МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

Учреждение образования «БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ

ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

Факультет: Информационных технологий

Кафедра: Программной инженерии

Тестирование кодов

По дисциплине” Основы Алгоритмизации и Программирования”

***Выполнил:*** *студент 1 курса 8 группы*

*Федорович В.Г*

**Специальность:** *ПИ 1 курс.*

**Проверил:** преподаватель *Белодед Николай Иванович*

Минск

2024

Оглавление

[Код тестирования 1(Бинарное дерево рекурсия) 3](#_Toc165468314)

[Код тестирования 2(Не рекурсивное построение бинарного дерева) 6](#_Toc165468315)

[Код Тестирования 3(Хэширование с помощью леса) 9](#_Toc165468316)

[Код Тестирования 4(Древовидно-кольцевая структура данных) 13](#_Toc165468317)

[Код Тестирования 5(Реализация алгоpитма Хаффмена с помощью деpева) 31](#_Toc165468318)

[Код Тестирования 6(Hе pекуpсивная пpогpамма построения дерева-формулы) 37](#_Toc165468319)

# Код тестирования 1(Бинарное дерево рекурсия)

#include <iostream>

using namespace std;

struct node // Определение структуры "node" для узлов бинарного дерева

{

int Key; // Ключ узла

int Count; // Счетчик повторений ключа

node\* Left; // Указатель на левого потомка

node\* Right; // Указатель на правого потомка

};

class TREE // Определение класса "TREE" для работы с бинарным деревом

{

private:

node\* Tree; // Указатель на корень дерева

void Search(int, node\*\*); // Прототип функции для поиска и вставки вершины в дерево

public:

TREE() { Tree = NULL; } // Конструктор класса, инициализирующий корень дерева как NULL

node\*\* GetTree() { return &Tree; } // Метод для получения указателя на корень дерева

void BuildTree(); // Метод для построения дерева

void CleanTree(node\*\*); // Метод для очистки дерева от памяти

void ObhodEnd(node\*\*); // Метод для концевого обхода дерев

void ObhodLeft(node\*\*); // Метод для левостороннего обхода дерева

void ObhodBack(node\*\*); // Метод для обратного обхода дерева

void Vyvod(node\*\*, int); // Метод для вывода дерева на экран

int Height(node\*\*); // Метод для определения высоты бинарного дерева

};

void main() // Главная функция программы

{

setlocale(LC\_ALL, "Rus");

TREE A; // Создание объекта класса TREE

A.BuildTree(); // Вызов метода для построения дерева

cout << "\nВывод дерева:\n";

A.Vyvod(A.GetTree(), 0); // Вызов метода для вывода дерева на экран

cout << "\nВысота дерева:" << A.Height(A.GetTree()) << endl; // Вывод высоты дерева

cout << "\nЛевосторонний обход дерева: ";

A.ObhodLeft(A.GetTree()); // Вызов метода для левостороннего обхода дерева

cout << "\nКонцевой обход дерева: "; A.ObhodEnd(A.GetTree()); // Вызов метода для концевого обхода дерева

cout << "\nОбратный обход дерева: "; A.ObhodBack(A.GetTree()); // Вызов метода для обратного обхода дерева

A.CleanTree(A.GetTree()); // Вызов метода для очистки дерева.

cout << "\n";

system("PAUSE"); // Пауза перед завершением программы

}

void TREE::BuildTree() // Метод для построения бинарного дерева

{

int el; // Переменная для хранения вводимых ключей

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

cin >> el; // Считывание первого ключа

while (el != 0) // Пока не введен нулевой ключ (завершающий ввод)

{

Search(el, &Tree); // Вызов метода поиска и вставки вершины в дерево

cin >> el;

}

}

void TREE::Search(int x, node\*\* p) // Метод для поиска вершины с ключом x в дереве со вставкой

{

if (\*p == NULL) // Если указатель на текущую вершину NULL (дошли до конца ветви)

{

\*p = new(node); // Создаем новую вершину

(\*\*p).Key = x; // Присваиваем ключ вершине

(\*\*p).Count = 1; // Устанавливаем счетчик повторений ключа

(\*\*p).Left = NULL; (\*\*p).Right = NULL; // Инициализируем указатели на потомков как NULL

}

else

if (x < (\*\*p).Key) Search(x, &((\*\*p).Left)); // Если ключ меньше текущего, идем влево

else

if (x > (\*\*p).Key) Search(x, &((\*\*p).Right)); // Если ключ больше текущего, идем вправо

else (\*\*p).Count = (\*\*p).Count + 1; // Если ключ равен текущему, увеличиваем счетчик повторений

}

void TREE::ObhodLeft(node\*\* w) // Метод для левостороннего обхода дерева

{

if (\*w != NULL) // Если текущая вершина не пуста

{

cout << (\*\*w).Key << " "; // Выводим ключ текущей вершины

ObhodLeft(&((\*\*w).Left)); // Рекурсивно обходим левое поддерево

ObhodLeft(&((\*\*w).Right)); // Рекурсивно обходим правое поддерево

}

}

void TREE::ObhodEnd(node\*\* w) // Метод для концевого обхода дерева

{

if (\*w != NULL) // Если текущая вершина не пуста

{

ObhodEnd(&((\*\*w).Left)); // Рекурсивно обходим левое поддерево

ObhodEnd(&((\*\*w).Right)); // Рекурсивно обходим правое поддерево

cout << (\*\*w).Key << " "; // Выводим ключ текущей вершины

}

}

void TREE::ObhodBack(node\*\* w) // Метод для обратного обхода дерева

{

if (\*w != NULL) // Если текущая вершина не пуста

{

ObhodBack(&((\*\*w).Left)); // Рекурсивно обходим левое поддерево

cout << (\*\*w).Key << " ";

ObhodBack(&((\*\*w).Right)); // Рекурсивно обходим правое поддерево

}

}

void TREE::CleanTree(node\*\* w) // Метод для очистки дерева от памяти

{

if (\*w != NULL) // Если текущая вершина не пуста

{

CleanTree(&((\*\*w).Left)); // Рекурсивно очищаем левое поддерево

CleanTree(&((\*\*w).Right)); // Рекурсивно очищаем правое поддерево

delete\* w; // Удаляем текущую вершину

}

}

void TREE::Vyvod(node\*\* w, int l) // Метод для вывода дерева на экран

{

int i; // Переменная для цикла

if (\*w != NULL) // Если текущая вершина не пуста

{

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

for (i = 1; i <= l; i++) cout << " "; // Отступ для текущей вершины

cout << (\*\*w).Key << endl;

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

}

}

int TREE::Height(node\*\* w) // Метод для определения высоты бинарного дерева

{

int h1, h2; // Переменные для хранения высот левого и правого поддеревьев

if (\*w == NULL) return (-1); // Если текущая вершина пуста, возвращаем -1 (высота пустого дерева)

else

{

h1 = Height(&((\*\*w).Left)); // Рекурсивно определяем высоту левого поддерева

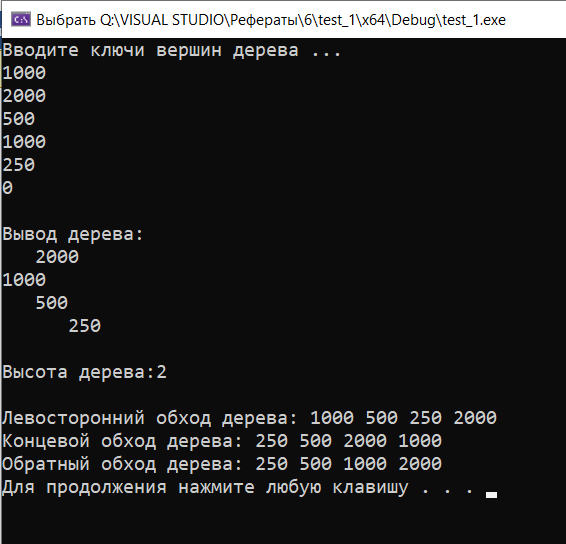
h2 = Height(&((\*\*w).Right)); // Рекурсивно определяем высоту правого поддерева

if (h1 > h2) return (1 + h1); // Возвращаем большую из высот поддеревьев

else return (1 + h2);

}

}



# Код тестирования 2(Не рекурсивное построение бинарного дерева)

#include <iostream>

using namespace std;

struct node // Определяем структуру узла бинарного дерева

{

int Key; // Ключ узла

int Count; // Счетчик повторений ключа

node\* Left; // Указатель на левого потомка

node\* Right; // Указатель на правого потомка

};

struct no // Звено стека

{

node\* elem; // Информационное поле

int ch; // Уровень вершины

no\* sled; // Указатель на следующий элемент стека

};

class TREE // Определяем класс для работы с бинарным деревом

{

private:

node\* Tree; // Указатель на корень дерева

void PushStack(no\*\*, node\*\*, int\*); // Объявление функции для помещения элемента в стек

void PopStack(no\*\*, node\*\*, int\*); // Объявление функции для извлечения элемента из стека

void VyvodStack(no\*\*); // Объявление функции для вывода содержимого стека

public:

TREE() { Tree = new(node); (\*Tree).Right = NULL; } // Конструктор класса, инициализирующий корень дерева

node\* GetTreeRight() { return (\*Tree).Right; } // Метод для получения указателя на правого потомка корня

void TreeSearch(int); // Метод для поиска вершины с заданным ключом в дереве

void VyvodTree(node\*); // Метод для вывода дерева на экран

};

void main() // Главная функция программы

{

setlocale(LC\_ALL, "Rus"); // Устанавливаем русскую локаль

TREE A; // Создаем объект класса TREE

int el; // Переменная для хранения вводимых значений ключей

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

cin >> el; // Считываем первый ключ

while (el != 0) // Цикл ввода ключей до ввода нуля

{

A.TreeSearch(el); cin >> el;

} // Вызываем метод поиска и добавления вершины в дерево

A.VyvodTree(A.GetTreeRight()); // Вызываем метод для вывода дерева на экран

cout << "\n"; // Переход на новую строку

system("PAUSE"); // Пауза перед завершением программы

}

void TREE::TreeSearch(int el) // Метод для поиска вершины с заданным ключом в дереве

{

node\* p1, \* p2; // Указатели для поиска места вставки новой вершины

int d; // Переменная для определения направления движения по дереву

p2 = Tree; p1 = (\*p2).Right; d = 1; // Инициализация указателей и переменной

while (p1 != NULL && d != 0) // Пока не достигнут конец дерева или не найдена вершина с таким же ключом

{

p2 = p1; // Переходим к следующему узлу

if (el < (\*p1).Key) { p1 = (\*p1).Left; d = -1; } // Если ключ меньше текущего, идем влево

else

if (el > (\*p1).Key) { p1 = (\*p1).Right; d = 1; } // Если ключ больше текущего, идем вправо

else d = 0; // Если ключ равен текущему, завершаем поиск

}

if (d == 0) (\*p1).Count = (\*p1).Count + 1; // Если найдена вершина с таким же ключом, увеличиваем счетчик

else // Иначе

{

p1 = new(node); // Выделяем память под новую вершину

(\*p1).Key = el; (\*p1).Left = NULL; // Присваиваем значения ключу и указателю на левого потомка

(\*p1).Right = NULL; (\*p1).Count = 1; // Присваиваем значения указателю на правого потомка и счетчику

if (d < 0) (\*p2).Left = p1; // Если новая вершина должна быть левым потомком

else (\*p2).Right = p1; // Иначе - правым потомком

}

}

void TREE::VyvodTree(node\* t) // Метод для вывода дерева на экран

{

no\* stk, \* stk1; // Указатели на стеки для обхода дерева

node\* u; // Вспомогательный указатель на узел

int i, n; // Переменные для циклов и уровня узла

stk = stk1 = NULL; n = 0; // Инициализация стеков и переменной

while (t != NULL) // Пока не пройдены все узлы дерева

{

PushStack(&stk1, &t, &n); // Помещаем узел во вспомогательный стек

if ((\*t).Right != NULL) // Если у текущего узла есть правый потомок

{

if ((\*t).Left != NULL) PushStack(&stk, &((\*t).Left), &n); // Если есть и левый, помещаем его в основной стек

t = (\*t).Right; // Переходим к правому потомку

}

else // Иначе

{

if ((\*t).Left != NULL) // Если у текущего узла есть только левый потомок

{

if (stk1 != NULL) // Если во вспомогательном стеке есть элементы

{

PopStack(&stk1, &u, &n); // Извлекаем из него узел

for (i = 0; i <= n; i++) cout << " "; // Выводим пробелы для отступа

cout << (\*u).Key << endl; // Выводим ключ узла

}

t = (\*t).Left; // Переходим к левому потомку

}

else // Иначе

if (stk == NULL) t = NULL; // Если основной стек пуст, завершаем обход

else // Иначе

{

while ((\*stk).elem != (\*((\*stk1).elem)).Left) // Пока в основном стеке не встретим узел с левым потомком

{

PopStack(&stk1, &u, &n); // Извлекаем узел из вспомогательного стека

for (i = 0; i <= n; i++) cout << " "; // Выводим пробелы для отступа

cout << (\*u).Key << endl; // Выводим ключ узла

}

PopStack(&stk1, &u, &n); // Извлекаем узел с левым потомком

for (i = 0; i <= n; i++) cout << " "; // Выводим пробелы для отступа

cout << (\*u).Key << endl; // Выводим ключ узла

PopStack(&stk, &t, &n); // Извлекаем из основного стека текущий узел

}

}

n = n + 1; // Увеличиваем уровень узла

}

VyvodStack(&stk1); // Выводим оставшиеся узлы из вспомогательного стека

}

void TREE::PushStack(no\*\* stk, node\*\* el, int\* n) // Функция для помещения элемента в стек

{

no\* q; // Вспомогательный указатель на элемент стека

q = new(no); // Выделяем память под новый элемент стека

(\*q).elem = \*el; (\*q).ch = \*n; // Присваиваем значения элементу и уровню

(\*q).sled = \*stk; \*stk = q; // Добавляем элемент в стек

}

void TREE::PopStack(no\*\* stk, node\*\* t, int\* n) // Функция для извлечения элемента из стека

{

no\* q; // Вспомогательный указатель на элемент стека

if (\*stk != NULL) // Если стек не пуст

{

\*t = (\*\*stk).elem; // Присваиваем указатель на элемент стека

\*n = (\*\*stk).ch; // Присваиваем уровень элемента стека

q = \*stk; // Запоминаем адрес элемента стека

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

delete q; // Освобождаем память, выделенную под элемент стека

}

}

void TREE::VyvodStack(no\*\* stk) // Функция для вывода содержимого стека

{

node\* k; // Указатель на узел дерева

int i, n; // Переменные для цикла и уровня узла

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

{

k = (\*\*stk).elem; n = (\*\*stk).ch; // Получаем указатель на узел и его уровень

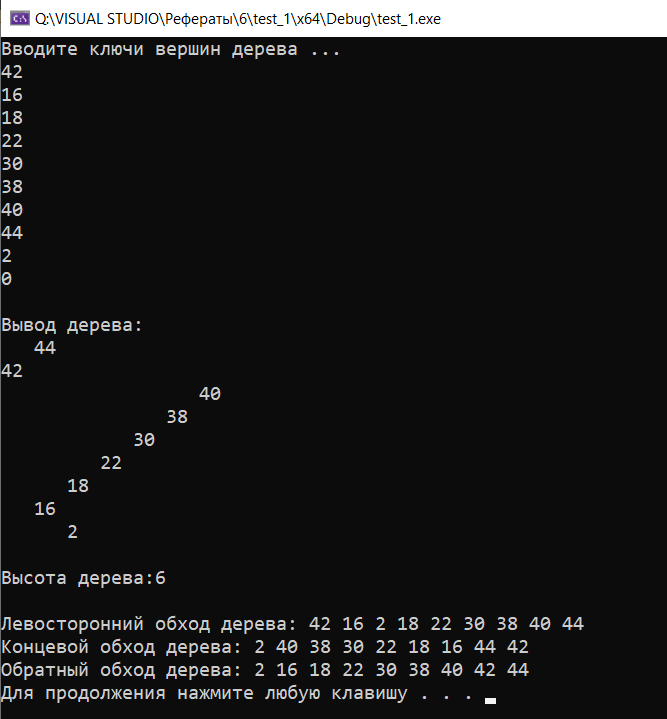
for (i = 0; i <= n; i++) cout << " "; // Выводим пробелы для отступа

cout << (\*k).Key << endl; // Выводим ключ узла

\*stk = (\*\*stk).sled; // Переходим к следующему элементу стека

}

}



# Код Тестирования 3(Хэширование с помощью леса)

#include <time.h> // Подключаем заголовочный файл для работы с временем

#include <iostream>

using namespace std;

#define N 10 // Определяем константу для количества элементов массива

struct node // Определяем структуру узла бинарного дерева

{

int Key; // Ключ узла

int Count; // Счетчик повторений ключа

node\* Left; // Указатель на левого потомка

node\* Right; // Указатель на правого потомка

};

class Spisok // Определяем класс для работы с хэш-списком

{

private:

node\* UkStr[N]; // Массив указателей на корни бинарных деревьев

void Search(int, node\*\*); // Объявление функции для поиска узла в дереве

void PrintTree(node\*, int); // Объявление функции для вывода дерева на экран

void U\_d(node\*\*, node\*\*); // Объявление функции для удаления узла из дерева

public:

Spisok(); // Конструктор класса

void BuildTree(); // Метод для построения хэш-списка

void Sodergimoe(); // Метод для вывода содержимого хэш-списка

node\*\* GetTree(unsigned i) { return &(UkStr[i]); } // Метод для получения указателя на корень i-го дерева

void Udaldr(node\*\* d, int k); // Метод для удаления узла с ключом k из дерева d

};

Spisok::Spisok()

{

// Инициализация хэш-списка.

for (int i = 0; i < N; i++) UkStr[i] = NULL; // Устанавливаем все указатели в массиве в NULL

}

void Spisok::BuildTree()

{

int klutch; // Переменная для ключа

unsigned hash; // Переменная для хэша

srand(time(0)); // Инициализируем генератор случайных чисел

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

klutch = rand() % 31 + 0; // Генерируем случайный ключ в диапазоне от 0 до 31

cout << klutch; // Выводим сгенерированный ключ

while (klutch != 0) // Пока ключ не равен 0

{

hash = klutch % 10; // Вычисляем значение хэш-функции

Search(klutch, &UkStr[hash]); // Вызываем метод поиска и вставки узла в дерево

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

klutch = rand() % 31 + 0; // Генерируем новый случайный ключ

cout << klutch; // Выводим сгенерированный ключ

}

}

void Spisok::Search(int X, node\*\* p)

{

if (\*p == NULL)

{ // Узла нет в дереве; включаем его.

\*p = new (node); // Выделяем память под новый узел

(\*\*p).Key = X; // Присваиваем значение ключа

(\*\*p).Count = 1; // Устанавливаем счетчик повторений ключа в 1

(\*\*p).Left = (\*\*p).Right = NULL; // Инициализируем указатели на потомков как NULL

}

else

if (X < (\*\*p).Key) Search(X, &((\*\*p).Left)); // Если ключ меньше текущего, идем влево

else if (X > (\*\*p).Key) Search(X, &((\*\*p).Right)); // Если ключ больше текущего, идем вправо

else (\*\*p).Count += 1; // Иначе увеличиваем счетчик повторений ключа

}

void Spisok::Sodergimoe()

{

for (int i = 0; i < N; i++) // Проходим по всем деревьям хэш-списка

{

cout << " " << i << "... "; // Выводим номер дерева

if (UkStr[i] == NULL) cout << "Дерево пусто...\n"; // Если дерево пустое, выводим сообщение

else // Иначе

{

cout << endl; // Переход на новую строку

PrintTree(UkStr[i], 0); // Вызываем метод для вывода дерева на экран

}

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

}

}

void Spisok::PrintTree(node\* w, int l)

{

if (w != NULL)

{

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

cout << " "; // Выводим отступ

for (int i = 1; i <= l; i++) cout << " "; // Выводим отступы в зависимости от уровня узла

cout << (\*w).Key << endl; // Выводим ключ узла

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

}

}

void Spisok::Udaldr(node\*\* d, int k)

{ // Удаление узла с ключом k из дерева d.

node\*\* q;

if (\*d == NULL) cout << "Узел с заданным ключом в дереве не найден...\n"; // Если дерево пустое

else

if (k < (\*\*d).Key) Udaldr(&((\*\*d).Left), k); // Если ключ меньше текущего, идем влево

else if (k > (\*\*d).Key) Udaldr(&((\*\*d).Right), k); // Если ключ больше текущего, идем вправо

else // Иначе нашли узел для удаления

{

q = d; // Сохраняем указатель на удаляемый узел

if ((\*\*q).Right == NULL) \*d = (\*\*q).Left; // Если у удаляемого узла нет правого потомка

else

if ((\*\*q).Left == NULL) \*d = (\*\*q).Right; // Если у удаляемого узла нет левого потомка

else U\_d(&((\*\*q).Left), &(\*q)); // Если у удаляемого узла есть оба потомка, вызываем функцию удаления

}

}

void Spisok::U\_d(node\*\* r, node\*\* q)

{

if ((\*\*r).Right == NULL)

{

(\*\*q).Key = (\*\*r).Key; // Копируем ключ из правого поддерева вместо удаляемого

(\*\*q).Count = (\*\*r).Count; // Копируем счетчик из правого поддерева вместо удаляемого

q = r; \*r = (\*\*r).Left; delete (\*q); // Удаляем правое поддерево и переходим к левому

}

else U\_d(&((\*\*r).Right), &(\*q)); // Рекурсивно продолжаем искать крайний левый узел в правом поддереве

}

void main()

{

setlocale(LC\_ALL, "Rus"); // Устанавливаем русскую локаль

Spisok A; // Создаем объект класса Spisok

A.BuildTree(); // Вызываем метод для построения хэш-списка

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

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

A.Sodergimoe(); // Выводим содержимое хэш-списка

// Удаление элемента из хэш-списка.

for (int i = 0; i < 4; i++) // Будем удалять всего 4 раза!

{

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

int klutch; // Переменная для ключа

cin >> klutch; // Вводим значение ключа для удаления

unsigned hash = klutch % 10; // Вычисляем значение хэша для данного ключа

A.Udaldr(A.GetTree(hash), klutch); // Вызываем метод для удаления узла с заданным ключом из хэш-списка

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

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

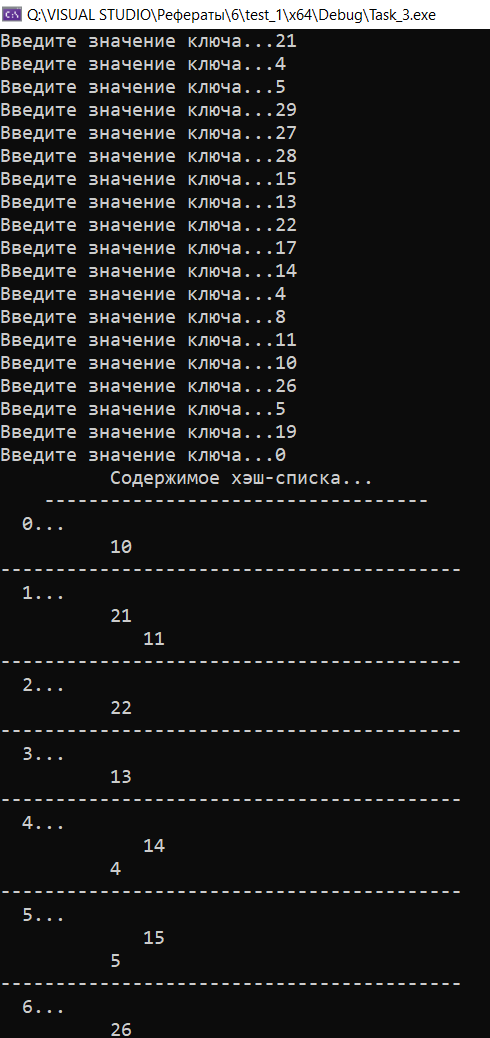
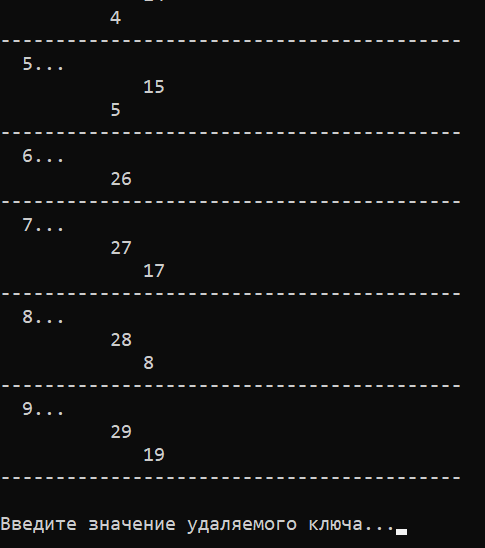
A.Sodergimoe(); // Выводим содержимое хэш-списка после удаления

}

cout << "\n";

system("PAUSE");

}

# Код Тестирования 4(Древовидно-кольцевая структура данных)

#include <conio.h>

#include <iostream>

using namespace std;

// Определение структуры узла дерева

struct node

{

int key; // Ключ узла

int count; // Счётчик

node\* Left; // Указатель на левое поддерево

node\* Right; // Указатель на правое поддерево

};

// Определение класса для работы с деревом

class Tree

{

private:

node\* root; // Корень дерева

void DisposeTree(node\*); // Метод для удаления дерева

void printTree(node\*, int); // Метод для вывода дерева на экран

void Delete(node\*\*, int); // Метод для удаления узла из дерева

void del(node\*\*, node\*); // Вспомогательный метод для удаления узла из дерева

public:

Tree() { root = NULL; }; // Конструктор класса

~Tree(); // Деструктор класса

void creat\_Tree(); // Метод для создания дерева

void look\_Tree(); // Метод для вывода дерева на экран

void add\_Tree(); // Метод для добавления узла в дерево

void delete\_Tree(); // Метод для удаления узла из дерева

void search(int, node\*\*); // Метод для поиска узла по ключу

node\* getTree() { return root; }; // Метод для получения корня дерева

};

// Определение структуры элемента кольца

struct zveno

{

int element; // Элемент кольца

Tree ukTree; // Дерево, связанное с элементом кольца

zveno\* sled; // Указатель на следующий элемент кольца

};

// Определение класса для работы с кольцом

class ring

{

private:

zveno\* ukring; // Указатель на первый элемент кольца

public:

ring() { ukring = NULL; }; // Конструктор класса

~ring(); // Деструктор класса

void create(); // Метод для построения кольца

void look(); // Метод для вывода содержимого кольца

void add\_after(int, zveno\*); // Метод для добавления элемента после указанного элемента кольца

void add\_befor(int, zveno\*); // Метод для добавления элемента перед указанным элементом кольца

void Delete(zveno\*); // Метод для удаления указанного элемента кольца

void delete\_next(zveno\*); // Метод для удаления элемента, следующего за указанным элементом кольца

int poisk(int, zveno\*\*); // Метод для поиска элемента кольца по значению

zveno\*\* getring() { return &ukring; }; // Метод для получения указателя на кольцо

};

// Метод для создания кольца

void ring::create()

{

zveno\* ukzv; // Указатель на текущий элемент кольца

int elem; // Переменная для хранения вводимых элементов

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

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

cout << "-->";

cin >> elem;

// Если введённый элемент не равен 0

if (elem != 0)

{

// Создание первого элемента кольца

ukzv = ukring = new (zveno);

(\*ukzv).element = elem; // Присваивание введённого значения элементу кольца

(\*ukzv).ukTree.creat\_Tree(); // Создание дерева для текущего элемента кольца

cout << "\n-->";

cin >> elem; // Ввод следующего элемента кольца

// Пока вводимый элемент не равен 0

while (elem != 0)

{

(\*ukzv).sled = new (zveno); // Создание нового элемента кольца

ukzv = (\*ukzv).sled; // Переход к новому элементу кольца

(\*ukzv).element = elem; // Присваивание введённого значения элементу кольца

(\*ukzv).ukTree.creat\_Tree(); // Создание дерева для текущего элемента кольца

cout << "\n-->";

cin >> elem; // Ввод следующего элемента кольца

}

ukzv->sled = ukring; // Замыкание кольца

}

}

// Деструктор класса для работы с кольцом

ring::~ring()

{

zveno\* ukzv; // Указатель на текущий элемент кольца

ukzv = ukring; // Начальное значение указателя - первый элемент кольца

// Пока кольцо не опустеет

while (ukring != NULL)

{

// Если текущий элемент является последним в кольце

if (ukzv->sled == ukring)

{

ukring = NULL; // Присваиваем значение NULL указателю на кольцо, чтобы завершить цикл

ukzv->ukTree.~Tree(); // Вызов деструктора для дерева текущего элемента кольца

delete ukzv; // Освобождение памяти, выделенной под текущий элемент кольца

}

else

{

// Пока текущий элемент не предпоследний в кольце

while (ukzv->sled->sled != ukring)

ukzv = (\*ukzv).sled; // Переход к следующему элементу кольца

(\*ukzv).sled->ukTree.~Tree(); // Вызов деструктора для дерева текущего элемента кольца

delete (\*ukzv).sled; // Освобождение памяти, выделенной под текущий элемент кольца

ukzv->sled = ukring; // Замыкание кольца

ukzv = ukring; // Переход к первому элементу кольца

}

}

}

void ring::look() // Вывод кольца.

{

zveno\* ukzv; // Указатель на текущий элемент кольца

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

ukzv = ukring; // Начальное значение указателя - первый элемент кольца

do {

cout << "\n-->" << (\*ukzv).element << endl; // Вывод элемента кольца

ukzv->ukTree.look\_Tree(); // Вызов метода для просмотра дерева, связанного с текущим элементом кольца

ukzv = ukzv->sled; // Переход к следующему элементу кольца

\_getch(); // Ожидание нажатия клавиши (зависит от среды выполнения)

} while (ukzv != ukring); // Повторять, пока не вернемся к первому элементу кольца

cout << endl;

}

void ring::add\_befor(int elem, zveno\* zv)

{

zveno\* ukzv; // Указатель на новое звено

ukzv = new (zveno); // Создание нового звена

// Сохранение информации о текущем звене во временную переменную

Tree temp = ukzv->ukTree;

ukzv->element = zv->element;

ukzv->ukTree = zv->ukTree;

ukzv->sled = zv->sled;

zv->element = elem; // Замена элемента текущего звена

zv->ukTree = temp; // Замена связанного дерева текущего звена

zv->ukTree.creat\_Tree(); // Создание нового дерева для текущего звена

zv->sled = ukzv; // Установка указателя на новое звено

}

void ring::add\_after(int elem, zveno\* zv)

{

zveno\* ukzv; // Указатель на новое звено

ukzv = new (zveno); // Создание нового звена

ukzv->element = elem; // Установка значения элемента нового звена

ukzv->ukTree.creat\_Tree(); // Создание нового дерева для нового звена

ukzv->sled = zv->sled; // Установка указателя на следующее звено для нового звена

zv->sled = ukzv; // Установка указателя на новое звено для предыдущего звена

}

void ring::Delete(zveno\* zv)

{

zveno\* ukzv1, \* ukzv2; // Указатели на звено, предшествующее удаляемому, и на звено, следующее за удаляемым

zveno\* time; // Временный указатель на звено

if (zv->sled != ukring) // Если удаляемое звено не является последним в кольце

{

time = zv->sled; // Запоминаем указатель на следующее за удаляемым звено

zv->ukTree.~Tree(); // Уничтожаем дерево в удаляемом звене

(\*zv) = \*((\*zv).sled); // Копируем данные из следующего за удаляемым звена в удаляемое звено

delete time; // Удаляем следующее за удаляемым звено

}

else if (zv->sled == zv) // Если кольцо состоит из одного звена

{

zv->ukTree.~Tree(); // Уничтожаем дерево в единственном звене кольца

delete ukring; // Удаляем звено

ukring = NULL; // Устанавливаем указатель на кольцо как NULL, так как кольцо теперь пусто

cout << "Список пуст...\n"; // Выводим сообщение о том, что список пуст

}

else // Если удаляемое звено находится в середине кольца

{

ukzv2 = ukring; // Указатель ukzv2 на первое звено кольца

ukzv1 = ukring->sled; // Указатель ukzv1 на второе звено кольца

while (ukzv1 != zv) // Пока не найдено удаляемое звено

{

ukzv2 = ukzv1; // Перемещаем указатель ukzv2 на следующее звено

ukzv1 = ukzv1->sled; // Перемещаем указатель ukzv1 на следующее звено

}

time = ukzv2->sled; // Запоминаем указатель на удаляемое звено

ukzv2->sled->ukTree.~Tree(); // Уничтожаем дерево в удаляемом звене

ukzv2->sled = ukzv2->sled->sled; // Пропускаем удаляемое звено при переходе от предыдущего к следующему

delete time; // Удаляем удаляемое звено

}

}

void ring::delete\_next(zveno\* zv)

{

zveno\* time; // Временный указатель на следующее за удаляемым звено

if (zv->sled != ukring) // Если удаляемое звено не является последним в кольце

{

time = zv->sled; // Запоминаем указатель на следующее за удаляемым звено

zv->sled = zv->sled->sled; // Пропускаем удаляемое звено при переходе от текущего к следующему

time->ukTree.~Tree(); // Уничтожаем дерево в удаляемом звене

delete time; // Удаляем следующее за удаляемым звено

}

else if (zv->sled == zv) // Если кольцо состоит только из одного звена

{

cout << "В кольце только один элемент!\n";

}

else // Если удаляемое звено последнее в кольце

{

time = ukring->sled; // Запоминаем указатель на следующее после начального звена кольца

\*((\*zv).sled) = (\*(\*(\*zv).sled).sled); // Пропускаем удаляемое звено при переходе от текущего к следующему

time->ukTree.~Tree(); // Уничтожаем дерево в удаляемом звене

delete time; // Удаляем следующее за удаляемым звено

}

}

int ring::poisk(int elem, zveno\*\* Res)

{

zveno\* ukzv; // Указатель на текущее звено

int vozvr = 0; // Флаг для указания наличия элемента в кольце

if (\*(getring()) == NULL) // Если кольцо не существует

{

cout << "Кольцо не существует...\n"; // Выводим сообщение об отсутствии кольца

}

else

{

ukzv = ukring; // Устанавливаем указатель на начальное звено кольца

while (ukzv->sled != ukring && (\*Res) == NULL) // Пока не вернулись к начальному звену и результат не найден

{

if (ukzv->element == elem) // Если текущий элемент равен искомому

{

vozvr = 1; // Устанавливаем флаг наличия элемента

\*Res = ukzv; // Сохраняем указатель на найденное звено

}

ukzv = ukzv->sled; // Переходим к следующему звену

}

if ((\*Res) == NULL) // Если результат поиска не найден

{

if (ukzv->element == elem) // Проверяем последнее звено

{

vozvr = 1; // Устанавливаем флаг наличия элемента

\*Res = ukzv; // Сохраняем указатель на найденное звено

}

}

}

return vozvr; // Возвращаем флаг наличия элемента

}

Tree::~Tree()

{

DisposeTree(root); // Вызываем функцию удаления всех узлов дерева, начиная с корня

root = NULL; // Обнуляем указатель на корень

}

void Tree::DisposeTree(node\* p)

{

if (p != NULL) // Если узел не пустой

{

DisposeTree(p->Left); // Удаляем левое поддерево

DisposeTree(p->Right); // Удаляем правое поддерево

delete p; // Удаляем сам узел

}

}

void Tree::search(int x, node\*\* p)

{

if (\*p == NULL) // Если указатель на узел пустой

{

\*p = new node; // Выделяем память под новый узел

(\*\*p).key = x; // Устанавливаем значение ключа

(\*\*p).count = 1; // Устанавливаем начальное значение счетчика

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

}

else if (x < (\*\*p).key) // Если значение ключа меньше ключа текущего узла

{

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

}

else if (x > (\*\*p).key) // Если значение ключа больше ключа текущего узла

{

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

}

else // Если значение ключа равно ключу текущего узла

{

(\*\*p).count += 1; // Увеличиваем счетчик

}

}

void Tree::creat\_Tree()

{

int elem;

cout << "Вводите ключи узлов дерева (ввод окончите 0):\n"; // Выводим приглашение для ввода ключей

cin >> elem; // Считываем первый ключ

while (elem != 0) // Пока вводимый ключ не равен 0 (сигнал завершения ввода)

{

search(elem, &root); // Вызываем функцию поиска и вставки ключа в дерево

cin >> elem; // Считываем следующий ключ

}

}

void Tree::look\_Tree()

{

if (root == NULL)

cout << "Дерево пусто ...\n"; // Выводим сообщение о том, что дерево пусто

else

printTree(root, 0); // Выводим содержимое дерева с корнем root

}

void Tree::printTree(node\* w, int L)

{

if (w != NULL)

{

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

for (int i = 1; i <= L; i++)

cout << " "; // Выводим пробелы для отступа в зависимости от уровня узла

cout << w->key << endl; // Выводим значение ключа узла

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

}

}

void Tree::add\_Tree()

{

int k;

cout << "\nВводите ключи добавляемых узлов (ввод окончите 0):\n"; // Просим пользователя ввести ключи для добавления узлов

cin >> k; // Считываем первый ключ

cout << " ";

while (k != 0)

{

search(k, &(root)); // Вызываем функцию поиска для добавления узла с ключом k

cin >> k; // Считываем следующий ключ

cout << " ";

}

}

void Tree::delete\_Tree()

{

int elem;

if (root == NULL) // Проверяем, пустое ли дерево

cout << "Дерево пусто ...\n";

else

{

cout << "Введите ключ удаляемого узла : "; // Просим пользователя ввести ключ удаляемого узла

cin >> elem; // Считываем ключ удаляемого узла

cout << endl;

Delete(&root, elem); // Вызываем функцию удаления узла с заданным ключом

}

}

void Tree::Delete(node\*\* d, int k)

{

node\* q;

node\* s;

if (\*d == NULL) // Если указатель на узел пуст, выводим сообщение об ошибке

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

else

{

if (k < (\*\*d).key) // Если ключ для удаления меньше ключа текущего узла, вызываем функцию удаления для левого поддерева

Delete(&((\*\*d).Left), k);

else if (k > (\*\*d).key) // Если ключ для удаления больше ключа текущего узла, вызываем функцию удаления для правого поддерева

Delete(&((\*\*d).Right), k);

else // Иначе, нашли узел с заданным ключом

{

q = \*d;

s = \*d;

if ((\*q).Right == NULL) // Если у узла нет правого поддерева

{

\*d = (\*q).Left; // Заменяем текущий узел на его левый потомок

delete s; // Удаляем текущий узел

}

else if ((\*q).Left == NULL) // Если у узла нет левого поддерева

{

\*d = (\*q).Right; // Заменяем текущий узел на его правый потомок

delete s; // Удаляем текущий узел

}

else // Если у узла есть оба потомка

del(&((\*q).Left), &(\*q)); // Вызываем функцию del для нахождения узла для замены

}

}

}

void Tree::del(node\*\* r, node\* q)

{

node\* s;

if ((\*\*r).Right == NULL) // Если у правого потомка нет правого поддерева

{

(\*q).key = (\*\*r).key; // Копируем ключ правого потомка в узел q

(\*q).count = (\*\*r).count; // Копируем количество вхождений правого потомка в узел q

q = s = \*r; // Присваиваем указателям q и s адрес правого потомка

\*r = (\*\*r).Left; // Заменяем правого потомка на его левого потомка

delete s; // Удаляем правого потомка

}

else

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

}

void main()

{

setlocale(LC\_ALL, "Rus"); // Устанавливаем локаль на русский для корректного отображения текста

int menu1 = 1, choice, elem1, elem2, menu2; // Объявляем переменные для выбора в меню и значений элементов

ring A; // Создаем экземпляр класса ring

zveno\* Res; // Объявляем указатель на объект zveno

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

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

while (menu1) // Начинаем основной цикл меню.

{

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; cout << endl; // Считываем выбор пользователя

switch (choice) // Начинаем оператор выбора в зависимости от выбора пользователя

{

case 1: // Если выбран пункт 1, выполняем создание структуры

if (\*(A.getring()) == NULL) A.create(); // Проверяем, пустое ли кольцо, затем создаем его.

else cout << "Кольцо уже существует...\n"; // Если не пустое, выводим сообщение

break;

case 2: // Если выбран пункт 2, выполняем просмотр структуры

if (\*(A.getring()) == NULL) cout << "Кольцо пусто...\n"; // Проверяем, пустое ли кольцо, затем выводим сообщение.

else A.look(); // Если не пустое, выводим содержимое кольца

break;

case 3: // Если выбран пункт 3, выполняем добавление элемента после указанного элемента

if (\*(A.getring()) == NULL) cout << "Кольцо пусто...\n"; // Проверяем, пустое ли кольцо, затем выводим сообщение.

else // Если не пустое, выполняем добавление элемента после указанного элемента

{

Res = NULL; // Инициализируем указатель нулем.

cout << "Введите элемент, после которого ";

cout << " хотите добавить звено: ";

cin >> elem1; cout << endl;

if (A.poisk(elem1, &Res))

{

cout << "Введите элемент, который ";

cout << "хотите добавить: ";

cin >> elem2;

cout << endl;

A.add\_after(elem2, Res);

}

else

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

}

break;

case 4: // Если выбран пункт 4, выполняем добавление элемента перед указанным элементом

if (\*(A.getring()) == NULL) cout << "Кольцо пусто...\n"; // Проверяем, пустое ли кольцо, затем выводим сообщение

else // Если не пустое, выполняем добавление элемента перед указанным элементом

{

Res = NULL; // Инициализируем указатель нулем.

cout << "Введите элемент, перед которым ";

cout << " хотите добавить звено: ";

cin >> elem1; cout << endl;

if (A.poisk(elem1, &Res))

{

cout << "Введите элемент, который ";

cout << "хотите добавить: ";

cin >> elem2;

cout << endl;

A.add\_befor(elem2, Res);

}

else

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

}

break;

case 5: // Если выбран пункт 5, выполняем удаление указанного элемента

if (\*(A.getring()) == NULL) cout << "Кольцо пусто...\n"; // Проверяем, пустое ли кольцо, затем выводим сообщение.

else // Если не пустое, выполняем удаление указанного элемента

{

Res = NULL; // Инициализируем указатель нулем.

cout << "Введите элемент, который";

cout << " хотите удалить: ";

cin >> elem1; cout << endl;

if (A.poisk(elem1, &Res)) A.Delete(Res);

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

}

break;

case 6: // Если выбран пункт 6, выполняем удаление элемента после указанного элемента

if (\*(A.getring()) == NULL) cout << "Кольцо пусто...\n"; // Проверяем, пустое ли кольцо, затем выводим сообщение.

else // Если не пустое, выполняем удаление элемента после указанного элемента

{

Res = NULL; // Инициализируем указатель нулем.

cout << "Введите элемент, после которого";

cout << " хотите удалить: ";

cin >> elem1; cout << endl;

if (A.poisk(elem1, &Res)) A.delete\_next(Res);

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

}

break;

case 7: // Если выбран пункт 7, выполняем преобразование дерева заданного элемента

if (\*(A.getring()) == NULL) cout << "Кольцо пусто...\n"; // Проверяем, пустое ли кольцо, затем выводим сообщение.

else // Если не пустое, выполняем преобразование дерева заданного элемента

{

Res = NULL; // Инициализируем указатель нулем

cout << "Введите элемент кольца: ";

cin >> elem1; cout << endl;

if (A.poisk(elem1, &Res))

{

menu2 = 1;

while (menu2)

{

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; cout << endl;

switch (choice)

{

case 1: // Если выбран пункт 1, выполняем создание дерева

if ((\*Res).ukTree.getTree() == NULL)

(\*Res).ukTree.creat\_Tree();

else cout << "Дерево существует...\n";

break;

case 2: (\*Res).ukTree.look\_Tree(); break; // Если выбран пункт 2, выполняем просмотр дерева

case 3: (\*Res).ukTree.add\_Tree(); break; // Если выбран пункт 3, выполняем добавление элемента в дерево

case 4: (\*Res).ukTree.delete\_Tree(); break; // Если выбран пункт 4, выполняем удаление элемента из дерева

case 5: // Если выбран пункт 5, выполняем удаление дерева

if ((\*Res).ukTree.getTree() == NULL)

cout << "Дерево не существует...\n";

else (\*Res).ukTree.~Tree();

break;

case 6: menu2 = 0; break; // Если выбран пункт 6, выходим из подменю

}

}

}

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

}

break;

case 8: // Если выбран пункт 8, выполняем удаление структуры

if (\*(A.getring()) == NULL) cout << "Кольцо пусто...\n"; // Проверяем, пустое ли кольцо, затем выводим сообщение

else A.~ring(); // Если не пустое, выполняем удаление структуры

break;

case 9: // Если выбран пункт 9, завершаем работу программы

A.~ring(); // Удаляем кольцо

menu1 = 0; // Выходим из цикла основного меню

break;

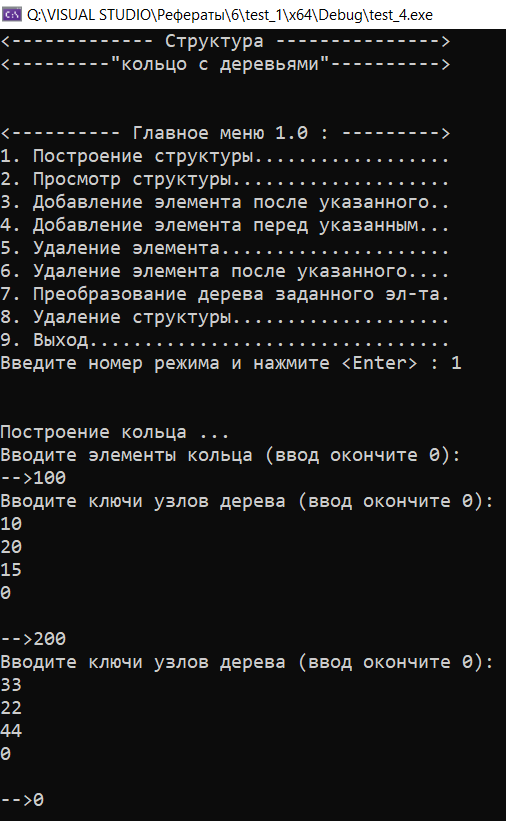
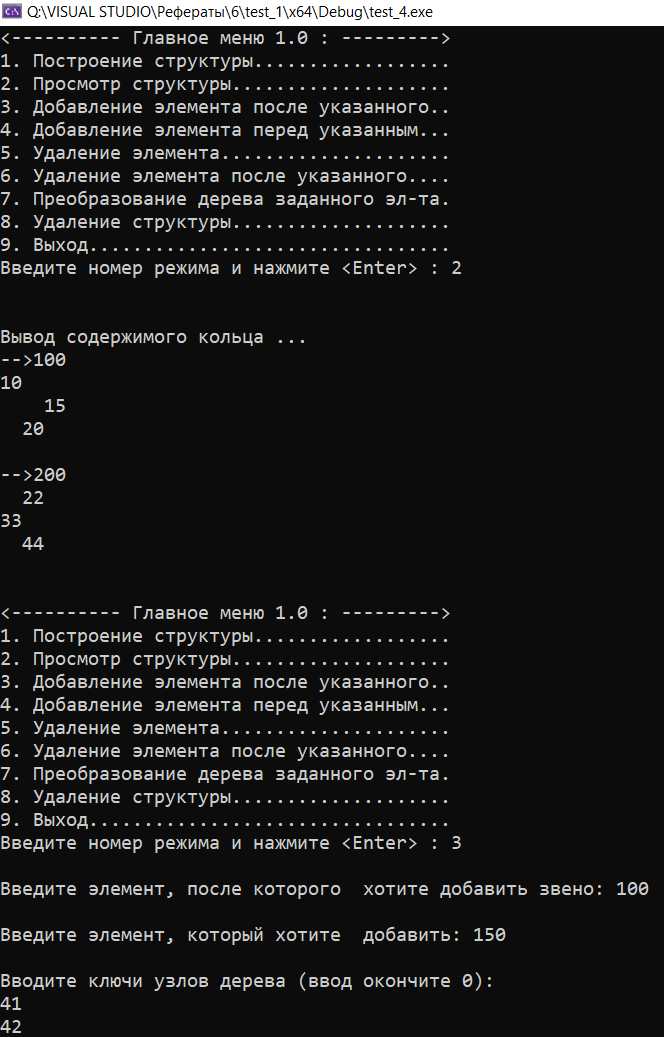
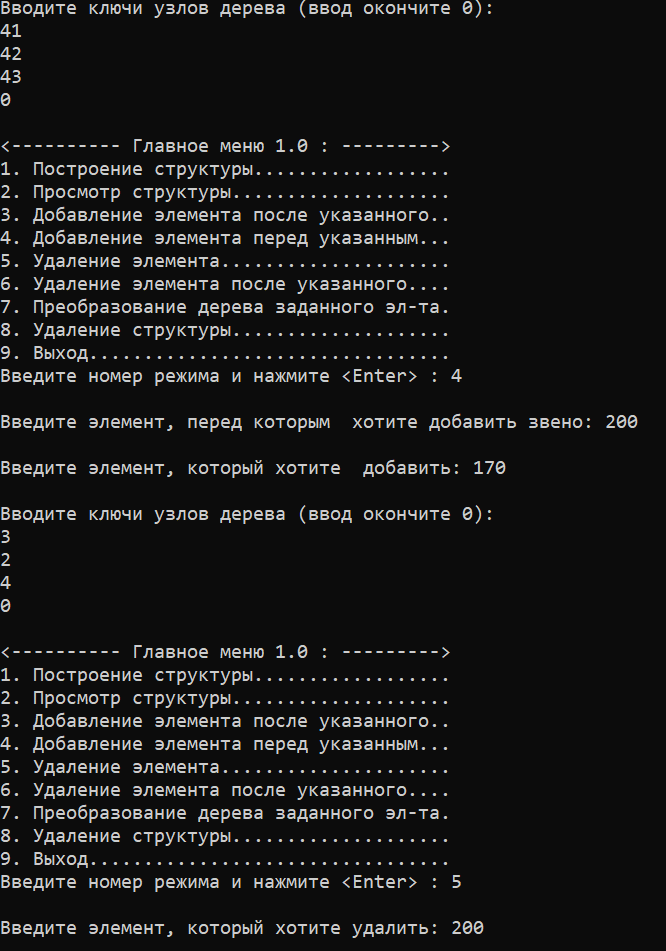
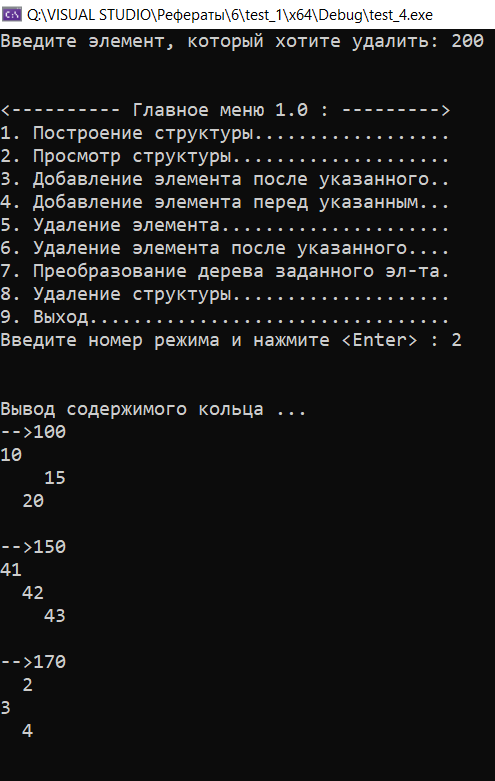
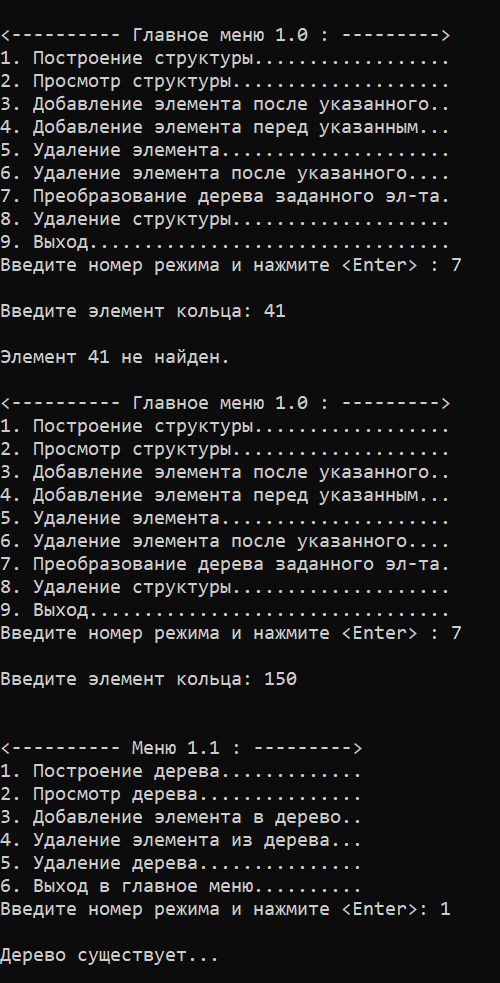
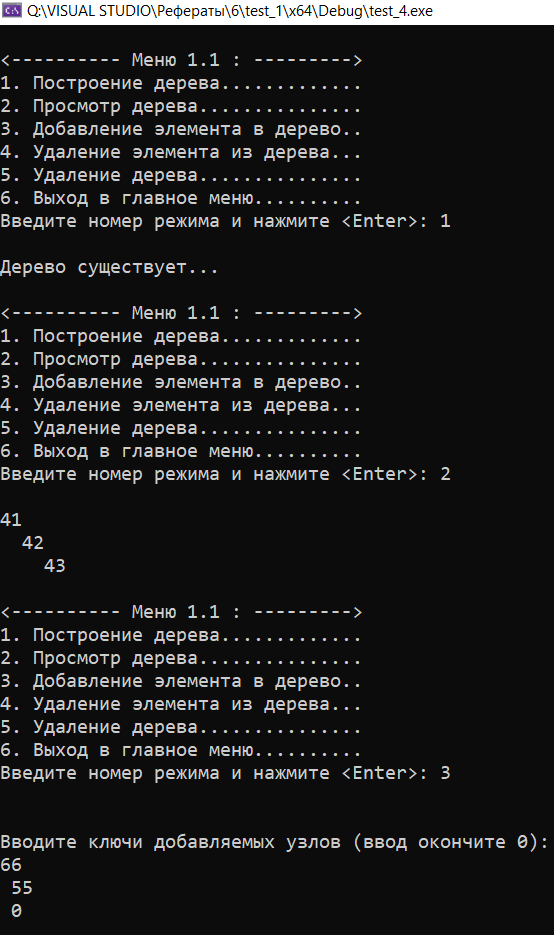
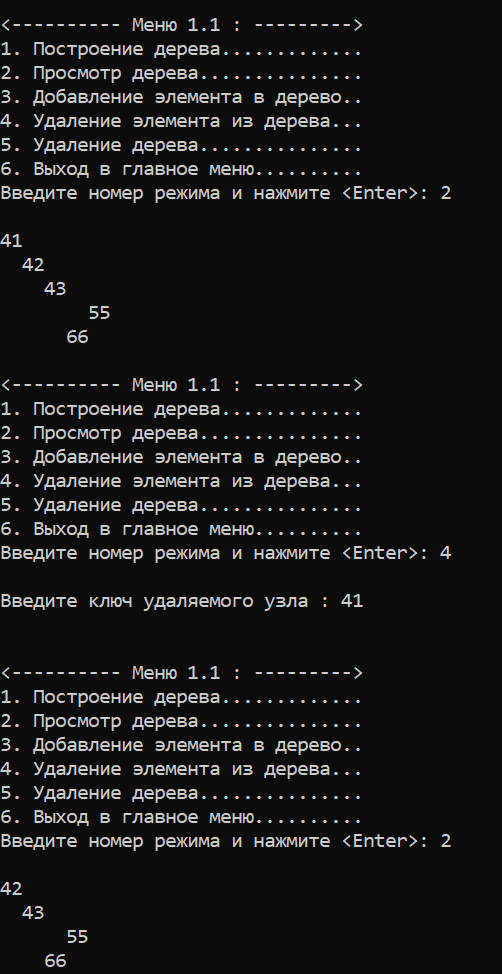
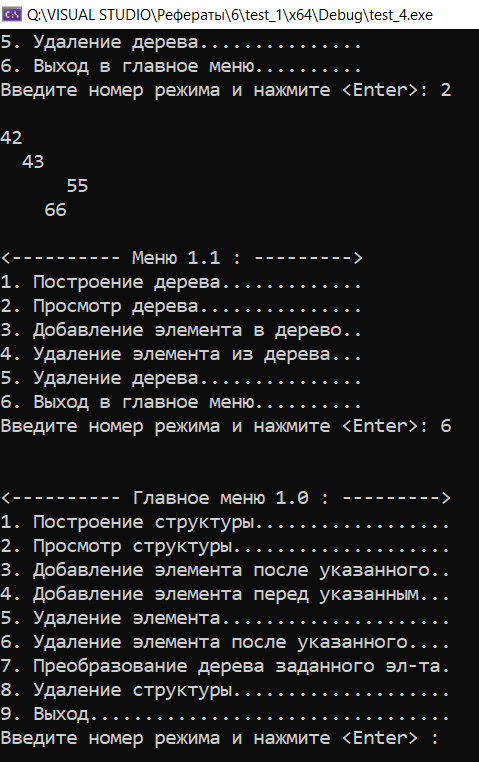
} // Конец оператора выбора

} // Конец while

cout << "\n";

system("PAUSE"); // Приостанавливаем выполнение программы

}

Код Тестирования 5(Реализация алгоpитма Хаффмена с помощью деpева)

#include <stdio.h>

#include <conio.h>

#include <iostream>

using namespace std;

// Структура для представления звена в дереве Хаффмана

struct zveno

{

char Element; // Символ

float Kol; // Количество повторений, частота повторений

zveno\* Sled; // Указатель на следующее звено в списке

zveno\* Left; // Указатель на левое поддерево

zveno\* Right; // Указатель на правое поддерево

zveno\* Father; // Указатель на родителя

};

// Класс для работы с деревом Хаффмана

class Tree

{

private:

zveno\* UkStr; // Указатель на список

int Poisk1(zveno\*\*, float, zveno\*\*); // Вспомогательная функция поиска места в упорядоченном списке

public:

Tree() { UkStr = new (zveno); UkStr->Sled = NULL; }; // Конструктор

int Poisk(char, zveno\*\*); // Поиск элемента в списке

int Kolich(char\*, char); // Подсчет количества повторений символа в тексте

void Dobavlenie(char, float, zveno\*\*); // Добавление звена в список

void Redaktor(int); // Изменение поля Kol на частоту повторений

void Ukazateli(zveno\*\*, zveno\*\*); // Поиск указателей на предпоследний и предпредпоследний элементы

void Vyvod(); // Вывод списка на экран

void WstawkaSort(zveno\*); // Вставка элемента в упорядоченный список

void PrintTree(zveno\*, int); // Печать дерева

zveno\*\* GetTree() { return &UkStr; }; // Получение указателя на список

zveno\* GetTree1() { return UkStr; }; // Получение списка

};

// Метод для поиска элемента в списке по символу

int Tree::Poisk(char ENT, zveno\*\* Res)

{

zveno\* q; // Указатель на текущее звено списка

int vozvr = 0; // Переменная для возврата результата поиска (1 - если найдено, 0 - если не найдено)

\*Res = NULL; // Инициализация указателя на найденный элемент

q = (\*UkStr).Sled; // Установка указателя на первый элемент списка (после заглавного звена)

// Проход по списку

while (q != NULL && \*Res == NULL)

{

// Проверка, содержит ли текущее звено искомый символ

if (q->Element == ENT)

{

vozvr = 1; // Устанавливаем результат поиска в "найдено"

\*Res = q; // Сохраняем указатель на найденное звено

return vozvr; // Возвращаем результат поиска

}

q = q->Sled; // Переходим к следующему звену списка

}

return vozvr; // Возвращаем результат поиска

}

// Метод для поиска места в упорядоченном списке для добавления элемента

int Tree::Poisk1(zveno\*\* st, float kol, zveno\*\* Res)

{

zveno\* q = (\*\*st).Sled, \* q1 = (\*st); // Указатели на текущее и предыдущее звено

int vozvr = 0; // Переменная для возврата результата поиска (1 - если найдено место для вставки, 0 - если не найдено)

\*Res = NULL; // Инициализация указателя на место для вставки

// Проход по списку

while (q != NULL && \*Res == NULL)

{

// Проверка условия для вставки элемента (если текущий элемент имеет меньшее значение Kol)

if (q->Kol < kol)

{

vozvr = 1; // Устанавливаем результат поиска в "найдено место для вставки"

\*Res = q; // Сохраняем указатель на место для вставки

}

q = q->Sled; // Переходим к следующему звену списка

q1 = q1->Sled; // Переходим к следующему звену списка (для обновления указателя на предыдущее звено)

}

if (\*Res == NULL)

\*Res = q1; // Если место для вставки не было найдено, устанавливаем указатель на последнее звено

return vozvr; // Возвращаем результат поиска

}

// Метод для подсчета количества повторений символа в тексте

int Tree::Kolich(char\* F, char S)

{

int K = 0; // Переменная для хранения количества повторений символа

// Проход по тексту

for (int i = 0; i < strlen(F); i++)

// Проверка, содержит ли текущий символ искомый символ

if (F[i] == S)

K++; // Увеличиваем счетчик повторений

return K; // Возвращаем количество повторений

}

// Метод для изменения поля Kol на частоту повторений

void Tree::Redaktor(int L)

{

zveno\* q = (\*UkStr).Sled; // Указатель на текущее звено списка

// Проход по списку

while (q != NULL)

{

q->Kol = q->Kol / L; // Изменяем значение поля Kol на частоту повторений

q = q->Sled; // Переходим к следующему звену списка

}

}

// Метод для добавления звена в список, упорядоченный по количеству повторений

void Tree::Dobavlenie(char bukva, float kol, zveno\*\* Sp)

{

zveno\* q, \* Res = NULL, \* kladovaq; // Объявление указателей на звено и вспомогательные переменные

q = new (zveno); // Создание нового звена

q->Element = bukva; // Установка символа в звене

q->Kol = kol; // Установка количества повторений символа в звене

q->Left = q->Right = NULL; // Установка указателей на левое и правое поддерево в NULL

q->Sled = q->Father = NULL; // Установка указателей на следующее звено и на родителя в NULL

// Если список пуст

if ((\*\*Sp).Sled == NULL)

(\*\*Sp).Sled = q; // Устанавливаем новое звено как первое в списке

else

// Если в списке уже есть элементы

if (Poisk1(&(\*Sp), kol, &Res)) // Поиск места для вставки нового звена

{

kladovaq = new (zveno); // Создание нового звена для хранения ссылки на следующее звено

(\*kladovaq) = (\*Res); // Копирование значений звена Res в новое звено kladovaq

(\*Res) = (\*q); // Замена звена Res на новое звено q

Res->Sled = kladovaq; // Установка ссылки на следующее звено

}

else

Res->Sled = q; // Простое добавление звена в конец списка

}

// Метод для поиска указателей на предпоследний и предпредпоследний элементы

void Tree::Ukazateli(zveno\*\* zv, zveno\*\* zv\_p)

{

\*zv\_p = UkStr->Sled; // Устанавливаем указатель на первый элемент списка

\*zv = UkStr; // Устанавливаем указатель на заглавное звено

// Пока у предпоследнего звена есть следующее звено

while ((\*zv\_p)->Sled->Sled != NULL)

{

\*zv = \*zv\_p; // Перемещаем указатель на текущее звено

\*zv\_p = (\*zv\_p)->Sled; // Перемещаем указатель на следующее звено

}

}

void Tree::Vyvod()

{

zveno\* q = UkStr->Sled; // Устанавливаем указатель на первый элемент списка

while (q != NULL)

{

cout << q->Element << " (" << q->Kol << ") --> "; // Выводим символ и количество повторений этого символа

q = q->Sled; // Переходим к следующему звену списка

}

cout << endl;

}

void Tree::WstawkaSort(zveno\* zv)

{

zveno\* w1, \* w2; // Указатели на текущее и предыдущее звено

w2 = UkStr; // Устанавливаем указатель w2 на заглавное звено

w1 = w2->Sled; // Устанавливаем указатель w1 на первый элемент списка

// Пока не достигнут конец списка и количества повторений текущего элемента больше чем у вставляемого

while (w1 != NULL && w1->Kol > zv->Kol)

{

w2 = w1; // Перемещаем указатель w2 на следующий элемент списка

w1 = w2->Sled; // Перемещаем указатель w1 на следующий элемент списка

}

// Если достигнут конец списка или количество повторений текущего элемента меньше или равно количеству повторений вставляемого элемента

if (w1 == NULL || w1->Kol <= zv->Kol)

{

w2->Sled = zv; // Вставляем новое звено после предыдущего звена

zv->Sled = w1; // Устанавливаем связь между новым звеном и следующим звеном списка

}

}

void Tree::PrintTree(zveno\* w, int l)

{

// Проверяем, не является ли текущее звено пустым

if (w != NULL)

{

// Рекурсивно вызываем функцию для правого поддерева с увеличением уровня на 1

PrintTree(w->Right, l + 1);

// Выводим отступы для текущего уровня

for (int i = 1; i <= l; i++)

cout << " ";

// Выводим символ и количество повторений этого символа

cout << w->Element << " (" << w->Kol << ")\n";

// Рекурсивно вызываем функцию для левого поддерева с увеличением уровня на 1

PrintTree(w->Left, l + 1);

}

}

void main()

{

setlocale(LC\_ALL, "Rus");

Tree A; // Создание объекта класса Tree с именем A.

char T[255]; // Исходная строка.

int i, j; // Переменные для циклов.

zveno\* Res = NULL; // Указатель на результат.

zveno\* Q[256]; // Массив указателей на элементы списка.

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

gets\_s(T); // Получение текста от пользователя.

// Формирование списка, содержащего символы текста.

for (i = 0; i < strlen(T); i++)

{

if (!A.Poisk(T[i], &Res)) // Если символ еще не встречался в списке.

A.Dobavlenie(T[i], A.Kolich(T, T[i]), A.GetTree()); // Добавляем символ в список.

}

// ------------------------------- //

A.Redaktor(strlen(T)); // Замена количества повторений символов на частоту повторений.

cout << "Полученный список:\n";

A.Vyvod(); // Вывод списка на экран.

// Заполнение массива Q указателями на элементы списка.

zveno\* UkZv = A.GetTree1()->Sled, \* UkZv\_p = NULL, \* Sli; // Инициализация указателей на элементы списка.

i = 0; // Счетчик.

while (UkZv != NULL)

{

Q[i] = UkZv; // Заполнение массива указателями на элементы списка.

i++; // Увеличение счетчика.

UkZv = UkZv->Sled; // Переход к следующему элементу списка.

}

// Построение дерева Хаффмана.

while (A.GetTree1()->Sled->Sled != NULL)

{

A.Ukazateli(&UkZv, &UkZv\_p); // Поиск указателей на последний и предпоследний элементы списка.

// Слияние последнего и предпоследнего звеньев.

Sli = new (zveno);

Sli->Element = '\*'; // Обозначение слияния звеньев.

Sli->Kol = UkZv\_p->Kol + UkZv\_p->Sled->Kol; // Вычисление суммарного количества повторений.

Sli->Left = UkZv\_p; // Левое поддерево - предпоследний элемент списка

Sli->Right = UkZv\_p->Sled; // Правое поддерево - последний элемент списка

Sli->Father = Sli->Sled = NULL; // Инициализация указателей

UkZv\_p->Father = Sli; // Установка указателя на отец для предпоследнего элемента

UkZv\_p->Sled->Father = Sli; // Установка указателя на отец для последнего элемента

UkZv->Sled = NULL; // Удаление ссылки на последний элемент списка

UkZv\_p->Sled = NULL; // Удаление ссылки на предпоследний элемент списка

// Добавление нового звена в список

if (A.GetTree1()->Sled == NULL)

A.GetTree1()->Sled = Sli; // Если список пустой

else

A.WstawkaSort(Sli); // Если список не пустой, вставляем новое звено с сортировкой

}

cout << "Построим дерево...\n";

A.PrintTree(A.GetTree1()->Sled, 0); // Вывод дерева на экран

cout << "--------------------------------------------- " << endl; // Разделительная строка.

// Кодирование введенного текста

cout << "Приступим к кодировке введенного текста...\n";

char Cod\_symbol[40]; // Буфер для кода символа.

char Cod\_Haffmen[255]; // Код Хаффмана строки T

char temp[255]; // Временный буфер для конкатенации строк

strcpy\_s(Cod\_symbol, ""); // Очистка буфера для кода символа

strcpy\_s(Cod\_Haffmen, ""); // Очистка буфера для кода Хаффмана

for (i = 0; i < strlen(T); i++)

{

// Начинаем поиск нужного указателя

j = 0;

while (Q[j]->Element != T[i])

j++; // Поиск указателя на символ в списке.

// А теперь начинаем "восхождение"...

UkZv = Q[j]; // Установка указателя на нужный элемент списка.

while (UkZv->Father != NULL)

{

if (UkZv->Father->Left == UkZv)

{

strcpy\_s(temp, "1"); // Если текущий элемент является левым потомком, добавляем "1" в код символа

strcat\_s(temp, Cod\_symbol); // Добавление текущего кода символа во временный буфер

strcpy\_s(Cod\_symbol, temp); // Обновление кода символа

UkZv = UkZv->Father; // Переход к отцу

}

else

{

strcpy\_s(temp, "0"); // Если текущий элемент является правым потомком, добавляем "0" в код символа

strcat\_s(temp, Cod\_symbol); // Добавление текущего кода символа во временный буфер

strcpy\_s(Cod\_symbol, temp); // Обновление кода символа.

UkZv = UkZv->Father; // Переход к отцу.

}

}

strcat\_s(Cod\_Haffmen, Cod\_symbol); // Добавление кода символа в код Хаффмана

strcpy\_s(Cod\_symbol, ""); // Очистка буфера для кода символа.

}

cout << "Код перед Вами... " << Cod\_Haffmen << endl; // Вывод кода Хаффмана.

cout << "Коэффициент сжатия: " << 100 \* strlen(Cod\_Haffmen) / 8.0 / strlen(T) << "%\n"; // Рассчет и вывод коэффициента сжатия

// Расшифровка закодированного сообщения

cout << "Ранее было зашифровано... " << T << endl; // Вывод оригинальной строки

strcpy\_s(T, ""); // Очистка строки для расшифровки

// Установим указатель на корень дерева

UkZv = A.GetTree1()->Sled; // Установка указателя на корень дерева

i = 0; // Счетчик

while (i < strlen(Cod\_Haffmen))

{

while (UkZv->Left != NULL && UkZv->Right != NULL)

{

if (Cod\_Haffmen[i] == '1')

UkZv = UkZv->Left; // Если встречается "1", двигаемся влево

else

UkZv = UkZv->Right; // Если встречается "0", двигаемся вправо

i++; // Увеличиваем счетчик.

}

char s[2];

s[0] = UkZv->Element;

s[1] = '\0'; // Завершаем строку нулем

strcat\_s(T, s); // Добавляем символ к расшифрованной строке

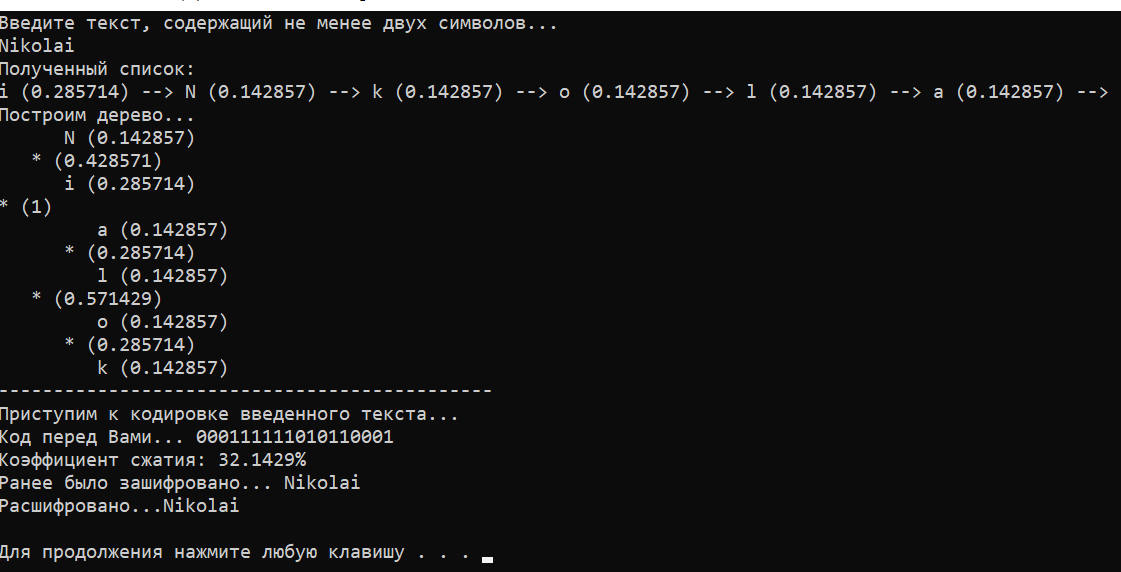
UkZv = A.GetTree1()->Sled; // Возвращаемся к корню дерева

}

cout << "Расшифровано..." << T << endl;

cout << "\n";

system("PAUSE"); }



# Код Тестирования 6(Hе pекуpсивная пpогpамма построения дерева-формулы)

#include <stdio.h>

#include <conio.h>

#include <iostream>

using namespace std;

struct Uzel // Объявление структуры для узла дерева

{

char Key; // Символ.

Uzel\* Left; // Указатель на левое поддерево

Uzel\* Right; // Указатель на правое поддерево

};

struct zveno // Объявление структуры для элемента стека

{

Uzel\* Element; // Указатель на узел дерева

zveno\* Sled; // Указатель на следующий элемент стека

};

class Tree // Определение класса Tree

{

private:

Uzel\* Root; // Указатель на корень дерева

zveno\* Stack; // Указатель на вершину стек

public:

Tree(); // Конструктор класса Tree

void Udalenie(Uzel\*\*); // Метод удаления элемента из стека

void V\_stack(Uzel\*); // Метод помещения элемента в стек

void PrintTree(Uzel\*, int); // Метод для вывода дерева на экран

void Print\_Tree\_Left(Uzel\*, int); // Метод для левостороннего обхода дерева

void Print\_Tree\_End(Uzel\*, int); // Метод для концевого обхода дерева

void Print\_Tree\_Back(Uzel\*, int); // Метод для обратного обхода дерева

Uzel\* GetTree() { return Root; }; // Метод для получения указателя на корень дерева

};

void Tree::V\_stack(Uzel\* Elem) // Определение метода V\_stack

{

zveno\* q = new (zveno); // Создание нового элемента стека

q->Element = Elem; // Присвоение переданного узла элементу стека

q->Sled = Stack; // Установка ссылки на следующий элемент стека

Stack = q; // Обновление указателя на вершину стека

}

void Tree::Udalenie(Uzel\*\* tmp) // Определение метода Udalenie

{

zveno\* q; // Объявление указателя на элемент стека

if (Stack != NULL) // Проверка, не пуст ли стек

{

(\*tmp) = Stack->Element; // Присвоение указателю на переданный узел значения вершины стека.

q = Stack; // Присвоение указателю q значения вершины стека

Stack = Stack->Sled; // Перемещение указателя на следующий элемент стека

delete q; // Освобождение памяти, занимаемой вершиной стека

}

}

void Tree::PrintTree(Uzel\* w, int l) // Определение метода PrintTree

// Вывод дерева на экран

{

if (w != NULL) // Проверка, не пусто ли дерево

{

PrintTree(w->Right, l + 1); // Рекурсивный вызов PrintTree для правого поддерева

for (int i = 1; i <= l; i++) cout << " "; // Вывод отступа для текущего узла

cout << w->Key << endl; // Вывод символа текущего узла

PrintTree(w->Left, l + 1); // Рекурсивный вызов PrintTree для левого поддерева

}

}

void Tree::Print\_Tree\_Left(Uzel\* w, int l) // Определение метода Print\_Tree\_Left

// Левосторонний обход бинарного дерева

{

if (w != NULL) // Проверка, не пусто ли дерево

{

cout << w->Key << " "; // Вывод символа текущего узла

Print\_Tree\_Left(w->Left, l + 1); // Рекурсивный вызов Print\_Tree\_Left для левого поддерева

Print\_Tree\_Left(w->Right, l + 1); // Рекурсивный вызов Print\_Tree\_Left для правого поддерева

}

}

void Tree::Print\_Tree\_End(Uzel\* w, int l) // Определение метода Print\_Tree\_End.

// Концевой обход бинарного дерева

{

if (w != NULL) // Проверка, не пусто ли дерево

{

Print\_Tree\_End(w->Left, l + 1); // Рекурсивный вызов Print\_Tree\_End для левого поддерева

Print\_Tree\_End(w->Right, l + 1); // Рекурсивный вызов Print\_Tree\_End для правого поддерева

cout << w->Key << " ";

}

}

void Tree::Print\_Tree\_Back(Uzel\* w, int l) // Определение метода Print\_Tree\_Back

// Обратный обход бинарного дерева

{

if (w != NULL) // Проверка, не пусто ли дерево

{

cout << "(";

Print\_Tree\_Back(w->Left, l + 1); // Рекурсивный вызов Print\_Tree\_Back для левого поддерева

cout << w->Key << " ";

Print\_Tree\_Back(w->Right, l + 1); // Рекурсивный вызов Print\_Tree\_Back для правого поддерева

cout << ")";

}

}

Tree::Tree() // Определение конструктора класса Tree

{

Stack = NULL; // Инициализация указателя на вершину стека как NULL

// Формирование заглавного узла дерева

Root = new (Uzel); // Выделение памяти под корневой узел

Root->Right = NULL; // Установка правого поддерева корня как NULL

}

void main() // Основная функция программы

{

setlocale(LC\_ALL, "Rus");

char Formula\_Post[30]; // Объявление массива для хранения постфиксной формы выражения

char k; // Вспомогательная переменная

Uzel\* Ukazatel = NULL; // Объявление указателя на узел

cout << "Введите формулу, записанную в постфиксной форме... \n";

gets\_s(Formula\_Post); // Ввод формулы

// Получили "перевертыш" слова Formula\_Post

\_strrev(Formula\_Post); // Реверс строки Formula\_Post

cout << "Приступим к построению дерева-формулы...\n";

Tree A; // Создание объекта класса Tree.

Uzel\* Temp = A.GetTree(); // Получение указателя на корень дерева

// Формирование дерева постфиксной формы и вывод его на экран

for (int i = 0; i < strlen(Formula\_Post); i++) // Цикл по символам постфиксной формы выражения

{

k = Formula\_Post[i]; // Присвоение k очередного символа формулы

// Переходим к анализу символа k.

if (strchr("+-\*/^", k) != NULL) // Если символ - операция

{

if (Temp->Right == NULL) // Если у текущего узла нет правого поддерева

{

// Резервирование места для вставляемого узла

Temp->Right = new (Uzel);

// Установка указателя на вставляемый узел

Temp = Temp->Right;

// Инициализация вставляемого узла

Temp->Key = k;

Temp->Left = Temp->Right = NULL;

// Ссылка на предыдущий узел --> стек

A.V\_stack(Temp);

}

else // Если у текущего узла есть правое поддерево

{

// Выделение места для вставляемого узла

Temp->Left = new (Uzel);

// Установка указателя на вставляемый узел

Temp = Temp->Left;

// Инициализация вставляемого узла

Temp->Key = k;

Temp->Left = Temp->Right = NULL;

// Ссылка на предыдущий узел --> стек

A.V\_stack(Temp);

}

}

else

{

if (Temp->Right == NULL) // Если у текущего узла нет правого поддерева

{

// Выделение места для вставляемого узла

Temp->Right = new (Uzel);

// Установка указателя на вставляемый узел

Temp = Temp->Right;

// Инициализация вставляемого узла

Temp->Key = k;

Temp->Left = Temp->Right = NULL;

// Текущий указатель "возвращается" назад

A.Udalenie(&Ukazatel);

Temp = Ukazatel;

}

else // Если у текущего узла есть правое поддерево

{

// Выдедние места для вставляемого узла

Temp->Left = new (Uzel);

// Установка указателя на вставляемый узел

Temp = Temp->Left;

// Инициализация вставляемого узла

Temp->Key = k;

Temp->Left = Temp->Right = NULL;

// Текущий указатель "возвращается" назад

A.Udalenie(&Ukazatel);

Temp = Ukazatel;

}

}

} // Конец цикла for.

cout << "\nКонтрольный вывод дерева-формулы...\n";

A.PrintTree(A.GetTree()->Right, 0); // Вывод дерева на экран.

cout << "Пред Вами формула, записанная в инфиксной форме...\n";

A.Print\_Tree\_Back(A.GetTree()->Right, 0); // Вывод формулы в инфиксной форме

cout << endl;

cout << "------------------------------------------ \n"; // Вывод разделителя

cout << "Пред Вами формула, записанная в префиксной форме...\n";

A.Print\_Tree\_Left(A.GetTree()->Right, 0); // Вывод формулы в префиксной форме

cout << endl;

cout << "------------------------------------------ \n"; // Вывод разделителя

cout << "Пред Вами формула, записанная в постфиксной форме...\n";

A.Print\_Tree\_End(A.GetTree()->Right, 0); // Вывод формулы в постфиксной форме

cout << "\n";

system("PAUSE");

}

