Белорусский государственный технологический университет

Факультет информационных технологий

Кафедра программной инженерии

Тестирование кода

По дисциплине «Основы алгоритмизации и программирования»

На тему «Деревья»

Выполнил:

Студент 1 курса 7 группы

Ленкевич Павел Андреевич

Преподаватель: Белодед Н. И.

2024, Минск

# Введение

В этом документе размещено тестирование пяти кодов с лекции за 31.04.2024. Большинство строк имеют пояснение в комментариях, а после кода представлен результат запуска и работы программы. Для того, чтоб быстро увидеть определённый код или результат его работы воспользуйтесь навигацией. Последний код откомментирован меньше всего, я оказался в очень жёстких временных рамках. Хорошего дня!

# Тестирование кода с лекции

## Левосторонний, концевой и обратный обход дерева

#include <iostream> // Подключаем библиотеку для ввода и вывода данных

using namespace std; // Используем пространство имен std, что позволяет нам не писать std:: перед каждым оператором ввода/вывода

// Определяем структуру "Узел"

struct node

{

int Key; // Ключ узла

int Count; // Количество узлов

node\* Left; // Указатель на левого потомка

node\* Right; // Указатель на правого потомка

};

// Определяем класс "Дерево"

class TREE

{

private:

node\* Tree; // Указатель на корень дерева

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

public:

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

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

void BuildTree(); // Метод для построения дерева

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

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

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

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

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

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

};

void main()

{

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

TREE A; // Создаем объект класса TREE

A.BuildTree(); // Строим дерево

cout << "\nВывод дерева:\n";

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

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

cout << "\nЛевосторонний обход дерева: ";

A.ObhodLeft(A.GetTree()); // Производим левосторонний обход дерева

cout << "\nКонцевой обход дерева: ";

A.ObhodEnd(A.GetTree()); // Производим концевой обход дерева

cout << "\nОбратный обход дерева: ";

A.ObhodBack(A.GetTree()); // Производим обратный обход дерева

cout << "\n";

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

}

void TREE::BuildTree()

// Функция считывает ключи узлов от пользователя и добавляем их в дерево

// Когда мы вводим 0, она прекращает считывание ключей и построение дерева

{

int el; // Объявляем переменную для хранения ключа узла

cout << "Вводите ключи вершин дерева:\n"; // Выводим приглашение для ввода ключей узлов

cin >> el; // Считываем ключ узла

while (el != 0) // Пока введенный ключ не равен нулю

{

Search(el, &Tree); // Ищем место для вставки узла в дерево и вставляем его

cin >> el; // Считываем следующий ключ узла

}

}

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

// Функция используется для поиска места в дереве, где должен быть вставлен новый узел

// Если узел с таким ключом уже существует, функция увеличивает счетчик этого узла на 1

// Если такого узла нет, функция создает новый узел с данным ключом

// Eсли ключ нового узла меньше ключа текущего узла, функция ищет место для вставки в левом поддереве, иначе - в правом

{

if (\*p == NULL) // Если узел пуст

{

\*p = new(node); // Создаем новый узел

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

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

(\*\*p).Left = NULL; // Устанавливаем указатель на левого потомка равным NULL

(\*\*p).Right = NULL; // Устанавливаем указатель на правого потомка равным NULL

}

else

if (x < (\*\*p).Key) // Если x меньше ключа узла

Search(x, &((\*\*p).Left)); // Ищем место для вставки узла в левом поддереве

else

if (x > (\*\*p).Key) // Если x больше ключа узла

Search(x, &((\*\*p).Right)); // Ищем место для вставки узла в правом поддереве

else

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

}

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

// Функция левостороннего обхода дерева

// Cначала обрабатывается корень, затем левое поддерево, а затем правое поддерево

// Если узел пуст, обход прекращается

// Если узел не пуст, функция выводит ключ узла, а затем рекурсивно обходит левое и правое поддеревья этого узла

{

if (\*w != NULL) // Если узел не пуст

{

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

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

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

}

}

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

// Функция концевого обхода

// Cначала обрабатывается левое поддерево, затем правое поддерево, а затем корень

// Если узел пуст, обход прекращается

// Если узел не пуст, функция рекурсивно обходит левое и правое поддеревья этого узла, а затем выводит ключ узла

{

if (\*w != NULL) // Если узел не пуст

{

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

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

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

}

}

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

// Функция обратного обхода

// Сначала обрабатывается левое поддерево, затем корень, а затем правое поддерево

// Если узел пуст, обход прекращается

// Если узел не пуст, функция рекурсивно обходит левое поддерево этого узла, выводит ключ узла, а затем рекурсивно обходит правое поддерево

{

if (\*w != NULL) // Если узел не пуст

{

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

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

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

}

}

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

// Функция очистки дерева

// Она рекурсивно обходит все узлы дерева и удаляет их

// Сначала очищает левое и правое поддеревья узла, а затем удаляет сам узел

// Если узел пуст, очистка прекращается

{

if (\*w != NULL) // Если узел не пуст

{

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

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

delete\* w; // Удаляем узел

}

}

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

// Функция вывода дерева

// Она рекурсивно обходит все узлы дерева и выводит их

// Сначала выводит правое поддерево, затем ключ узла, а затем левое поддерево

// Если узел пуст, вывод прекращается

{

int i;

if (\*w != NULL) // Если узел не пуст

{

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

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

cout << " "; // Выводим пробелы для отступа

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

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

}

}

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

// Функция вычисления высоты дерева

// Она рекурсивно обходит все узлы дерева и вычисляет их высоту

// Если узел пуст, функция возвращает -1

// Если узел не пуст, функция вычисляет высоту левого и правого поддеревьев этого узла, а затем возвращает максимальную из этих двух высот, увеличенную на 1

{

int h1, h2;

if (\*w == NULL) // Если узел пуст

return(-1); // Возвращаем -1

else // Если узел не пуст

{

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

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

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

return(1 + h1); // Возвращаем высоту левого поддерева + 1

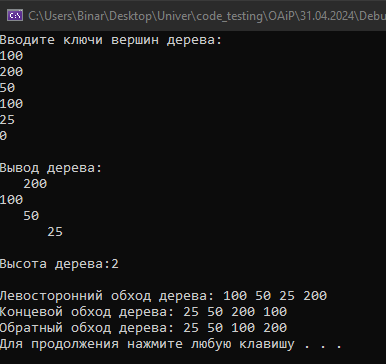
else

return(1 + h2); // Возвращаем высоту правого поддерева + 1

}

}

### Результат работы программы:



## Тестирование второго кода

#include <iostream> // Подключаем библиотеку для ввода и вывода данных

using namespace std; // Используем пространство имен std, что позволяет нам не писать std:: перед каждым оператором ввода/вывода

// Определение структуры узла для дерева

struct node

{

int Key; // Ключ узла

int Count; // Количество вхождений ключа

node\* Left; // Указатель на левого потомка

node\* Right; // Указатель на правого потомка

};

// Определение структуры для стека

struct no

{

node\* elem; // Элемент стека

int ch; // Счетчик

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

};

// Определение класса TREE

class TREE

{

private:

node\* Tree; // Указатель на корень дерева

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

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

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

public:

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

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

void TreeSearch(int); // Метод для поиска элемента в дереве

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

};

void main()

{

setlocale(LC\_ALL, "Rus"); // Установка русской локали

TREE A; // Создание объекта класса TREE

int el; // Переменная для хранения вводимого значения

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

cin >> el; // Ввод значения

while (el != 0) // Пока введенное значение не равно 0

{

A.TreeSearch(el); // Поиск введенного значения в дереве

cin >> el; // Ввод нового значения

}

A.VyvodTree(A.GetTreeRight()); // Вывод дерева

cout << "\n";

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

}

void TREE::TreeSearch(int el)

// Функция выполняет поиск в бинарном дереве поиска и добавляет новый узел, если введенное значение не найдено

// Если значение найдено, она увеличивает счетчик вхождений этого значения

{

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)

// Функция выполняет обход дерева и выводит его элементы

// В процессе обхода использует 2 стека для хранения узлов дерева

// Начинает обход с переданного ей узла, и выводит ключ каждого узла

{

no\* stk, \* stk1; // Указатели на элементы стека

node\* u; // Указатель на узел дерева

int i, n; // Переменные для хранения счетчика и индекса

stk = stk1 = NULL; // Инициализация указателей на стек

n = 0; // Инициализация счетчика

// Пока указатель на узел дерева не NULL

while (t != NULL)

{

PushStack(&stk1, &t, &n); // Добавление элемента в стек stk1

// Если у узла есть правый потомок

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

{

// Если у узла есть левый потомок

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

PushStack(&stk, &((\*t).Left), &n); // Добавление левого потомка в стек stk

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

}

else

{

// Если у узла есть левый потомок

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

{

// Если стек stk1 не пуст

if (stk1 != NULL)

{

PopStack(&stk1, &u, &n); // Удаление элемента из стека stk1

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

cout << " "; // Вывод пробелов для форматирования

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

}

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

}

else

// Если стек stk пуст

if (stk == NULL)

t = NULL;

else

{

// Пока элемент стека stk не равен левому потомку элемента стека stk1

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

{

PopStack(&stk1, &u, &n); // Удаление элемента из стека stk1

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

cout << " "; // Вывод пробелов для форматирования

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

}

PopStack(&stk1, &u, &n); // Удаление элемента из стека stk1

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

cout << " "; // Вывод пробелов для форматирования

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

PopStack(&stk, &t, &n); // Удаление элемента из стека stk

}

}

n = n + 1; // Увеличение счетчика

}

VyvodStack(&stk1); // Вывод стека stk1

}

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

// Функция добавляет новый элемент в стек

// Она принимает три аргумента: указатель на стек, указатель на узел дерева и указатель на счетчик

// Cоздает новый элемент стека, присваивает ему переданные значения и добавляет его в стек

// После работы функции вершина стека указывает на новый элемент

{

no\* q; // Указатель на элемент стека

q = new(no); // Создание нового элемента стека

(\*q).elem = \*el; // Присваивание элементу стека указателя на узел дерева

(\*q).ch = \*n; // Присваивание элементу стека значения счетчика

(\*q).sled = \*stk; // Присваивание элементу стека указателя на следующий элемент стека

\*stk = q; // Обновление указателя на вершину стека

}

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

// Функция удаляет элемент из стека

// Она принимает три аргумента: указатель на стек, указатель на узел дерева и указатель на счетчик

// Если стек не пуст, функция присваивает переданным указателям значения вершины стека, обновляет вершину стека и удаляет старую вершину

{

no\* q; // Указатель на элемент стека

// Если стек не пуст

if (\*stk != NULL)

{

\*t = (\*\*stk).elem; // Присваивание указателю на узел дерева значения элемента стека

\*n = (\*\*stk).ch; // Присваивание счетчику значения счетчика элемента стека

q = \*stk; // Сохранение указателя на вершину стека

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

delete q; // Удаление элемента стека

}

}

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

// Функция для вывода ключей элементов стека

// Принимает указатель на стек

// Если стек не пуст, функция присваивает указателю на узел дерева и счетчику значения вершины стека, выводит ключ узла с отступом,

// который зависит от значения счетчика, и обновляет вершину стека

{

node\* k; // Указатель на узел дерева

int i, n; // Переменные для хранения счетчика и индекса

// Пока стек не пуст

while (\*stk != NULL)

{

k = (\*\*stk).elem; // Присваивание указателю на узел дерева значения элемента стека

n = (\*\*stk).ch; // Присваивание счетчику значения счетчика элемента стека

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

cout << " "; // Вывод пробелов для форматирования

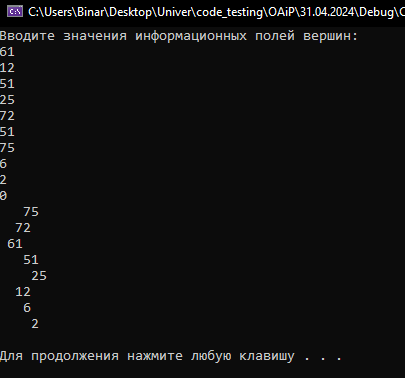
cout << (\*k).Key << endl; // Вывод ключа узла

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

}

}

### Результат работы программы:



## Тестирование третьего кода

#include<time.h> // Подключаем библиотеку для работы со временем

#include<iostream> // Подключаем библиотеку для работы с вводом-выводом

using namespace std; // Используем пространство имен std

#define N 10 // Определяем константу N равную 10

// Определяем структуру "узел"

struct node

{

int Key; // Ключ узла

int Cout; // Счетчик узла

node\* Left; // Указатель на левый узел

node\* Right; // Указатель на правый узел

};

// Определяем класс "Список"

class Spisok {

private:

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

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

void PrintTree(node\*, int); // Метод печати дерева

void U\_d(node\*\*, node\*\*); // Метод удаления узла

public:

Spisok(); // Конструктор класса

void BuildTree(); // Метод построения дерева

void Sodergimoe(); // Метод вывода содержимого

node\*\* GetTree(unsigned i) { return &(UkStr[i]); } // Метод получения дерева

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

};

// Реализация конструктора класса

Spisok::Spisok()

{

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

UkStr[i] = NULL; // Инициализируем все элементы массива указателей как NULL

}

// Реализация метода построения дерева

void Spisok::BuildTree()

{

int klutch; // Ключ узла

unsigned hash; // Хэш-значение

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

cout << "\nВведите значение ключа: ";

//cin>>klutch;

// Для ввода чисел с клавиатуры закомментируйте следующую строку и раскомментируйте предыдущую

klutch = rand() % 31 + 0; // Генерируем случайное число от 0 до 30

cout << klutch;

while (klutch != 0)

{

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

Search(klutch, &UkStr[hash]); // Ищем узел с данным ключом

cout << "\nВведите значение ключа: ";

//cin>>klutch;

klutch = rand() % 31 + 0; // Генерируем новое случайное число

cout << klutch;

}

}

// Реализация метода поиска узла

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

{

if (\*p == NULL) // Если узел не найден

{

\*p = new(node); // Создаем новый узел

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

(\*\*p).Cout = 1; // Инициализируем счетчик единицей

(\*\*p).Left = (\*\*p).Right = NULL; // Устанавливаем указатели на левый и правый узлы как NULL

}

else

if (X < (\*\*p).Key) // Если X меньше ключа

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

else

if (X > (\*\*p).Key) // Если X больше ключа

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

else

(\*\*p).Cout += 1; // Если X равно ключу, увеличиваем счетчик

}

// Реализация метода вывода содержимого

void Spisok::Sodergimoe()

{

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

{

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

if (UkStr[i] == NULL) // Если дерево пусто

cout << "Дерево пусто.\n";

else

{

cout << endl;

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

}

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

}

}

// Реализация метода печати дерева

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

{

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

{

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

cout << " ";

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

cout << " ";

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

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

}

}

// Реализация метода удаления узла

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

{

node\*\* q;

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

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

else

if (k < (\*\*d).Key) // Если k меньше ключа

Udaldr(&((\*\*d).Left), k); // Удаляем узел в левом поддереве

else

if (k > (\*\*d).Key) // Если k больше ключа

Udaldr(&((\*\*d).Right), k); // Удаляем узел в правом поддереве

else

{

q = d;

if ((\*\*q).Right == NULL) // Если правый узел отсутствует

\*d = (\*\*q).Left; // Устанавливаем указатель на левый узел

else

if ((\*\*q).Left == NULL) // Если левый узел отсутствует

\*d = (\*\*q).Right; // Устанавливаем указатель на правый узел

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

}

}

// Реализация метода удаления узла

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

{

if ((\*\*r).Right == NULL) // Если правый узел отсутствует

{

(\*\*q).Key = (\*\*r).Key; // Копируем ключ

(\*\*q).Cout = (\*\*r).Cout; // Копируем счетчик

q = r;

\*r = (\*\*r).Left; // Устанавливаем указатель на левый узел

delete(\*q); // Удаляем узел

}

else

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

}

// Главная функция

void main()

{

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

Spisok A; // Создаем объект класса "Список"

int klutch; // Ключ узла

unsigned hash; // Хэш-значение

A.BuildTree(); // Строим дерево

cout << "\n";

cout << "\n";

A.Sodergimoe(); // Выводим содержимое

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

{

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

cin >> klutch; // Вводим ключ удаляемого узла

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

A.Udaldr(A.GetTree(hash), klutch); // Удаляем узел

cout << " Содержимое хэш-списка: \n";

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

A.Sodergimoe(); // Выводим содержимое

}

cout << "\n";

system("PAUSE"); // Остановка работы программы

}

### Результат работы программы:

## 4.Тестирование четвёртого кода

#include <stdio.h> // Подключение библиотеки для работы с вводом/выводом

#include<conio.h> // Подключение библиотеки для работы с консолью

#include<iostream> // Подключение библиотеки для работы с потоками ввода/вывода

using namespace std; // Использование пространства имен std

// Определение структуры "Узел"

struct Uzel

{

char Key; // Ключ узла

Uzel\* Left; // Указатель на левого потомка

Uzel\* Right; // Указатель на правого потомка

};

// Определение структуры "Звено"

struct zveno

{

Uzel\* Element; // Указатель на элемент

zveno\* Sled; // Указатель на следующее звено

};

// Определение класса "Дерево"

class Tree

{

private:

Uzel\* Root; // Корень дерева

zveno\* Stack; // Стек

public:

Tree(); // Конструктор

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

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

void PrintTree(Uzel\*, int); // Функция печати дерева

void Print\_Tree\_Left(Uzel\*, int); // Функция печати дерева (левый обход)

void Print\_Tree\_End(Uzel\*, int); // Функция печати дерева (конечный обход)

void Print\_Tree\_Back(Uzel\*, int); // Функция печати дерева (обратный обход)

Uzel\* GetTree() { return Root; }; // Функция получения корня дерева

};

// Функция добавления элемента в стек

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

{

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

q->Element = Elem; // Присваивание элемента звену

q->Sled = Stack; // Присваивание следующего звена

Stack = q; // Обновление вершины стека

}

// Функция удаления узла

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

{

zveno\* q;

if (Stack != NULL) // Если стек не пуст

{

(\*tmp) = Stack->Element; // Получение элемента из вершины стека

q = Stack; // Сохранение указателя на вершину стека

Stack = Stack->Sled; // Обновление вершины стека

delete q; // Удаление звена

}

}

// Функция печати дерева

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

{

if (w != NULL) // Если узел не пуст

{

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

for (int i = 1; i <= l; i++) // Вывод отступов

cout << " ";

cout << w->Key << endl; // Вывод ключа узла

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

}

}

// Функция печати дерева (левый обход)

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

{

if (w != NULL) // Если узел не пуст

{

cout << w->Key << " "; // Вывод ключа узла

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

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

}

}

// Функция печати дерева (конечный обход)

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

{

if (w != NULL) // Если узел не пуст

{

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

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

cout << w->Key << " "; // Вывод ключа узла

}

}

// Функция печати дерева (обратный обход)

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

{

if (w != NULL) // Если узел не пуст

{

cout << "("; // Вывод открывающей скобки

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

cout << w->Key << " "; // Вывод ключа узла

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

cout << ")"; // Вывод закрывающей скобки

}

}

// Конструктор класса "Дерево"

Tree::Tree()

{

Stack = NULL; // Инициализация стека

Root = new(Uzel); // Создание корня дерева

Root->Right = NULL; // Инициализация правого потомка корня

}

void main()

{

setlocale(LC\_ALL, "Rus"); // Установка русской локали для корректного отображения кириллицы

char Formula\_Post[30]; // Массив для хранения формулы

char k; // Переменная для хранения текущего символа формулы

Uzel\* Ukazatel = NULL; // Указатель на текущий узел дерева

cout << "Введите формулу записанную в постфиксной форме..\n"; // Вывод приглашения для ввода формулы

gets\_s(Formula\_Post); // Считывание формулы с консоли

\_strrev(Formula\_Post); // Реверсирование формулы (переворачивание строки)

cout << "Приступим к построению дерева формулы..\n"; // Вывод сообщения о начале построения дерева

Tree A; // Создание объекта класса "Дерево"

Uzel\* Temp = A.GetTree(); // Получение корня дерева

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

{

k = Formula\_Post[i];

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

if (Temp->Right == NULL)

{

Temp->Right = new(Uzel);

Temp = Temp->Right;

Temp->Key = k;

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

A.V\_stack(Temp);

}

else

{

Temp->Left = new(Uzel);

Temp = Temp->Left;

Temp->Key = k;

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

A.V\_stack(Temp);

}

else

if (Temp->Right == NULL)

{

Temp->Right = new(Uzel);

Temp = Temp->Right;

Temp->Key = k;

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

A.Udalenie(&Ukazatel);

Temp = Ukazatel;

}

else

{

Temp->Left = new(Uzel);

Temp = Temp->Left;

Temp->Key = k;

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

A.Udalenie(&Ukazatel);

Temp = Ukazatel;

}

}

cout << "\nКонтрольный вывод дерева-формулы..\n"; // Вывод сообщения о начале контрольного вывода дерева

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

cout << "Перед Вами формула, записанная в инфиксной форме..\n"; // Вывод сообщения о начале вывода формулы в инфиксной форме

A.Print\_Tree\_Back(A.GetTree()->Right, 0); // Печать дерева (обратный обход)

cout << endl;

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

cout << "Перед Вами формула, записанная в префиксной форме..\n"; // Вывод сообщения о начале вывода формулы в префиксной форме

A.Print\_Tree\_Left(A.GetTree()->Right, 0); // Печать дерева (левый обход)

cout << endl;

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

cout << "Перед Вами формула, записанная в постфиксной форме..\n"; // Вывод сообщения о начале вывода формулы в постфиксной форме

A.Print\_Tree\_End(A.GetTree()->Right, 0); // Печать дерева (конечный обход)

cout << "\n";

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

}

### Результат работы программы:

## Тестирование пятого кода

// Подключение библиотек

#include<conio.h>

#include<iostream>

// Использование пространства имен std

using namespace std;

// Структура узла дерева

struct node

{

int key; // Ключ узла

int count; // Количество узлов

node\* Left; // Указатель на левый узел

node\* Right; // Указатель на правый узел

};

// Класс дерева

class Tree

{

private:

node\* root; // Корень дерева

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

void printTree(node\*, int); // Метод для печати дерева

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

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

public:

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

~Tree(); // Деструктор дерева

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

void look\_Tree(); // Метод для просмотра дерева

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 в дереве

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

};

// Структура звена кольца

struct zveno

{

int element; // Элемент звена

Tree ukTree; // Дерево, связанное с звеном

zveno\* sled; // Указатель на следующее звено

};

// Класс кольца

class ring

{

private:

zveno\* ukring; // Указатель на текущее звено кольца

public:

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

~ring(); // Деструктор кольца

void create(); // Метод для создания кольца

void look(); // Метод для просмотра кольца

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

void add\_befor(int, zveno\*); // Метод для добавления звена перед указанным звеном

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

void delete\_next(zveno\*); // Метод для удаления звена после указанного звена

int poisk(int, zveno\*\*); // Метод для поиска элемента в кольце

zveno\*\* getring() { return &ukring; }; // Метод для получения текущего звена кольца

};

void ring::create()

{

zveno\* ukzv;

int elem;

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

cout << "Вводите элементы кольца (ввод окончить 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()

{

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;

}

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;

}

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;

}

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;

}

}

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;

}

}

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 обавляемых 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 ключи 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 ((\*d)->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"); // подключение русского языка

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

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 хотите добавить звено: ";

cin >> elem1;

cout << endl;

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

{

cout << "Введите элемент который 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 хотите добавить звено: ";

cin >> elem1;

cout << endl;

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

{

cout << "Введите элемент который 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 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 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 << "<----------- Меню 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 << "Введеите 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;

default:

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");

}

### Результат работы программы:

