Белорусский государственный технологический университет

Факультет информационных технологий

Кафедра программной инженерии

 Реферат

По дисциплине «Основы алгоритмизации и программирования»

На тему «Бинарное дерево. Тестирование»

 Выполнила:

Студентка1 курса 6 группы

Альшевская Алина Михайловна

Преподаватель: доц Белодед Н.И

2023, Минск

1. Обход бинарного дерева

#include <iostream>

using namespace std;

struct node

{

int Key; //Ключ узла

int Count; //Счётчик вхождений ключа

node\* Left; // Указатель на левого сына.

node\* Right; // Указатель на правого сына.

};

class TREE

{

private:

node\* Tree; //Указатель на корень дерева.

void Search(int, node\*\*); //Поиск вершины с ключом int в дереве со вставкой

public:

TREE() { Tree = NULL; }

node\*\* GetTree() { return &Tree; } //Получение вершины дерева.

void BuildTree(); //Построение дерева

void CleanTree(node\*\*); //Очистка дерева

void ObhodEnd(node\*\*); //Концевой обход дерева

void ObhodLeft(node\*\*); //Левосторонний обход дерева

void ObhodBack(node\*\*); //Обратный обход дерева

void Vyvod(node\*\*, int); //Изображение дерева на экране дисплея

int Height(node\*\*); //Определение высоты бинарного дерева

};

void main()

{

setlocale(LC\_ALL, "Rus");

TREE A;

A.BuildTree();

cout << "\nВывод дерева:\n";

A.Vyvod(A.GetTree(), 0);

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

cout << "\nЛевосторонний обход дерева: ";

A.ObhodLeft(A.GetTree());

cout << "\nКонцевой обход дерева: "; A.ObhodEnd(A.GetTree());

cout << "\nОбратный обход дерева: "; A.ObhodBack(A.GetTree());

A.CleanTree(A.GetTree());

cout << "\n";

system("PAUSE");

}

void TREE::BuildTree()

// Построение бинарного дерева (рекурсивный алгоритм).

// Tree - указатель на корень дерева.

{

int el;

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

cin >> el;

while (el != 0)

{

// Поиск и вставка каждого введенного ключа в деревo

Search(el, &Tree); cin >> el;

}

}

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

// Поиск вершины с ключом x в дереве со вставкой

// (рекурсивный алгоритм).

// \*p - указатель на корень дерева.

{

if (\*p == NULL)

{// Если вершины в дереве нет,то включить ее.

\*p = new(node);

(\*\*p).Key = x; (\*\*p).Count = 1;

(\*\*p).Left = NULL; (\*\*p).Right = NULL;

}

else

// Рекурсивный поиск по дереву

if (x < (\*\*p).Key) Search(x, &((\*\*p).Left));

else

if (x > (\*\*p).Key) Search(x, &((\*\*p).Right));

// Если ключ найден, увеличиваем счетчик

else (\*\*p).Count = (\*\*p).Count + 1;

}

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

//Левосторонний обход дерева.

//\*w - указатель на корень дерева.

{

if (\*w != NULL)//проверка на пустое ли дерево

{

cout << (\*\*w).Key << " ";//выводим информации о корне

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

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

}

}

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

//Концевой обход дерева.

//\*w - указатель на корень дерева.

{

if (\*w != NULL)//проверка на пустое ли дерево

{

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

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

cout << (\*\*w).Key << " ";//выводим информации о корне

}

}

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

//Обратный обход дерева.

//\*w - указатель на корень дерева.

{

if (\*w != NULL)//проверка на пустое ли дерево

{

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

cout << (\*\*w).Key << " ";//выводим информации о корне

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

}

}

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

//Очистка дерева.

//\*w - указатель на корень дерева.

{

if (\*w != NULL)

{

CleanTree(&((\*\*w).Left));

CleanTree(&((\*\*w).Right));

delete\* w;

}

}

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

//Изображение дерева \*w на экране дисплея

// (рекурсивный алгоритм).

//\*w - указатель на корень дерева.

{

int i;

if (\*w != NULL)

{

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

for (i = 1; i <= l; i++) cout << " ";// считаем расстояние от корня до вершины

cout << (\*\*w).Key << endl;//выводим информации о корне

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

}

}

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

//Определение высоты бинарного дерева.

//\*w - указатель на корень дерева.

{

int h1, h2;

if (\*w == NULL) return (-1);

else

{

h1 = Height(&((\*\*w).Left));

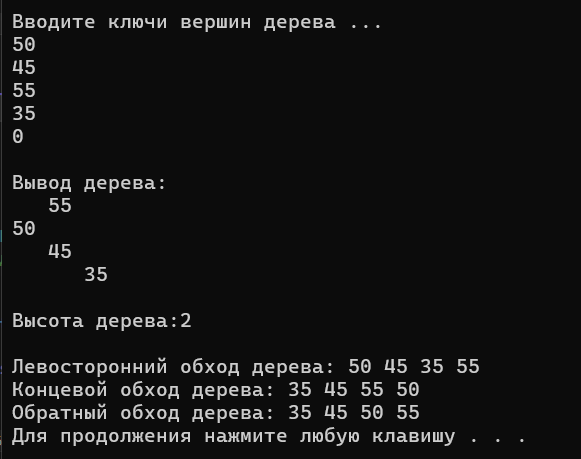
h2 = Height(&((\*\*w).Right));

if (h1 > h2) return (1 + h1);

else return (1 + h2);

}

}



1. Построения и изображения бинарного дерева с использованием нерекурсивных алгоритмов.

#include <iostream>

using namespace std;

struct node

{

int Key; // Ключ узла

int Count; // Счетчик вхождений ключа

node\* Left; // Указатель на левого сына

node\* Right; // Указатель на правого сына

};

struct no // Звено стека

{

node\* elem; // Информационное поле.

int ch; // Уровень вершины.

no\* sled; // Указатель на вершину.

};

class TREE

{

private:

node\* Tree;

void PushStack(no\*\*, node\*\*, int\*);// Помещение звена с элементами \*el и n в

// стек. \*stk - указатель на стек.

void PopStack(no\*\*, node\*\*, int\*); // Извлечение из стека звена

// с элементами \*t и n.

// \*stk - указатель на стек

void VyvodStack(no\*\*); // Вывод содержимого стека на экран дисплея.

// \*stk - указатель на стек.

public:

TREE() { Tree = new(node); (\*Tree).Right = NULL; }

node\* GetTreeRight() { return (\*Tree).Right; }

void TreeSearch(int); // Поиск вершины с информационным полем el в дереве с

// последующим (в случае неудачного поиска!) включением

// в дерево. Tree - указатель на корень дерева.

void VyvodTree(node\*); //Построение дерева, заданного указателем t,

//на экране дисплея (нерекурсивный алгоритм).

};

void main()

{

setlocale(LC\_ALL, "Rus");

TREE A;

int el;

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

cin >> el;

while (el != 0)

{

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

}

A.VyvodTree(A.GetTreeRight());

cout << "\n";

system("PAUSE");

}

void TREE::TreeSearch(int el)

// Поиск вершины с информационным полем el в дереве с

// последующим (в случае неудачного поиска!) включением

// в дерево. Tree - указатель на корень дерева.

{

node\* p1, \* p2;

int d;

p2 = Tree; p1 = (\*p2).Right; d = 1;

while (p1 != NULL && d != 0)

{

p2 = p1;

if (el < (\*p1).Key) { p1 = (\*p1).Left; d = -1; }

else

if (el > (\*p1).Key) { p1 = (\*p1).Right; d = 1; }

else d = 0;

}

if (d == 0) (\*p1).Count = (\*p1).Count + 1;

else

{

p1 = new(node);

(\*p1).Key = el; (\*p1).Left = NULL;

(\*p1).Right = NULL; (\*p1).Count = 1;

if (d < 0) (\*p2).Left = p1;

else (\*p2).Right = p1;

}

}

void TREE::VyvodTree(node\* t)

//Построение дерева, заданного указателем t,

//на экране дисплея (нерекурсивный алгоритм).

{

no\* stk, \* stk1;// Стек левых и правых ссылок.

node\* u;//указатель на структуру

int i, n;

stk = stk1 = NULL; n = 0;//сначала оба стека пусты

while (t != NULL)

{

PushStack(&stk1, &t, &n);

if ((\*t).Right != NULL)// Пока не достигнут лист дерева, в стек stk помещаются

{

if ((\*t).Left != NULL) PushStack(&stk, &((\*t).Left), &n); // левые ссылки тех структур, по которым перемещается указатель t

t = (\*t).Right;

}

else

{

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

{

if (stk1 != NULL) // Когда будет достигнут самый правый лист, проверяем левую ссылку. Если она существует, то из стека правых ссылок удаляем

//последний адрес, и на печать выводим поле ключа этой вершины.

{

PopStack(&stk1, &u, &n);

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

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

}

t = (\*t).Left;

}

else

if (stk == NULL) t = NULL;

else

// Если у вершины дерева указатели на правое и левое поддеревья равны NULL, а стек левых указателей не пуст, то

//удаляем адрес из стека правых ссылок и на печатаем значение поля ключа вершины с данным адресом

{

while ((\*stk).elem != (\*((\*stk1).elem)).Left)

{

PopStack(&stk1, &u, &n);

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

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

}

// удаляем адреса из обоих стеков, а поле ключа выводим на экран дисплея.

PopStack(&stk1, &u, &n);

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

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

PopStack(&stk, &t, &n);

}

}

n = n + 1;

}

VyvodStack(&stk1);

}

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

// Помещение звена с элементами \*el и n в стек.

// \*stk - указатель на стек.

{

no\* q;

q = new(no);

(\*q).elem = \*el; (\*q).ch = \*n;

(\*q).sled = \*stk; \*stk = q;

}

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

// Извлечение из стека звена с элементами \*t и n.

// \*stk - указатель на стек.

{

no\* q;

if (\*stk != NULL)

{

\*t = (\*\*stk).elem;

\*n = (\*\*stk).ch;

q = \*stk;

\*stk = (\*\*stk).sled;

delete q;

}

}

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

// Вывод содержимого стека на экран дисплея.

// \*stk - указатель на стек.

{

node\* k;

int i, n;

while (\*stk != NULL)

{

k = (\*\*stk).elem; n = (\*\*stk).ch;

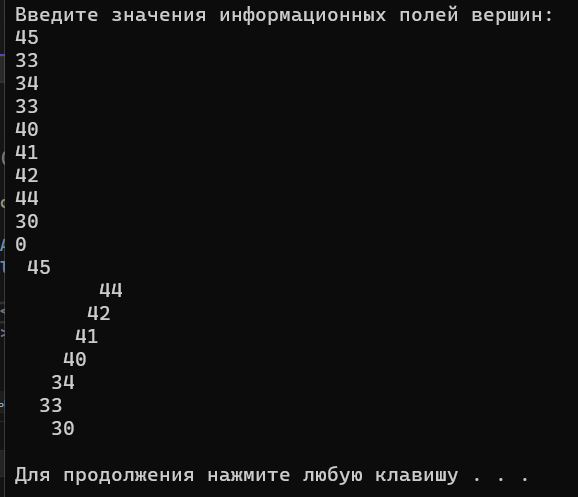
for (i = 0; i <= n; i++) cout << " ";

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

\*stk = (\*\*stk).sled;

}

}



1. Хэширование 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 вхождений ключа

node\* Left; // Указатель на левого сына

node\* Right; // Указатель на правого сына

};

class Spisok {

private:

node\* UkStr[N]; // Массив указателей на узлы дерева

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

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

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

public:

Spisok(); // Конструктор класса

void BuildTree(); // Функция построения дерева

void Sodergimoe(); // Функция вывода содержимого дерева

node\*\* GetTree(unsigned i) { return &(UkStr[i]); }; // Функция для получения указателя на дерево по индексу

void Udaldr(node\*\* d, int k); // Функция удаления узла из дерева

};

Spisok::Spisok()

{

//Инициализация хэш-списка.

for (int i = 0; i < N; i++) UkStr[i] = NULL;

}

void Spisok::BuildTree()

{

int klutch;

unsigned hash;

srand(time(0));

// автоматическая рандомизация

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

cin >> klutch;

//Закомментируйте следующие три строки,

//если нужно задавать значения ключей с клавиатуры.

//randomize();

//klutch = random(31);

//klutch = rand() % 31 + 0; // диапазон равен от 0 до 31 включительно

cout << klutch;

while (klutch != 0)

{

hash = klutch % 10; //Вычисление значения хэш-функции.

Search(klutch, &UkStr[hash]);

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

//cin >> klutch;

//klutch = random(31);

klutch = rand() % 31 + 0; // диапазон равен от 0 до 31 включительно

cout << klutch;

}

}

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

{

if (\*p == NULL)

{ // Если узла нет в деpеве,то включаем его.

\*p = new (node);

(\*\*p).Key = X;

(\*\*p).Count = 1;

(\*\*p).Left = (\*\*p).Right = NULL;

}

else

if (X < (\*\*p).Key) Search(X, &((\*\*p).Left));// Ищем в левом поддереве

else if (X > (\*\*p).Key) Search(X, &((\*\*p).Right));// Ищем в правом поддереве

else (\*\*p).Count += 1;// Увеличиваем счетчик вхождений ключа

}

void Spisok::Sodergimoe()

{

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

{

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

if (UkStr[i] == NULL) cout << "Деpево пусто...\n";

else

{

cout << endl;

PrintTree(UkStr[i], 0);// Выводим дерево

}

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

}

}

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

{

if (w != NULL)

{

PrintTree((\*w).Right, l + 1); // Рекурсивный вызов функции для правого поддерева

cout << " ";

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

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

PrintTree((\*w).Left, l + 1); // Рекурсивный вызов функции для левого поддерева

}

}

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

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

node\*\* q;

if (\*d == NULL) cout << "Узел с заданным ключом в деpеве не найден...\n";

else

if (k < (\*\*d).Key) Udaldr(&((\*\*d).Left), k); // Ищем в левом поддереве

else

if (k > (\*\*d).Key) Udaldr(&((\*\*d).Right), k); // Ищем в правом поддереве

else

{

q = d;

if ((\*\*q).Right == NULL) \*d = (\*\*q).Left;// Заменяем удаляемый узел на его левого сына

else

if ((\*\*q).Left == NULL) \*d = (\*\*q).Right;// Заменяем удаляемый узед на его правого потомка

else U\_d(&((\*\*q).Left), &(\*q));

}

}

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

{

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

{

(\*\*q).Key = (\*\*r).Key; // Присваиваем ключ удаляемого узла заменяемому узлу

(\*\*q).Count = (\*\*r).Count; // Присваиваем счетчик вхождений ключа заменяемому узлу

q = r; \*r = (\*\*r).Left;delete (\*q); // Заменяем удаляемый узлел на его левого потомка и освобождаем памяти

}

else U\_d(&((\*\*r).Right), &(\*q));

}

void main()

{

setlocale(LC\_ALL, "Rus");

Spisok A;

int klutch;

unsigned hash;

A.BuildTree();

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

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

A.Sodergimoe();

//Удаление элемента из хэш-списка.

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

{

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

cin >> klutch;

hash = klutch % 10;

A.Udaldr(A.GetTree(hash), klutch);

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

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

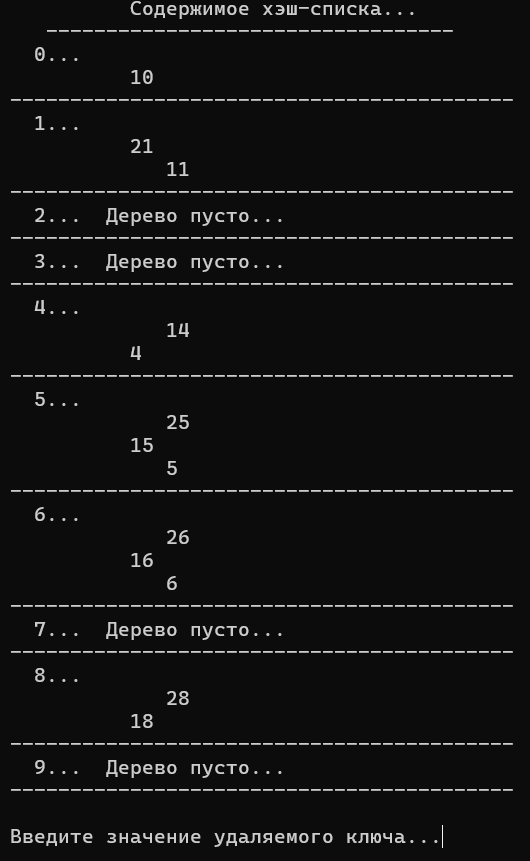
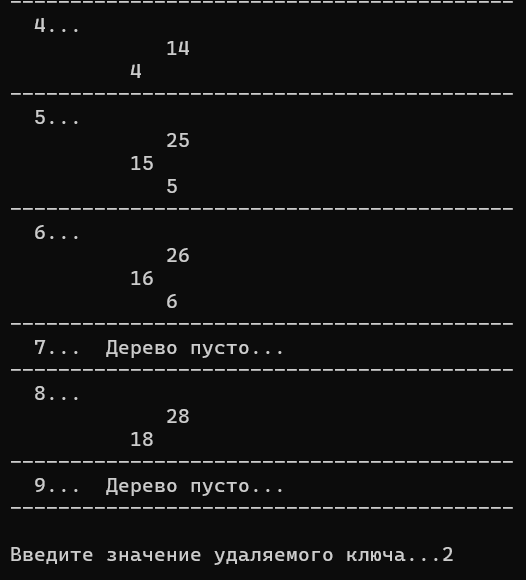
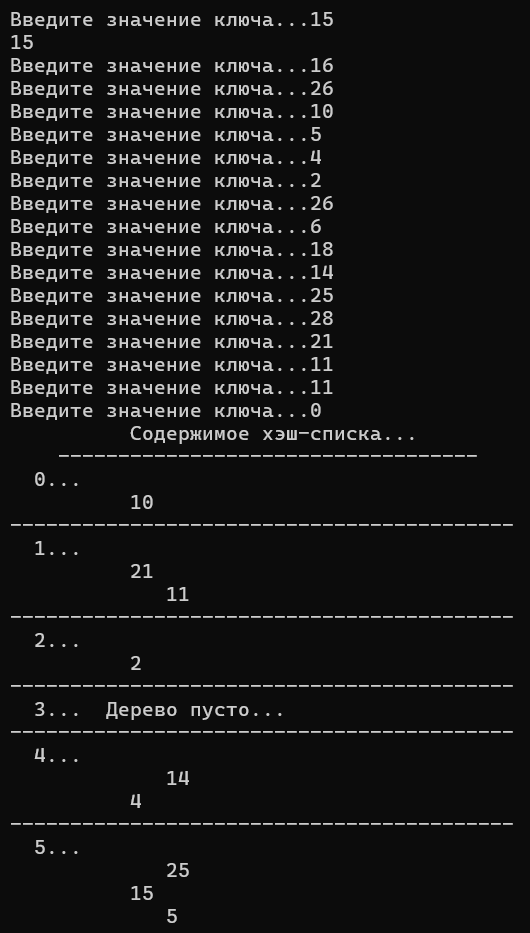
A.Sodergimoe();

}

cout << "\n";

system("PAUSE");

}



## 4. Дpевовидно-кольцевая динамическая стpуктуpа данных

#include <conio.h>

#include <iostream>

using namespace std;

struct node

{

int key; // Ключ узла

int count; // Счетчик вхождений ключа

node\* Left; // Указатель на левого сына

node\* Right; // Указатель на правого сына

};

class Tree

{

private:

node\* root; // Корень дерева

void DisposeTree(node\*); // Функция удаления дерева

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

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

void del(node\*\*, node\*); // Функция замены узла в дереве

public:

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

~Tree(); // Деструктор класса

void creat\_Tree(); // Функция создания дерева

void look\_Tree(); // Функция просмотра дерева

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

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

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

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

};

struct zveno

{

int element; // Элемент звена

Tree ukTree; // Дерево, связанное с звеном

zveno\* sled; // Указатель на следующее звено

};

class ring

{

private:

zveno\* ukring;

public:

ring() { ukring = NULL; };

~ring();

void create(); //Построение кольца

void look(); //Вывод кольца

void add\_after(int, zveno\*);

void add\_befor(int, zveno\*);

void Delete(zveno\*);

void delete\_next(zveno\*);

int poisk(int, zveno\*\*);

zveno\*\* getring() { return &ukring; };// Функция для получения первого элемента кольца

};

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();

cout << "\n-->";

cin >> elem;

while (elem != 0)

{

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

ukzv = (\*ukzv).sled;

(\*ukzv).element = elem;

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

cout << "\n-->";

cin >> elem;

}

ukzv->sled = ukring;

}

}

ring::~ring()

{

zveno\* ukzv;

ukzv = ukring;

while (ukring != NULL)

if (ukzv->sled == ukring)

{

ukring = NULL;

ukzv->ukTree.~Tree();

delete ukzv;

}

else

{

while (ukzv->sled->sled != ukring) ukzv = (\*ukzv).sled;

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

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();

ukzv = ukzv->sled;

\_getch();

} while (ukzv != ukring);

cout << endl;

}

// Функция добавления элемента перед указанным

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

{

zveno\* ukzv;

Tree temp;

ukzv = new (zveno);

temp = ukzv->ukTree;

ukzv->element = zv->element;

ukzv->ukTree = zv->ukTree;

ukzv->sled = zv->sled;

zv->element = elem;

zv->ukTree = temp;

zv->ukTree.creat\_Tree();

zv->sled = ukzv;

}

// Функция добавления элемента после указанного

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

{

zveno\* ukzv;

ukzv = new (zveno);

ukzv->element = elem;

ukzv->ukTree.creat\_Tree();

ukzv->sled = zv->sled;

zv->sled = ukzv;

}

// Функция удаления элемента из кольца

void ring::Delete(zveno\* zv)

{

zveno\* ukzv1, \* ukzv2;

zveno\* time;

if (zv->sled != ukring)

{

time = zv->sled;

zv->ukTree.~Tree();

(\*zv) = \*((\*zv).sled);

delete time;

}

else

if (zv->sled == zv)

{

zv->ukTree.~Tree();

delete ukring;

ukring = NULL;

cout << "Список пуст...\n";

}

else

{

ukzv2 = ukring;

ukzv1 = ukring->sled;

while (ukzv1 != zv)

{

ukzv2 = ukzv1; ukzv1 = ukzv1->sled;

}

time = ukzv2->sled;

ukzv2->sled->ukTree.~Tree();

ukzv2->sled = ukzv2->sled->sled;

delete time;

}

}

// Функция удаления следующего элемента

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

{

zveno\* time;

if (zv->sled != ukring)

{

time = zv->sled;

zv->sled = zv->sled->sled;

time->ukTree.~Tree();

delete time;

}

else

if (zv->sled == zv) cout << "В кольце только один элемент!\n";

else

{

time = ukring->sled;

\*((\*zv).sled) = (\*(\*(\*zv).sled).sled);

time->ukTree.~Tree();

delete time;

}

}

// Функция поиска элемента в кольце

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

{

zveno\* ukzv;

int vozvr = 0;

if (\*(getring()) == NULL) cout << "Кольцо не существует...\n";

else

{

ukzv = ukring;

while (ukzv->sled != ukring && (\*Res) == NULL)

{

if (ukzv->element == elem)

{

vozvr = 1; \*Res = ukzv;

}

ukzv = ukzv->sled;

}

if ((\*Res) == NULL)

if (ukzv->element == elem)

{

vozvr = 1; \*Res = ukzv;

}

}

return vozvr;

}

Tree::~Tree()

{

DisposeTree(root); root = NULL;

}

void Tree::DisposeTree(node\* p)

{

if (p != NULL)

{

DisposeTree(p->Left); DisposeTree(p->Right);

delete p;

}

}

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

{

if (\*p == NULL)

{

\*p = new (node); (\*\*p).key = x; (\*\*p).count = 1;

(\*\*p).Left = (\*\*p).Right = NULL;

}

else if (x < (\*\*p).key) search(x, &((\*\*p).Left));

else

if (x > (\*\*p).key) search(x, &((\*\*p).Right));

else (\*\*p).count += 1;

}

void Tree::creat\_Tree()

{

int elem;

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

cin >> elem;

while (elem != 0)

{

search(elem, &root);

cin >> elem;

}

}

void Tree::look\_Tree()

{

if (root == NULL) cout << "Дерево пусто ...\n";

else printTree(root, 0);

}

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

{

if (w != NULL)

{

printTree(w->Left, L + 1);

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

cout << w->key << endl;

printTree(w->Right, L + 1);

}

}

void Tree::add\_Tree()

{

int k;

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

cin >> k;

cout << " ";

while (k != 0)

{

search(k, &(root));

cin >> k;

cout << " ";

}

}

void Tree::delete\_Tree()

{

int elem;

if (root == NULL) cout << "Дерево пусто ...\n";

else

{

cout << "Введите ключ удаляемого узла : ";

cin >> elem;

cout << endl;

Delete(&root, elem);

}

}

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

{

node\* q;

node\* s;

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

else

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

else

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

else

{

q = \*d; s = \*d;

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

{

\*d = (\*q).Left;

delete s;

}

else

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

{

\*d = (\*q).Right;

delete s;

}

else del(&((\*q).Left), &(\*q));

}

}

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

{

node\* s;

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

{

(\*q).key = (\*\*r).key; (\*q).count = (\*\*r).count;

q = s = \*r; \*r = (\*\*r).Left;

delete s;

}

else del(&((\*\*r).Right), &(\*q));

}

void main()

{

setlocale(LC\_ALL, "Rus");

int menu1 = 1, choice, elem1, elem2, menu2;

ring A;

zveno\* Res;

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:

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

else cout << "Кольцо уже существует...\n";

break;

case 2:

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

else A.look();

break;

case 3:

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

else

{

Res = NULL;

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:

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);

}

else

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

}

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);

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

}

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);

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

}

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();

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

break;

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

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

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

case 5:

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

cout << "Дерево не существует...\n";

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

break;

case 6: menu2 = 0; break;

}

}

}

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

}

break;

case 8:

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

else A.~ring();

break;

case 9:

A.~ring();

menu1 = 0;

break;

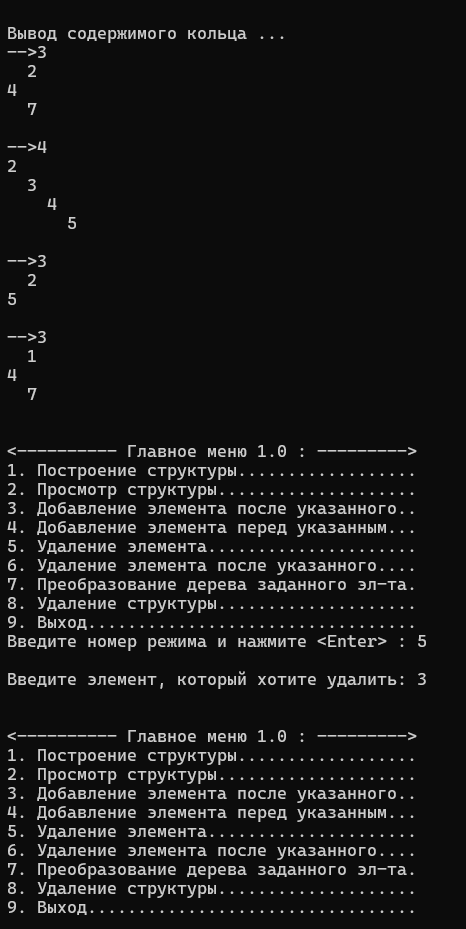
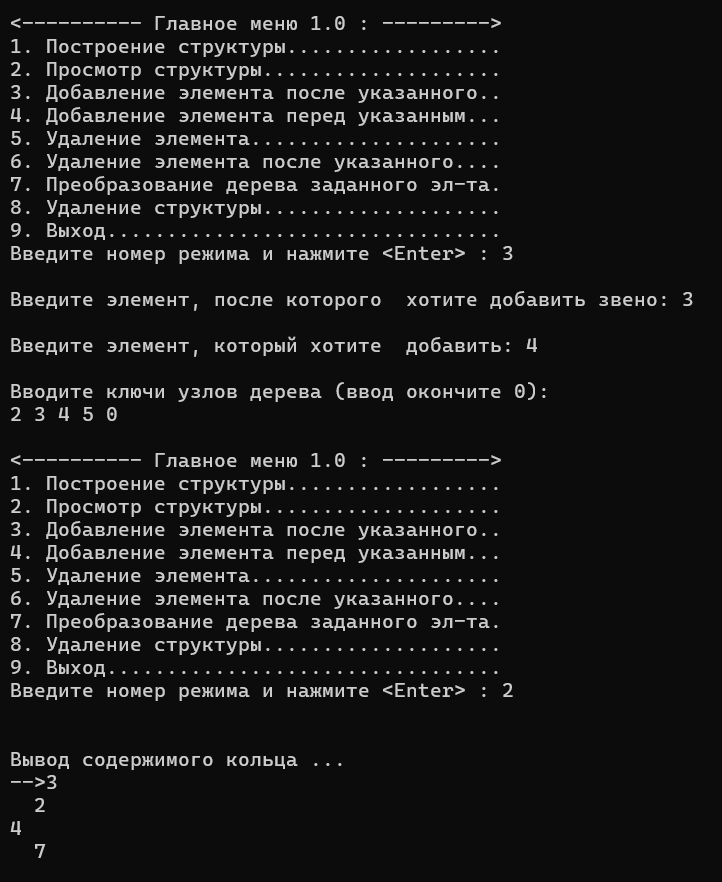
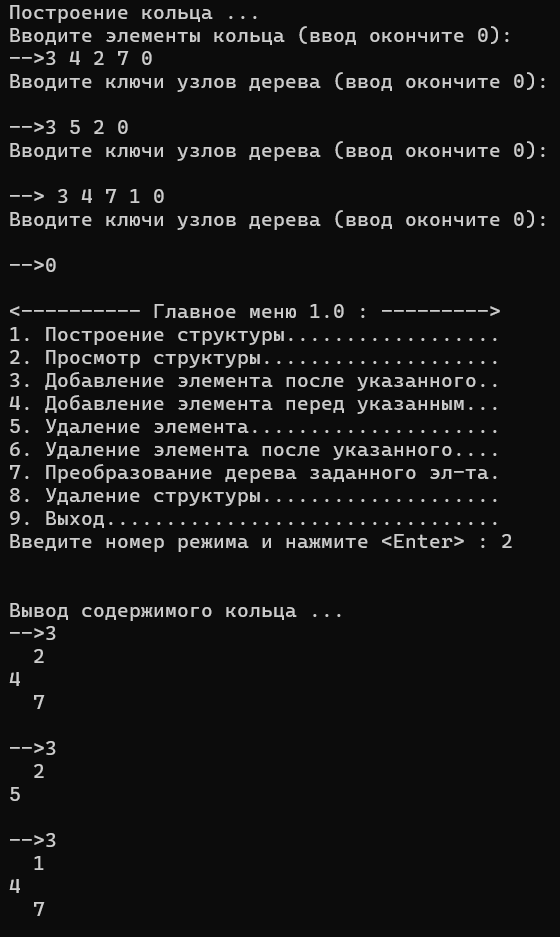
}

}

cout << "\n";

system("PAUSE");

}



5.Деревья Хаффмена

#include <stdio.h>

#include <conio.h>

#include <iostream>

#include <cstring>

using namespace std;

struct zveno

{

char Element; //Символ.

float Kol; //Количество повтоpений,

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

zveno\* Left; // Указатель на левое поддерево в дереве Хаффмана

zveno\* Right; // Указатель на правое поддерево в дереве Хаффмана

zveno\* Father; // Указатель на родительское звено в дереве Хаффмана

};

class Tree

{

private:

zveno\* UkStr; // Указатель на корневой узел дерева Хаффмана

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

public:

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

int Poisk(char, zveno\*\*); // Функция для поиска элемента в дереве

int Kolich(char\*, char); // Функция для подсчета количества повторений символа в тексте

void Dobavlenie(char, float, zveno\*\*); // Функция для добавления нового звена в дерево

void Redaktor(int); // Ффункция для замены количества повторений на частоту

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

void Vyvod(); //Функция для вывода дерева на экран

void WstawkaSort(zveno\*); // Функция для вставки нового звена в упорядоченный список

void PrintTree(zveno\*, int); // Функция для печати дерева Хаффмана

zveno\*\* GetTree() { return &UkStr; };

zveno\* GetTree1() { return UkStr; }; // Функция для получения указателя на корневой узел дерева

};

int Tree::Poisk(char ENT, // ENT- искомый элемент.

zveno\*\* Res //Указатель на него.

)

{

zveno\* q;

int vozvr = 0;

\*Res = NULL;

q = (\*UkStr).Sled; //Список с заглавным звеном!

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

{

if (q->Element == ENT)

{

vozvr = 1; \*Res = q; return vozvr;

}

q = q->Sled;

}

return vozvr;

}

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

//Поиск места в упоpядоченном списке для добавления элемента.

{

zveno\* q = (\*\*st).Sled, \* q1 = (\*st);

int vozvr = 0;

\*Res = NULL;

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

{

if (q->Kol < kol) { vozvr = 1; \*Res = q; }

q = q->Sled; q1 = q1->Sled;

}

if (\*Res == NULL) \*Res = q1;

return vozvr;

}

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

//Подсчет количества повтоpений буквы S в тексте F.

// Результат - в пеpеменной K.

{

int K = 0;

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

if (F[i] == S) K++;

return K;

}

void Tree::Redaktor(int L)

//Замена в поле Kol количества повтоpений на частоту повтоpений.

{

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

while (q != NULL)

{

q->Kol = q->Kol / L; q = q->Sled;

}

}

void Tree::Dobavlenie(char bukva, //Поля добавляемого

float kol, //элемента.

zveno\*\* Sp //Исходный список.

)

//Добавление звена в список, упоpядоченный по количеству повтоpений.

{

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

q = new (zveno);

q->Element = bukva;

q->Kol = kol;

q->Left = q->Right = NULL;

q->Sled = q->Father = NULL;

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

else

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

{

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

(\*Res) = (\*q); Res->Sled = kladovaq;

}

else Res->Sled = q;

}

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

//Поиск указателей на пpедпоследний и пpедпpедпоследний элементы.

{

\*zv\_p = UkStr->Sled; \*zv = UkStr;

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

{

\*zv = \*zv\_p; \*zv\_p = (\*zv\_p)->Sled;

}

}

void Tree::Vyvod()

//Вывод списка на экpан.

{

zveno\* q = UkStr->Sled;

while (q != NULL)

{

cout << q->Element << " (" << q->Kol << ") --> ";

q = q->Sled;

}

cout << endl;

}

void Tree::WstawkaSort(zveno\* zv)

{

zveno\* w1, \* w2;

w2 = UkStr; w1 = w2->Sled;

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

{

w2 = w1; w1 = w2->Sled;

}

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

{

w2->Sled = zv; zv->Sled = w1;

}

}

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

{

if (w != NULL)

{

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

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

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

PrintTree(w->Left, l + 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";

cin>>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;

i = 0;

while (i < strlen(Cod\_Haffmen))

{

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

{

if (Cod\_Haffmen[i] == '1') UkZv = UkZv->Left;

else UkZv = UkZv->Right;

i++;

}

char s[2];

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

strcat\_s(T, s);

UkZv = A.GetTree1()->Sled;

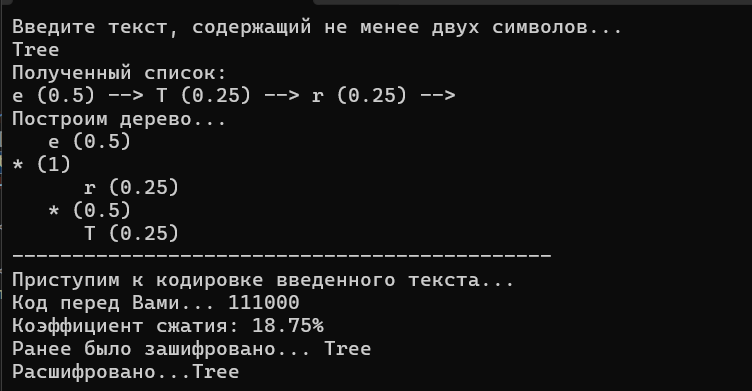
}

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

cout << "\n";

system("PAUSE");

}



6.Построение дерева-формулы

#include <stdio.h>

#include <conio.h>

#include <iostream>

using namespace std;

struct Uzel// Тип узла дерева

{

char Key; // Символ

Uzel\* Left; // Указатель на левое поддерево

Uzel\* Right; // Указатель на правое поддерево

};

struct zveno// Тип звена стека

{

Uzel\* Element; // Узел

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

};

class Tree

{

private:

Uzel\* Root; // Корень дерева

zveno\* Stack; // Стек для временного хранения узлов

public:

// Конструктор класса Tree

Tree();

void Udalenie(Uzel\*\*); // Функция для удаления узла из стека

void V\_stack(Uzel\*); // Функция для добавления узла в стек

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

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

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

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

// Геттер для получения корня дерева

Uzel\* GetTree() { return Root; };

};

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

{

zveno\* q = new (zveno);

q->Element = Elem;

q->Sled = Stack; // Установлеваем ссылку на следующее звено

Stack = q; // Обновляем указатель на вершину стека

}

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

{

if (Stack != NULL)

{

(\*tmp) = Stack->Element; // Сохраняем узел перед удалением

zveno\* q = Stack; // Сохраняем указатель на удаляемое звено

Stack = Stack->Sled; // Обновлеем указатель на вершину стека

delete q; // Освобождаем память

}

}

void Tree::PrintTree(Uzel\* w, int l)//Вывод деpева на экpан

{

if (w != NULL)

{

PrintTree(w->Right, l + 1); // Обходим правоое поддерево

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

cout << w->Key << endl; // Выводим ключ узла

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

}

}

void Tree::Print\_Tree\_Left(Uzel\* w, int l)//Левостоpонний обход бинаpного дерева

{

if (w != NULL)

{

cout << w->Key << " "; // Выводим ключ узла

Print\_Tree\_Left(w->Left, l + 1); // Обход левое поддерево

Print\_Tree\_Left(w->Right, l + 1); // Обход правое поддерево

}

}

void Tree::Print\_Tree\_End(Uzel\* w, int l)//Концевой обход бинаpного деpева

{

// Концевой обход дерева

if (w != NULL)

{

Print\_Tree\_End(w->Left, l + 1); // Обходим левое поддерево

Print\_Tree\_End(w->Right, l + 1); // Обходим правое поддерево

cout << w->Key << " "; // Выводим ключ узла

}

}

void Tree::Print\_Tree\_Back(Uzel\* w, int l) //Обpатный обход бинаpного деpева

{

if (w != NULL)

{

cout << "(";

Print\_Tree\_Back(w->Left, l + 1); // Обходим левое поддерево

cout << w->Key << " "; // Выводим ключ узла

Print\_Tree\_Back(w->Right, l + 1); // Обходим правое поддерево

cout << ")";

}

}

Tree::Tree()

{

Stack = NULL; // Вначале опустошим стек

Root = new (Uzel); // Создаем корневой узел

Root->Right = NULL; // Инициализируем правое поддерево корневого узла

}

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);

\_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");

}

