Дисциплина **Структуры и алгоритмы обработки данных**

Лабораторная работа №2. **Бинарные деревья**

Студент **Бекошин Никита Вячеславович**

Группа **1ПИб-02-3оп-23-1пг**

**Отчет**

**Вариант задания**

**Вариант 20.**

Написать функцию, которая находит все элементы дерева, встречающиеся ровно один раз.

Требования к решению решаемой задачи:

Написать функцию формирования бинарного дерева, состоящего из целых чисел.

Для представления дерева использовать динамические структуры данных.

Количество элементов дерева, а также его вид задаются случайным образом.

Произвести вывод элементов дерева тремя видами обхода.

Используя информацию о выведенном дереве изобразить структуру одного из построенных деревьев в отчете. Выполнить по вариантам следующие задания:

Листинг с подробными комментариями

#include <iostream>

#include <ctime>

/\*2-я лаба - функцию осуществления дерева придется перестроить\*/

struct Node //структура узла

{

int data; //значение элемента дерева

Node \*left, \*right; // Он означает, что переменная left имеет не само значение типа Node, а указывает на адрес, где находится другой объект типа Node.

};

/\*Брать ссылку на корень дерева и бежать туда, куда направляем - умное решение для обработки пар\*/

/\*функция printtree однозначно рекурсивная\*/

void add(Node \*&node, int value) //\*& только при записи, при чтении только \*

//Первый параметр - Обратиться по адресу к структуре Node (node),

//которая является ссылочной для указателя root. При этом не создается

//копии указателя, а значит это помогает избавиться от ситуации двойного

//указателя, происходит работа со структурой Node напрямую.

{ //& - апперсанд означает, что функция add работает со структурой Node напрямую, не создавая копию этой структуры.

if (node == NULL) //если данного узла еще не существует, то

//есть он имеет значение 0

{ // имеется в виду, если данного

// корня еще не существует, или как если у родителя есть один

//потомок, но нет второго, и на месте второго будет значение NULL

node = new Node;

//Выделить для указателя память из кучи,

// чтобы она могла хранить данные каждого элемента дерева

node->data = value; //в первую очередь обращать внимание

//на эту команду "node->data = value;

//node больше не NULL, значит после

//после выхода из тела условия "node==NULL" начнет проверяться

//условие else ("node!=NULL"), которое выполнится

//значение корня, под которого выделена динамическая

//память. значение data изменяется через указатель node

/\*Создание потомков для текущего корня node при каждом

добавлении новой ветки, от этой ветки отходят два потомка,

но в них пока ничего нет, значит они имеют значение

пустого указателя\*/

node->left = NULL; //Указатель устанавливает значение

//другого указателя (\*left, являющегося полем структуры

//Node и указывающего на ячейку памяти левого потомка) в NULL

node->right = NULL; //аналогично для правого потомка

}

/\*Важно понимать, что в противном случае (невыполнения условия

node==NULL), так как при каждом вызове add каждый раз происходит

проход, начиная с главного корня дерева при условии, если он

содержит данные, которые добавились при вызове функции add

в первый раз\*/

else //реализация добавления элементов рекурсией предусматривает,

{

/\*Алгоритм заполнения дерева случайной структуры: если текущее значение элемента дерева больше, чем чем значение данных родителя,

то происходит \*/

/\*Если ни одно из нижеперечисленных условий не выполнится, то это говорит

о том, что пока в дерево добавлен один элемент, главный корень дерева\*/

/\*Важно помнить, что подаваемое в функции add значение value генерируется новым

случайным числом за каждый проход по циклу, в котором

вызывается эта функция (add). В нижеперечисленных условиях нет третьего условия,

когда value == node->data.

Оно не нужно, так как "srand(time(0));" позволяет генерировать уникальные значения

(повторения случайных чисел невозможно)\*/

if (value < node->data) // Если случайное число value

//меньше значения поля data ), на который указывает node, то:

add(node->left, value); /\*Вызов функции add - переход

// на левый потомок с целью заполнения его хранимых данных и

инициализации его потомков\*/

if (value > node->data) //если value больше data, то :

add(node->right, value); //иначе вызов функции add - переход на правый потомок

}

}

/\*Рекурсивная функция проверки на уникальность содержимого элементов\*/

bool Proverka(Node \*node, int value)

{

if (node == NULL) //Если после возвращения функции Proverka

//при переходе на правый или левый элемент, корень имеет

//node указывывает на NULL, значит данного корня нет, нет

//нет элемента и его содержимого, указатель считается нулевым.

//Это условие нужно для того, чтобы понять, что раз дошли

//до одного из листов дерева, то на данной ветви повторяющихся

//как минимум один раз элементов нет

{

return true; //вернет true - узел уникален

} //сравнивается равенство содержимого текущего элемента,

//которого мы рассматриваем и пытаемся понять, уникален ли он,

//значением value, то есть его потомка (левого или правого)

if (node->data == value) //если содержимое текущего элемента

//дерева равно текущему элементу дерева, е

{

return false; // Значение уже встретилось

}

//рекурсивно возвращаются две функции, но для левого и правого

//потомков

return Proverka(node->left, value) && Proverka(node->right, value);

// переход на левое и правое деревья с целью дальнейшего поиска

// уникальных узлов

}

int cnt1 = 0; //инициализация счетчика как ноль (начальное значение)

void FindSingleElement(Node \*node) //в качестве параметра подаётся

//указатель на элементы структуры Node

{

if (node != NULL) //проверка на то, что дерево состоит

//как минимум из одного элемента

{

/\*В теле цикла вызывается функция "Proverka",

где в качестве первого параметра подается указатель

на указатель содержимого левого потомка, в качестве

второго параметра - содержимое родителя для сравнения,

чтобы проверить, имеет ли родительский потомок такое же

значение, хранящееся в содержимом, как и у его предка.

Вызывается эта функция дважды: для левого и правого потомков

текущего корня-родителя.

Если в обоих случаях функции возвращают истину, то

элемент дерева содержит уникальный элемент, протсходит его

вывод и подсчет количества таких элементов\*/

//если проверка на укальность наконец произошла в левой

//и правой ветвях дерева (не осталось сомнений, что

//для найденных в этой ветвях больше нет повторяющихся элементов),

//выводится данный уникальный элемент, равных которому нет

//в дальнейших его потомках

if (Proverka(node->left, node->data) && Proverka(node->right, node->data))

{

std::cout << node->data << " "; //вывод содержимого

//элемента дерева

cnt1++; //подсчет количества уникальных элементов дерева

}

/\*Вызываются функции с целью перейти на левые и правые

потомки с целью найти все больше уникальных элементов

в рассматриваемом дереве\*/

FindSingleElement(node->left);

FindSingleElement(node->right);

//Так на каждом дереве 1 или 2, и на каждом таком потомке

//по 1-2 подпотомка, то рекурсивный вызов функций

//"FindSingleElement" нужен для того, чтобы зафиксировать

//ветвь, на которой будет искаться уникальные элементы

}

}

// переделать рекурсивную функцию, чтобы она еще и считала количество элементов

void print1(Node \*node)

{ // прямой код

if (node != NULL)

{

std::cout << node->data << " ";

/\*std::cout<<node->data<<" "; // Перенести для симетричного кода на 28 и 29\*/

print1(node->left); // перешли на левого потомка

print1(node->right); // перешли на правого потомка

}

}

void print2(Node \*node)

{ // обратный обход

if (node != NULL)

{

print2(node->left);

print2(node->right);

std::cout << node->data << " ";

}

}

void print3(Node \*node) // симметричный код

{

if (node != NULL)

{

print3(node->left);

std::cout << node->data << " ";

print3(node->right);

}

}

/\*int cnt = 0;

void SingleElement(Node \*node)

{ // прямой код

if (node != NULL)

{

std::cout << node->data << " ";

SingleElement(node->left); // перешли на левого потомка

SingleElement(node->right); // перешли на правого потомка

}

}

\*/

void main()

{

setlocale(LC\_ALL, "ru\_RU.UTF-8");

srand(time(0));

int d;

std::cout << "Введите диапазон генерации случайного числа";

std::cout << " (например,число 60, значит будет сгенерировано одно";

std::cout << " случайное число от 0 до 59, оно будет количеством элементов ";

std::cin >> d;

std::cout << "от нуля до ";

std::cout << d << " не включительно" << std::endl;

/\*изначально дерева никакого не было. Инициализация корня дерева как NULL\*/

Node \*root = NULL;

int cnt = rand() % d; //генерация случайного числа в диапазоне d

std::cout << std::endl;

std::cout << "Сгенерированное количество узлов " << cnt<< std::endl;

// суть такова, что мы создаем переменную типа указатель на node. Он

// каждый раз за уменьшение cnt на 1 меняется, например, был root, стал root->left

for (int i = 0; i < cnt; i++) // цикл для заполнения

// дерева случайным образом

{

int value = rand() % 100; //за каждое прохождение цикла

//генерируется число, которое поместится в значение элемента дерева

add(root, value); //вызов функции добавления элемента в дерево

/\*В качестве первого параметра подаётся root: рекурсивно

происходит построение случайным образом дерева с

элементами value с разбросом от 0 до 99

При первом вызове в цикле этой функции, так как указатель root

указывает на NULL, то для root выделится память в куче. А после

произойдет инициализация потомков как NULL

При втором прохождении по циклу так как функция add вызовется,

то root уже не будет указывать на NULL,

\*/

}

/\*Выводы деревьев разными способами\*/

std::cout<<"Прямой обход дерева: ";

print1(root);

std::cout << std::endl;

std::cout<<"Обратный обход дерева: ";

print2(root);

std::cout << std::endl;

std::cout<<"Симметричный обход дерева: ";

print3(root);

std::cout << std::endl;

std::cout << "Элементы, которые встретились только 1 раз: ";

FindSingleElement(root);

std::cout << std::endl;

std::cout << "Количество уникальных элементов дерева: " << cnt1 << std::endl;

}

Скриншоты кода

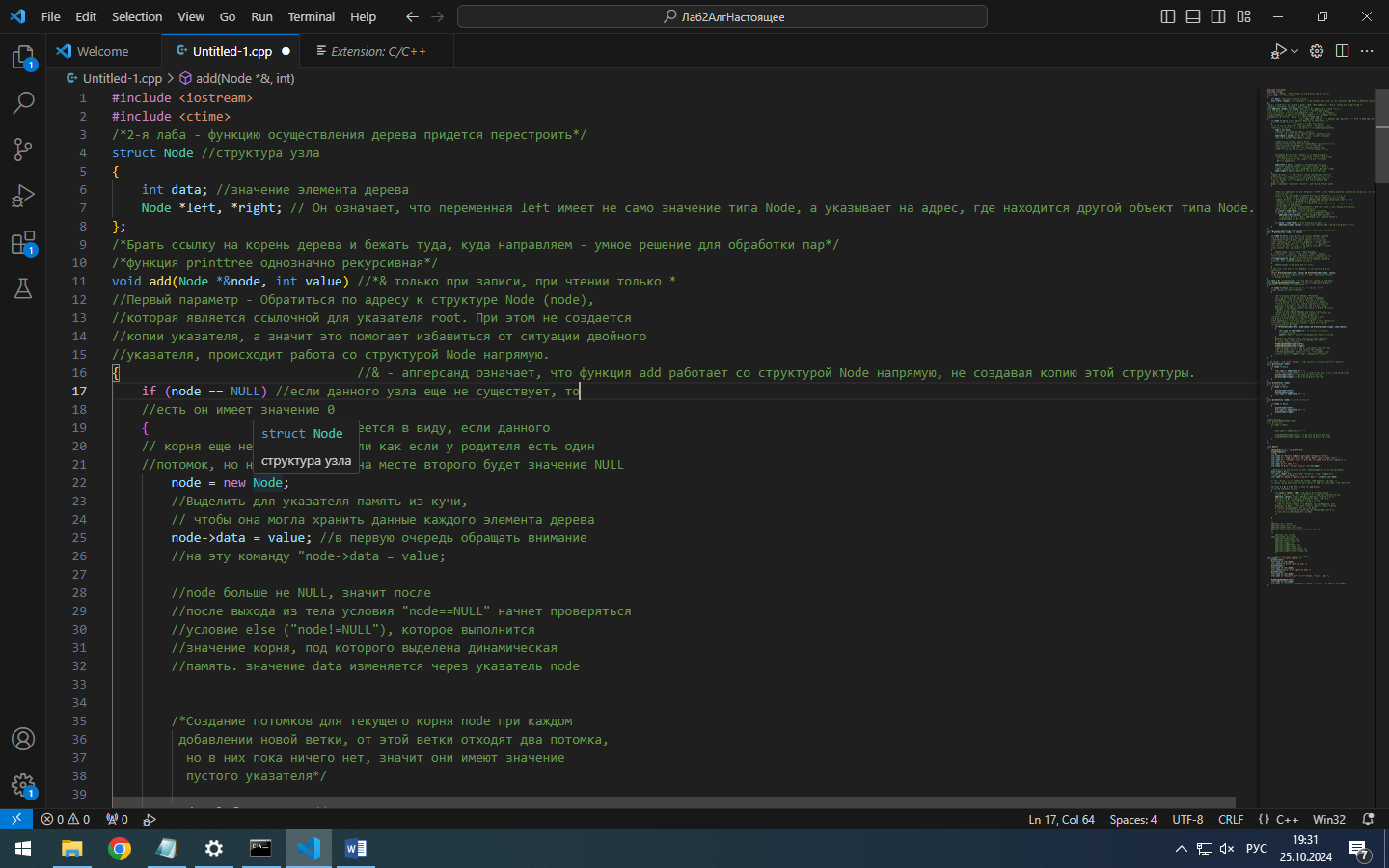


Рис. 1

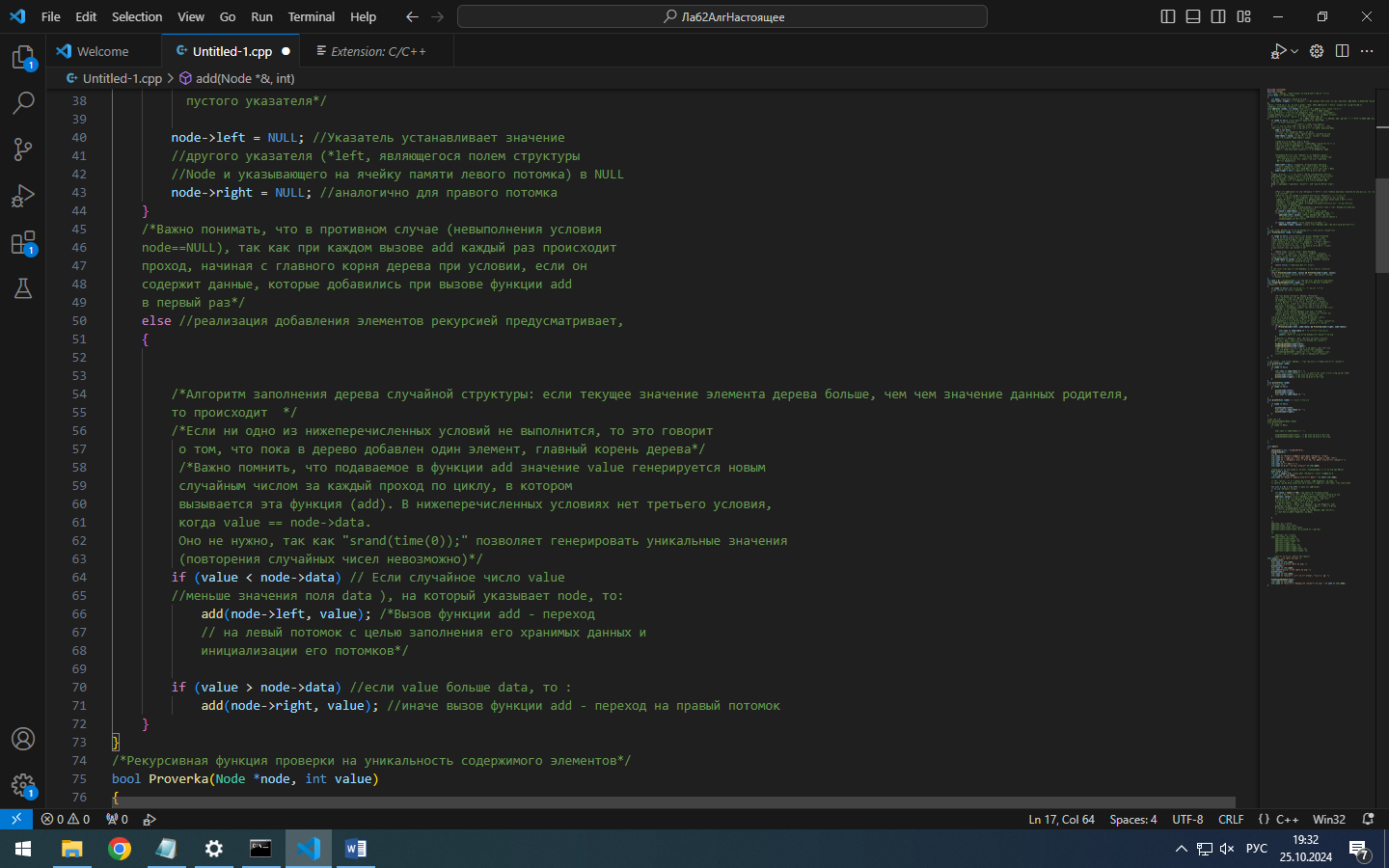
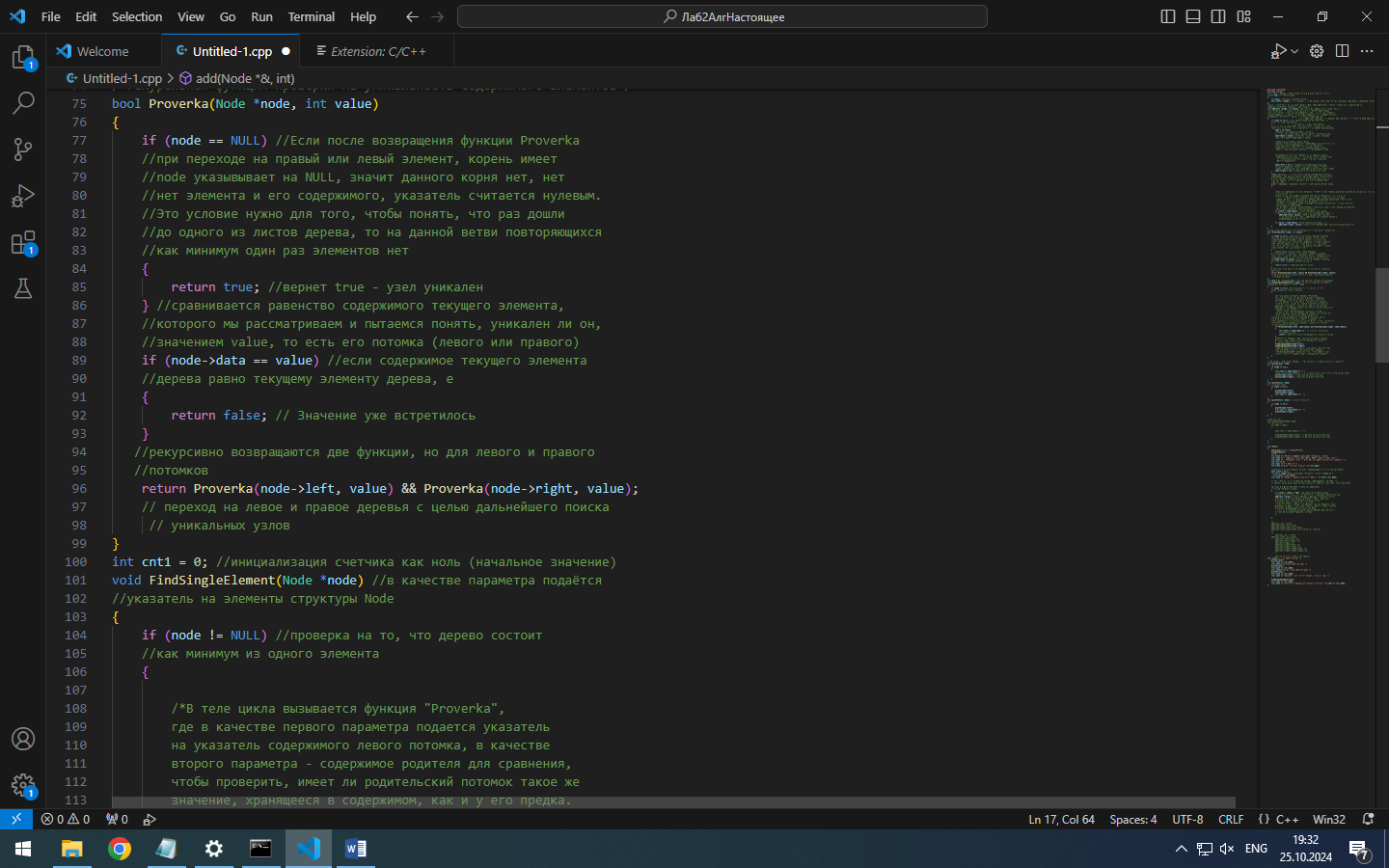
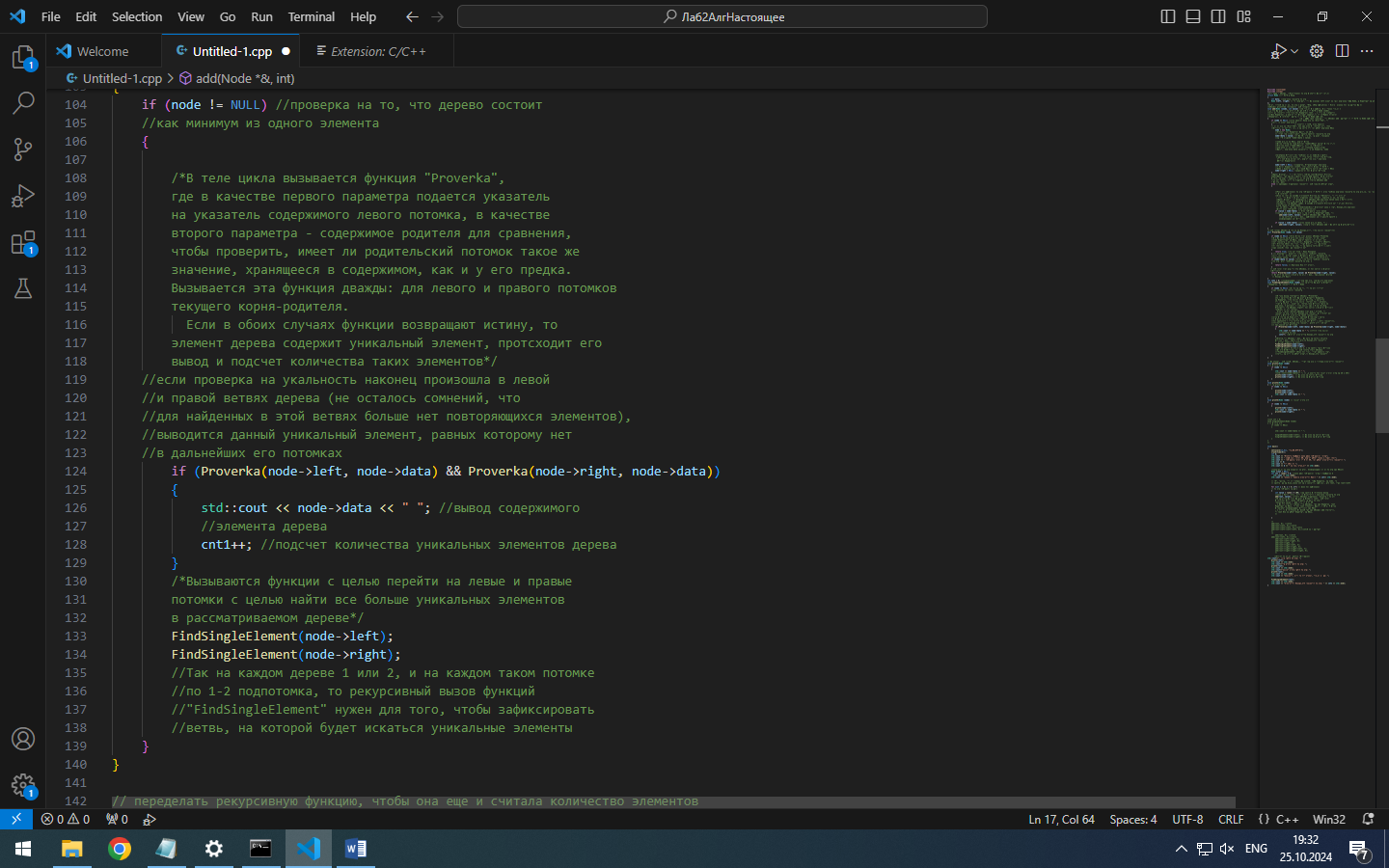


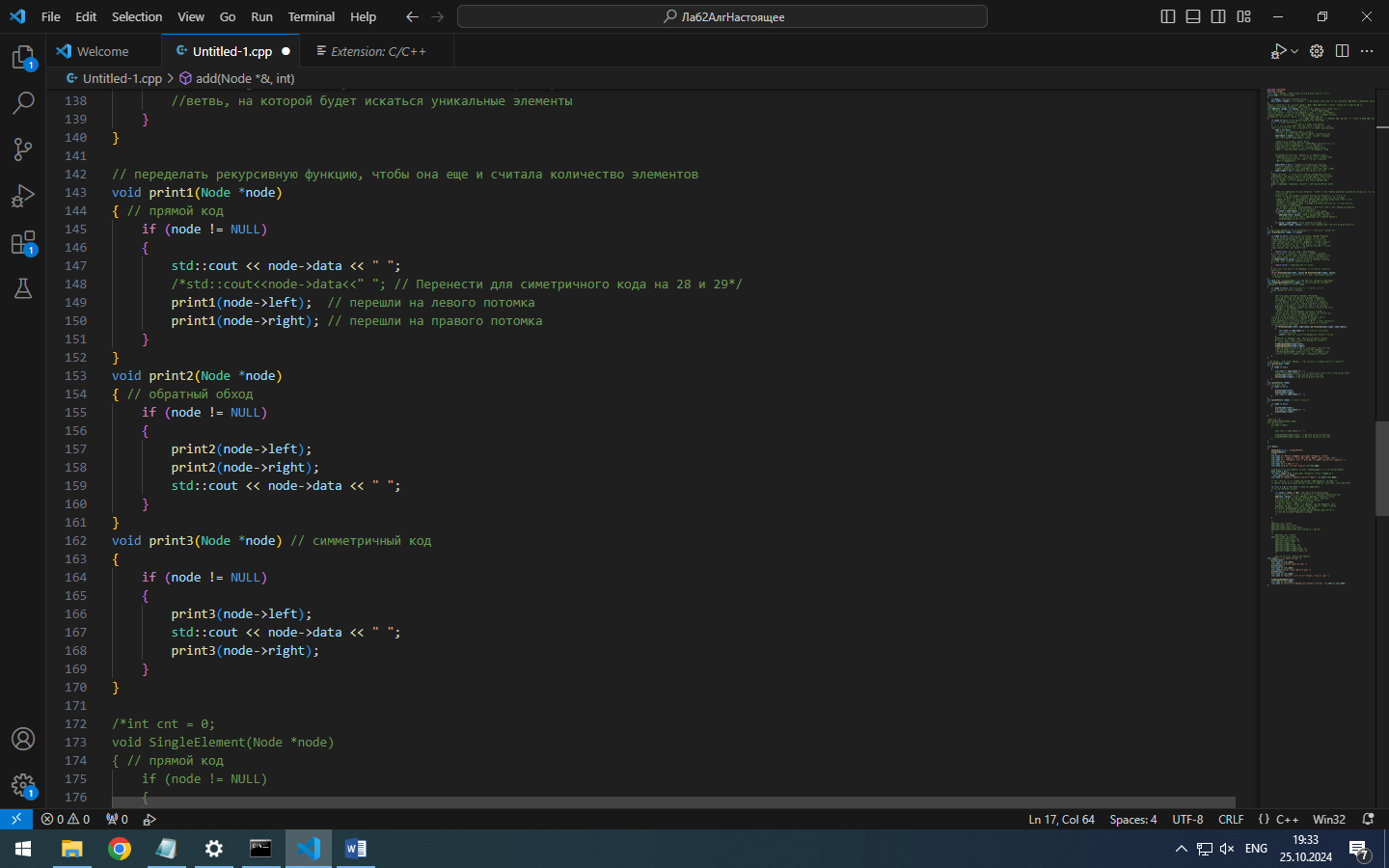
Рис. 2



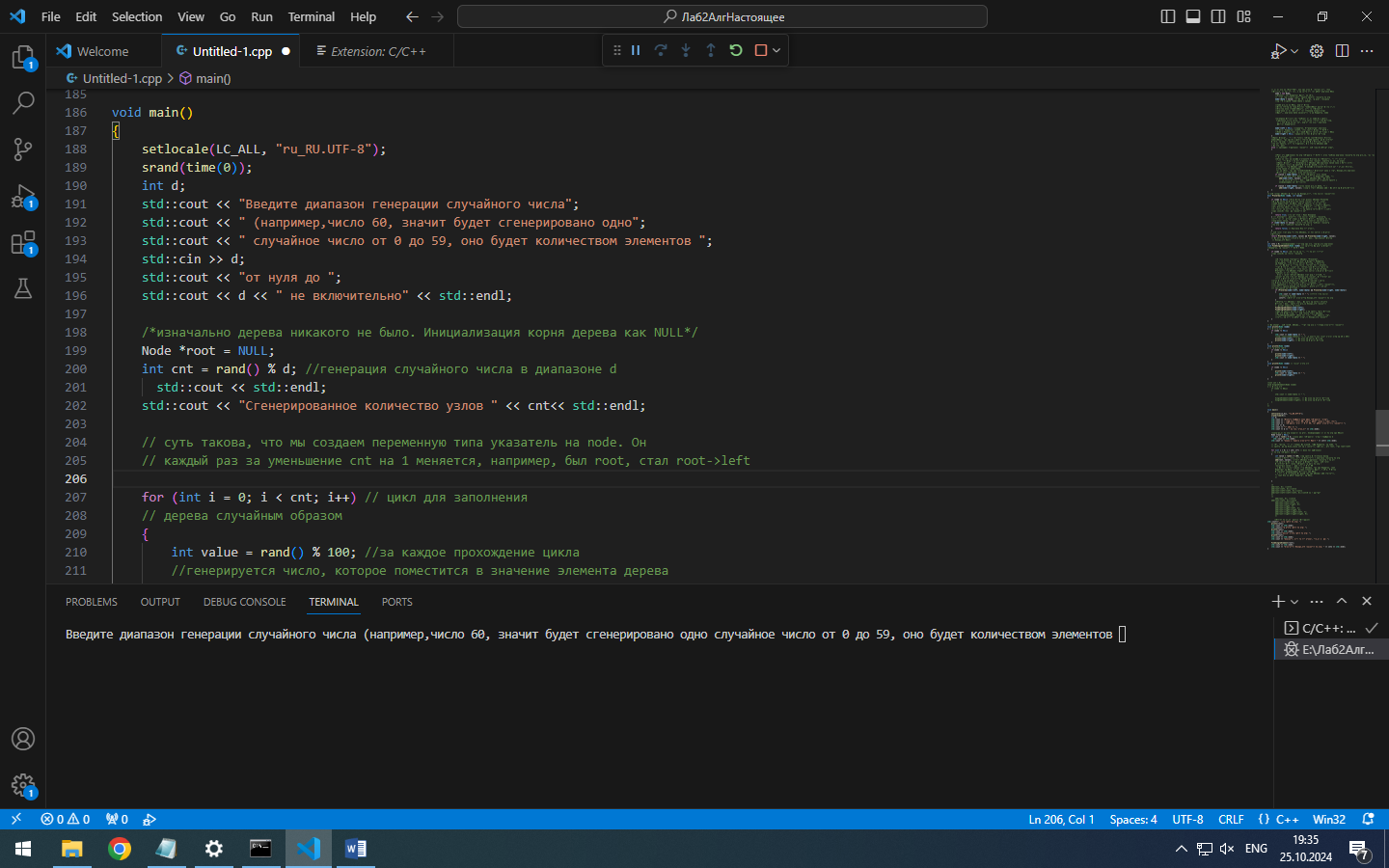
**Рис. 3**



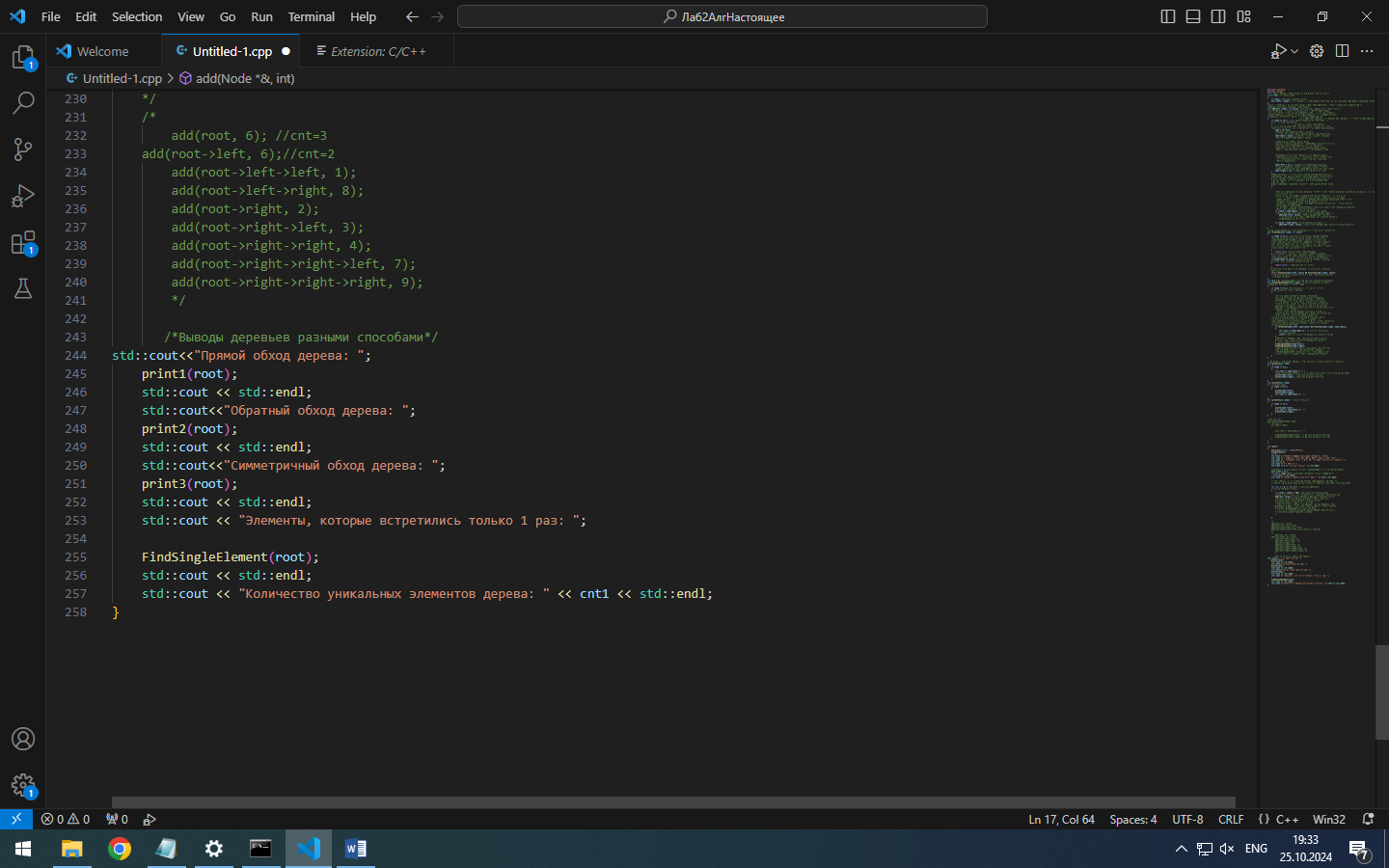
**Рис. 4**



**Рис. 5**



**Рис. 6**



**Рис. 7**

Скриншоты результатов работы программы

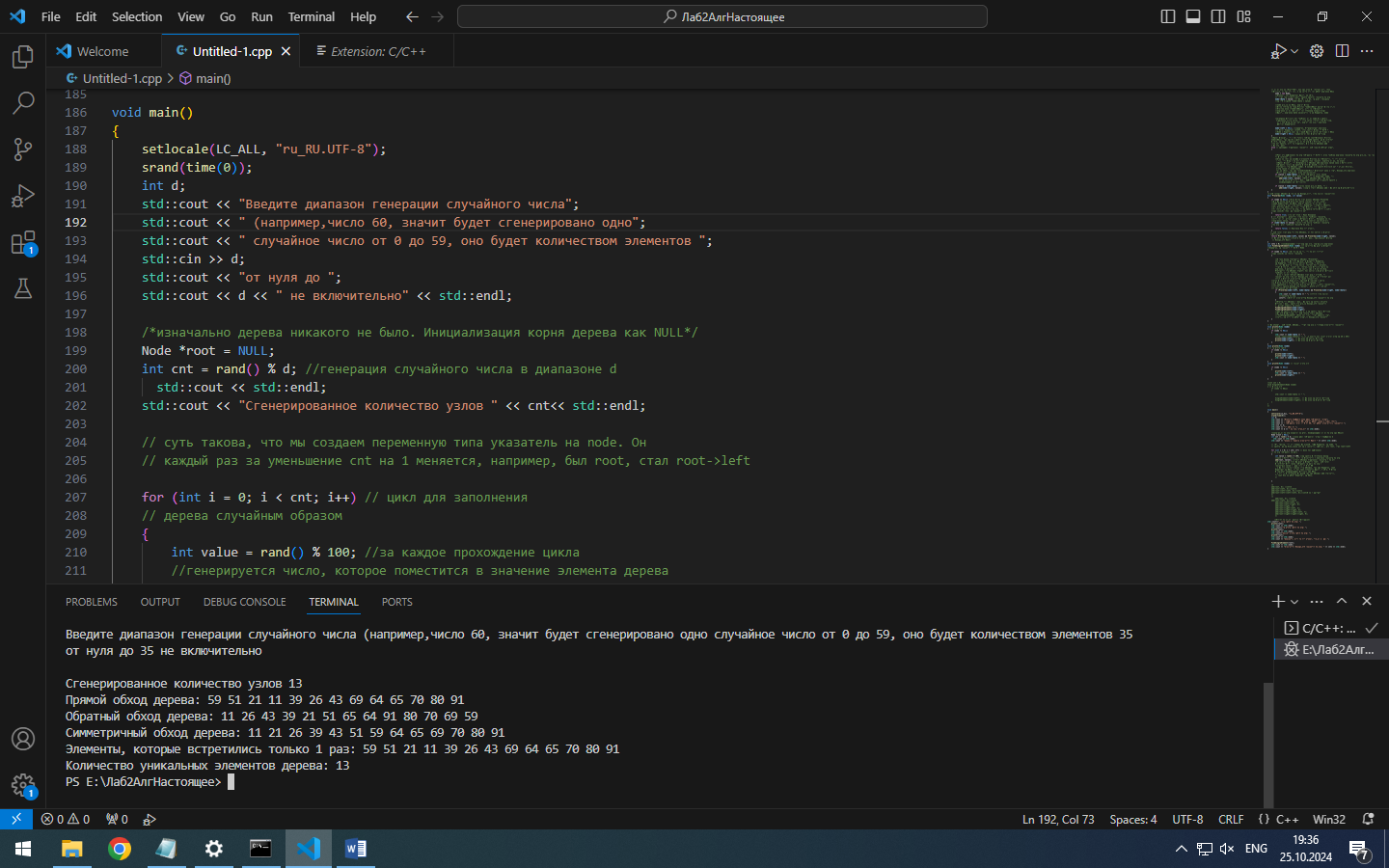


Рис. 8

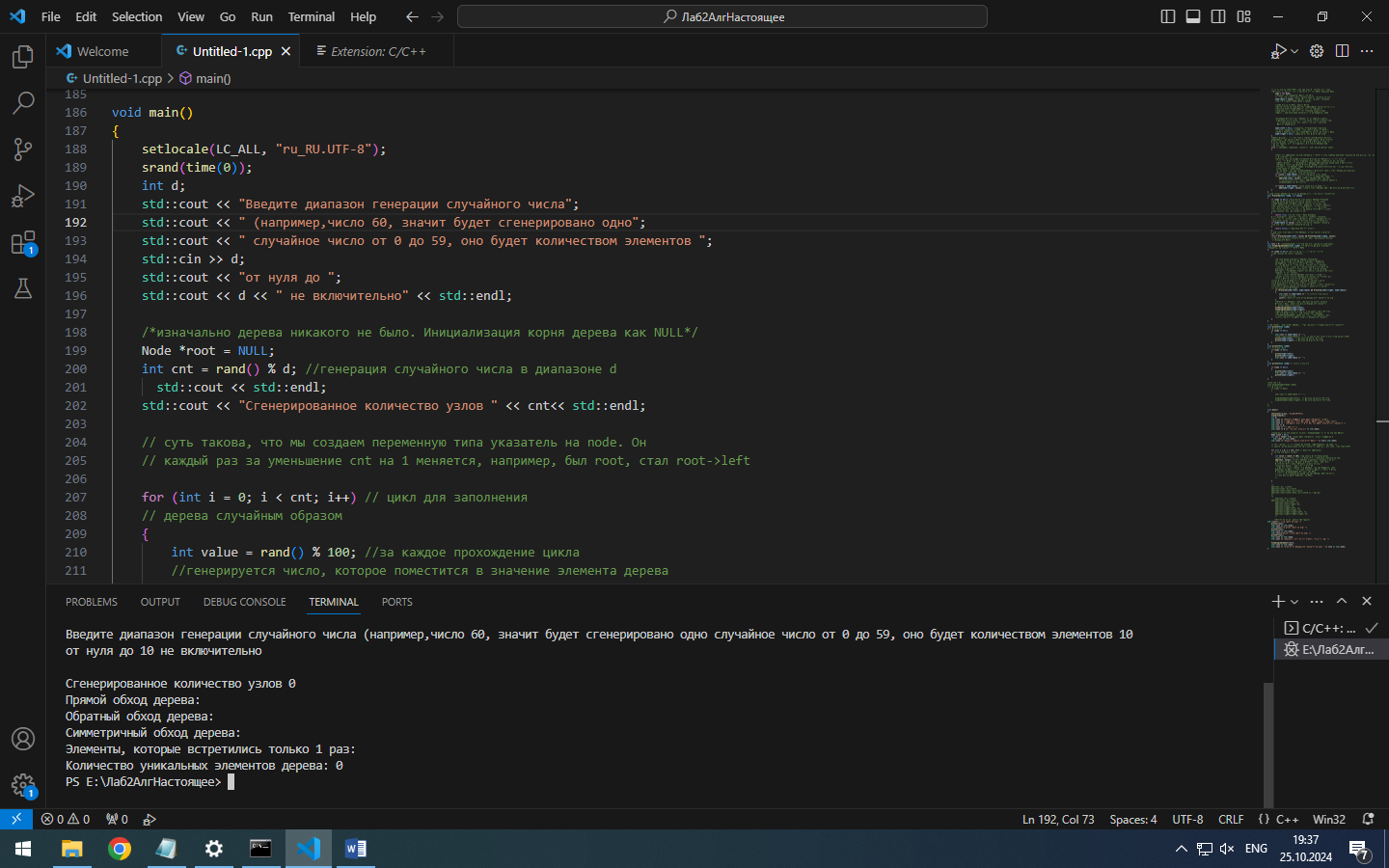


Рис. 9

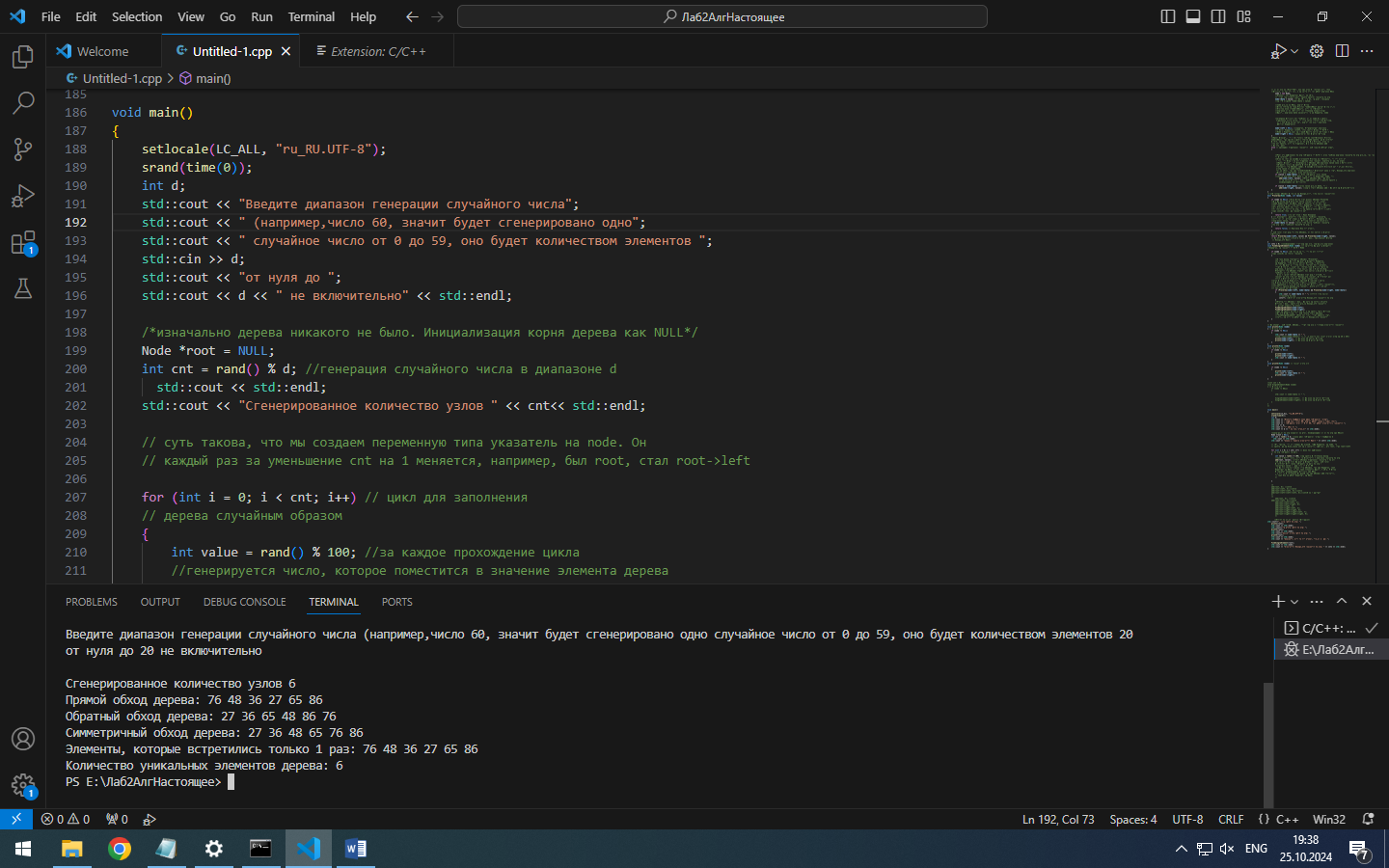


Рис. 10

Дерево к рис. 10

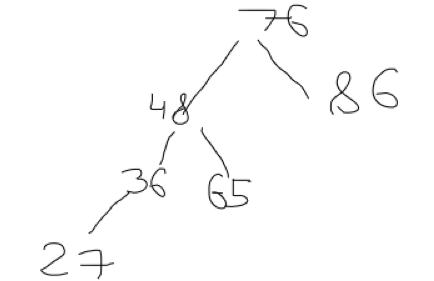


Рис.11