Белорусский государственный технологический университет

Факультет информационных технологий

Кафедра программной инженерии

 Реферат

По дисциплине «Основы алгоритмизации и программирования»

На тему «Тестирование кода бинарного дерева с рекурсией и без, хеширование с помощью леса, древовидно-кольцевой структуры, алгоритма Хаффмена, построение дерева-формулы»

Выполнил:

Студент 1 курса 8 группы

Лужецкий Владислав Константинович

Проверил:

Белодед Николай Иванович

2024, Минск

Оглавление

[Тестирование кода бинарного дерева с рекурсией 2](#_Toc165459001)

[Тестирование кода бинарного дерева без рекурсии 5](#_Toc165459002)

[Тестирование кода хеширования с помощью леса 9](#_Toc165459003)

[Тестирование кода древовидно-кольцевой структура 14](#_Toc165459004)

[Тестирование кода алгоритма Хаффмена 30](#_Toc165459005)

[Тестирование кода построения дерева-формулы 36](#_Toc165459006)

# Тестирование кода бинарного дерева с рекурсией

#include <iostream>

#include <Windows.h>

using namespace std;

struct node {

int key; // Ключ узла

int count; // Счетчик

node\* left; // Указатель на левое поддерево

node\* right; // Указатель на правое поддерево

};

class TREE {

private:

node\* Tree; // Указатель на корень дерева

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

public:

TREE() { // Конструктор класса

Tree = NULL; // Инициализация указателя в NULL

}

node\*\* GetTree() { // Функция для получения указателя на корень дерева

return &Tree; // Возвращает адрес указателя на корень дерева

}

void BuildTree(); // Функция для построения дерева

void CleanTree(node\*\*); // Функция для очистки дерева

void ObhodEnd(node\*\*); // Функция для обхода дерева (концевой)

void ObhodLeft(node\*\*); // Функция для обхода дерева (Левосторонний)

void ObhodBack(node\*\*); // Функция для обхода дерева (Обратный)

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

int Height(node\*\*); // Функция для вычисления высоты дерева

};

int main() {

SetConsoleOutputCP(1251);

SetConsoleCP(1251);

TREE A;

A.BuildTree(); // Построение дерева

cout << "\nВывод дерева\n";

A.Vyvod(A.GetTree(), 0); // Вывод дерева на экран

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

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

A.ObhodLeft(A.GetTree()); // Левосторонний обход дерева

cout << "\nКонцевой обход дерева\n";

A.ObhodEnd(A.GetTree()); // Концевой обход дерева

cout << "\nОбратный обход дерева\n";

A.ObhodBack(A.GetTree()); // Обратный обход дерева

A.CleanTree(A.GetTree()); // Очистка дерева

cout << '\n';

}

void TREE::BuildTree() {

int el;

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

cin >> el; // Ввод элемента

while (el != 0) {

Search(el, &Tree); // Поиск места для вставки нового узла в дерево

cin >> el;

}

}

void TREE::Search(int x, node\*\* p) {

if (\*p == NULL) {

\*p = new node; // Выделение памяти для нового узла

(\*\*p).key = x; // Установка значения ключа нового узла

(\*\*p).count = 1; // Инициализация счетчика нового узла в 1

(\*\*p).left = NULL; // Инициализация указателя на левое поддерево в NULL

(\*\*p).right = NULL; // Инициализация указателя на правое поддерево в NULL

}

else {

if (x < (\*\*p).key) { // Если x меньше текущего узла

Search(x, &((\*\*p).left)); // Рекурсивный вызов функции Search для левого поддерева

}

else {

if (x > (\*\*p).key) { // Если x больше текущего узла

Search(x, &((\*\*p).right)); // Рекурсивный вызов функции Search для правого поддерева

}

else { // Если x равен текущему узлу

(\*\*p).count = (\*\*p).count + 1; // Увеличение счетчика текущего узла на 1

}

}

}

}

void TREE::ObhodLeft(node\*\* w) {

if (\*w != NULL) { // Если указатель w != NULL

cout << (\*\*w).key << ' '; // Вывод значения ключа текущего узла

ObhodLeft(&((\*\*w).left)); // Рекурсивный вызов функции для левого поддерева

ObhodLeft(&((\*\*w).right)); // Рекурсивный вызов функции для правого поддерева

}

}

void TREE::ObhodEnd(node\*\* w) {

if (\*w != NULL) { // Если указатель w != NULL

ObhodEnd(&((\*\*w).left)); // Рекурсивный вызов функции для левого поддерева

ObhodEnd(&((\*\*w).right)); // Рекурсивный вызов функции для правого поддерева

cout << (\*\*w).key << ' '; // Вывод значения ключа текущего узла

}

}

void TREE::ObhodBack(node\*\* w) {

if (\*w != NULL) { // Если указатель != NULL

ObhodBack(&((\*\*w).left)); // Рекурсивный вызов функции для левого поддерева

cout << (\*\*w).key << ' '; // Вывод значения ключа текущего узла

ObhodBack(&((\*\*w).right)); // Рекурсивный вызов функции для правого поддерева

}

}

void TREE::CleanTree(node\*\* w) {

if (\*w != NULL) { // Если указатель w != NULL

CleanTree(&((\*\*w).left)); // Рекурсивный вызов функции для левого поддерева

CleanTree(&((\*\*w).right)); // Рекурсивный вызов функции для правого поддерева

delete\* w; // Освобождение памяти

}

}

void TREE::Vyvod(node\*\* w, int l) {

int i; // Переменная для цикла

if (\*w != NULL) { // Если указатель w != NULL

Vyvod(&((\*\*w).right), l + 1); // Рекурсивный вызов функции для правого поддерева с увеличением уровня на 1

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

cout << " "; // Вывод пробелов для отступа в соответствии с уровнем дерева

}

cout << (\*\*w).key << '\n'; // Вывод значения ключа текущего узла на новой строке

Vyvod(&((\*\*w).left), l + 1); // Рекурсивный вызов функции для левого поддерева с увеличением уровня на 1

}

}

int TREE::Height(node\*\* w) {

int h1 = 0, h2 = 0; // Переменные для хранения высот левого и правого поддеревьев

if (\*w == NULL) { // Если указатель w = NULL, то дерево пустое и высота равна -1

return (-1);

}

h1 = Height(&((\*\*w).left)); // Рекурсивный вызов функции для левого поддерева

h2 = Height(&((\*\*w).right)); // Рекурсивный вызов функции для правого поддерева

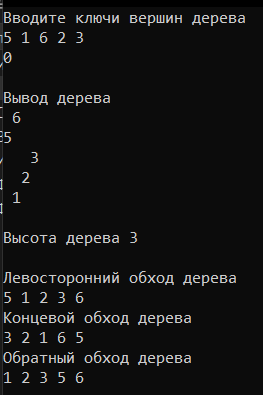
if (h1 > h2) { // Если высота левого поддерева больше высоты правого поддерева

return (1 + h1); // Возвращается высота левого поддерева, увеличенная на 1

}

return (1 + h2); // Возвращается высота правого поддерева, увеличенная на 1

}



# Тестирование кода бинарного дерева без рекурсии

#include <iostream>

#include <Windows.h>

using namespace std;

struct node {

int Key; // Ключ узла

int Count; // Счетчик узла

node\* Left; // Указатель на левое поддерево

node\* Right; // Указатель на правое поддерево

};

struct no {

node\* elem; // Указатель на узел

int ch;

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

};

class TREE {

private:

node\* Tree; // Указатель на корень дерева

void PushStack(no\*\*, node\*\*, int\*); // Функция для добавления элемента в стек

void PopStack(no\*\*, node\*\*, int\*); // Функция для удаления элемента из стека

void VyvodStack(no\*\*); // Функция для вывода стека

public:

TREE() {

Tree = new(node); // Выделение памяти для корневого узла дерева

(\*Tree).Right = NULL; // Инициализация указателя на правое поддерево как NULL

}

node\* GetTreeRight() {

return (\*Tree).Right; // Возвращаем указатель на правое поддерево корня дерева

}

void TreeSearch(int); // Функция поиска узла в дереве

void VyvodTree(node\*); // Функция вывода дерева

};

void main()

{

SetConsoleCP(1251);

SetConsoleOutputCP(1251);

TREE A; // Создание объекта класса с именем A

int el; // Переменная для ввода значения информационного поля вершины

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

cin >> el; // Ввод первого значения информационного поля вершины

while (el != 0) { // Цикл ввода значений информационных полей вершин, пока не будет введен 0

A.TreeSearch(el); // Вызов функции для поиска или добавления узла в дерево

cin >> el; // Ввод следующего значения информационного поля вершины

}

A.VyvodTree(A.GetTreeRight()); // Вызов функции для вывода дерева, начиная с правого поддерева корня дерева A

cout << "\n";

}

void TREE::TreeSearch(int el) {

node\* p1, \* p2; // Указатели на узлы

int d;

p2 = Tree; // Указатель p2 на корневой узел дерева

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

d = 1; // Инициализация переменной d значением 1

while (p1 != NULL && d != 0) { // Цикл поиска места для вставки нового узла

p2 = p1; // Перемещение указателя p2 на текущий узел

if (el < (\*p1).Key) {

p1 = (\*p1).Left; // Перемещение указателя p1 на левое поддерево текущего узла

d = -1; // Значение d устанавливается на -1

}

else if (el > (\*p1).Key) { // Если значение el больше ключа текущего узла

p1 = (\*p1).Right; // Перемещение указателя p1 на правое поддерево текущего узла

d = 1; // Значение d устанавливается на 1

}

else

d = 0; // Значение d устанавливается на 0

}

if (d == 0) {

(\*p1).Count = (\*p1).Count + 1; // Увеличение счетчика узла, если найден узел с таким же ключом

}

else {

p1 = new(node); // ВЫделение памяти для нового узла

(\*p1).Key = el; // Запись значения el в ключ нового узла

(\*p1).Left = NULL; // Инициализация указателя на левое поддерево нового узла в NULL

(\*p1).Right = NULL; // Инициализация указателя на правое поддерево нового узла в NULL

(\*p1).Count = 1; // Инициализация счетчика нового узла как 1

if (d < 0) {

(\*p2).Left = p1; // Присоединение нового узла к левому поддереву родительского узла

(\*p2).Left = p1;

}

else {

(\*p2).Right = p1; // Присоединение нового узла к правому поддереву родительского узла

}

}

}

void TREE::VyvodTree(node\* t) {

no\* stk, \* stk1; // Указатели на стеки

node\* u; // Указатель на узел

int i, n; // Переменные для циклов и счетчиков

stk = stk1 = NULL; // Инициализация стеков как пустых

n = 0; // Инициализация счетчика уровня узлов как 0

while (t != NULL) {

PushStack(&stk1, &t, &n); // Добавление указателя на узел в стек stk1

if ((\*t).Right != NULL) { // Если есть правое поддерево

if ((\*t).Left != NULL) // Если есть левое поддерево

PushStack(&stk, &((\*t).Left), &n); // Добавление указателя на левое поддерево в стек stk

t = (\*t).Right; // Переход к правому поддереву

}

else {

if ((\*t).Left != NULL) {

if (stk1 != NULL) {

PopStack(&stk1, &u, &n); // Извлечение узла и счетчика уровня узлов из стека stk1

for (i = 0; i <= n; i++) cout << " "; // Вывод пробелов для отступа

cout << (\*u).Key << endl; // Вывод ключа узла

}

t = (\*t).Left; // Переход к левому поддереву

}

else if (stk == NULL) // Если и стек stk пустой

t = NULL; // Завершение цикла

else {

while ((\*stk).elem != (\*((\*stk1).elem)).Left) { // Пока указатель на левое поддерево узла в стеке stk не совпадает с текущим узлом

PopStack(&stk1, &u, &n); // Извлечение узла и счетчика уровня узлов из стека stk1

for (i = 0; i <= n; i++) cout << " ";

cout << (\*u).Key << endl; // Вывод ключа узла

}

PopStack(&stk1, &u, &n); // Извлечение узла и счетчика уровня узлов из стека stk1

for (i = 0; i <= n; i++) cout << " ";

cout << (\*u).Key << endl; // Вывод ключа узла

PopStack(&stk, &t, &n); // Извлечение узла и счетчика уровня узлов из стека stk

}

}

n = n + 1; // Увеличение счетчика уровня узлов

}

VyvodStack(&stk1); // Вызов функции для вывода узлов

}

void TREE::PushStack(no\*\* stk, node\*\* el, int\* n) {

no\* q; // Указатель на новый элемент стека

q = new(no); // Выделение памяти для нового элемента стека

(\*q).elem = \*el; // Запись указателя на узел в элемент стека

(\*q).ch = \*n; // Запись счетчика уровня узлов в элемент стека

(\*q).sled = \*stk; // Присоединение элемента стека к стеку

\*stk = q; // Обновление указателя на стек

}

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

no\* q; // Указатель на элемент стека для удаления

if (\*stk != NULL) {

\*t = (\*\*stk).elem; // Извлечение указателя на узел из элемента стека

\*n = (\*\*stk).ch; // Извлечение счетчика уровня узлов из элемента стека

q = \*stk;

\*stk = (\*\*stk).sled; // Обновление указателя на стек

delete q; // Удаление элемента стека

}

}

void TREE::VyvodStack(no\*\* stk) {

node\* k; // Указатель на узел

int i, n; // Переменные для циклов и счетчиков

while (\*stk != NULL) {

k = (\*\*stk).elem; // Извлечение указателя на узел из элемента стека

n = (\*\*stk).ch; // Извлечение счетчика уровня узлов из элемента стека

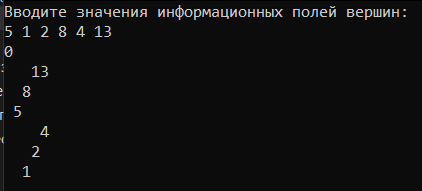
for (i = 0; i <= n; i++) cout << " ";

cout << (\*k).Key << endl; // Вывод ключа узла

\*stk = (\*\*stk).sled; // Обновление указателя на стек

}

}



# Тестирование кода хеширования с помощью леса

#include <time.h>

#include <iostream>

using namespace std;

#define N 10 //Количество элементов массива.

struct node {

int Key; // Ключ узла

int Count; // Счетчик

node\* Left; // Указатель на левое поддерево

node\* Right; // Указатель на правое поддерево

};

class Spisok {

private:

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

void Search(int, node\*\*); // Поиск места для вставки узла в дерево

void PrintTree(node\*, int); // Вывод дерева на экран

void U\_d(node\*\*, node\*\*); // Удаление узла из дерева

public:

Spisok(); // Конструктор класса

void BuildTree(); // Построение дерева

void Sodergimoe(); // Вывод содержимого дерева

node\*\* GetTree(unsigned i) { // Получение указателя на дерево по индексу

return &(UkStr[i]);

}

void Udaldr(node\*\* d, int k); // Удаление узла с заданным ключом из дерева

};

Spisok::Spisok() {

for (int i = 0; i < N; i++) UkStr[i] = NULL; // Инициализация массива указателей на деревья в NULL

}

void Spisok::BuildTree() {

int klutch;

unsigned hash; // Хэш-значение ключа

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

cin >> klutch; // Чтение значения ключа

while (klutch != 0) { // Цикл ввода ключей до тех пор, пока не будет введен 0

hash = klutch % 10; // Вычисление хэш-значения ключа

Search(klutch, &UkStr[hash]); // Поиск места для вставки узла с ключом в соответствующее дерево

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

cin >> klutch; // Чтение следующего значения ключа

}

}

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

if (\*p == NULL) { // Если указатель на текущий узел == NULL, то создаем новый узел

\*p = new (node);

(\*\*p).Key = X; // Присваиваем ключу нового узла значение X

(\*\*p).Count = 1; // Устанавливаем счетчик нового узла в 1

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

}

else {

if (X < (\*\*p).Key) // Если значение X меньше ключа текущего узла, вызываем Search для левого поддерева

Search(X, &((\*\*p).Left));

else if (X > (\*\*p).Key) // Если значение X больше ключа текущего узла, вызываем Search для правого поддерева

Search(X, &((\*\*p).Right));

else

(\*\*p).Count += 1; // Если значение X равно ключу текущего узла, увеличиваем счетчик на 1

}

}

void Spisok::Sodergimoe() {

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

cout << " " << i << "... ";

if (UkStr[i] == NULL)

cout << "Дерево пусто...\n"; // Если указатель на корень дерева равен NULL, выводим сообщение о том, что дерево пусто

else {

cout << endl;

PrintTree(UkStr[i], 0); // Выводим содержимое дерева с корнем

}

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

}

}

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

if (w != NULL) { // Если указатель на текущий узел != NULL

PrintTree((\*w).Right, l + 1); // Рекурсивно вызываем PrintTree для правого поддерева с уровнем l+1

cout << " ";

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

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

PrintTree((\*w).Left, l + 1); // Рекурсивно вызываем PrintTree для левого поддерева с уровнем l+1

}

}

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

node\*\* q;

if (\*d == NULL)

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

else {

if (k < (\*\*d).Key)

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

else if (k > (\*\*d).Key)

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

else {

q = d;

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

\*d = (\*\*q).Left; // Если правое поддерево отсутствует, заменяем удаляемый узел левым поддеревом

else if ((\*\*q).Left == NULL)

\*d = (\*\*q).Right; // Если левое поддерево отсутствует, заменяем удаляемый узел правым поддеревом

else

U\_d(&((\*\*q).Left), &(\*q)); // Иначе вызываем вспомогательную функцию для удаления узла с двумя дочерними узлами

}

}

}

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

if ((\*\*r).Right == NULL) { // Если правое поддерево узла r отсутствует

(\*\*q).Key = (\*\*r).Key; // Копируем ключ из узла r в узел q

(\*\*q).Count = (\*\*r).Count; // Копируем счетчик из узла r в узел q

q = r;

\*r = (\*\*r).Left; // Заменяем узел r его левым поддеревом

delete (\*q); // Удаляем узел q

}

else

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

}

int main() {

setlocale(LC\_ALL, "Rus");

Spisok A;

int klutch;

unsigned hash;

A.BuildTree(); // Построение дерева

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

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

A.Sodergimoe(); // Вывод содержимого дерева

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

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

cin >> klutch;

hash = klutch % 10;

A.Udaldr(A.GetTree(hash), klutch); // Удаление узла с заданным ключом из дерева

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

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

A.Sodergimoe(); // Вывод содержимого дерева после удаления узла

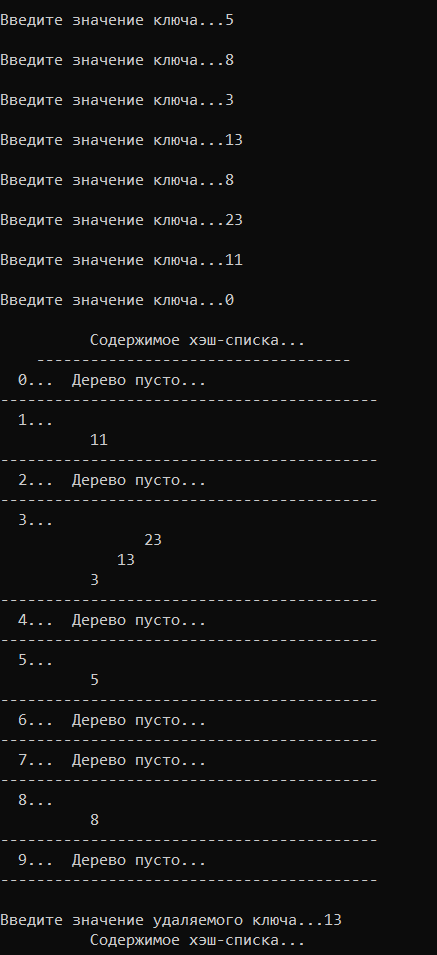
}

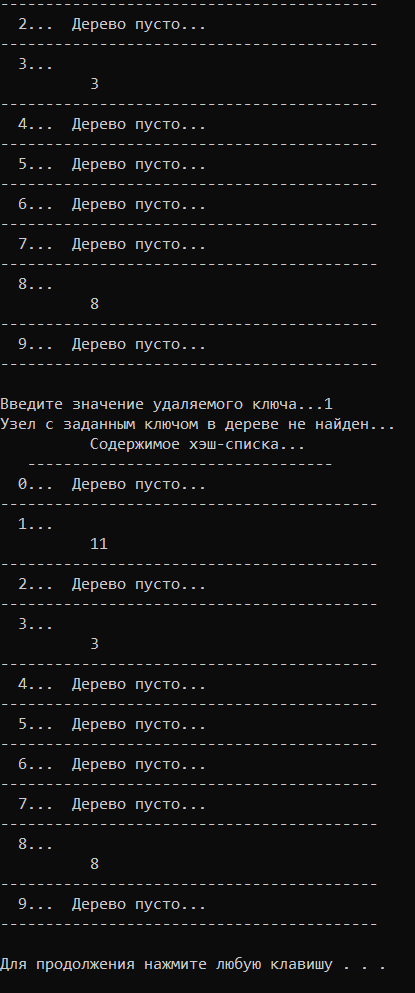
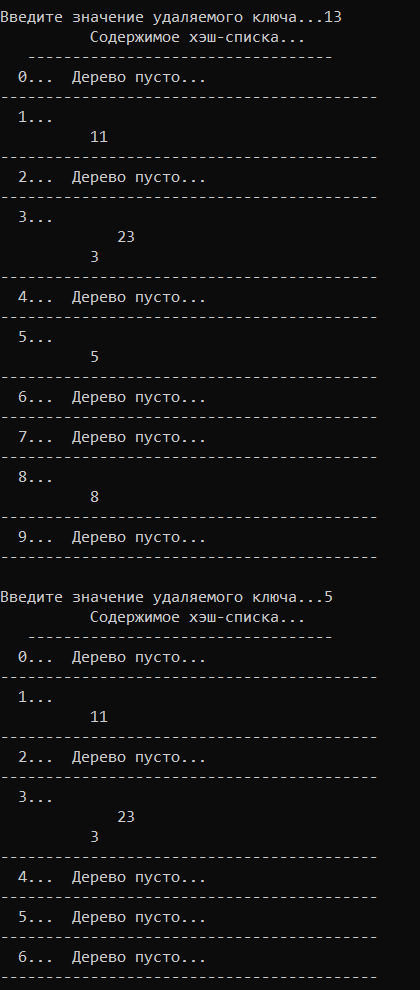
cout << "\n";

system("PAUSE");

return 0;

}





# Тестирование кода древовидно-кольцевой структура

#include <conio.h>

#include <iostream>

using namespace std;

struct node

{

int key; // Ключ узла

int count; // Счетчик узла

node\* Left; // Указатель на левого потомка

node\* Right; // Указатель на правого потомка

};

class Tree

{

private:

node\* root;

void DisposeTree(node\*); // Функция для освобождения памяти, используемой деревом

void printTree(node\*, int); // Функция для вывода содержимого дерева

void Delete(node\*\*, int); // Функция для удаления узла с заданным ключом из дерева

void del(node\*\*, node\*); // Функция для удаления узла с двумя дочерними узлами

public:

Tree() { root = NULL; };

~Tree();

void creat\_Tree(); // Ф-ция для создания дерева

void look\_Tree(); // Ф-ция для вывода содержимого дерева

void add\_Tree(); // Ф-ция для добавления узла в дерево

void delete\_Tree(); // Ф-ция для удаления узла из дерева

void search(int, node\*\*); // Ф-ция для поиска узла с заданным ключом в дереве

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

};

struct zveno

{

int element; // Элемент кольца

Tree ukTree; // Объект класса Tree

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

};

class ring

{

private:

zveno\* ukring;

public:

ring() { ukring = NULL; };

~ring();

void create(); // Ф-ция для построения кольца

void look(); // Ф-ция для вывода содержимого кольца

void add\_after(int, zveno\*); // Ф-ция для добавления элемента после указанного элемента в кольце

void add\_befor(int, zveno\*); // Ф-ция для добавления элемента перед указанным элементом в кольце

void Delete(zveno\*); // Ф-ция для удаления указанного элемента из кольца

void delete\_next(zveno\*); // Ф-ция для удаления элемента, следующего за указанным элементом в кольце

int poisk(int, zveno\*\*); // Ф-ция для поиска элемента с заданным значением в кольце

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

};

void ring::create() // Построение кольца.

{

zveno\* ukzv;

int elem;

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

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

cout << "-->";

cin >> elem;

if (elem != 0)

{

ukzv = ukring = new (zveno);

(\*ukzv).element = elem;

(\*ukzv).ukTree.creat\_Tree(); // Создание дерева, связанного с элементом кольца

cout << "\n-->";

cin >> elem;

while (elem != 0)

{

(\*ukzv).sled = new (zveno);

ukzv = (\*ukzv).sled;

(\*ukzv).element = elem;

(\*ukzv).ukTree.creat\_Tree(); // Создание дерева, связанного с элементом кольца

cout << "\n-->";

cin >> elem;

}

ukzv->sled = ukring; // Замыкание кольца, последний элемент указывает на первый элемент

}

}

ring::~ring()

{

zveno\* ukzv;

ukzv = ukring;

while (ukring != NULL)

if (ukzv->sled == ukring) // Если есть только один элемент в кольце

{

ukring = NULL;

ukzv->ukTree.~Tree(); // Уничтожение дерева, связанного с элементом кольца

delete ukzv;

}

else

{

while (ukzv->sled->sled != ukring) ukzv = (\*ukzv).sled; // Поиск предыдущего элемента перед ukring

(\*ukzv).sled->ukTree.~Tree(); // Уничтожение дерева, связанного с элементом кольца

delete (\*ukzv).sled;

ukzv->sled = ukring;

ukzv = ukring;

}

}

void ring::look() // Вывод кольца.

{

zveno\* ukzv;

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

ukzv = ukring;

do {

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

ukzv->ukTree.look\_Tree(); // Вывод содержимого связанного дерева

ukzv = ukzv->sled;

\_getch();

} while (ukzv != ukring);

cout << endl;

}

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

{

zveno\* ukzv;

Tree temp;

ukzv = new (zveno);

temp = ukzv->ukTree; // Сохранение дерева, связанного с новым элементом

ukzv->element = zv->element;

ukzv->ukTree = zv->ukTree;

ukzv->sled = zv->sled;

zv->element = elem; // Замена значения текущего элемента на новый элемент

zv->ukTree = temp; // Присвоение сохраненного дерева текущему элементу

zv->ukTree.creat\_Tree(); // Создание нового дерева, связанного с текущим элементом

zv->sled = ukzv; // Установка указателя на новый элемент

}

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

{

zveno\* ukzv;

ukzv = new (zveno);

ukzv->element = elem; // Создание нового элемента с заданным значением

ukzv->ukTree.creat\_Tree(); // Создание дерева, связанного с новым элементом

ukzv->sled = zv->sled; // Установка указателя на следующий элемент после текущего

zv->sled = ukzv; // Установка указателя текущего элемента на новый элемент

}

void ring::Delete(zveno\* zv)

{

zveno\* ukzv1, \* ukzv2;

zveno\* time;

if (zv->sled != ukring) // Если следующий элемент после zv не является первым элементом кольца

{

time = zv->sled; // Сохраняем указатель на следующий элемент

zv->ukTree.~Tree(); // Уничтожаем связанное с zv дерево

(\*zv) = \*((\*zv).sled); // Копируем значения следующего элемента в текущий элемент

delete time; // Удаляем временно сохраненный следующий элемент

}

else if (zv->sled == zv) // Если в кольце только один элемент и zv указывает на него

{

zv->ukTree.~Tree(); // Уничтожаем связанное с zv дерево

delete ukring; // Удаляем единственный элемент кольца

ukring = NULL; // Устанавливаем указатель на кольцо в NULL

cout << "Список пуст...\n";

}

else

{

ukzv2 = ukring;

ukzv1 = ukring->sled;

while (ukzv1 != zv) // Поиск элемента zv в кольце

{

ukzv2 = ukzv1;

ukzv1 = ukzv1->sled;

}

time = ukzv2->sled; // Сохраняем указатель на удаляемый элемент

ukzv2->sled->ukTree.~Tree(); // Уничтожаем связанное с удаляемым элементом дерево

ukzv2->sled = ukzv2->sled->sled; // Пропускаем удаляемый элемент в кольце

delete time; // Удаляем временно сохраненный элемент

}

}

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

{

zveno\* time;

if (zv->sled != ukring) // Если следующий элемент после zv не является первым элементом кольца

{

time = zv->sled; // Сохраняем указатель на следующий элемент

zv->sled = zv->sled->sled; // Пропускаем следующий элемент в кольце

time->ukTree.~Tree(); // Уничтожаем связанное с временно сохраненным элементом дерево

delete time; // Удаляем временно сохраненный элемент

}

else if (zv->sled == zv) // Если в кольце только один элемент и zv указывает на него

{

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

}

else

{

time = ukring->sled; // Сохраняем указатель на следующий элемент после первого элемента

\*((\*zv).sled) = (\*(\*(\*zv).sled).sled); // Копируем значения следующего элемента в элемент, следующий за zv

time->ukTree.~Tree(); // Уничтожаем связанное с временно сохраненным элементом дерево

delete time; // Удаляем временно сохраненный элемент

}

}

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

{

zveno\* ukzv;

int vozvr = 0;

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

{

cout << "Кольцо не существует...\n";

}

else

{

ukzv = ukring;

while (ukzv->sled != ukring && (\*Res) == NULL) // Поиск элемента в кольце

{

if (ukzv->element == elem) // Если найден элемент совпадающий с искомым элементом

{

vozvr = 1; // Устанавливаем флаг успешного поиска

\*Res = ukzv; // Присваиваем указателю Res адрес найденного элемента

}

ukzv = ukzv->sled; // Переходим к следующему элементу в кольце

}

if ((\*Res) == NULL) // Если после прохода по всем элементам кольца Res все еще NULL

{

if (ukzv->element == elem) // Проверяем последний элемент в кольце

{

vozvr = 1; // Устанавливаем флаг успешного поиска

\*Res = ukzv; // Присваиваем указателю Res адрес последнего элемента

}

}

}

return vozvr; // Возвращаем результат поиска

}

Tree::~Tree()

{

DisposeTree(root); // Вызываем функцию для освобождения памяти, начиная с корневого узла

root = NULL; // Устанавливаем указатель на корневой узел в NULL

}

void Tree::DisposeTree(node\* p)

{

if (p != NULL) // Если текущий узел не является пустым

{

DisposeTree(p->Left); // Освобождаем память для левого поддерева

DisposeTree(p->Right); // Освобождаем память для правого поддерева

delete p; // Удаляем текущий узел

}

}

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

{

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

{

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

(\*\*p).key = x; // Присваиваем ключу нового узла значение x

(\*\*p).count = 1; // Инициализируем счетчик нового узла значением 1

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

}

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

{

search(x, &((\*\*p).Left)); // Вызываем функцию для левого поддерева

}

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

{

search(x, &((\*\*p).Right)); // Вызываем функцию для правого поддерева

}

else // Если x равно ключу текущего узла

{

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

}

}

void Tree::creat\_Tree()

{

int elem;

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

cin >> elem;

while (elem != 0) // Пока не будет введен 0

{

search(elem, &root); // Вызываем функцию для вставки нового узла с ключом в дерево

cin >> elem;

}

}

void Tree::look\_Tree()

{

if (root == NULL) // Если корневой узел == NULL, то дерево пусто

{

cout << "Дерево пусто ...\n";

}

else

{

printTree(root, 0); // Выводим содержимое дерева, начиная с корневого узла

}

}

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

{

if (w != NULL) // Если текущий узел != NULL

{

printTree(w->Left, L + 1); // Вызываем функцию для левого поддерева с увеличением уровня на 1

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

cout << w->key << endl; // Выводим ключ текущего узла

printTree(w->Right, L + 1); // Вызываем функцию для правого поддерева с увеличением уровня на 1

}

}

void Tree::add\_Tree()

{

int k;

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

cin >> k;

cout << " ";

while (k != 0) // Пока не будет введен 0

{

search(k, &(root)); // Вызываем функцию для вставки нового узла с ключом в дерево

cin >> k;

cout << " ";

}

}

void Tree::delete\_Tree()

{

int elem;

if (root == NULL) // Если корневой узел == NULL, то дерево пусто

{

cout << "Дерево пусто ...\n";

}

else

{

cout << "Введите ключ удаляемого узла : ";

cin >> elem;

cout << endl;

Delete(&root, elem); // Вызываем функцию для удаления узла с ключом из дерева

}

}

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

{

node\* q;

node\* s;

if (\*d == NULL) // Если текущий узел == NULL, ключ не найден в дереве

{

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

}

else

{

if (k < (\*\*d).key) // Если ключ меньше ключа текущего узла

{

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

}

else if (k > (\*\*d).key) // Если ключ больше ключа текущего узла

{

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

}

else

{

q = \*d;

s = \*d;

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

{

\*d = (\*q).Left; // Левый потомок заменяет текущий узел

delete s; // Освобождаем память для текущего узла

}

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

{

\*d = (\*q).Right; // Правый потомок заменяет текущий узел

delete s; // Освобождаем память для текущего узла

}

else // Если у текущего узла есть и левый, и правый потомки

{

del(&((\*q).Left), &(\*q)); // Вызываем функцию для нахождения наименьшего узла в правом поддереве

}

}

}

}

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

{

node\* s;

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

{

(\*q).key = (\*\*r).key; // Копируем ключ текущего узла в q

(\*q).count = (\*\*r).count; // Копируем счёт текущего узла в q

q = s = \*r; // Запоминаем указатели q и s на текущий узел

\*r = (\*\*r).Left; // Левый потомок заменяет текущий узел

delete s; // Освобождаем память для текущего узла

}

else

{

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

}

}

void main()

{

setlocale(LC\_ALL, "Rus");

int menu1 = 1, choice, elem1, elem2, menu2; // Объявление переменных menu1, choice, elem1, elem2 и menu2

ring A; // Создание объекта типа ring с именем A

zveno\* Res; // Объявление указателя Res типа zveno

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

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

while (menu1)

{

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;

switch (choice)

{

case 1:

// Проверяем, существует ли уже кольцо

if (\*(A.getring()) == NULL)

A.create(); // Если кольца не существует, создаем его

else

cout << "Кольцо уже существует...\n"; // Если кольцо уже существует, выводим сообщение

break;

case 2:

// Проверяем, пустое ли кольцо

if (\*(A.getring()) == NULL)

cout << "Кольцо пусто...\n"; // Если кольцо пустое, выводим сообщение

else

A.look(); // Если кольцо не пустое, выводим его содержимое

break;

case 3:

// Проверяем, пустое ли кольцо

if (\*(A.getring()) == NULL)

cout << "Кольцо пусто...\n"; // Если кольцо пустое, выводим сообщение

else

{

Res = NULL;

cout << "Введите элемент, после которого ";

cout << "хотите добавить звено: ";

cin >> elem1;

cout << endl;

// Поиск элемента в кольце

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

{

cout << "Введите элемент, который ";

cout << "хотите добавить: ";

cin >> elem2;

cout << endl;

A.add\_after(elem2, Res); // Добавляем звено после найденного элемента

}

else

cout << "Элемент " << elem1 << " не найден.\n"; // Если элемент не найден, выводим сообщение

}

break;

case 4:

if (\*(A.getring()) == NULL)

cout << "Кольцо пусто...\n";

else

{

Res = NULL;

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); // Добавляем звено перед найденным элементом

}

else

cout << "Элемент " << elem1 << " не найден.\n"; // Если элемент не найден, выводим сообщение

}

break;

case 5:

if (\*(A.getring()) == NULL)

cout << "Кольцо пусто...\n";

else

{

Res = NULL;

cout << "Введите элемент, который";

cout << " хотите удалить: ";

cin >> elem1;

cout << endl;

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

A.Delete(Res); // Удаляем найденный элемент

else

cout << "Элемент отсутствует...\n"; // Если элемент не найден, выводим сообщение

}

break;

case 6:

if (\*(A.getring()) == NULL)

cout << "Кольцо пусто...\n";

else

{

Res = NULL;

cout << "Введите элемент, после которого";

cout << " хотите удалить: ";

cin >> elem1;

cout << endl;

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

A.delete\_next(Res); // Удаляем звено после найденного элемента

else

cout << "Элемент отсутствует...\n"; // Если элемент не найден, выводим сообщение

}

break;

case 7:

if (\*(A.getring()) == NULL)

cout << "Кольцо пусто...\n";

else

{

Res = NULL;

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:

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

(\*Res).ukTree.creat\_Tree(); // Если дерево не существует, создаем его

else

cout << "Дерево существует...\n"; // Если дерево уже существует, выводим сообщение

break;

case 2:

(\*Res).ukTree.look\_Tree(); // Выводим содержимое дерева

break;

case 3:

(\*Res).ukTree.add\_Tree(); // Добавляем элемент в дерево

break;

case 4:

(\*Res).ukTree.delete\_Tree(); // Удаляем элемент из дерева

break;

case 5:

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

cout << "Дерево не существует...\n"; // Если дерево не существует, выводим сообщение

else

(\*Res).ukTree.~Tree(); // Удаляем дерево

break;

case 6:

menu2 = 0; // Выход из вложенного меню

break;

}

}

}

else

cout << "Элемент " << elem1 << " не найден.\n"; // Если элемент не найден, выводим сообщение

}

break;

case 8:

if (\*(A.getring()) == NULL) cout << "Кольцо пусто...\n";

else A.~ring();

break;

case 9:

A.~ring();

menu1 = 0;

break;

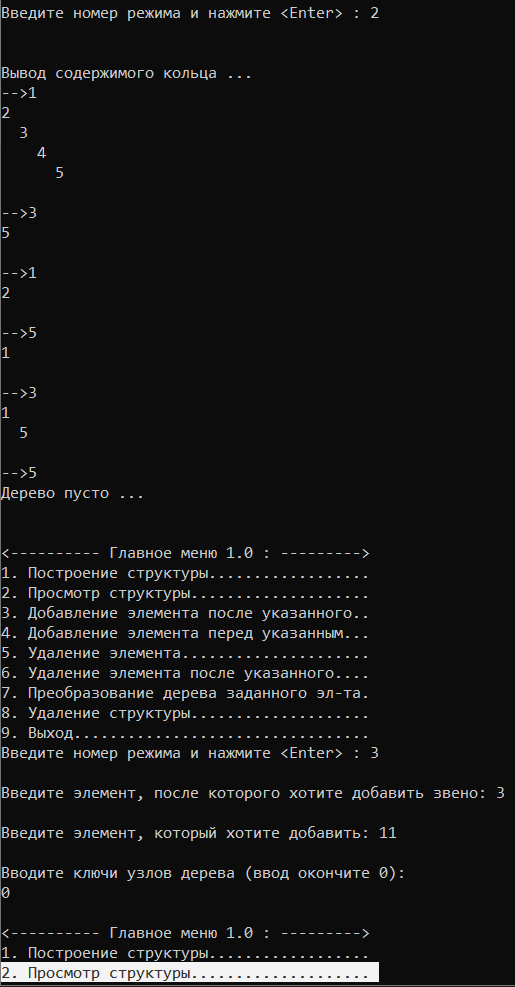
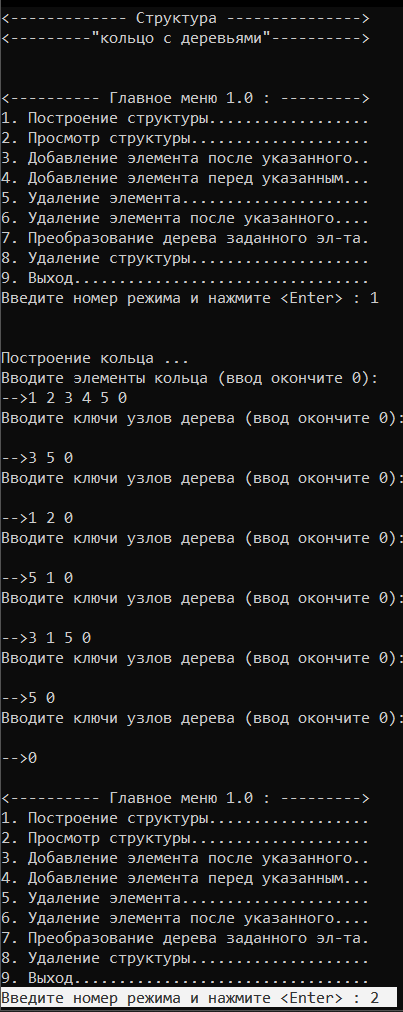
}

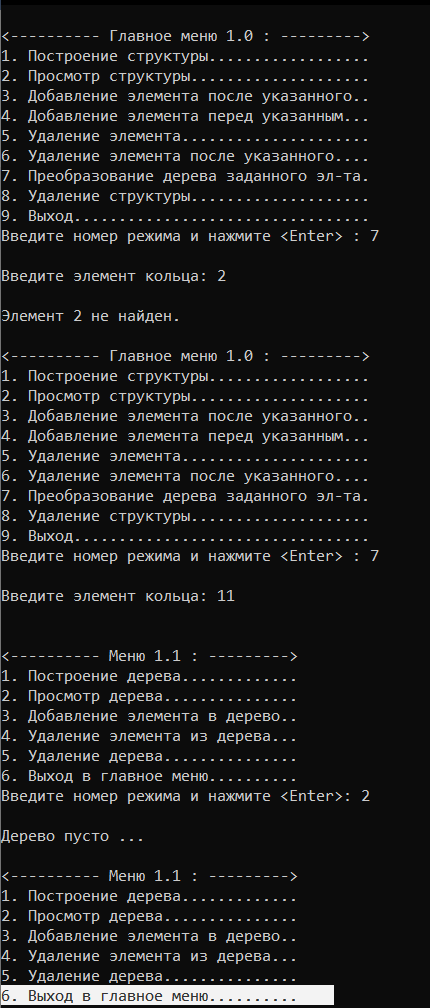
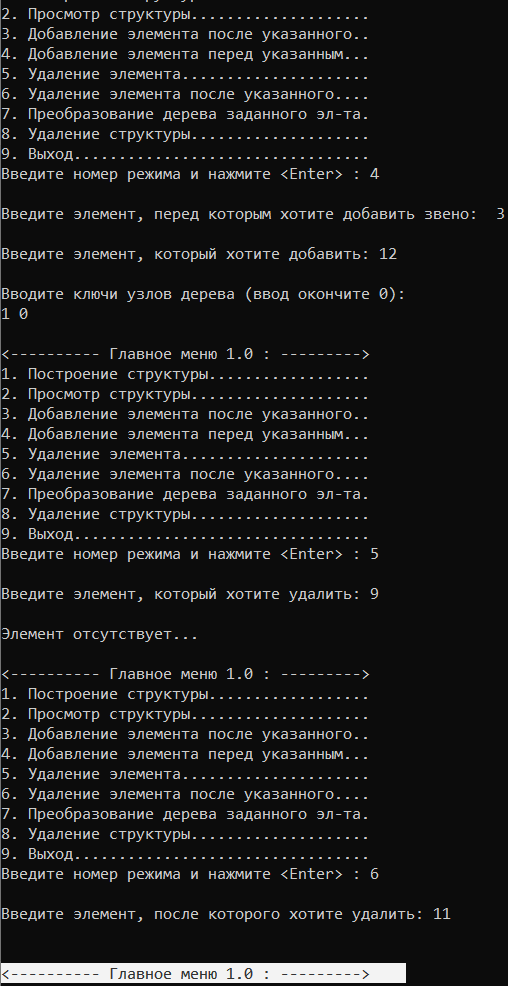
}

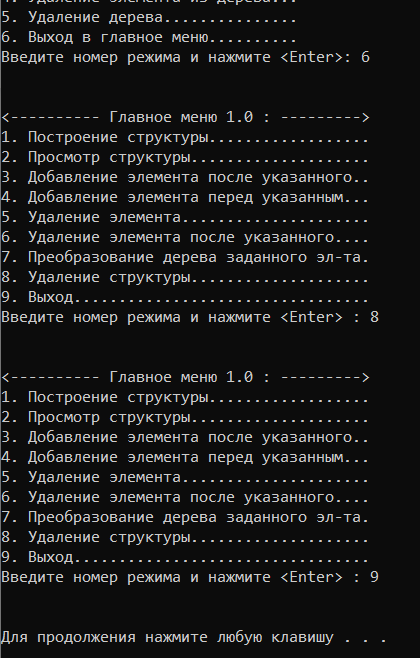
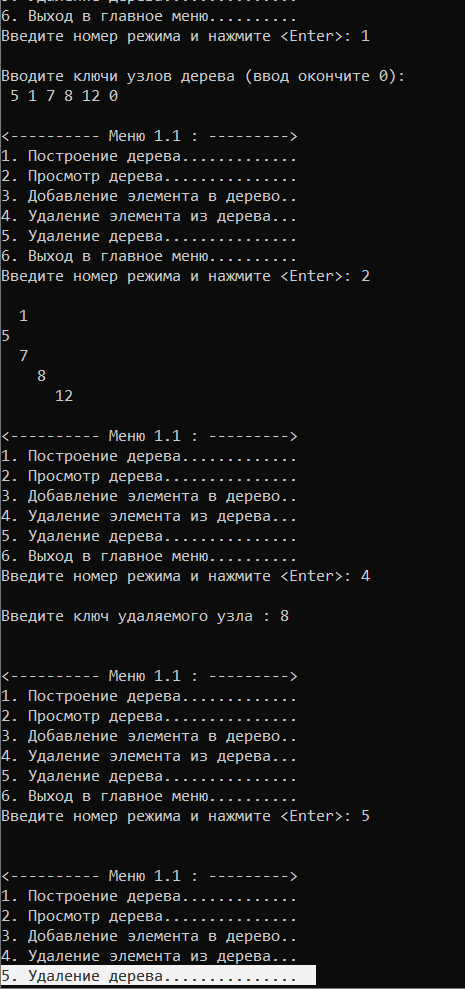
cout << "\n";

system("PAUSE");

}







# Тестирование кода алгоритма Хаффмена

#include <stdio.h>

#include <conio.h>

#include <iostream>

using namespace std;

struct zveno

{

char Element; // Символ

float Kol; // Количество повторений

zveno\* Sled; // Указатель на следующий элемент

zveno\* Left; // Указатель на левый дочерний элемент в дереве

zveno\* Right; // Указатель на правый дочерний элемент в дереве

zveno\* Father; // Указатель на родительский элемент в дереве

};

class Tree

{

private:

zveno\* UkStr; //Указатель на список.

int Poisk1(zveno\*\*, float, zveno\*\*);

public:

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

int Poisk(char, zveno\*\*);// Поиск элемента в дереве по символу

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

void Dobavlenie(char, float, zveno\*\*); // Добавление элемента в дерево

void Redaktor(int); // Редактирование значения элемента в дереве

void Ukazateli(zveno\*\*, zveno\*\*); // Обновление указателей в дереве

void Vyvod(); // Вывод дерева

void PrintTree(zveno\*, int);

void WstawkaSort(zveno\*);

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

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

};

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

{

zveno\* q;

int vozvr = 0;

\*Res = NULL;

q = (\*UkStr).Sled; // Начинаем с элемента после заглавного звена

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

{

if (q->Element == ENT)

{

vozvr = 1; // Элемент найден

\*Res = q; // Устанавливаем указатель на найденный элемент

return vozvr;

}

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

}

return vozvr; // Возвращаем

}

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

{

zveno\* q = (\*\*st).Sled; // Начинаем с элемента после заглавного звена

zveno\* q1 = (\*st);

int vozvr = 0;

\*Res = NULL;

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

{

if (q->Kol < kol)

{

vozvr = 1; // Место для добавления элемента найдено

\*Res = q; // Устанавливаем указатель на место для добавления элемента

}

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

q1 = q1->Sled;

}

if (\*Res == NULL)

\*Res = q1; // Если место для добавления элемента не найдено, устанавливаем указатель на последний элемент

return vozvr;

}

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

{

int K = 0;

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

{

if (F[i] == S)

{

K++; // Увеличиваем счетчик повторений, если символ совпадает с искомым символом

}

}

return K; // Возвращаем количество повторений

}

void Tree::Redaktor(int L)

{

zveno\* q = (\*UkStr).Sled;

while (q != NULL)

{

q->Kol = q->Kol / L; // Заменяем количество повторений на частоту повторений, деленную на L

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

}

}

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

{

zveno\* q, \* Res = NULL, \* kladovaq;

q = new (zveno); // Выделяем память под q

q->Element = bukva; // Присваивает значение переменной нового элемента

q->Kol = kol; // Присваивает значение переменной нового элемента

q->Left = q->Right = NULL; // Устанавливает указатели Left и Right нового элемента в NULL

q->Sled = q->Father = NULL; // Устанавливает указатели Sled и Father нового элемента в NULL

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

{

(\*\*Sp).Sled = q; // Если список пуст, добавляем элемент в качестве первого элемента списка

}

else

{

if (Poisk1(&(\*Sp), kol, &Res))

{

kladovaq = new (zveno);

(\*kladovaq) = (\*Res);

(\*Res) = (\*q); // Вставляем элемент перед найденным местом в упорядоченном списке

Res->Sled = kladovaq;

}

else

{

Res->Sled = q; // Если место для добавления элемента не найдено, добавляем его в конец списка

}

}

}

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

{

\*zv\_p = UkStr->Sled; // Устанавливаем указатель на элемент, следующий за UkStr

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

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

{

\*zv = \*zv\_p; // Сдвигаем указатель на следующий элемент

\*zv\_p = (\*zv\_p)->Sled; // Сдвигаем указатель на следующий элемент

}

}

void Tree::Vyvod()

{

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

while (q != NULL)

{

cout << q->Element << " (" << q->Kol << ") --> "; // Выводим значение элемента

q = q->Sled; // Сдвигаем указатель на следующий элемент

}

cout << endl;

}

void Tree::WstawkaSort(zveno\* zv)

{

zveno\* w1, \* w2;

w2 = UkStr; // Устанавливаем на UkStr

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

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

{

w2 = w1;

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

}

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

{

w2->Sled = zv; // Вставляем элемент zv после w2

zv->Sled = w1; // Устанавливаем следующий элемент для zv на w1

}

}

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

{

if (w != NULL) // Проверяем, что текущий узел не является пустым

{

PrintTree(w->Right, l + 1); // Вызываем метод для правого поддерева, увеличивая уровень на 1

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

cout << w->Element << " (" << w->Kol << ")\n"; // Выводим значение поля Element и Kol текущего узла

PrintTree(w->Left, l + 1); // Вызываем метод для левого поддерева, увеличивая уровень на 1

}

}

void main()

{

setlocale(LC\_ALL, "Rus");

Tree A;

char T[255]; //Исходная стpока.

int i, j;

zveno\* Res = NULL;

zveno\* Q[256];

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

gets\_s(T);

//Фоpмиpование списка, содеpжащего символы текста.

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

{

if (!A.Poisk(T[i], &Res))

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

}

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

A.Redaktor(strlen(T));

cout << "Полученный список:\n";

A.Vyvod();

//Заполнение массива Q указателей на элементы списка.

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

i = 0;

while (UkZv != NULL)

{

Q[i] = UkZv; i++; UkZv = UkZv->Sled;

}

//Постpоение деpева Хаффмена.

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

{

A.Ukazateli(&UkZv, &UkZv\_p);

//Слияние последнего и пpедпоследнего звена.

Sli = new (zveno);

Sli->Element = '\*';

Sli->Kol = UkZv\_p->Kol + UkZv\_p->Sled->Kol;

Sli->Left = UkZv\_p;

Sli->Right = UkZv\_p->Sled;

Sli->Father = Sli->Sled = NULL;

UkZv\_p->Father = Sli;

UkZv\_p->Sled->Father = Sli;

//Уничтожаем ссылки на последнее и пpедпоследнее звенья.

UkZv->Sled = NULL;

UkZv\_p->Sled = NULL;

//Помещаем звено, заданное указателем Sli в список.

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

else A.WstawkaSort(Sli);

}

cout << "Постpоим деpево...\n";

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

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

//Кодиpование заданного текста.

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

char Cod\_symbol[40];

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

char temp[255];

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

strcpy\_s(Cod\_Haffmen, "");

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

{

//Hачнем поиски нужного указателя.

j = 0;

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

//А тепеpь начинаем "восхождение"...

UkZv = Q[j];

while (UkZv->Father != NULL)

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

{

strcpy\_s(temp, "1");

strcat\_s(temp, Cod\_symbol);

strcpy\_s(Cod\_symbol, temp);

UkZv = UkZv->Father;

}

else

{

strcpy\_s(temp, "0");

strcat\_s(temp, Cod\_symbol);

strcpy\_s(Cod\_symbol, temp);

UkZv = UkZv->Father;

}

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

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

}

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; // Устанавливаем указатель UkZv на элемент списка дерева

i = 0; // Инициализируем переменную i

while (i < strlen(Cod\_Haffmen)) // Пока не достигнут конец строки Cod\_Haffmen

{

while (UkZv->Left != NULL && UkZv->Right != NULL) // Пока текущий узел имеет и левого, и правого потомка

{

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

UkZv = UkZv->Left; // Если текущий символ в Cod\_Haffmen равен '1', переходим к левому потомку

else

UkZv = UkZv->Right; // Если текущий символ в Cod\_Haffmen не равен '1', переходим к правому потомку

i++; // Увеличиваем i

}

char s[2]; // Создаем временный массив s с размером 2

s[0] = UkZv->Element; // Присваиваем первому элементу массива значение Element текущего узла

s[1] = '\0'; // Устанавливаем второй элемент массива в '\0'

strcat\_s(T, s); // Добавляем строку s в конец строки T

UkZv = A.GetTree1()->Sled; // Возвращаем указатель UkZv к началу списка дерева

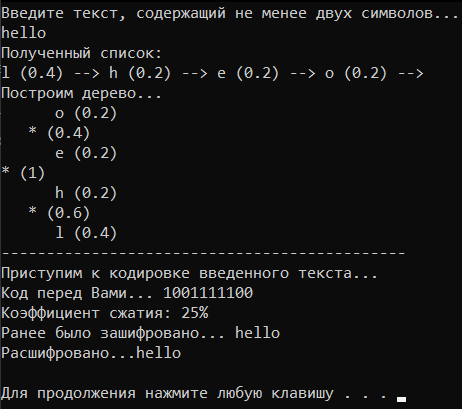
}

cout << "Расшифровано..." << T << endl; // Выводим результат расшифровки

cout << "\n";

system("PAUSE");

}



# Тестирование кода построения дерева-формулы

#include <stdio.h>

#include <conio.h>

#include <iostream>

using namespace std;

struct Uzel {

char Key; // Символ

Uzel\* Left; // Левый потомок узла

Uzel\* Right; // Правый потомок узла

};

struct zveno {

Uzel\* Element; // Указатель на узел дерева

zveno\* Sled; // Указатель на следующее звено стека

};

class Tree{

private:

Uzel \* Root; // Указатель на корень дерева

zveno\* Stack; // Указатель на стек

public:

Tree(); // Конструктор класса Tree

void Udalenie(Uzel\*\*); // Удаление узла

void V\_stack(Uzel\*); // Помещение узла в стек

void PrintTree(Uzel\*, int); // Вывод дерева на экран дисплея

void Print\_Tree\_Left(Uzel\*, int); // Левосторонний обход бинарного дерева

void Print\_Tree\_End(Uzel\*, int); // Концевой обход бинарного дерева

void Print\_Tree\_Back(Uzel\*, int); // Обратный обход бинарного дерева

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

};

void Tree::V\_stack(Uzel\* Elem)

{

zveno\* q = new zveno; // Выделяем память под новое звено стека

q->Element = Elem; // Устанавливаем указатель на узел дерева в звене стека

q->Sled = Stack; // Устанавливаем указатель на предыдущее звено стека в качестве следующего звена

Stack = q; // Обновляем указатель на вершину стека

}

void Tree::Udalenie(Uzel\*\* tmp)

{

zveno\* q;

if (Stack != NULL)

{

(\*tmp) = Stack->Element; // Получаем указатель на узел дерева из вершины стека

q = Stack; // Сохраняем указатель на вершину стека

Stack = Stack->Sled; // Перемещаем указатель на следующее звено стека

delete q; // Удаляем звено стека

}

}

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

{

if (w != NULL)

{

PrintTree(w->Right, l + 1); // Вызываем ф-цию для правого поддерева с увеличением уровня на 1.

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

cout << " ";

cout << w->Key << endl; // Выводим символ текущего узла.

PrintTree(w->Left, l + 1); // Вызываем метод для левого поддерева с увеличением уровня на 1.

}

}

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

// Левосторонний обход бинарного дерева.

{

if (w != NULL)

{

cout << w->Key << " "; // Выводим символ текущего узла.

Print\_Tree\_Left(w->Left, l + 1); // Рекурсивно вызываем ф-цию для левого поддерева.

Print\_Tree\_Left(w->Right, l + 1); // Рекурсивно вызываем ф-цию для правого поддерева.

}

}

void Tree::Print\_Tree\_End(Uzel\* w, int l)

// Концевой обход бинарного дерева.

{

if (w != NULL)

{

Print\_Tree\_End(w->Left, l + 1); // Рекурсивно вызываем ф-цию для левого поддерева.

Print\_Tree\_End(w->Right, l + 1); // Рекурсивно вызываем ф-цию для правого поддерева.

cout << w->Key << " "; // Выводим символ текущего узла.

}

}

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

// Обратный обход бинарного дерева.

{

if (w != NULL)

{

cout << "("; // Выводим открывающую скобку.

Print\_Tree\_Back(w->Left, l + 1); // Рекурсивно вызываем ф-цию для левого поддерева.

cout << w->Key << " "; // Выводим символ текущего узла.

Print\_Tree\_Back(w->Right, l + 1); // Рекурсивно вызываем ф-цию для правого поддерева.

cout << ")"; // Выводим закрывающую скобку.

}

}

Tree::Tree()

{

Stack = NULL; //Вначале опустошим стек.

//Фоpмиpование заглавного звена деpева.

Root = new (Uzel);

Root->Right = NULL;

}

void main()

{

setlocale(LC\_ALL, "Rus");

char Formula\_Post[30];

char k; //Вспомогательная пеpеменная.

Uzel\* Ukazatel = NULL;

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

gets\_s(Formula\_Post);

//Получили "пеpевеpтыш" слова Formula\_Post.

\_strrev(Formula\_Post);

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

Tree A;

Uzel\* Temp = A.GetTree(); //Текущий указатель.

//Фоpмиpование деpева поиска и вывод его на экpан.

for (int i = 0; i < strlen(Formula\_Post); i++)

{

k = Formula\_Post[i];

//Пеpеходим к анализу символа k.

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

{ //Символ - опеpация.

if (Temp->Right == NULL) //Отсутствует пpавая дуга.

{

//Резеpвиpование 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 пpедыдущий узел --> стек.

A.V\_stack(Temp);

}

else //Есть пpавая дуга.

{ //Резеpвиpование 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 пpедыдущий узел --> стек.

A.V\_stack(Temp);

}

}

else //Символ - опеpанд.

if (Temp->Right == NULL) //Отсутствует пpавая дуга.

{

//Резеpвиpование 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 "возвpащается" назад.

A.Udalenie(&Ukazatel);

Temp = Ukazatel;

}

else //Есть пpавая дуга.

{ //Резеpвиpование 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 "возвpащается" назад.

A.Udalenie(&Ukazatel);

Temp = Ukazatel;

}

} //Конец for.

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

A.PrintTree(A.GetTree()->Right, 0);

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

A.Print\_Tree\_Back(A.GetTree()->Right, 0);

cout << endl;

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

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

A.Print\_Tree\_Left(A.GetTree()->Right, 0);

cout << endl;

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

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

A.Print\_Tree\_End(A.GetTree()->Right, 0);

cout << "\n";

system("PAUSE");

}

