МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

Учреждение образования «БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ

ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

Факультет: Информационных технологий

Кафедра: Программной инженерии

Выполнила: студентка 1 курса 5 группы

специальности ПОИТ Дзивнель М.А.

Проверил: преподаватель

Белодед Николай Иванович

**ТЕСТИРОВАНИЕ КОДА**

По дисциплине “Основы алгоритмизации и программирования”

Минск

2023

Содержание

[**ВВЕДЕНИЕ** 3](file:///C:\Users\VivoBook\Downloads\Telegram%20Desktop\Динамические_структуры_данныхБинарные_деревья_C++_Тестирование_кода.docx#_Toc134158454)

[ПРОГРАММА №1 3](file:///C:\Users\VivoBook\Downloads\Telegram%20Desktop\Динамические_структуры_данныхБинарные_деревья_C++_Тестирование_кода.docx#_Toc134158455)

[ПРОГРАММА №2 5](file:///C:\Users\VivoBook\Downloads\Telegram%20Desktop\Динамические_структуры_данныхБинарные_деревья_C++_Тестирование_кода.docx#_Toc134158456)

[ПРОГРАММА №3 7](file:///C:\Users\VivoBook\Downloads\Telegram%20Desktop\Динамические_структуры_данныхБинарные_деревья_C++_Тестирование_кода.docx#_Toc134158457)

[ПРОГРАММА №4 10](file:///C:\Users\VivoBook\Downloads\Telegram%20Desktop\Динамические_структуры_данныхБинарные_деревья_C++_Тестирование_кода.docx#_Toc134158458)

[ПРОГРАММА №5 14](file:///C:\Users\VivoBook\Downloads\Telegram%20Desktop\Динамические_структуры_данныхБинарные_деревья_C++_Тестирование_кода.docx#_Toc134158459)

# **ВВЕДЕНИЕ**

Тестирование кода и дополнение его комментариями, реализующего алгоритмы операций над бинарными деревьями.

# ПРОГРАММА №1

Построение дерева отрезков и вывод его в консоль.

#include <iostream>

using namespace std;

struct node // определение структуры node

{

int KeyMin; // минимальное значение ключа вершины

int KeyMax; // максимальное значение ключа вершины

node\* Left; // Указатель на "левого" потомка

node\* Right; // Указатель на "правого" потомка

};

class TREE // определение класса TREE

{

private:// указывает, что все переменные будут доступны только внутри класса

node\* Tree; // указатель на корень дерева

void Search(int, int, node\*\*); // прототип функции построения дерева отрезков

public:

TREE() { Tree = NULL; } // конструктор класса, инициализирующий указатель Tree значением NULL

void BuildTree(); // прототип функции построения дерева отрезков

node\*\* GetTree() { return &Tree; } // прототип функции получения вершины дерева

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

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

};

void main() // главная функция

{

setlocale(LC\_ALL, "Rus"); // подключение русского языка в консоль

TREE A; // создание объекта А типа TREE

A.BuildTree(); // вызов функции построения дерева отрезков

cout << "\nВывод дерева:\n"; // вывод сообщения пользователю

A.Vyvod(A.GetTree(), 0); // вызов функции вывода дерева

A.CleanTree(A.GetTree()); // вызов функции очистки дерева

cout << "\n";

system("PAUSE");// вызов командного интерпретатора с параметром pause

}

void TREE::BuildTree() // реализация функции построения дерева

// Tree - указатель на корень дерева.

{

int k1, k2; // объявления двух числовых переменных

cout << "Введите два целых числа...\n";

cin >> k1; // ввод первой переменной

cin >> k2; // ивод второй переменной

Search(k1, k2, &Tree); // Вызов функции поиска

}

void TREE::Search(int k1, int k2, node\*\* p) // функция построения дерева отрезков

// p - указатель на корень дерева.

{

if (k2 - k1 > 1) // если отрезок больше 1

{

\*p = new (node); // выделение памяти для новой структуры node

(\*\*p).KeyMin = k1; // запись минимального значения ключа в узел

(\*\*p).KeyMax = k2; // запись максимального значения ключа в узел

(\*\*p).Left = (\*\*p).Right = NULL; // зануление указателей на левое и правое поддеревья

Search(k1, (k1 + k2) / 2, &((\*\*p).Left)); // рекурсивный вызов поиска для левого поддерева

Search((k1 + k2) / 2, k2, &((\*\*p).Right)); // рекурсивный вызов поиска для правого поддерева

}

else // иначе

{

\*p = new (node); // выделение памяти для новой структуры node

(\*\*p).KeyMin = k1; // запись минимального значения ключа в узел

(\*\*p).KeyMax = k2; // запись максимального значения ключа в узел

(\*\*p).Left = (\*\*p).Right = NULL; // зануление указателей на левое и правое поддеревья

}

}

void TREE::CleanTree(node\*\* w) // реализация функции очистки дерева

//\*w - указатель на корень дерева.

{

if (\*w != NULL)// если узел не пустой

{

CleanTree(&((\*\*w).Left)); // рекурсивный вызов очистки для левого поддерева

CleanTree(&((\*\*w).Right)); // рекурсивный вызов очистки для правого поддерева

delete\* w; // удаление узла

}

}

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

//\*w - указатель на корень дерева.

{

int i; // объявление переменной i

if (\*w != NULL) // если узел не пустой

{

Vyvod(&((\*\*w).Right), l + 1); // рекурсивный вызов вывода для правого поддерева

for (i = 1; i <= l; i++) cout << " "; // вывод отступов

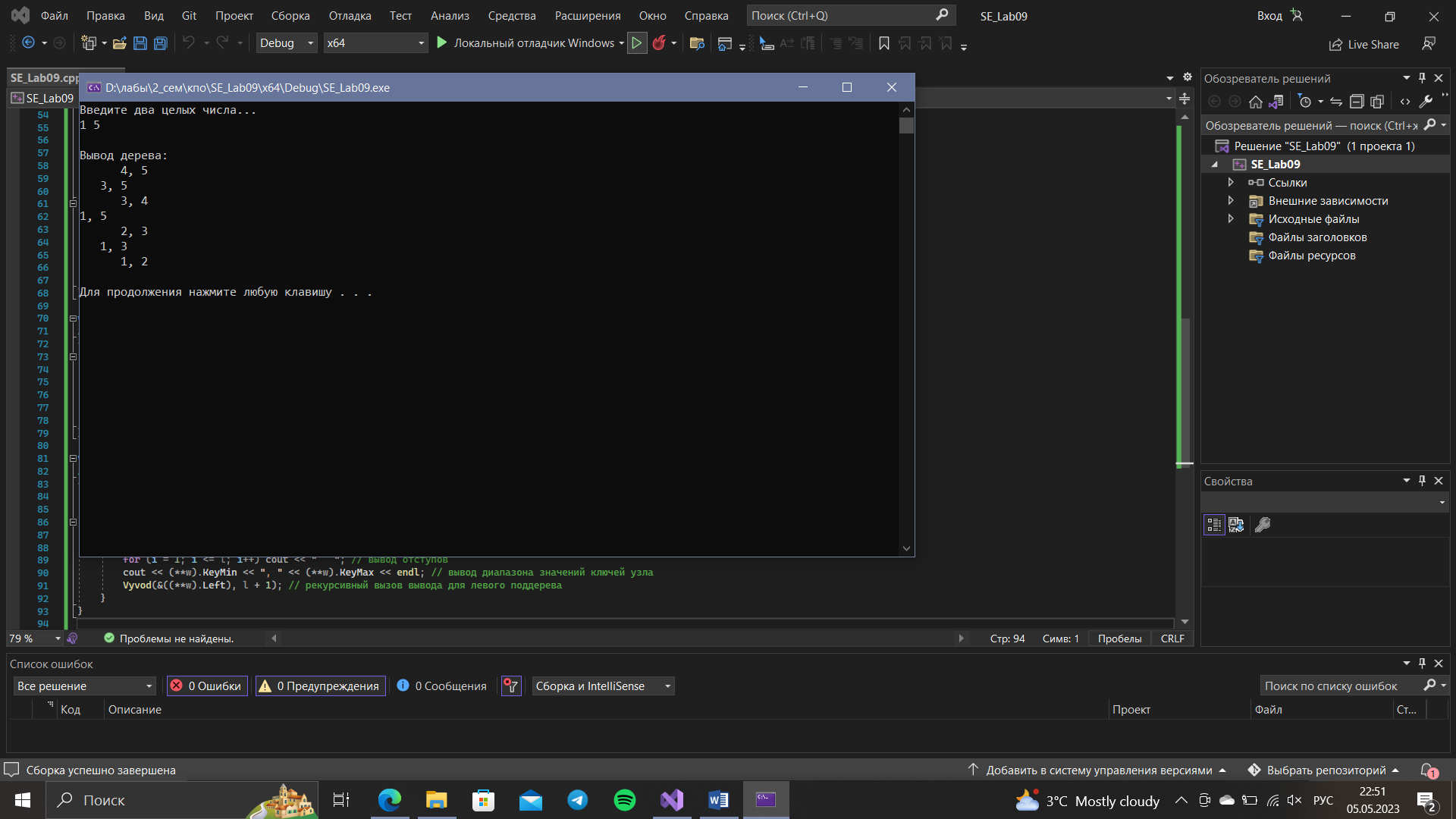
cout << (\*\*w).KeyMin << ", " << (\*\*w).KeyMax << endl; // вывод диапазона значений ключей узла

Vyvod(&((\*\*w).Left), l + 1); // рекурсивный вызов вывода для левого поддерева

}

}

Результат программы:



# ПРОГРАММА №2

Нахождение числа интервалов, содержащих точку X, с помощью двоичного поиска.

#include <iostream>

using namespace st;

struct node // определение структуры node

{

int KeyMin; // минимальный ключ вершины

int KeyMax; // максимальный ключ вершины

node\* Left; // указатель на "левого" потомка

node\* Right; // Указатель на "правого" потомка

};

class TREE // определение класса TREE

{

private:// указывает, что все переменные будут доступны только внутри класса

node\* Tree; //указатель на корень дерева

int S; //количество вхождений заданной точки в дерево

void Search(int, int, node\*\*); // прототип функции построения дерева отрезков

public:

TREE() { Tree = NULL; S = 0; }// конструктор класса, инициализирующий указатель Tree значением NULL, а количество вхождений нулем

void BuildTree(); //прототип функции построения дерева отрезков

node\*\* GetTree() { return &Tree; } //прототип функции получения вершины дерева

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

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

int GetCount() { return S; } // прототип функции получения количества интервалов

void Count(node\*\*, float);// прототип функции подсчёта интервалов

};

void main()// главная функция

{

setlocale(LC\_ALL, "Rus"); подключение русского языка

TREE A;// создание объекта А типа TREE

float X; // инициализация абсциссы точки

A.BuildTree(); // вызов функции построение дерева

cout << "\nВывод дерева:\n;

A.Vyvod(A.GetTree(), 0); //вывод дерева

cout << "\nВведите абсциссу точки:”;

cin >> X; // ввод абсциссы

A.Count(A.GetTree(), X); // вызов функции подсчёта интервалов

cout << "Точка принадлежит " << A.GetCount() << " интервалам"; // вывод сообщения пользователю и количества интервалов

A.CleanTree(A.GetTree()); // вызов функции очистки дерева

cout << "\n";

system("PAUSE");// вызов командного интерпретатора с параметром pause

}

void TREE::BuildTree() // реализация функции построения дерева

// Tree - указатель на корень дерева.

{

int k1, k2; // объявления двух числовых переменных

cout << "Введите два целых числа...\n"; // вывод сообщения пользователю

cin >> k1; // ввод первой переменной

cin >> k2; // ввод второй переменной

Search(k1, k2, &Tree); // вызов функции построения дерева отрезков

}

void TREE::Search(int k1, int k2, node\*\* p) // функция построения дерева отрезков

// p - указатель на корень дерева.

{

if (k2 - k1 > 1) // если отрезок больше 1

{

\*p = new (node); // выделение памяти для новой структуры node

(\*\*p).KeyMin = k1; // запись минимального значения ключа в узел

(\*\*p).KeyMax = k2; // запись максимального значения ключа в узел

(\*\*p).Left = (\*\*p).Right = NULL; // зануление указателей на левое и правое поддеревья

Search(k1, (k1 + k2) / 2, &((\*\*p).Left)); // рекурсивный вызов поиска для левого поддерева

Search((k1 + k2) / 2, k2, &((\*\*p).Right)); // рекурсивный вызов поиска для правого поддерева

}

else // иначе

{

\*p = new (node); // выделение памяти для новой структуры node

(\*\*p).KeyMin = k1; // запись минимального значения ключа в узел

(\*\*p).KeyMax = k2; // запись максимального значения ключа в узел

(\*\*p).Left = (\*\*p).Right = NULL; // зануление указателей на левое и правое поддеревья

}

}

void TREE::Count(node\*\* p, float X) // реализация функции подсчёта интервалов

// содеpжащих точку X

{

if (\*p != NULL) // если узел не пустой

{

Count(&((\*\*p).Right), X); // рекурсивный вызов подсчёта интервалов для правого поддерева

if (X >= (\*\*p).KeyMin && X <= (\*\*p).KeyMax) S++; // если абсцисса находится между максимальным и минимальным узлами, то счётчик интервалов увеличиваем на один

Count(&((\*\*p).Left), X);// рекурсивный вызов подсчёта интервалов для левого поддерева

}

}

void TREE::CleanTree(node\*\* w) // реализация функции очистки дерева

//\*w - указатель на корень дерева.

{

if (\*w != NULL)// если узел не пустой

{

CleanTree(&((\*\*w).Left)); // рекурсивный вызов очистки для левого поддерева

CleanTree(&((\*\*w).Right)); // рекурсивный вызов очистки для правого поддерева

delete\* w; // удаление узла

}

}

void TREE::Vyvod(node\*\* w, int l) //реализация функции изображение дерева \*w на экране дисплея

//\*w - указатель на корень дерева.

{

int i; // объявление переменной i

if (\*w != NULL) // если узел не пустой

{

Vyvod(&((\*\*w).Right), l + 1); // рекурсивный вызов вывода для правого поддерева

for (i = 1; i <= l; i++) cout << " "; // вывод отступов

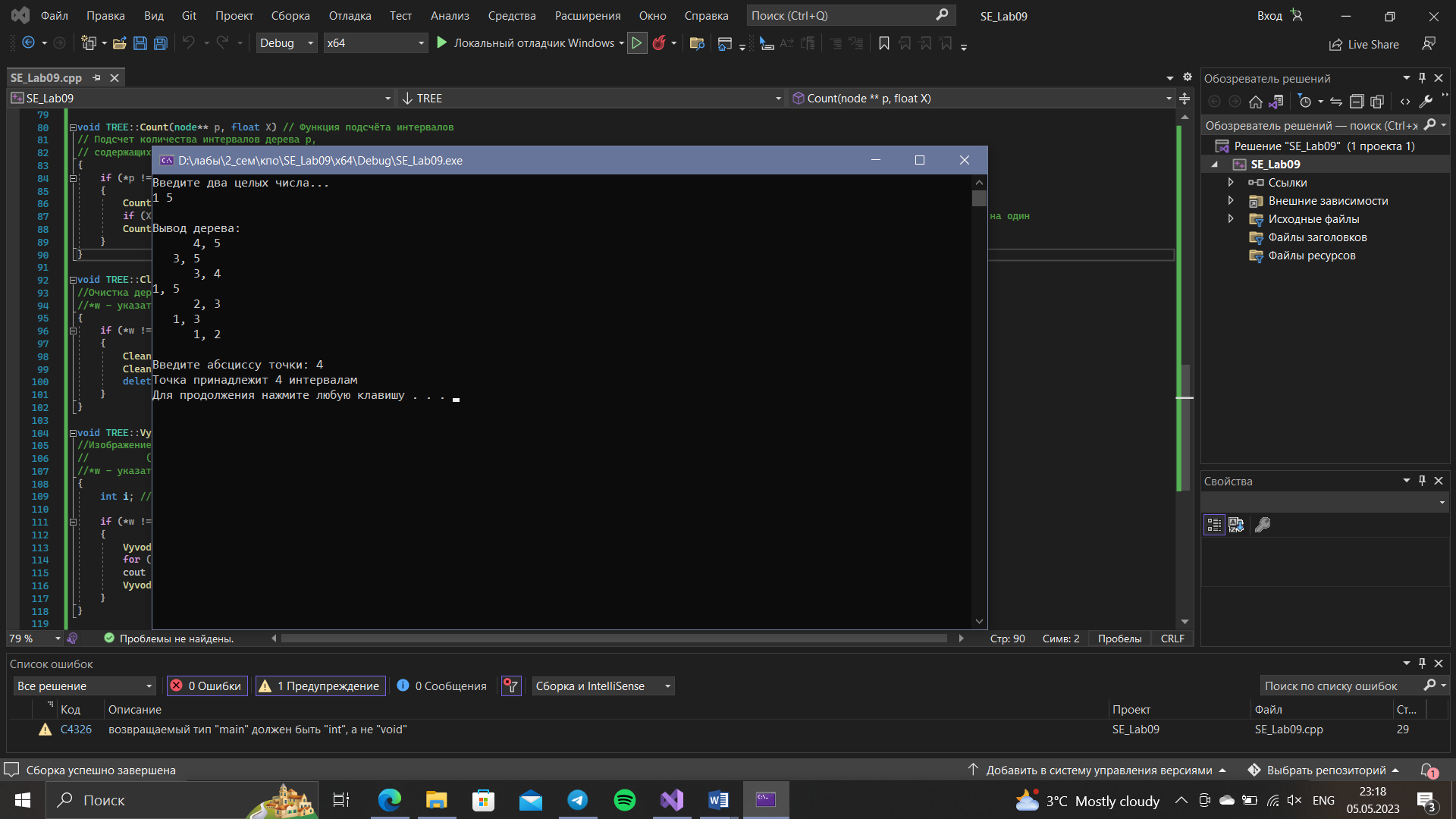
cout << (\*\*w).KeyMin << ", " << (\*\*w).KeyMax << endl; // вывод диапазона значений ключей узла

Vyvod(&((\*\*w).Left), l + 1); // рекурсивный вызов вывода для левого поддерева

}

}

Результат программы:



# ПРОГРАММА №3

Нахождение числа интервалов, содержащих точку X, с помощью двоичного поиска.

#include <iostream>

using namespace std;

struct node // определение структуры node

{

int Key; // ключ узла дерева

int Count; // количество узлов

node\* Left; // указатель на "левого" потомка

node\* Right; // указатель на "правого" потомка

};

class TREE // определение класса TREE

{

private: // указывает, что все переменные будут доступны только внутри класса

node\* Tree; //указатель на корень дерева

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

public:

TREE() { Tree = NULL; } // конструктор класса, инициализирующий указатель Tree значением NULL

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

void BuildTree(); // прототип функции построение дерева

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

void ObhodEnd(node\*\*); // прототип функции концевой обход дерева

void ObhodLeft(node\*\*); // прототип функции левосторонний обход дерева

void ObhodBack(node\*\*); // прототип функции обратный обход дерева

void Vyvod(node\*\*, int); // прототип функции изображение дерева на экране дисплея

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

};

void main()// главная функция

{

setlocale(LC\_ALL, "Rus"); //подключение русского языка

TREE A;// создание объекта А типа TREE

A.BuildTree(); // строим дерево

cout << "\nВывод дерева:\n"; // вывод сообщения для пользователя

A.Vyvod(A.GetTree(), 0); // вывод дерева

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

cout << "\nЛевосторонний обход дерева: ";// вывод сообщения для пользователя

A.ObhodLeft(A.GetTree()); // вызов функции обхода слева

cout << "\nКонцевой обход дерева: "; A.ObhodEnd(A.GetTree()); // вызов функции кольцевого обхода дерева

cout << "\nОбратный обход дерева: "; A.ObhodBack(A.GetTree()); // вызов функции обратного обхода дерева

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

cout << "\n";

system("PAUSE"); // вызов командного интерпретатора с параметром pause

}

void TREE::BuildTree() // реализация функции построения дерева

// Tree - указатель на корень дерева.

{

int el; // инициализация элемента

cout << "Вводите ключи вершин дерева ...\n";// вывод сообщения для пользователя

cin >> el; // ввод элемента дерева

while (el != 0) // пока элемент не ноль

{

Search(el, &Tree); cin >> el; // поиск вершины с ключом x в дереве со вставкой

}

}

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

// \*p - указатель на корень дерева.

{

if (\*p == NULL) // если в вершины нет

{

\*p = new(node); // выделяем место для нового элемента

(\*\*p).Key = x; // присваиваем значение корню дерева

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

(\*\*p).Left = NULL; // указатель на правое поддерево NULL

(\*\*p).Right = NULL; // указатель на левое поддерево NULL

}

else // иначе

if (x < (\*\*p).Key) Search(x, &((\*\*p).Left)); // если ключ искомого узла меньше ключа вершины, то производим рекурсивный поиск по левому поддереву

else // иначе

if (x > (\*\*p).Key) Search(x, &((\*\*p).Right)); //если ключ искомого узла больше ключа вершины, то производим рекурсивный поиск по правому поддереву

else // иначе

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

}

void TREE::ObhodLeft(node\*\* w) // реализация функции левосторонний обход дерева

//\*w - указатель на корень дерева.

{

if (\*w != NULL) // если узел не пустой

{

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

ObhodLeft(&((\*\*w).Left)); // рекурсивный вызов левостороннего обхода для левого поддерева

ObhodLeft(&((\*\*w).Right));// рекурсивный вызов левостороннего обхода для правого поддерева

}

}

void TREE::ObhodEnd(node\*\* w) // реализация функции концевой обход дерева

//\*w - указатель на корень дерева.

{

if (\*w != NULL)// если узел не пустой

{

ObhodEnd(&((\*\*w).Left));// рекурсивный вызов концевого обхода для левого поддерева

ObhodEnd(&((\*\*w).Right)); // рекурсивный вызов концевого обхода для правого поддерева

cout << (\*\*w).Key << " "; // Вывод ключа

}

}

void TREE::ObhodBack(node\*\* w) // реализация функции обратный обход дерева

//\*w - указатель на корень дерева.

{

if (\*w != NULL) // если узел не пустой

{

ObhodBack(&((\*\*w).Left)); // рекурсивный вызов обратного обхода для левого поддерева

cout << (\*\*w).Key << " "; // вывод ключа

ObhodBack(&((\*\*w).Right)); // рекурсивный вызов обратного обхода для правого поддерева

}

}

void TREE::CleanTree(node\*\* w) // реализация функции очистки дерева

//Очистка дерева.

//\*w - указатель на корень дерева.

{

if (\*w != NULL) // если узел не пустой

{

CleanTree(&((\*\*w).Left)); // рекурсивный вызов очистки для левого поддерева

CleanTree(&((\*\*w).Right)); // рекурсивный вызов очистки для правого поддерева

delete\* w; // удаление узла

}

}

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

//Изображение дерева \*w на экране дисплея

// (рекурсивный алгоритм).

//\*w - указатель на корень дерева.

{

int i; // объявление переменной i

if (\*w != NULL) // если узел не пустой

{

Vyvod(&((\*\*w).Right), l + 1); // рекурсивный вызов вывода для правого поддерева

for (i = 1; i <= l; i++) cout << " "; // вывод отступов

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

Vyvod(&((\*\*w).Left), l + 1); Vyvod(&((\*\*w).Left), l + 1); // рекурсивный вызов вывода для левого поддерева

}

}

int TREE::Height(node\*\* w) // реализация функции определение высоты бинарного дерева

//\*w - указатель на корень дерева.

{

int h1, h2; // инициализация переменных

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

return (-1); // завершаем работу программы с кодом -1

else // иначе

{

h1 = Height(&((\*\*w).Left)); // присваиваем h1 результат от рекурсивного определение высоты левого поддерева

h2 = Height(&((\*\*w).Right));// присваиваем h2 результат от рекурсивного определение высоты правого поддерева

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

return (1 + h1); // возвращаем высоту левого

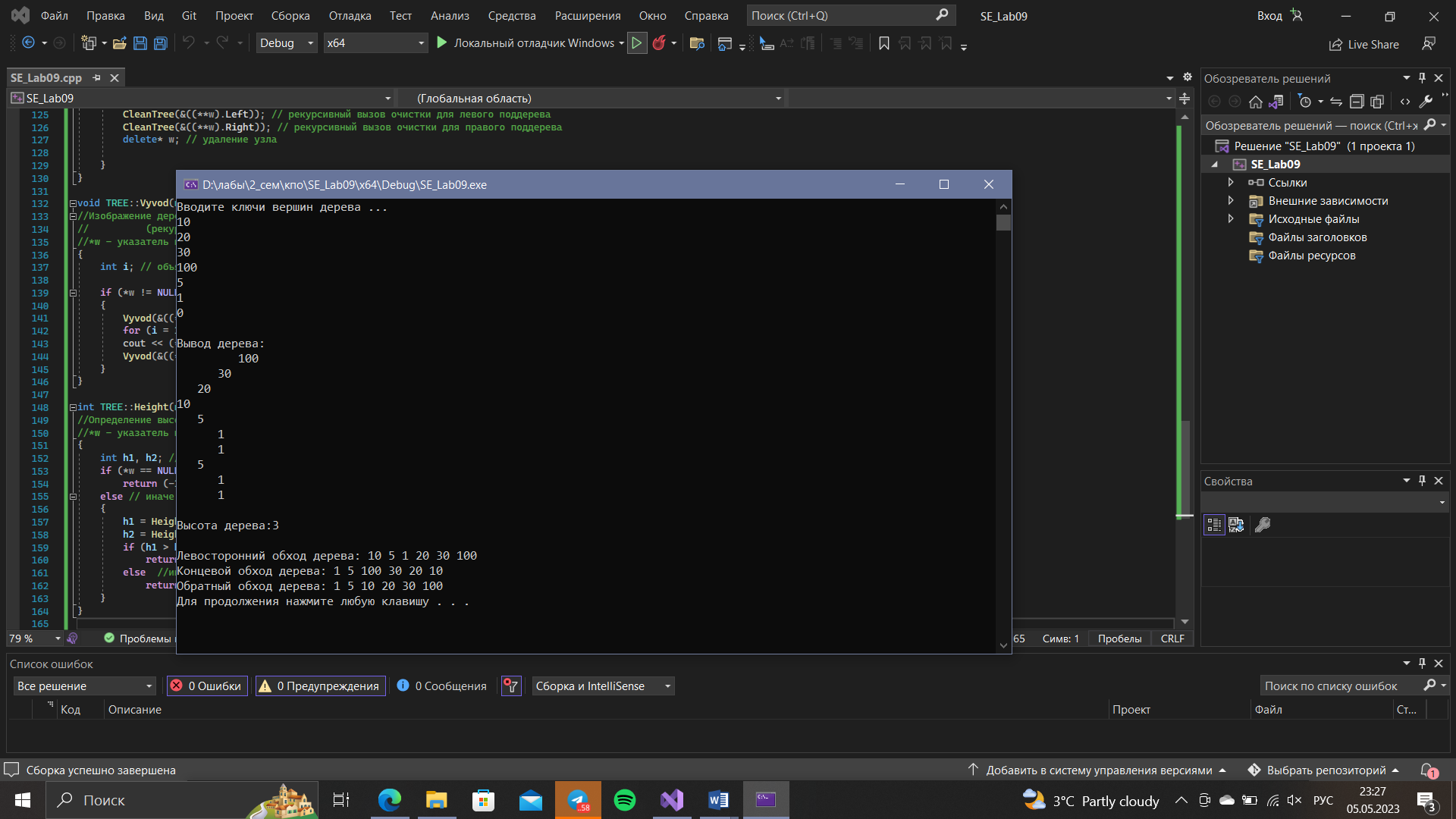
else //иначе

return (1 + h2); // правого

}

}

Результат программы:



# ПРОГРАММА №4

Не рекурсивное построение бинарного дерева и его изображение на экране дисплея.

#include <iostream>

using namespace std;

struct node {

int Key; // ключ узла дерева

int Count; // количество узлов

node\* Left; // указатель на "левого" сына.

node\* Right; // указатель на "правого" сына.

};

struct node // звено стека

{

node\* elem; // информационное поле.

int ch; // уровень вершины.

no\* sled; // указатель на вершину.

};

class TREE

{

private:// указывает, что все переменные будут доступны только внутри класса

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

void PushStack(no\*\*, node\*\*, int\*);// уомещение 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; } // инициализация корня дерева и создается нового узла node, указатель на правое поддерево равен нулю

node\* GetTreeRight() { return (\*Tree).Right; } // возвращает указатель на правого сына корня дерева.

void TreeSearch(int); // прототип функции 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\*); //прототип функции построения дерева, заданного указателем t,

//на экране дисплея (не рекурсивный алгоритм).

};

void main() // главная функция

{

setlocale(LC\_ALL, "Rus");// подключенеи русского языка

TREE A;// создание объекта А типа TREE

int el;

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

cin >> el; // Ввод элемента дерева

while (el != 0) // пока элемент не ноль

{

A.TreeSearch(el); cin >> el; // поиск вершины с информационным полем el в дереве с последующим (в случае неудачного поиска!) включением в дерево и вывод элемента

}

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

cout << "\n";

system("PAUSE"); // вызов командного интерпретатора с параметром pause

}

void TREE::TreeSearch(int el) // поиск вершины с информационным полем el в дереве с последующим (в случае неудачного поиска!) включением в дерево

{

node\* p1, \* p2;// объявление указателей

int d; // объявление переменной

p2 = Tree; p1 = (\*p2).Right; d = 1; // устанавливаем начальные значения для переменных

while (p1 != NULL && d != 0) // пока указатель p1 не станет равен NULL или пока значение d не станет равным нулю

{

p2 = p1; // присваиваем значение р1 р2

if (el < (\*p1).Key) { p1 = (\*p1).Left; d = -1; } // если el меньше ключа, то указатель p1 перенаправляется на левого потомка и значение d устанавливается в -1

else // иначе

if (el > (\*p1).Key) { p1 = (\*p1).Right; d = 1; } // если el больше ключа, то указатель p1 перенаправляется на правого потомка и значение d устанавливается в 1

else d = 0; // если el равен ключу вершины, то значение d устанавливается в 0

}

if (d == 0) (\*p1).Count = (\*p1).Count + 1; // если значение d равно нулю, то увеличивается счётчик вершины

else // иначе

{

p1 = new(node); // выделяется место для новой вершины

(\*p1).Key = el; //значение элемента присваивается ключу

(\*p1).Left = NULL; (\*p1).Right = NULL; // указатели и правого и левого потомков равны NULL

(\*p1).Count = 1; // счётчик увеличивается на один

if (d < 0) (\*p2).Left = p1; // если d меньше нуля, то новый узел становится левым потомком p2

else (\*p2).Right = p1; //иначе правым

}

}

void TREE::VyvodTree(node\* t)

//реализация функции построения дерева, заданного указателем t,

//на экране дисплея (не рекурсивный алгоритм).

{

no\* stk, \* stk1; // объявление стеков

node\* u; // объявление переменной u типа node\*

int i, n; // объявление численных переменных

stk = stk1 = NULL; n = 0; //инициализация стеков

while (t != NULL) // пока указатель на узел не NULL

{

PushStack(&stk1, &t, &n); // добавляем текущий узел во вспомогательный стек 1

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

{

if ((\*t).Left != NULL) PushStack(&stk, &((\*t).Left), &n); // если есть правое поддерево, добавляем левое поддерево в стек

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

}

else // иначе

{

if ((\*t).Left != NULL) // переходим к левому и если указатель не равен NULL

{

if (stk1 != NULL) // и если указатель на стек 1 не ноль

{

PopStack(&stk1, &u, &n); //извлекаем узел из вспомогательного стека 1

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

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

}

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

}

else // иначе если узел не имеет потомков

if (stk == NULL) t = NULL; // Если он пуст, то переходим к следующему узлу

else // иначе извлекаем узел и переходим к его правому поддереву

{

while ((\*stk).elem != (\*((\*stk1).elem)).Left) // пока указатель на элемент, на который указывает верхушка стека stk, не равен левому потомку элемента, на который указывает верхушка стека stk1.

{

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

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

cout << (\*u).Key << endl; // выводим ключи узлов

}

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

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

cout << (\*u).Key << endl; // выводим ключи узлов

PopStack(&stk, &t, &n); // удаляем элемент из вершины стека

}

}

n = n + 1; // увеличиваем счетчик глубины узла

}

VyvodStack(&stk1); // выводим оставшиеся элементы из стека 1

}

void TREE::PushStack(no\*\* stk, node\*\* el, int\* n) // реализация функции заполнение стека

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

// \*stk - указатель на стек.

{

no\* q; //объявление переменной с типом no\*

q = new(no); // выделение памяти для нового звена стека

(\*q).elem = \*el; (\*q).ch = \*n; // // сохранение переданных элементов и параметров в звене

(\*q).sled = \*stk; \*stk = q; // // добавление звена в стек

}

void TREE::PopStack(no\*\* stk, node\*\* t, int\* n) // реализация функции извлечение из стека

// \*stk - указатель на стек.

{

no\* q; //объявление переменной с типом no\*

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

{

\*t = (\*\*stk).elem; // сохраняем значение элемента из вершины стека

\*n = (\*\*stk).ch; // сохраняем значение параметра из вершины стека

q = \*stk; // сохраняем указатель на звено в вершине стека

\*stk = (\*\*stk).sled; // переустанавливаем вершину стека на следующее звено

delete q; // удаляем звено из вершины стека

}

}

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

// \*stk - указатель на стек.

{

node\* k; //объявление переменной с типом node\*

int i, n; //объевление числовых переменных

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

{

k = (\*\*stk).elem; n = (\*\*stk).ch; // присваиваем значение k вершины элемента, а n значение уровня вершин

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

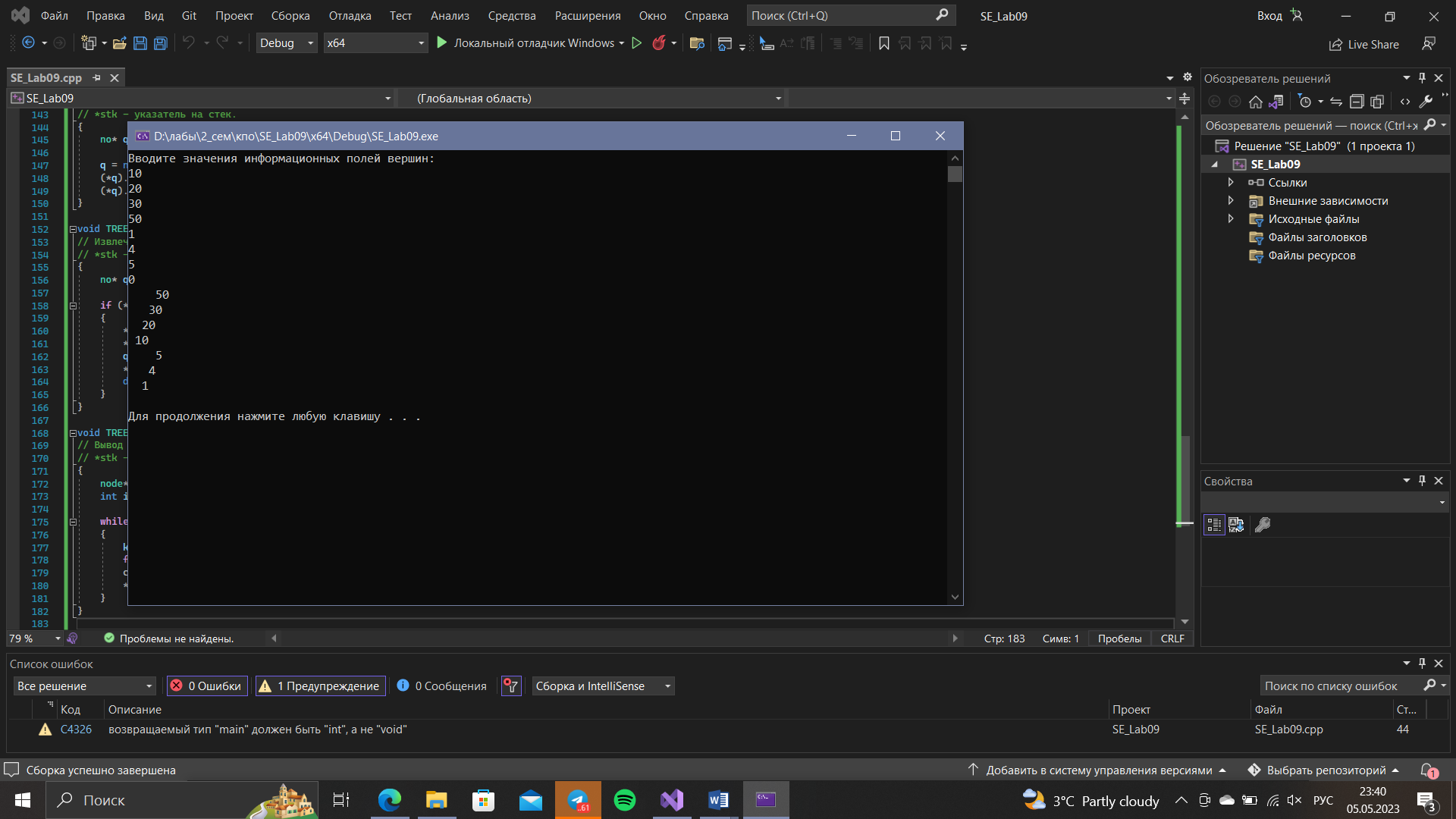
cout << (\*k).Key << endl; // вывод ключей

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

}

}

Результат программы:



# ПРОГРАММА №5

Программа, иллюстрирующая поиск заданной вершины в дереве, добавление вершины в дерево, удаление вершины из дерева.

#include <iostream>

using namespace std;

#define TRUE 1 // определение константы TRUE со значением 1

#define FALSE 0 // определение константы FALSE со значением 0

struct node {

int Key; // ключ узла дерева

int Count; // количество узлов

node\* Left; // указатель на "левого" потомка

node\* Right; // указатель на "правого" потомка

};

class TREE

{

private: // указывает, что все переменные будут доступны только внутри класса

node\* Tree;//указатель на корень дерева

node\* Res;//указатель на найденную вершину

int B; //Признак нахождения вершины в дереве

void Search(int, node\*\*); //прототип функции поиск звена x в бинарном дереве

//вершина с заданным ключом имеет две исходящие дуги

void Delete\_1(node\*\*, node\*\*); // прототип функции которая удаляет вершину с двумя исходящими дугами

public:

TREE() { Tree = NULL; } // конструктор класса TREE, который устанавливает указатель Tree в значение NULL

node\*\* GetTree() { return &Tree; } // возвращает указатель на Tree

void BuildTree();//прототип функции построение бинарного дерева.

//Вывод дерева на экран (рекурсивный алгоритм).

void Vyvod(node\*\*, int); // прототип функции которая выводит дерево на экран

//Поиск вершины в дереве (не рекурсивный алгоритм).

int Poisk(int); // прототип функции которая ищет вершину

//Поиск вершины в дереве (рекурсивный алгоритм).

node\* Poisk\_1(int, node\*\*); // прототип функции поиск вершины с помощью рекурсивного алгоритма

//Добавление вершины в дерево (не рекурсивный алгоритм).

void Addition(int); // прототип функции которая возвращает указатель на Tree

// Удаление вершины из дерева.

void Delete(node\*\*, int); // прототип функции котораяудаляет вершину

};

void main() // Главная функция

{

setlocale(LC\_ALL, "Rus"); подключение русского языка TREE A;// Создание объекта А типа TREE

int el;

A.BuildTree(); A.Vyvod(A.GetTree(), 0); // вызов построения и вывода дерева

cout << "Введите ключ вершины, которую нужно найти в дереве: "; // вывод сообщения для пользователя

cin >> el;// ввод элемента

if (A.Poisk(el)) cout << "В дереве есть такая вершина!\n"; // если вершина найдена, то выводится соответствующее сообщение

else cout << "В дереве нет такой вершины!\n"; // вывод сообщения для пользователя

cout << "Введите ключ вершины, которую нужно найти в дереве: "; // вывод сообщения для пользователя

cin >> el; // ввод элемента

if (A.Poisk\_1(el, A.GetTree()) != NULL) //если вершина найдена

cout << "В дереве есть такая вершина!\n"; // вывод сообщения для пользователя

else cout << "В дереве нет такой вершины!\n"; // вывод сообщения для пользователя

cout << "Введите ключ добавляемой вершины: "; // вывод сообщения для пользователя

cin >> el; // ввод элемента

A.Addition(el); A.Vyvod(A.GetTree(), 0); // добавляем вершину и выводим дерево на экран

cout << "Введите ключ удаляемой вершины: "; cin >> el; // вывод сообщения для пользователя

A.Delete(A.GetTree(), el); A.Vyvod(A.GetTree(), 0); // удаляем вершину и выводим дереао на экран

cout << "\n";

system("PAUSE");// вызов командного интерпретатора с параметром pause

}

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

//Tree - указатель на вершину дерева.

{

int el; // инициализируем элемент

cout << "Вводите ключи вершин дерева ...\n";// вывод сообщения для пользователя

cin >> el; // ввод элемента дерева

while (el != 0) // пока элемент не ноль

{

Search(el, &Tree); cin >> el; // поиск вершины с ключом x в дереве со вставкой

}

}

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

// реализация функциии зображение дерева w на экране дисплея

// (рекурсивный алгоритм).

//\*w - указатель на корень дерева.

{

int i; // объявление переменной i

if (\*w != NULL) // если узел не пустой

{

Vyvod(&((\*\*w).Right), l + 1); // рекурсивный вызов вывода для правого поддерева

for (i = 1; i <= l; i++) cout << " "; // вывод отступов

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

Vyvod(&((\*\*w).Left), l + 1); Vyvod(&((\*\*w).Left), l + 1); // рекурсивный вызов вывода для левого поддерева

}

}

void TREE::Search(int x, node\*\* p) // реализация функции поиск вершины с ключом x в дереве со вставкой

//\*p - указатель на вершину дерева.

{

if (\*p == NULL) // если в вершины нет

{// вершины в дереве нет; включить ее.

\*p = new(node); //

\*p = new(node); // выделяем место для нового элемента

(\*\*p).Key = x; // присваиваем значение корню дерева

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

(\*\*p).Left = NULL; // указатель на правое поддерево NULL

(\*\*p).Right = NULL; // указатель на левое поддерево NULL

}

else // иначе

if (x < (\*\*p).Key) Search(x, &((\*\*p).Left)); // если ключ искомого узла меньше ключа вершины, то производим рекурсивный поиск по левому поддереву

else // иначе

if (x > (\*\*p).Key) Search(x, &((\*\*p).Right)); // если ключ искомого узла больше ключа вершины, то производим рекурсивный поиск по правому поддереву

else // иначе

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

}

void TREE::Addition(int k)

// реализация функции поиск звена k в бинарном дереве со вставкой

// (не рекурсивный алгоритм).

// Tree - указатель на вершину дерева.

{

node\* s;// Объявление указателя на структуру

Poisk(k); // проверяет, есть ли в дереве вершина с ключом k.

if (!B) //если B будет равен 0.

{

s = new(node); // выделяется память под новую вершину

(\*s).Key = k; // присваиваем значение корню дерева

(\*s).Count = 1; // количество элементов 1

(\*s).Left = (\*s).Right = NULL; // указатель на правое и левое поддеревья NULL

if (Tree == NULL) Tree = s; //если дерево пусто, то новая вершина будет корневой.

else // иначе

if (k < (\*Res).Key) (\*Res).Left = s; // если ключ меньше клуча вершины, то он становиться левым потомком

else (\*Res).Right = s; // иначе правым

}

else (\*Res).Count += 1; // если вершина с ключом k уже существует, то увеличивается счётчик вхождений переменной

}

int TREE::Poisk(int k) // реализация функции поиск вершины с ключом k в дереве

{

node\* p, \* q = NULL; // объявление указателей на узлы

B = FALSE; // инициализация флага нахождения элемента в дереве

p = Tree; // начало поиска с корня дерева

if (Tree != NULL) // если дерево не пустое

do // то

{

q = p; // запоминаем родительский узел перед переходом к следующему узлу

if ((\*p).Key == k) B = TRUE; // если нашли элемент в дереве, то значение флага равно правде

else // иначе

{

q = p; // запоминаем родительский узел перед переходом к следующему узлу

if (k < (\*p).Key) p = (\*p).Left; // переходим в левого сына

else p = (\*p).Right; // переходим в правого потомка

}

} while (!B && p != NULL); // выполняем поиск, пока элемент не найден или не прошли всё дерево

Res = q; // запоминаем узел, в котором закончился поиск

return B; // возвращаем флаг нахождения элемента в дереве

}

node\* TREE::Poisk\_1(int k, node\*\* p) // реализация функции поиск вершины с ключом k в дереве

{

if (\*p == NULL) return (NULL); // если дерево пустое, вернуть NULL

else// иначе

if ((\*\*p).Key == k) return (\*p); // // Если ключ корневой вершины совпадает с k, вернуть указатель на корень

else // иначе

if (k < (\*\*p).Key) return Poisk\_1(k, &((\*\*p).Left)); // // Если k меньше ключа корневой вершины, искать в левом поддереве

else return Poisk\_1(k, &((\*\*p).Right)); // иначе искать в правом поддереве

}

void TREE::Delete(node\*\* p, int k) // реализация функции, которая удаляет вершину

{

node\* q; // объявление указателя на вершину

if (\*p == NULL) cout << "Вершина с заданным ключом не найдена!\n"; // если дерево пустое, то вывод сообщение

else // иначе

if (k < (\*\*p).Key) Delete(&((\*\*p).Left), k); // если ключ удаляемой вершины меньше ключа текущей вершины, то вызывается рекурсивно функция удаления вершины для левого поддерева текущей вершины

else // иначе

if (k > (\*\*p).Key) Delete(&((\*\*p).Right), k); // если ключ удаляемой вершины больше ключа текущей вершины, то вызывается рекурсивно функция удаления вершин для правого поддерева текущей вершины

else // иначе

{

q = \*p; // сохраняем указатель на удаляемую вершину в переменную q

if ((\*q).Right == NULL) { \*p = (\*q).Left; delete q; } // если удаляемая вершина не имеет правого поддерева, то на место удаляемой вершины ставится ее левый потомок и вершина удаляется

else // иначе

if ((\*q).Left == NULL) { \*p = (\*q).Right; delete q; } // если удаляемая вершина не имеет левого поддерева, то на место удаляемой вершины ставится ее правый потомок и вершина удаляется.

else Delete\_1(&((\*q).Left), &q); // иначе вызывается функция Delete\_1

}

}

void TREE::Delete\_1(node\*\* r, node\*\* q) // реализация функции, которая удаляет вершину с двумя исходящими дугами

{

node\* s; // объявление указателя на вершин

if ((\*\*r).Right == NULL) // если текущая вершина не имеет правого потомка

{

(\*\*q).Key = (\*\*r).Key; (\*\*q).Count = (\*\*r).Count; // заменяет соответствующие значения удаляемой вершины на значения текущей

\*q = \*r; //изменяем указатель на текущую вершину

s = \*r; \*r = (\*\*r).Left; delete s; // изменяем указатель на текущую вершину на указатель на её левого потомка, а затем сохраняет адрес текущей вершины в s и удаляем вершину

}

else Delete\_1(&((\*\*r).Right), q); // иначе рекурсивно вызываем функцию для правого потомка

}

Результат программы:

