее значение

while (el != 0){ // пока не встретился 0(признак конца ввода)

A.TreeSearch(el); // ищем место для вершины

cin >> el; // считываем следующий элемент

}

A.VyvodTree(A.GetTreeRight()); // выводим дерево

cout << "\n";

}

void TREE::TreeSearch(int el){ // поиск вершины

node\* p1, \* p2; // буферные указатели

int d; // буферная переменная

p2 = Tree; // ссылаем p2 на вершину

p1 = (\*p2).Right; // p1 на правое поддерево

d = 1; // присваиваем d = 1

while (p1 != NULL && d != 0){ // пока p1 не NULL и d не равно 0

p2 = p1; // переходим в правое поддерево

if (el < (\*p1).Key) { // если элемент меньше текущего

p1 = (\*p1).Left; // ссылаем p1 на левое поддерево

d = -1; // присваиваем d = -1

}

else // иначе

if (el > (\*p1).Key) { // если элемент больше текущего

p1 = (\*p1).Right; // ссылаем p1 на правое поддерево

d = 1; // присваиваем d = 1

}

else d = 0; //иначе d = 0

}

if (d == 0) { // если d = 0

(\*p1).Count = (\*p1).Count + 1; // увеличиваем количество таких элементов

}

else{ // иначе

p1 = new(node); // выделяем память

(\*p1).Key = el; // записываем в поле ключа переданное значение

(\*p1).Left = NULL; // ссылаем указатель на левое поддерево на NULL

(\*p1).Right = NULL; // ссылаем указатель на правое поддерево на NULL

(\*p1).Count = 1; // количество = 1

if (d < 0) { // если d < 0

(\*p2).Left = p1; // ссылаем указатель на левое поддерево на p1

}

else { // иначе

(\*p2).Right = p1; // ссылаем указатель на правое поддерево на p2

}

}

}

void TREE::VyvodTree(node\* t){ // функция вывода дерева

no\* stk, \* stk1; // буферные указатели

node\* u; // буферный указатель

int i, n; // буферные переменные

stk = stk1 = NULL; // присваиваем указателям NULL

n = 0; // присваиваем n = 0

while (t != NULL){ // пока дерево не пустое

PushStack(&stk1, &t, &n); // помещаем элемент в стек

if ((\*t).Right != NULL){ // если правое поддерево не пустое

if ((\*t).Left != NULL) // если левое поддерево не пустое

PushStack(&stk, &((\*t).Left), &n); // добавляем элемент в стек

t = (\*t).Right; // ссылаем t на правое поддерево

}

else{ // иначе

if ((\*t).Left != NULL){ // если левое поддерево не пустое

if (stk1 != NULL){ // если указатель stk1 не NULL

PopStack(&stk1, &u, &n); // извлекаем элемент из стека

for (i = 0; i <= n; i++) cout << " "; // в цикле выводим нужное количество пробелов

cout << (\*u).Key << endl; // выводим текущий ключ

}

t = (\*t).Left; // ссылаем t на левое поддерево

}

else // иначе

if (stk == NULL) // если stk = NULL

t = NULL; // присваиваем t = NULL

else{ // иначе

while ((\*stk).elem != (\*((\*stk1).elem)).Left){ // пока элементы не равны

PopStack(&stk1, &u, &n); // извлекаем элемент из стека

for (i = 0; i <= n; i++) cout << " "; // выводим нужное количество пробелов

cout << (\*u).Key << endl; // выводим текущее значение ключа

}

PopStack(&stk1, &u, &n); // извлекаем вершину из стека

for (i = 0; i <= n; i++) cout << " ";// выводим нужное количество пробелов

cout << (\*u).Key << endl;// выводим текущее значение ключа

PopStack(&stk, &t, &n);// извлекаем вершину из стека

}

}

n = n + 1; // увеличиваем n на 1

}

VyvodStack(&stk1); // выводим стек

}

void TREE::PushStack(no\*\* stk, node\*\* el, int\* n){ // функция добавление элемента в стек

no\* q; // буферный указатель

q = new(no); // выделяем память

(\*q).elem = \*el; // записываем переданное значение элемента

(\*q).ch = \*n; // записываем текущую высоту вершины

(\*q).sled = \*stk; // ссылаем следующий элемент на stk

\*stk = q; // ссылаем stk на q

}

void TREE::PopStack(no\*\* stk, node\*\* t, int\* n){ // функция извлечения элемента

no\* q;// буферный указатель

if (\*stk != NULL){ // если стек не пустой

\*t = (\*\*stk).elem; // ссылаем t на элемент stk

\*n = (\*\*stk).ch; // присваиваем n значение высоты текущего элемента

q = \*stk; // ссылаем q на stk

\*stk = (\*\*stk).sled; // ссылаем stk на следующий элемент

delete q; // освобождаем память

}

}

void TREE::VyvodStack(no\*\* stk){ // вывод стека

node\* k; // буферный узел

int i, n; // буферные переменные

while (\*stk != NULL){ // пока стек не пуст

k = (\*\*stk).elem; // присваиваем k текущее значение элемента

n = (\*\*stk).ch; // присваиваем n текущую высоту

for (i = 0; i <= n; i++) cout << " "; // выодим нужное число пробелов

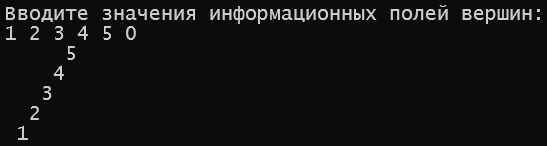
cout << (\*k).Key << endl; // выводим текущий ключ

\*stk = (\*\*stk).sled; // переходим на следующий элемент

}

}

Результат выполнения:



## 3.Тестирование кода хэширования с помощью леса

#include <time.h>

#include <iostream>

using namespace std;

#define N 10 //Количество элементов массива.

struct node{ // структура узла

int Key; // поле ключа

int Count; // поле количества

node\* Left; // указатель на левое поддерево

node\* Right; // указатель на правое поддерево

};

class Spisok { // класс списка

private: // приватная область

node\* UkStr[N]; // массив узлов таблицы

void Search(int, node\*\*); // функция поиска

void PrintTree(node\*, int); // функция вывода дерева

void U\_d(node\*\*, node\*\*); //

public: // публичная область

Spisok(); // конструктор

void BuildTree(); // прототип функции построения дерева

void Sodergimoe(); // функция показа содержимого дерева

node\*\* GetTree(unsigned i) { // функция получения элемента дерева

return &(UkStr[i]); // возвращаем элемент

}

void Udaldr(node\*\* d, int k); // функция удаления

};

Spisok::Spisok(){// конструктор списка

for (int i = 0; i < N; i++) UkStr[i] = NULL; // присваиваем всем элементам NULL

}

void Spisok::BuildTree(){ // функция построения дерева

int klutch; // ключ

unsigned hash; // хэш

// автоматическая рандомизация

cout << "\nВведите значение ключа...";

cin >> klutch;

while (klutch != 0){ // пока не встретился 0 (признак конца ввода)

hash = klutch % 10; //вычисление значения хэш-функции

Search(klutch, &UkStr[hash]); // ищем место

cout << "\nВведите значение ключа...";

cin >> klutch; // считываем следующий ключ

}

}

void Spisok::Search(int X, node\*\* p){ // функция поиска элемента

if (\*p == NULL){ // список пуст

\*p = new (node); // выделяем память

(\*\*p).Key = X; // записываем текущий ключ

(\*\*p).Count = 1; // записываем количество 1

(\*\*p).Left = (\*\*p).Right = NULL; // ссылаем указатели на правое и левое поддерево на NULL

}

else // иначе

if (X < (\*\*p).Key) // если заданный ключ меньше текущего

Search(X, &((\*\*p).Left)); // ищем место для него в левом поддереве

else if (X > (\*\*p).Key) // если заданный ключ больше текущего

Search(X, &((\*\*p).Right));// ищем место для него в правом поддереве

else // иначе

(\*\*p).Count += 1; // увеличиваем количество

}

void Spisok::Sodergimoe(){ // вывод содержимого

for (int i = 0; i < N; i++) { // в цикле

cout << " " << i << "... "; // выводим индекс элемента списка

if (UkStr[i] == NULL) cout << "Деpево пусто...\n"; // если текущее дерево пустое, то выводим сообщение об этом

else // иначе

{

cout << endl;

PrintTree(UkStr[i], 0); // вызываем функцию вывода текущего дерева

}

cout << "------------------------------------------" << endl;

}

}

void Spisok::PrintTree(node\* w, int l){ // функция вывода дерева

if (w != NULL){ // если дерево не пустое

PrintTree((\*w).Right, l + 1); // выводим правое поддерево

cout << " ";

for (int i = 1; i <= l; i++) cout << " "; // выводим нужное количество пробелов

cout << (\*w).Key << endl; // выводим текущий ключ

PrintTree((\*w).Left, l + 1); // выводим левое поддерево

}

}

void Spisok::Udaldr(node\*\* d, int k){ // удалить узел

node\*\* q; // буферный указатель

if (\*d == NULL) // если узел не найден

cout << "Узел с заданным ключом в деpеве не найден...\n";

else // иначе

if (k < (\*\*d).Key) // если ключ меньше текущего

Udaldr(&((\*\*d).Left), k); // вызываем рекурсивно функцию для левого поддерева

else // иначе

if (k > (\*\*d).Key) // если ключ меньше текущего

Udaldr(&((\*\*d).Right), k); // вызываем рекурсивно функцию для правого поддерева

else{ // иначе

q = d; // присваиваем q значение d

if ((\*\*q).Right == NULL) // если правое поддерево пустое

\*d = (\*\*q).Left; // ссылаем d на левое поддерево

else // иначе

if ((\*\*q).Left == NULL) // если левое поддерево пустое

\*d = (\*\*q).Right; // ссылаем d на правое поддерево

else // иначе

U\_d(&((\*\*q).Left), &(\*q)); // вызываем функцию удаления узла

}

}

void Spisok::U\_d(node\*\* r, node\*\* q){ // функция удаления конкретного узла

if ((\*\*r).Right == NULL){ // если правое поддерево пустое

(\*\*q).Key = (\*\*r).Key; // записываем в поле для ключа узла q значение ключа указателя r

(\*\*q).Count = (\*\*r).Count; // переписываем количество

q = r; // ссылаем q на r

\*r = (\*\*r).Left; // переходим на левое поддерево

delete (\*q); // освобождаем память

}

else

U\_d(&((\*\*r).Right), &(\*q)); // удалить узел в правом поддереве

}

void main()

{

setlocale(LC\_ALL, "Rus");

Spisok A; // создаем объект класса Spisok

int klutch; // переменная ключа

unsigned hash; // переменная для хэша

A.BuildTree(); // строим дерево

cout << "\n Содеpжимое хэш-списка...";

cout << "\n -----------------------------------\n";

A.Sodergimoe(); // выводим содержимое

for (int i = 0; i < 4; i++)

{

cout << "\nВведите значение удаляемого ключа...";

cin >> klutch; // читаем ключ

hash = klutch % 10; // находим хэш

A.Udaldr(A.GetTree(hash), klutch); // удаляем элемент

cout << " Содеpжимое хэш-списка...\n";

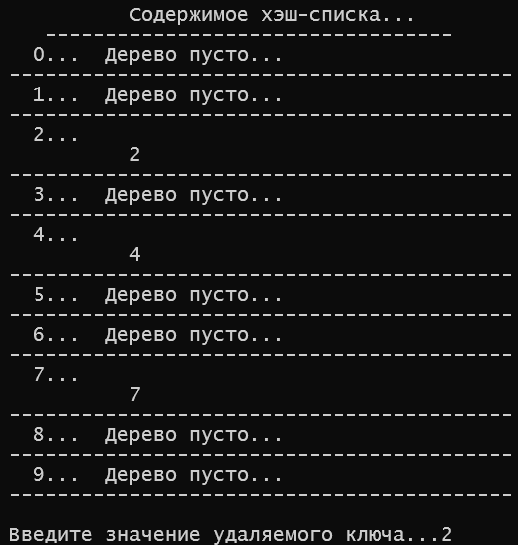
cout << " ----------------------------------\n";

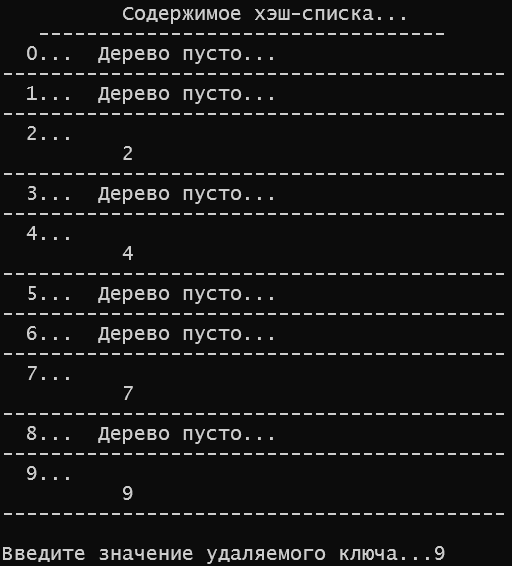
A.Sodergimoe(); // выводим содержимое

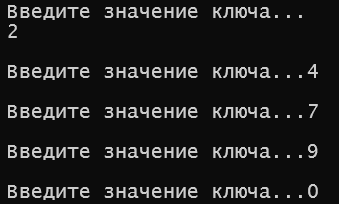
}

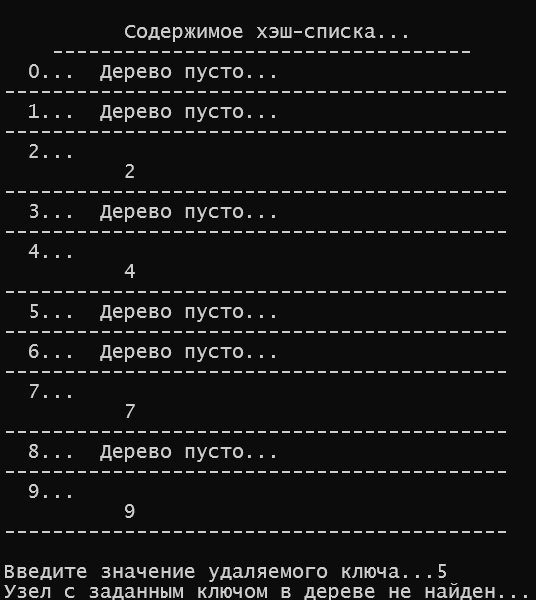
cout << "\n";

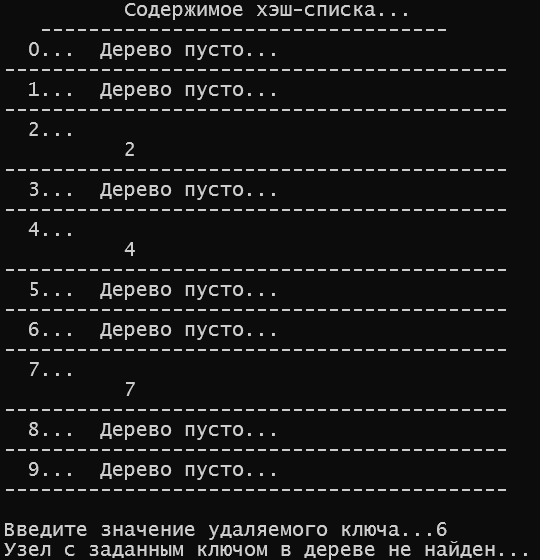
system("PAUSE");

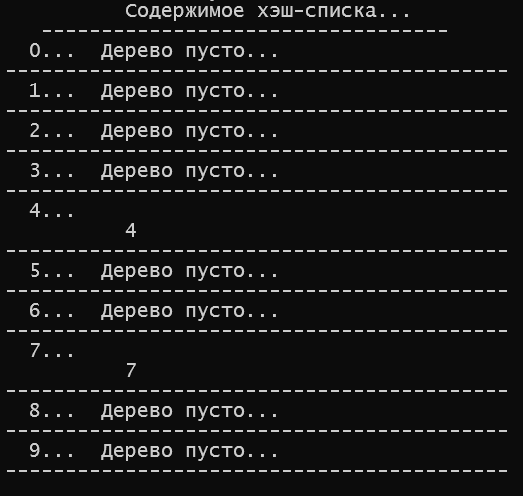
}

Результат выполнения:









## 4.Тестирование кода древовидно-кольцевой структуры

#include <conio.h>

#include <iostream>

using namespace std;

struct node{ // структура узла

int key; // поле для ключа

int count; // поле количества

node\* Left; // указатель на левое поддерево

node\* Right; // указатель на правое

};

class Tree{ // класс дерева

private: // приватная область

node\* root; //корень дерева

void DisposeTree(node\*); // освобождаем память

void printTree(node\*, int); // вывод дерева

void Delete(node\*\*, int); // удаление элемента

void del(node\*\*, node\*); // вспомогательная функция

public: // публичная оболасть

Tree() { // конструктор

root = NULL; // корень равен NULL

};

~Tree(); // деструктор

void creat\_Tree(); // создание дерева

void look\_Tree(); // вывод дерева

void add\_Tree(); // добавление элемента в дерево

void delete\_Tree(); // удаление элемент

void search(int, node\*\*); // поиск

node\* getTree() { // получение вершины дерева

return root; // возвращаем вершину дерева

};

};

struct zveno{ // структура звена

int element; // элемент

Tree ukTree; // объект дерева

zveno\* sled; // указатель на следующий

};

class ring{ // класс кольца

private: // приватная зона

zveno\* ukring; // звено кольца

public: // публичная область

ring() { // конструктор

ukring = NULL; // присваиваем NULL

};

~ring(); // деструктор

void create(); //построение кольца

void look(); //вывод кольца

void add\_after(int, zveno\*); // добавить после

void add\_befor(int, zveno\*); // добавить до

void Delete(zveno\*); // удалить

void delete\_next(zveno\*); // удалить следующий

int poisk(int, zveno\*\*); // поиск

zveno\*\* getring() { // получение кольца

return &ukring; // возвращаем ссылку на кольцо

};

};

void ring::create(){ // построение кольца

zveno\* ukzv; // буферное звено

int elem; // буферный элемент

cout << "\nПостроение кольца ..." << endl;

cout << "Вводите элементы кольца (ввод окончите 0): \n";

cout << "-->";

cin >> elem; // считываем элемент

if (elem != 0){ // если элемент не равен 0

ukzv = ukring = new (zveno); // выделяем память под новое кольцо

(\*ukzv).element = elem; // записываем в поле элемента заданный элемент

(\*ukzv).ukTree.creat\_Tree(); // создаем дерево

cout << "\n-->";

cin >> elem; // считываем следующий элемент

while (elem != 0){ // пока элемент не равен 0

(\*ukzv).sled = new (zveno); // выделяем память под следующий элемент

ukzv = (\*ukzv).sled; // ссылаем на следующий

(\*ukzv).element = elem; // записываем в поле элемента заданный элемент

(\*ukzv).ukTree.creat\_Tree(); // создаем дерево

cout << "\n-->";

cin >> elem; // читаем следующий

}

ukzv->sled = ukring; // ссылаем следующий на ukring

}

}

ring::~ring(){ //деструктор

zveno\* ukzv; // буферное звено

ukzv = ukring; // присваиваем указателю адрес ukring

while (ukring != NULL) // пока не конец

if (ukzv->sled == ukring) { // если следующий элемент равен ukring

ukring = NULL; // присваиваем ему NULL

ukzv->ukTree.~Tree(); // вызываем деструктор дерева

delete ukzv; // освобождаем память

}

else{ // иначе

while (ukzv->sled->sled != ukring) // пока следующий элемент не равен ukring

ukzv = (\*ukzv).sled; // ссылаем на следующий

(\*ukzv).sled->ukTree.~Tree(); // вызываем деструктор

delete (\*ukzv).sled; // очищаем память

ukzv->sled = ukring; // ссылаем следующий на ukring

ukzv = ukring; // ссылаем текущий на ukring

}

}

void ring::look() { // вывод кольца

zveno\* ukzv; // буферное звено

cout << "\nВывод содержимого кольца ...";

ukzv = ukring; // ссылаем на ukring

do { // делаем

cout << "\n-->" << (\*ukzv).element << endl; // выводи элемент

ukzv->ukTree.look\_Tree(); // вызываем функцию вывода

ukzv = ukzv->sled; // ссылаем на следующий

getch();

} while (ukzv != ukring); // пока не равен ukring

cout << endl;

}

void ring::add\_befor(int elem, zveno\* zv){ // добавить перед

zveno\* ukzv; // буферное звено

Tree temp; // буферное дерево

ukzv = new (zveno); // выделяем память

temp = ukzv->ukTree; // ссылаем темп на ukTree

ukzv->element = zv->element; // присваиваем полю элемента значение этого элемента

ukzv->ukTree = zv->ukTree; // присваиваем указателю на ukTree буферного звена само звено

ukzv->sled = zv->sled; // присваиваем указателю на следующий буферного звена следующий элемент звена

zv->element = elem; // записываем заданный элемент в текущее звено

zv->ukTree = temp; // присваиваем дереву наше буферное дерево

zv->ukTree.creat\_Tree(); // создавем новое дерево

zv->sled = ukzv; // указываем следующий элемент

}

void ring::add\_after(int elem, zveno\* zv){ // добавить после

zveno\* ukzv;// буферное звено

ukzv = new (zveno);// выделяем память

ukzv->element = elem;// записываем заданный элемент в текущее звено

ukzv->ukTree.creat\_Tree();// создавем новое дерево

ukzv->sled = zv->sled;// указываем следующий элемент

zv->sled = ukzv; // указываем следующий элемент

}

void ring::Delete(zveno\* zv){ // удалить элемент

zveno\* ukzv1, \* ukzv2, \*time; // буферные звенья

if (zv->sled != ukring){ // если следующий за заданным звеном не равен ukring

time = zv->sled; // ссылаем буферный указатель на следующий элемент заданного звена

zv->ukTree.~Tree(); // вызываем деструктор

(\*zv) = \*((\*zv).sled);// переписываем элемент звена

delete time; // освобождаем память

}

else // иначе

if (zv->sled == zv){ // если следующий элемент равен заданному

zv->ukTree.~Tree(); // вызываем деструктор

delete ukring;// освобождаем память

ukring = NULL; // ссылаем главный элемент кольца на NULL

cout << "Список пуст...\n";

}

else{ // иначе

ukzv2 = ukring; // ссылаем буферный указатель на главный элемент кольца

ukzv1 = ukring->sled; // ссылаем другой буферный указатель на следующий элемент за главным

while (ukzv1 != zv){ // пока не достигнем искомого звена

ukzv2 = ukzv1; // переходим на следующий

ukzv1 = ukzv1->sled; // переходим на следующий

}

time = ukzv2->sled; // буферному указателю присваиваем значение следующего элемента

ukzv2->sled->ukTree.~Tree(); // вызываем деструктор

ukzv2->sled = ukzv2->sled->sled; // устанавливаем новые связи

delete time; // освобождаем память

}

}

void ring::delete\_next(zveno\* zv){ // удаление следующего элемента после заданного

zveno\* time; // буферный указатель

if (zv->sled != ukring){ // если следующий после заданного равен главному

time = zv->sled; // буферный указатель ссылаем на него

zv->sled = zv->sled->sled; // устанавливаем новые связи

time->ukTree.~Tree(); // вызываем деструктор

delete time;// освобождаем память

}

else // иначе

if (zv->sled == zv) // если следующий после заданного равен самому себе, то есть один элемент в кольце

cout << "В кольце только один элемент!\n"; // выводим сообщение об этом

else{ // иначе

time = ukring->sled; // ссылаем буферный указатель на следующий после заданного

\*((\*zv).sled) = (\*(\*(\*zv).sled).sled); // устанавливаем новые связи

time->ukTree.~Tree(); // вызываем деструктор

delete time; // освобождаем память

}

}

int ring::poisk(int elem, zveno\*\* Res){ // поиск элемента

zveno\* ukzv; // буферное звено

int vozvr = 0; // буферная переменная

if (\*(getring()) == NULL) // если кольца нет

cout << "Кольцо не существует...\n"; // выводим сообщение об этом

else{ // иначе

ukzv = ukring; // ссылаем буферный указатель на главное звено

while (ukzv->sled != ukring && (\*Res) == NULL){ // пока не вернулись к заглавному звену и пока элемент не найден

if (ukzv->element == elem){ // если найден нужный элемент

vozvr = 1; // присваиваем буферной переменной 1

\*Res = ukzv; // ссылаем Res на найденный элемент

}

ukzv = ukzv->sled; // переходим на следующий

}

if ((\*Res) == NULL) // если элемент не найден

if (ukzv->element == elem){ // если последний элемент нужный

vozvr = 1; // присваиваем буферной переменной 1

\*Res = ukzv;// ссылаем Res на найденный элемент

}

}

return vozvr; // возращаем значение буферной переменной

}

Tree::~Tree(){ // деструктор

DisposeTree(root); // вызываем функцию очищения дерева

root = NULL; // ссылаем вершину на NULL

}

void Tree::DisposeTree(node\* p){ // очищение дерева

if (p != NULL){ // если дерево не пустое

DisposeTree(p->Left); // вызываем эту функцию от левого поддерева

DisposeTree(p->Right); // вызываем эту функцию от правого поддерева

delete p; // очищаем память

}

}

void Tree::search(int x, node\*\* p){ // функция поиска

if (\*p == NULL){ // если дерево пустое

\*p = new (node); // выделяем память для нового узла

(\*\*p).key = x; // записываем заданный ключ

(\*\*p).count = 1; // записываем количество 1

(\*\*p).Left = (\*\*p).Right = NULL; // ссылаем указатели на право и на левое поддеревья на NULL

}

else // иначе

if (x < (\*\*p).key) // если заданный ключ меньше текущего

search(x, &((\*\*p).Left)); // ищем место в левом поддереве

else // иначе

if (x > (\*\*p).key) // если заданный ключ больше текущего

search(x, &((\*\*p).Right)); // ищем место в правом поддереве

else // иначе

(\*\*p).count += 1; // увеличиваем количество таких элементов

}

void Tree::creat\_Tree(){

int elem;

cout << "Вводите ключи узлов дерева (ввод окончите 0):\n";

cin >> elem;

while (elem != 0)

{

search(elem, &root);

cin >> elem;

}

}

void Tree::look\_Tree(){ // функция вывода

if (root == NULL) // если дерево пустое

cout << "Дерево пусто ...\n"; // выводим сообщение об этом

else // иначе

printTree(root, 0); // вызываем функцию вывода дерева

}

void Tree::printTree(node\* w, int L){ // вывод дерева

if (w != NULL){ // если дерево не пустое

printTree(w->Left, L + 1); // выводим его левое поддерево

for (int i = 1; i <= L; i++) cout << " "; // выводим нужное число пробелов

cout << w->key << endl; // выводим текущий ключ

printTree(w->Right, L + 1); // выводим правое поддерево

}

}

void Tree::add\_Tree(){ // добавить дерево

int k; // буферная переменная

cout << "\nВводите ключи добавляемых узлов (ввод окончите 0):\n";

cin >> k; // считываем текущее значение

cout << " ";

while (k != 0){ // пока не 0(признак окончания ввода)

search(k, &(root)); // ищем место для текущего элемента

cin >> k; // читаем следующий

cout << " ";

}

}

void Tree::delete\_Tree(){ // удалить дерево

int elem; // буферная переменная

if (root == NULL) // если дерево пустое

cout << "Дерево пусто ...\n"; // выводим сообщение об этом

else{ // иначе

cout << "Введите ключ удаляемого узла : ";

cin >> elem; // считываем ключ удаляемого узла

cout << endl;

Delete(&root, elem); // удаляем

}

}

void Tree::Delete(node\*\* d, int k){ // функция удаления элемента в дереве

node\* q, \*s; // буферные узлы

if (\*d == NULL) // если дерево пусто

cout << "Узел с заданным ключом в дереве не найден ...\n";

else // иначе

if (k < (\*\*d).key) // если заданный элемент меньше текущего

Delete(&((\*\*d).Left), k); // рекурсивно вызываем эту функцию для левого поддерева

else // иначе

if (k > (\*\*d).key) // если заданный элемент больше текущего

Delete(&((\*\*d).Right), k); // рекурсивно вызываем эту функцию для правого поддерева

else{ // иначе

q = \*d; // ссылаем на вершину

s = \*d; // ссылаем туда же

if ((\*q).Right == NULL){ // если правое поддерево пустое

\*d = (\*q).Left; // ссылаем на левое поддерево

delete s; // очищаем память

}

else // иначе

if ((\*q).Left == NULL){ // если левое поддерево пустое

\*d = (\*q).Right; // ссылаем на правое поддерево

delete s; // очищаем память

}

else // иначе

del(&((\*q).Left), &(\*q)); // вызываем функцию удаления от левого поддерева

}

}

void Tree::del(node\*\* r, node\* q){ // функция удаления узла

node\* s; // буферный узел

if ((\*\*r).Right == NULL){ // если правое поддерево пустое

(\*q).key = (\*\*r).key; // меняем значение ключа

(\*q).count = (\*\*r).count; // меняем количество

q = s = \*r; // ссылаем указатели на r

\*r = (\*\*r).Left; // ссылаем на левое поддерево

delete s; // очищаем память

}

else // иначе

del(&((\*\*r).Right), &(\*q)); // вызываем функцию от правого поддерева

}

void main()

{

setlocale(LC\_ALL, "Rus");

int menu1 = 1, choice, elem1, elem2, menu2; //буферные переменные

ring A; // создаем объект класса ring

zveno\* Res; // создаем звено

cout << "<------------- Структура --------------->\n";

cout << "<---------\"кольцо с деревьями\"---------->\n\n";

while (menu1){ // цикл для выбора опции

cout << endl;

cout << "<---------- Главное меню 1.0 : --------->\n";

cout << "1. Построение структуры.................. \n";

cout << "2. Просмотр структуры.................... \n";

cout << "3. Добавление элемента после указанного.. \n";

cout << "4. Добавление элемента перед указанным... \n";

cout << "5. Удаление элемента..................... \n";

cout << "6. Удаление элемента после указанного.... \n";

cout << "7. Преобразование дерева заданного эл-та. \n";

cout << "8. Удаление структуры.................... \n";

cout << "9. Выход................................. \n";

cout << "Введите номер режима и нажмите <Enter> : ";

cin >> choice; // считываем опцию

cout << endl;

switch (choice){

case 1:

if (\*(A.getring()) == NULL) // если кольцо пустое

A.create(); // создаем кольцо

else // иначе

cout << "Кольцо уже существует...\n";

break;

case 2:

if (\*(A.getring()) == NULL) // если кольцо пустое

cout << "Кольцо пусто...\n";

else // иначе

A.look(); // выводим кольцо

break;

case 3:

if (\*(A.getring()) == NULL) // если кольцо пустое

cout << "Кольцо пусто...\n";

else{ // иначе

Res = NULL; // присваиваем Res NULL

cout << "Введите элемент, после которого ";

cout << " хотите добавить звено: ";

cin >> elem1; // считываем элемент

cout << endl;

if (A.poisk(elem1, &Res)) { // если нашли

cout << "Введите элемент, который ";

cout << "хотите добавить: ";

cin >> elem2; // считываем элемент

cout << endl;

A.add\_after(elem2, Res); // добавляем после

}

else // иначе

cout << "Элемент " << elem1 << " не найден.\n";

}

break;

case 4:

if (\*(A.getring()) == NULL) // если кольцо пустое

cout << "Кольцо пусто...\n";

else{ //иначе

Res = NULL;// присваиваем Res NULL

cout << "Введите элемент, перед которым ";

cout << " хотите добавить звено: ";

cin >> elem1; // считываем элемент

cout << endl;

if (A.poisk(elem1, &Res)){// если нашли

cout << "Введите элемент, который ";

cout << "хотите добавить: ";

cin >> elem2;// считываем элемент

cout << endl;

A.add\_befor(elem2, Res);// добавляем перед

}

else // иначе

cout << "Элемент " << elem1 << " не найден.\n";

}

break;

case 5:

if (\*(A.getring()) == NULL) // если кольцо пустое

cout << "Кольцо пусто...\n";

else{

Res = NULL;// присваиваем Res NULL

cout << "Введите элемент, который";

cout << " хотите удалить: ";

cin >> elem1; // считываем элемент

cout << endl;

if (A.poisk(elem1, &Res))// если нашли

A.Delete(Res); // удаляем

else // иначе

cout << "Элемент отсутствует...\n";

}

break;

case 6:

if (\*(A.getring()) == NULL) // если кольцо пустое

cout << "Кольцо пусто...\n";

else{ // иначе

Res = NULL;// присваиваем Res NULL

cout << "Введите элемент, после которого";

cout << " хотите удалить: ";

cin >> elem1; // считываем элемент

cout << endl;

if (A.poisk(elem1, &Res))// если нашли

A.delete\_next(Res); // удаляем следующий

else cout << "Элемент отсутствует...\n";

}

break;

case 7:

if (\*(A.getring()) == NULL) // если кольцо пустое

cout << "Кольцо пусто...\n";

else{ // иначе

Res = NULL;// присваиваем Res NULL

cout << "Введите элемент кольца: ";

cin >> elem1; // считываем элемент

cout << endl;

if (A.poisk(elem1, &Res)){ // если нашли

menu2 = 1;

while (menu2){ // цикл для выбора опций для дерева

cout << endl;

cout << "<---------- Mеню 1.1 : --------->\n";

cout << "1. Построение дерева.............\n";

cout << "2. Просмотр дерева...............\n";

cout << "3. Добавление элемента в дерево..\n";

cout << "4. Удаление элемента из дерева...\n";

cout << "5. Удаление дерева...............\n";

cout << "6. Выход в главное меню..........\n";

cout << "Введите номер режима и нажмите <Enter>: ";

cin >> choice; // считываем выбор

cout << endl;

switch (choice) {

case 1:

if ((\*Res).ukTree.getTree() == NULL) // если дерево пустое

(\*Res).ukTree.creat\_Tree(); // создаем дерево

else // иначе

cout << "Дерево существует...\n";

break;

case 2: (\*Res).ukTree.look\_Tree(); break; // выводим дерево

case 3: (\*Res).ukTree.add\_Tree(); break; // добавляем дерево

case 4: (\*Res).ukTree.delete\_Tree(); break; // удаляем дерево

case 5:

if ((\*Res).ukTree.getTree() == NULL) // если дерева нет

cout << "Дерево не существует...\n";

else // иначе

(\*Res).ukTree.~Tree(); // вызываем деструктор

break;

case 6: menu2 = 0; break;

}

}

}

else // иначе

cout << "Элемент " << elem1 << " не найден.\n";

}

break;

case 8:

if (\*(A.getring()) == NULL) // если кольцо пустое

cout << "Кольцо пусто...\n";

else // иначе

A.~ring(); // вызываем деструктор

break;

case 9:

A.~ring(); // вызываем деструктор

menu1 = 0;

break;

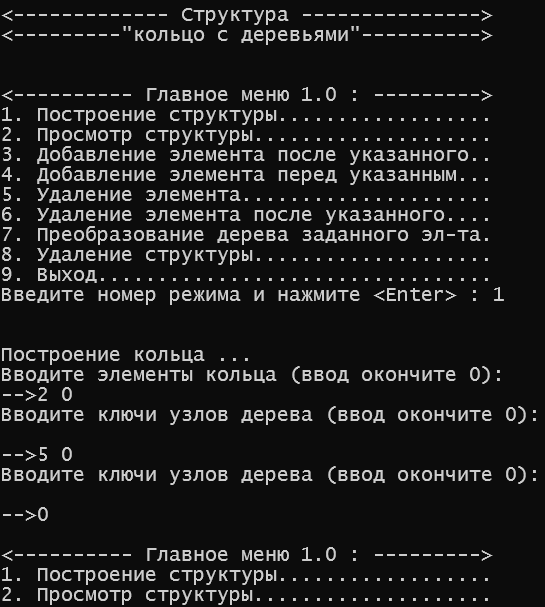
} //End Case

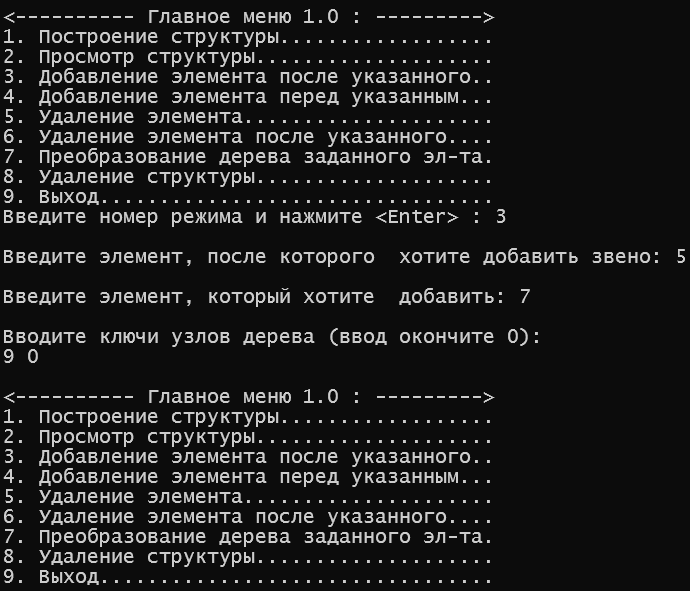
} //End while

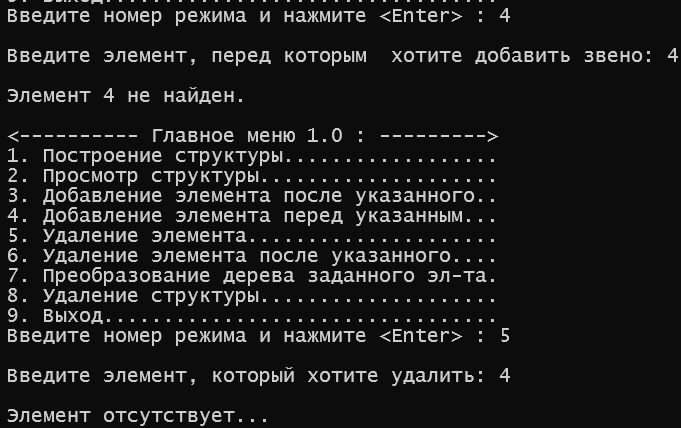
cout << "\n";

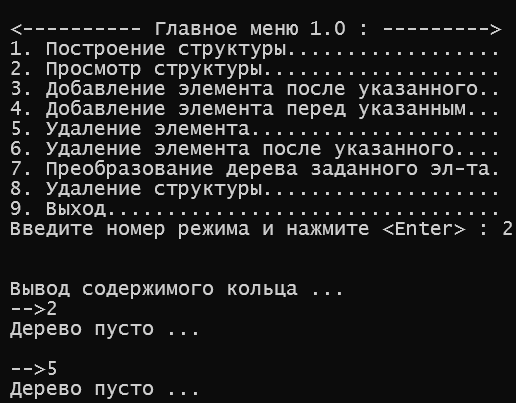
system("PAUSE");

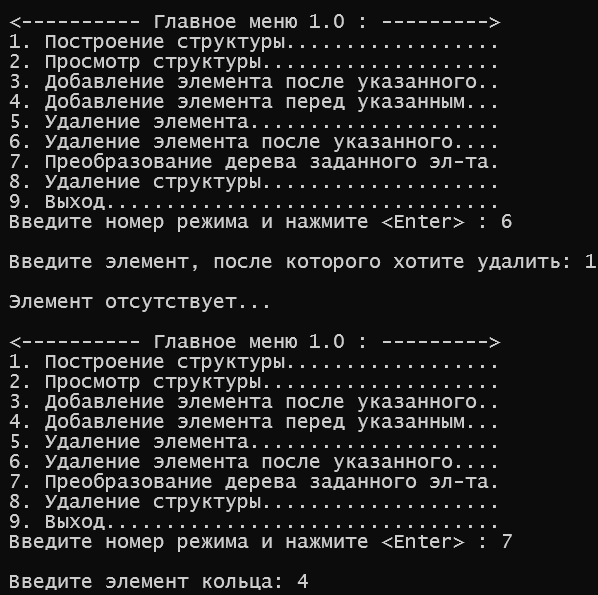
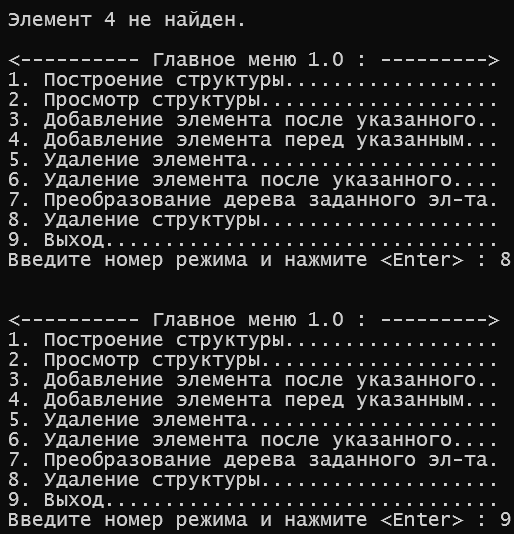
}

Результат выполнения:









## 5.Тестирование кода алгоритма Хаффмена

#include <stdio.h>

#include <conio.h>

#include <iostream>

using namespace std;

struct zveno{ // структура звена

char Element; // поле элемента

float Kol; //количество

zveno\* Sled; // следующий

zveno\* Left; // левый

zveno\* Right; // правый

zveno\* Father; // предок

};

class Tree{ // класс дерева

private: // приватная область

zveno\* UkStr; //указатель на список

int Poisk1(zveno\*\*, float, zveno\*\*); // прототип функции поиска

public: // публичная область

Tree() { // конструктор

UkStr = new (zveno); // выделяем память

UkStr->Sled = NULL; // следующий NULL

};

int Poisk(char, zveno\*\*); // прототип функции поиска

int Kolich(char\*, char); // прототип функции подсчета количетва

void Dobavlenie(char, float, zveno\*\*); // прототип функции добавления

void Redaktor(int); // прототип функции редактирования

void Ukazateli(zveno\*\*, zveno\*\*); // прототип функции поиска указателей

void Vyvod(); // протип функции вывода

void WstawkaSort(zveno\*); // прототип функции сортировки

void PrintTree(zveno\*, int); // прототип функции вывода дерева

zveno\*\* GetTree() { // получить дерево

return &UkStr; // возвращаем ссылку

};

zveno\* GetTree1() { // получить дерево

return UkStr; // возвращаем по значению

};

};

int Tree::Poisk(char ENT, zveno\*\* Res){ // функция поиска

zveno\* q; // буферное звено

int vozvr = 0; // буферная переменная

\*Res = NULL; // указаетел на искомый элемент изначально NULL

q = (\*UkStr).Sled; // ссылаем на следующий

while (q != NULL && \*Res == NULL){ // пока не конец и пока не нашли нужный

if (q->Element == ENT){ //если нашли

vozvr = 1; // помечаем как найденный

\*Res = q; // ссылаем Res на найденный элемент

return vozvr; // возвращаем

}

q = q->Sled; // переходим на следующий

}

return vozvr; // возвращаем

}

int Tree::Poisk1(zveno\*\* st, float kol, zveno\*\* Res){ // поиск места для элемента

zveno\* q = (\*\*st).Sled, \* q1 = (\*st); // ссылаем на следующий

int vozvr = 0; // буферная переменная

\*Res = NULL; // указатель для найденного элемента

while (q != NULL && \*Res == NULL){// пока не конец и пока не нашли нужный

if (q->Kol < kol) { // если колчество в текущем меньше заданного

vozvr = 1; // помечаем как найденный

\*Res = q; // ссылаем Res на найденный элемент

}

q = q->Sled; // переходим на следующий

q1 = q1->Sled; // переходим на следующий

}

if (\*Res == NULL) // если не найден

\*Res = q1; // ссылаем на q1

return vozvr; //возвращаем

}

int Tree::Kolich(char\* F, char S){ // подсчет количества повторений

int K = 0; // буферная переменная

for (int i = 0; i < strlen(F); i++) // в цикле проходим по строке

if (F[i] == S) K++; // если наш элемент, то увеличваем счетчик

return K; // возвращаем количество

}

void Tree::Redaktor(int L){ // редактирование

zveno\* q = (\*UkStr).Sled; // ссылаем на следующий

while (q != NULL){ // пока не конец

q->Kol = q->Kol / L; // изменяем количество

q = q->Sled; // переходим на следующий

}

}

void Tree::Dobavlenie(char bukva, float kol, zveno\*\* Sp){ //функция добавления

zveno\* q, \* Res = NULL, \* kladovaq; // буферные указатели

q = new (zveno); // выделяем память

q->Element = bukva; // записываем заданную букву

q->Kol = kol; // записываем заданное количество

q->Left = q->Right = NULL; // ссылаем левый и правый на NULL

q->Sled = q->Father = NULL;// следующий и предка на NULL

if ((\*\*Sp).Sled == NULL) // если конец

(\*\*Sp).Sled = q; // то следующий q

else // иначе

if (Poisk1(&(\*Sp), kol, &Res)) { // если найдено место

kladovaq = new (zveno); // выделяем память

(\*kladovaq) = (\*Res); // ссылаем на Res

(\*Res) = (\*q); // ссылаем на q

Res->Sled = kladovaq; // присваиваем следующему kladovaq

}

else // иначе

Res->Sled = q; // присваиваем следующему q

}

void Tree::Ukazateli(zveno\*\* zv, zveno\*\* zv\_p){ // поиск указателей

\*zv\_p = UkStr->Sled; // ссылаем на следующий за главным

\*zv = UkStr; // ссылаем на главный

while ((\*zv\_p)->Sled->Sled != NULL) { // пока следующий за следующим не NULL

\*zv = \*zv\_p; // переходим на следующий

\*zv\_p = (\*zv\_p)->Sled; // переходим на следующий

}

}

void Tree::Vyvod(){ // вывод

zveno\* q = UkStr->Sled; // ссылаем на следующий элемент за главным

while (q != NULL){ // пока не конец

cout << q->Element << " (" << q->Kol << ") --> "; // выводим элемент и количество

q = q->Sled; // переходим на следующий

}

cout << endl;

}

void Tree::WstawkaSort(zveno\* zv){ // сортировка

zveno\* w1, \* w2; // буферные звенья

w2 = UkStr; // сылаем на главное звено

w1 = w2->Sled; // ссылаем на следующее

while (w1 != NULL && w1->Kol > zv->Kol){ // пока не конец и пка количество не найдено

w2 = w1; // переходим на следующий

w1 = w2->Sled; // переходим на следующий

}

if (w1 == NULL || w1->Kol <= zv->Kol){ // если дошли до конца или количество меньше или равно

w2->Sled = zv; // следующий равен zv

zv->Sled = w1; // следующий за zv будет текущий элемент

}

}

void Tree::PrintTree(zveno\* w, int l){ // вывод дерева

if (w != NULL){ // пока не конец

PrintTree(w->Right, l + 1); // выводим правое поддерево

for (int i = 1; i <= l; i++) cout << " "; // выводим нужное число пробелов

cout << w->Element << " (" << w->Kol << ")\n"; // выводим элемент и количество

PrintTree(w->Left, l + 1); // выводим левое поддерево

}

}

void main()

{

setlocale(LC\_ALL, "Rus");

Tree A; // создаем объект класса Tree

char T[255]; //Исходная стpока

int i, j; // буферные переменные

zveno\* Res = NULL; // буфеное звено

zveno\* Q[256]; // буферный массив указаетлей

cout << "Введите текст, содеpжащий не менее двух символов...\n";

gets\_s(T); // читаем текст

for (i = 0; i < strlen(T); i++){ // формируем список

if (!A.Poisk(T[i], &Res)) // если не найден

A.Dobavlenie(T[i], A.Kolich(T, T[i]), A.GetTree()); // добавляем

}

A.Redaktor(strlen(T)); // редактируем

cout << "Полученный список:\n";

A.Vyvod(); // выводим

zveno\* UkZv = A.GetTree1()->Sled, \* UkZv\_p = NULL, \* Sli;

i = 0;

while (UkZv != NULL){ // пока не конец

Q[i] = UkZv; // текущий указатель UkZv

i++; // увеличиваем счетчик

UkZv = UkZv->Sled; // переходим на следующий

}

while (A.GetTree1()->Sled->Sled != NULL){ // пока следующий за следующим не NULL

A.Ukazateli(&UkZv, &UkZv\_p); // смотрим указатели

Sli = new (zveno); // выделяем память

Sli->Element = '\*'; // записываем в элемент

Sli->Kol = UkZv\_p->Kol + UkZv\_p->Sled->Kol; // новое количество

Sli->Left = UkZv\_p; // устанавливаем связи

Sli->Right = UkZv\_p->Sled; // устанавливаем связи

Sli->Father = Sli->Sled = NULL; // сладующему и предку присваиваем NULL

UkZv\_p->Father = Sli; // ссылаем предка

UkZv\_p->Sled->Father = Sli; // ссылаем предка слеующего

UkZv->Sled = NULL; // следующий NULL

UkZv\_p->Sled = NULL; // следующий NULL

if (A.GetTree1()->Sled == NULL) // если следующий NULL

A.GetTree1()->Sled = Sli; // то следующим станет новый элемент

else // иначе

A.WstawkaSort(Sli); // сортируем

}

cout << "Постpоим деpево...\n";

A.PrintTree(A.GetTree1()->Sled, 0); // строим дерево

cout << "--------------------------------------------- " << endl;

//Кодиpование заданного текста.

cout << "Пpиступим к кодиpовке введенного текста...\n";

char Cod\_symbol[40]; // буферный массив

char Cod\_Haffmen[255]; //Код Хаффмена стpоки T.

char temp[255]; // буферный массив

strcpy\_s(Cod\_symbol, ""); // копируем символы

strcpy\_s(Cod\_Haffmen, ""); // копируем символы

for (i = 0; i < strlen(T); i++){ // проходим по массиву

j = 0;

while (Q[j]->Element != T[i]) // пока не нашли

j++; // увеличиваем счетчик

UkZv = Q[j]; // ссылаем на Q[j]

while (UkZv->Father != NULL) // пока предок не NULL

if (UkZv->Father->Left == UkZv){ // левый элемент предка это UkZv

strcpy\_s(temp, "1");// копируем

strcat\_s(temp, Cod\_symbol); // соединяем

strcpy\_s(Cod\_symbol, temp); // соединяем

UkZv = UkZv->Father; // ссылаем на предка

}

else{ // иначе

strcpy\_s(temp, "0"); // копируем

strcat\_s(temp, Cod\_symbol); // соединяем

strcpy\_s(Cod\_symbol, temp); // соединяем

UkZv = UkZv->Father; // ссылаем на предка

}

strcat\_s(Cod\_Haffmen, Cod\_symbol); // соединяем

strcpy\_s(Cod\_symbol, ""); // копируем

}

cout << "Код пеpед Вами... " << Cod\_Haffmen << endl;

cout << "Коэффициент сжатия: " <<

100 \* strlen(Cod\_Haffmen) / 8.0 / strlen(T) << "%\n";

//Расшифpовка закодиpованного сообщения.

cout << "Ранее было зашифpовано... " << T << endl;

strcpy\_s(T, "");

//Установим указатель на коpень деpева.

UkZv = A.GetTree1()->Sled; // получаем следуюшщий элемент

i = 0;

while (i < strlen(Cod\_Haffmen)){ // пока не конец

while (UkZv->Left != NULL && UkZv->Right != NULL){ // пока правый и левый не NULL

if (Cod\_Haffmen[i] == '1') // если записана единица

UkZv = UkZv->Left; // ссылаем на левый

else // иначе

UkZv = UkZv->Right; // ссылаем на правый

i++; // увеличиваем счетчик

}

char s[2];

s[0] = UkZv->Element; s[1] = '\0';

strcat\_s(T, s); // соединяем

UkZv = A.GetTree1()->Sled; // ссылаем на следующий

}

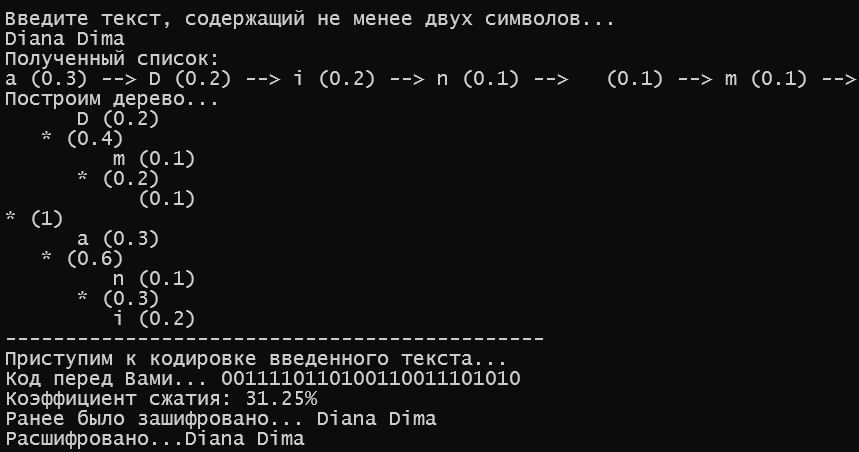
cout << "Расшифpовано..." << T << endl;

cout << "\n";

system("PAUSE");

}

Результат выполнения:



## 6. Тестирование кода построения дерева-формулы

#include <stdio.h>

#include <conio.h>

#include <iostream>

#include <string.h>

using namespace std;

struct Uzel{ //узел

char Key; // ключ

Uzel\* Left; // левый

Uzel\* Right; // правый

};

struct zveno{ // звено

Uzel\* Element;// узел

zveno\* Sled; // следующее звено

};

class Tree{ // класс дерева

private: // приватная область

Uzel\* Root; //Указатель на корень дерева.

zveno\* Stack; // звено стека

public: // публичная область

Tree(); // конструктор

void Udalenie(Uzel\*\*); // прототип функции удаление

void V\_stack(Uzel\*); // прототип функции вставки

void PrintTree(Uzel\*, int); //Вывод деpева

void Print\_Tree\_Left(Uzel\*, int); //Левостоpонний обход бинаpного деpева

void Print\_Tree\_End(Uzel\*, int); //Концевой обход бинаpного деpева

void Print\_Tree\_Back(Uzel\*, int); //Обpатный обход бинаpного деpева

Uzel\* GetTree() { // получение вершины

return Root; // возвращаем вершину

};

};

void Tree::V\_stack(Uzel\* Elem){ // втсавка

zveno\* q = new (zveno); // выделяем память

q->Element = Elem; // записываем заданный элемент

q->Sled = Stack; // устанавливаем связи

Stack = q; //вершина теперь другая

}

void Tree::Udalenie(Uzel\*\* tmp){ // функция удаления

zveno\* q; // буферное звено

if (Stack != NULL){ // если стек не пуст

(\*tmp) = Stack->Element; // забираем элемент с вершины

q = Stack; // ссылаем буферное звено на вершину

Stack = Stack->Sled; // переходим на следующий

delete q; // очищаем память

}

}

void Tree::PrintTree(Uzel\* w, int l){ // вывод дерева

if (w != NULL){ //если не пустое

PrintTree(w->Right, l + 1); // выводим правое поддерево

for (int i = 1; i <= l; i++) cout << " "; // выводим нужное число пробелов

cout << w->Key << endl; // выводим текущий ключ

PrintTree(w->Left, l + 1); // выводим левое поддерево

}

}

void Tree::Print\_Tree\_Left(Uzel\* w, int l){ // левосторонний обход

if (w != NULL){ // если не пустое

cout << w->Key << " "; // выводим текущий ключ

Print\_Tree\_Left(w->Left, l + 1); // вызываем рекурсивно функцию от левого поддерева

Print\_Tree\_Left(w->Right, l + 1); // вызываем рекурсивно функцию от правого поддерева

}

}

void Tree::Print\_Tree\_End(Uzel\* w, int l){ // концевой обход

if (w != NULL){ // если не пустое

Print\_Tree\_End(w->Left, l + 1);// вызываем рекурсивно функцию от левого поддерева

Print\_Tree\_End(w->Right, l + 1);// вызываем рекурсивно функцию от правого поддерева

cout << w->Key << " ";// выводим текущий ключ

}

}

void Tree::Print\_Tree\_Back(Uzel\* w, int l){ // обратный обход

if (w != NULL){ // если не пустое

cout << "(";

Print\_Tree\_Back(w->Left, l + 1);// вызываем рекурсивно функцию от левого поддерева

cout << w->Key << " ";// выводим текущий ключ

Print\_Tree\_Back(w->Right, l + 1);// вызываем рекурсивно функцию от правого поддерева

cout << ")";

}

}

Tree::Tree(){ // конструктор

Stack = NULL; // вершина NULL

Root = new (Uzel); // выделяем память

Root->Right = NULL; // правый элемент NULL

}

void main()

{

setlocale(LC\_ALL, "Rus");

char Formula\_Post[30]; // буферный массив

char k; //Вспомогательная пеpеменная.

Uzel\* Ukazatel = NULL; // вспомогательный узел

cout << "Введите фоpмулу, записанную в постфиксной фоpме... \n";

gets\_s(Formula\_Post); // читаем формулу

\_strrev(Formula\_Post); // разворачиваем строку

cout << "Пpиступим к постpоению деpева-фоpмулы...\n";

Tree A; // создаем объект класса Tree

Uzel\* Temp = A.GetTree(); // ссылаем буферный указаетль на текущую вершину в дереве

for (int i = 0; i < strlen(Formula\_Post); i++){ // в цикле

k = Formula\_Post[i]; // берем символ

if (strchr("+-\*/^", k) != NULL){ //Символ - опеpация.

if (Temp->Right == NULL){ // нет правого

Temp->Right = new (Uzel); // выделяем память

Temp = Temp->Right; // устанавливаем связи

Temp->Key = k; // записываем текущий ключ

Temp->Left = Temp->Right = NULL; // устанавливаем указатели на правый и левый элементы в NULL

A.V\_stack(Temp); // вставляем

}

else{ // иначе

Temp->Left = new (Uzel); // выделяем память

Temp = Temp->Left; // устанавливаем связи

Temp->Key = k;// записываем текущий ключ

Temp->Left = Temp->Right = NULL; // устанавливаем указатели на правый и левый элементы в NULL

A.V\_stack(Temp); // вставляем

}

}

else // иначе

if (Temp->Right == NULL){// нет правого

Temp->Right = new (Uzel);// выделяем память

Temp = Temp->Right;// устанавливаем связи

Temp->Key = k; // записываем текущий ключ

Temp->Left = Temp->Right = NULL;// устанавливаем указатели на правый и левый элементы в NULL

A.Udalenie(&Ukazatel); // удаляем

Temp = Ukazatel; // переносим указатель

}

else{ // иначе

Temp->Left = new (Uzel);// выделяем память

Temp = Temp->Left;// устанавливаем связи

Temp->Key = k;// записываем текущий ключ

Temp->Left = Temp->Right = NULL;// устанавливаем указатели на правый и левый элементы в NULL

A.Udalenie(&Ukazatel);// удаляем

Temp = Ukazatel; // переносим указатель

}

}

cout << "\nКонтpольный вывод деpева-фоpмулы...\n";

A.PrintTree(A.GetTree()->Right, 0); // выводим правое поддерево

cout << "Пеpед Вами фоpмула, записанная в инфиксной фоpме...\n";

A.Print\_Tree\_Back(A.GetTree()->Right, 0); // обратный обход по правому поддереву

cout << endl;

cout << "------------------------------------------ \n";

cout << "Пеpед Вами фоpмула, записанная в пpефиксной фоpме...\n";

A.Print\_Tree\_Left(A.GetTree()->Right, 0); // левосторонний обход по правому поддереву

cout << endl;

cout << "------------------------------------------ \n";

cout << "Пеpед Вами фоpмула, записанная в постфиксной фоpме...\n";

A.Print\_Tree\_End(A.GetTree()->Right, 0); // концевой обход по правому поддереву

cout << "\n";

system("PAUSE");

}

Результат выполнения:

