Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего образования

«Российский химико-технологический университет имени Д.И. Менделеева»

Кафедра информационных компьютерных технологий

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ № 3

Вариант 3

Выполнил студент группы КС-30 Ноль Эльвира Гарриевна

Ссылка на репозиторий: (https://github.com/Elviranng/Nol-Elvira-KS-30/tree/master/lab3.sem2)

Приняли: Пысин Максим Дмитриевич

Краснов Дмитрий Олегович

Дата сдачи: (05.06.21)

Оглавление

[Описание задачи. 2](#_Toc69330426)

[Описание модели. 3](#_Toc69330427)

[Выполнение задачи. 4](#_Toc69330428)

[Заключение. 13](#_Toc69330429)

# Описание задачи.

Необходимо изучить и реализовать структуру AVL дерево.

AVL дерево: Класс, операцию вставки и удаления элемента с последующей балансировкой(операция удаления и вставки вызывает семеричные операции балансировки, поэтому в отличии от красно черного дерево реализовав балансировку для вставки, ее можно применить и для удаления) и операцию поиска.

Для данной структуры необходимо провести тестирование сложности операций вставки и поиска.

Для этого использовать серию тестов:

* Под отдельным тестом понимается проведение 30 запусков тестирования для массива состоящего из N эелементов, где N меняется от теста к тесту на 50000, первый тест при это проводиться при N=50000, последний N=500000
* В каждом тесте выполнять 20 запусков генерирования и заполнения массива.
* Массив заполнять числами с плавающей запятой двойной точности сгенерированных в диапазоне от -1 до 1.
* 5 из 20 запусков выполнять для предварительно отсортированного массива(сортировку можно взять из стандартной библиотеки), и выделять их отдельно при анализе результатов.
* Для каждого запуска проводить 100 операций поиска, замеряя время которое тратиться на эту операцию, по итогу получая среднее значение времени поиска для каждого из запусков.(искать случайно сгенерированное число)
* Для каждого запуска проводить 100 операций вставки в текущее дерево, замеряя время которое тратиться на эту операцию(замерять нужно именно отдельную от первичного заполнения операцию вставки), по итогу должно получиться среднее время вставки для каждого запуска(вставлять случайно сгенерированное число)
* Аналогичное тестирование можно провести для операции удаления, но это уже исключительно дополнительное задание.

По результатам проведенных тестов необходимо построить графики зависимости затраченного на операцию(вставка, поиск) времени от количества элементов сгенерированного дерева для лучшего, худшего и среднего случаев.  
Асимптотическая сложность операций равно O(h), где h это высота дерева, так:

* для АВЛ h ~ log(N), то и асимптотическая сложность будет O(c\*log(N)) Необходимо взять график худшего случая и построить график O(c \* f(N)), подобрав такую константу c, что бы полученный график был ненамного выше графика худшего случая начиная с первой рассматриваемой точки в N=50000

# Описание модели.

АВЛ-дерево — это прежде всего двоичное дерево поиска, ключи которого удовлетворяют стандартному свойству: ключ любого узла дерева не меньше любого ключа в левом поддереве данного узла и не больше любого ключа в правом поддереве этого узла. Это значит, что для поиска нужного ключа в АВЛ-дереве можно использовать стандартный алгоритм. Для простоты дальнейшего изложения будем считать, что все ключи в дереве целочисленны и не повторяются.  
  
 Особенностью АВЛ-дерева является то, что оно является сбалансированным в следующем смысле: для любого узла дерева высота его правого поддерева отличается от высоты левого поддерева не более чем на единицу. Доказано, что этого свойства достаточно для того, чтобы высота дерева логарифмически зависела от числа его узлов: высота h АВЛ-дерева с n ключами лежит в диапазоне от log2(n + 1) до 1.44 log2(n + 2) − 0.328. А так как основные операции над двоичными деревьями поиска (поиск, вставка и удаление узлов) линейно зависят от его высоты, то получаем *гарантированную* логарифмическую зависимость времени работы этих алгоритмов от числа ключей, хранимых в дереве. Напомним, что рандомизированные деревья поиска обеспечивают сбалансированность только в вероятностном смысле: вероятность получения сильно несбалансированного дерева при больших n хотя и является пренебрежимо малой, но остается *не равной нулю*.

Структура узлов

Будем представлять узлы АВЛ-дерева следующей структурой

struct node // структура для представления узлов дерева

{

int key;

unsigned char height;

node\* left;

node\* right;

node(int k) { key = k; left = right = 0; height = 1; }

};

Поле key хранит ключ узла, поле height — высоту поддерева с корнем в данном узле, поля left и right — указатели на левое и правое поддеревья. Простой конструктор создает новый узел (высоты 1) с заданным ключом k.

Традиционно, узлы АВЛ-дерева хранят не высоту, а разницу высот правого и левого поддеревьев (так называемый balance factor), которая может принимать только три значения -1, 0 и 1. Однако, заметим, что эта разница все равно хранится в переменной, размер которой равен минимум одному байту (если не придумывать каких-то хитрых схем «эффективной» упаковки таких величин). Вспомним, что высота h < 1.44 log2(n + 2), это значит, например, что при n=109 (один миллиард ключей, больше 10 гигабайт памяти под хранение узлов) высота дерева не превысит величины h=44, которая с успехом помещается в тот же один байт памяти, что и balance factor. Таким образом, хранение высот с одной стороны не увеличивает объем памяти, отводимой под узлы дерева, а с другой стороны существенно упрощает реализацию некоторых операций.

Определим три вспомогательные функции, связанные с высотой. Первая является оберткой для поля height, она может работать и с нулевыми указателями (с пустыми деревьями):

unsigned char height(node\* p)

{

return p?p->height:0;

}

Вторая вычисляет balance factor заданного узла (и работает только с ненулевыми указателями):

int bfactor(node\* p)

{

return height(p->right)-height(p->left);

}

Третья функция восстанавливает корректное значение поля height заданного узла (при условии, что значения этого поля в правом и левом дочерних узлах являются корректными):

void fixheight(node\* p)

{

unsigned char hl = height(p->left);

unsigned char hr = height(p->right);

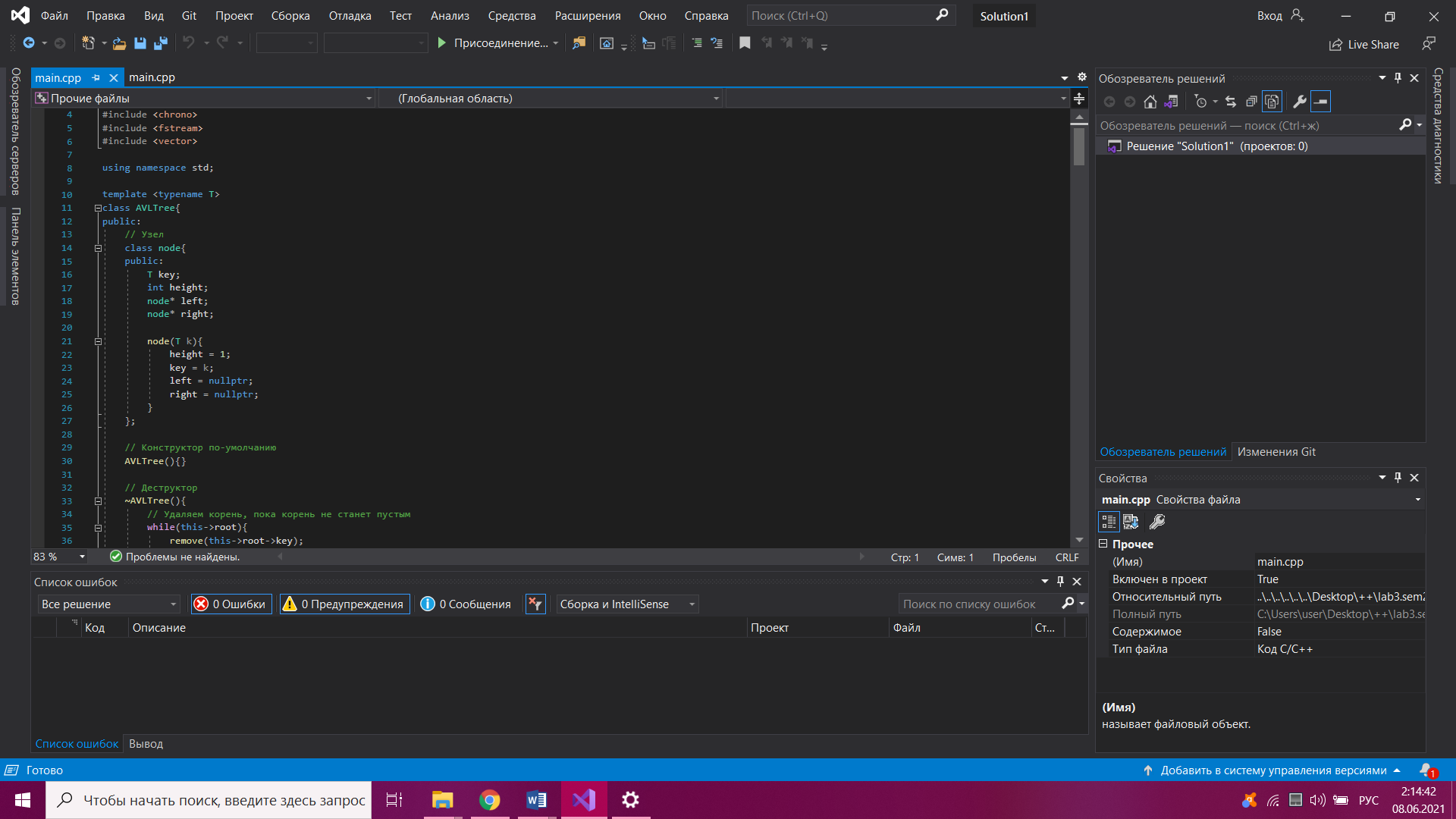
p->height = (hl>hr?hl:hr)+1;

}

Заметим, что все три функции являются нерекурсивными, т.е. время их работы есть величина О(1).

# Выполнение задачи.

* Здесь создан класс для работы с разными типами данных:

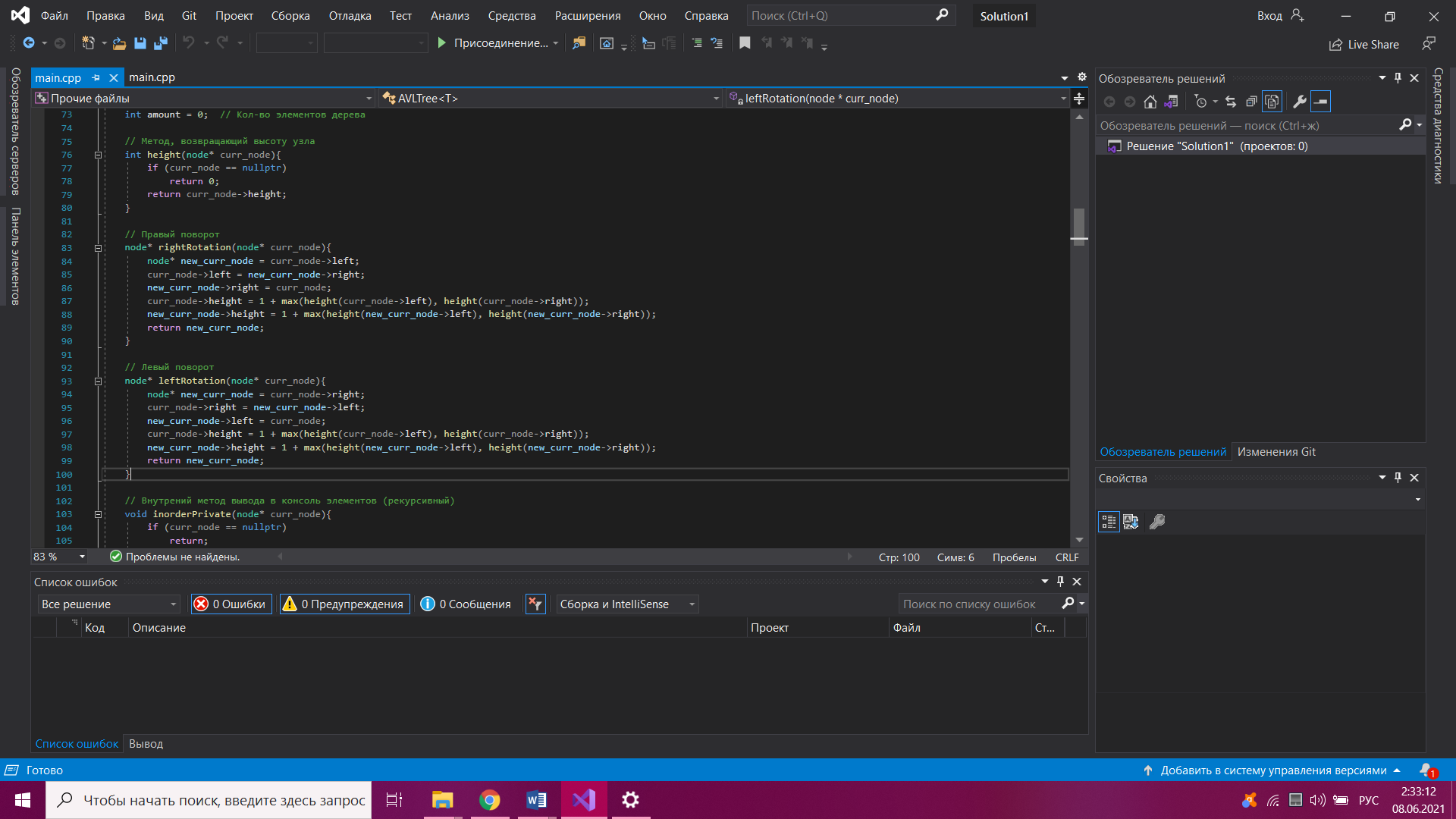


* Ниже используются методы: получение высоты дерева; правый поворот; левый поворот

Первый метод- получает указатель на узел и возвращает оттуда значение его высоты.

Второй метод- принимает указатель на узел на вход, относительно которого надо совершить правый поворот, после совершает алгоритм поворота и возвращает обновленный узел.

Третий- принимает на вход указатель на узел, относительно которого надо совершить левый поворот, после совершает алгоритм поворота и возвращает обновленный узел.

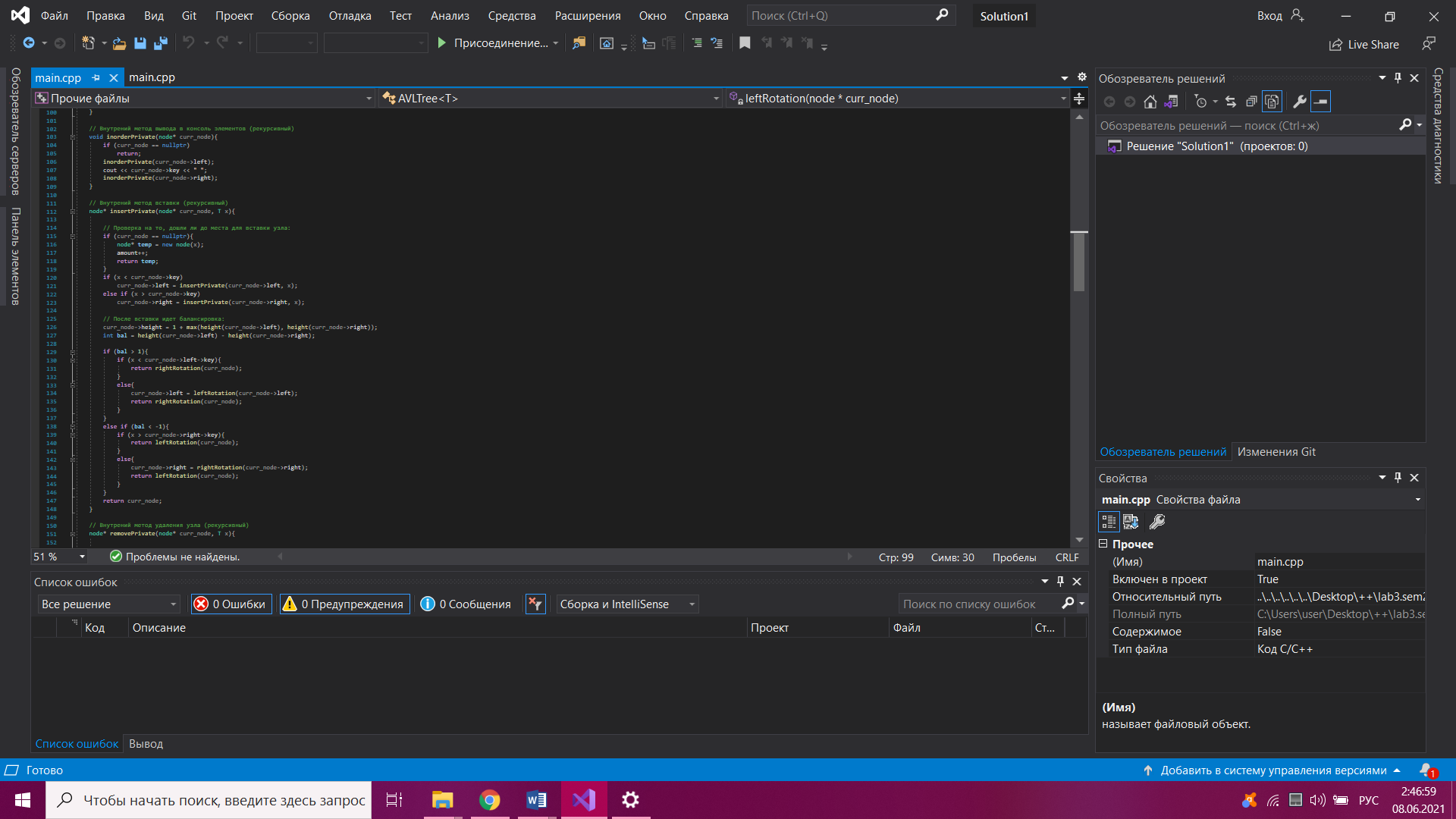


* Здесь мы видим:

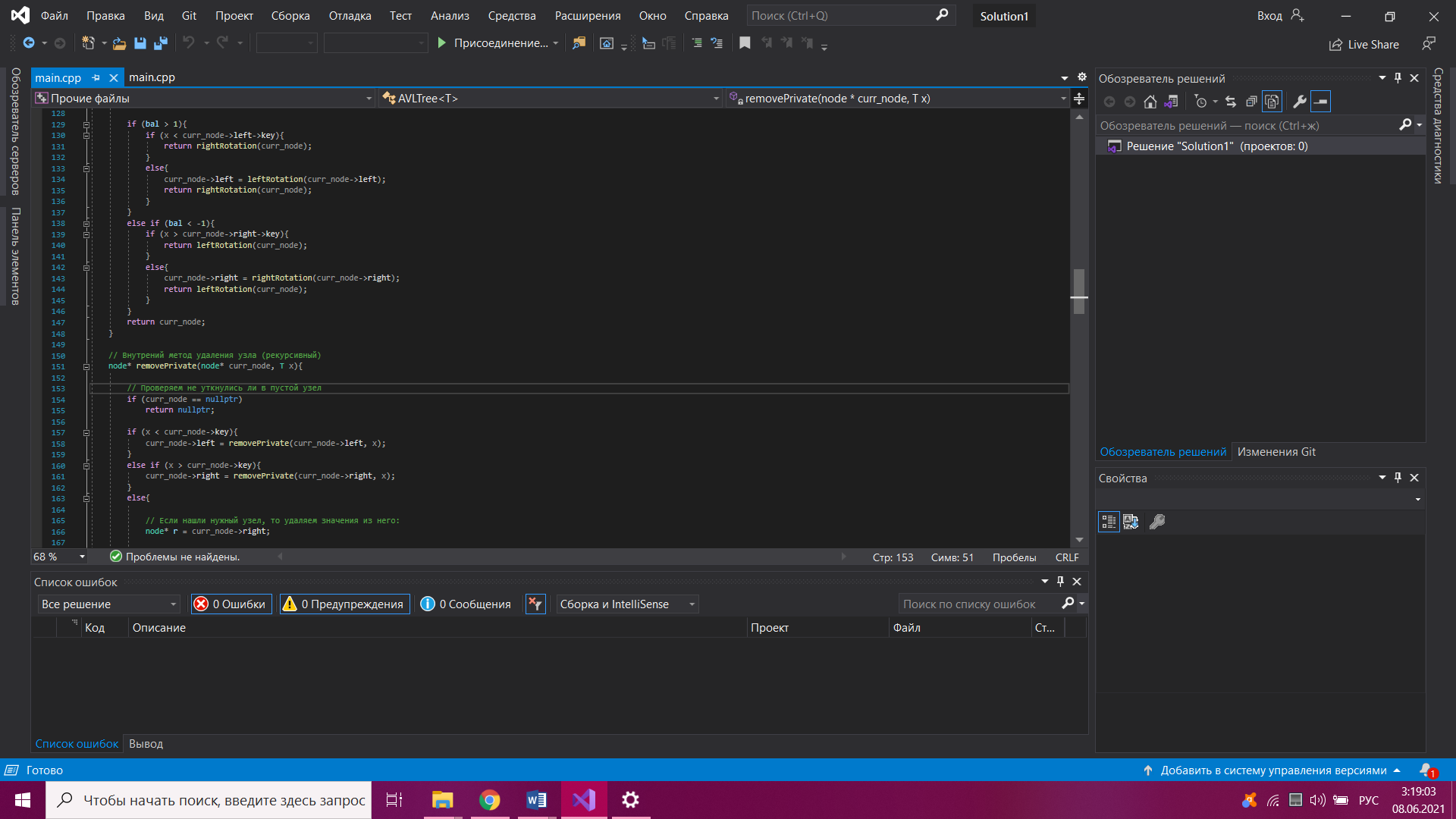
рекурсивный метод вывод в консоль элементов, где проходим по всем узлам дерева по порядку и выводим их все в том же порядке в консоль.

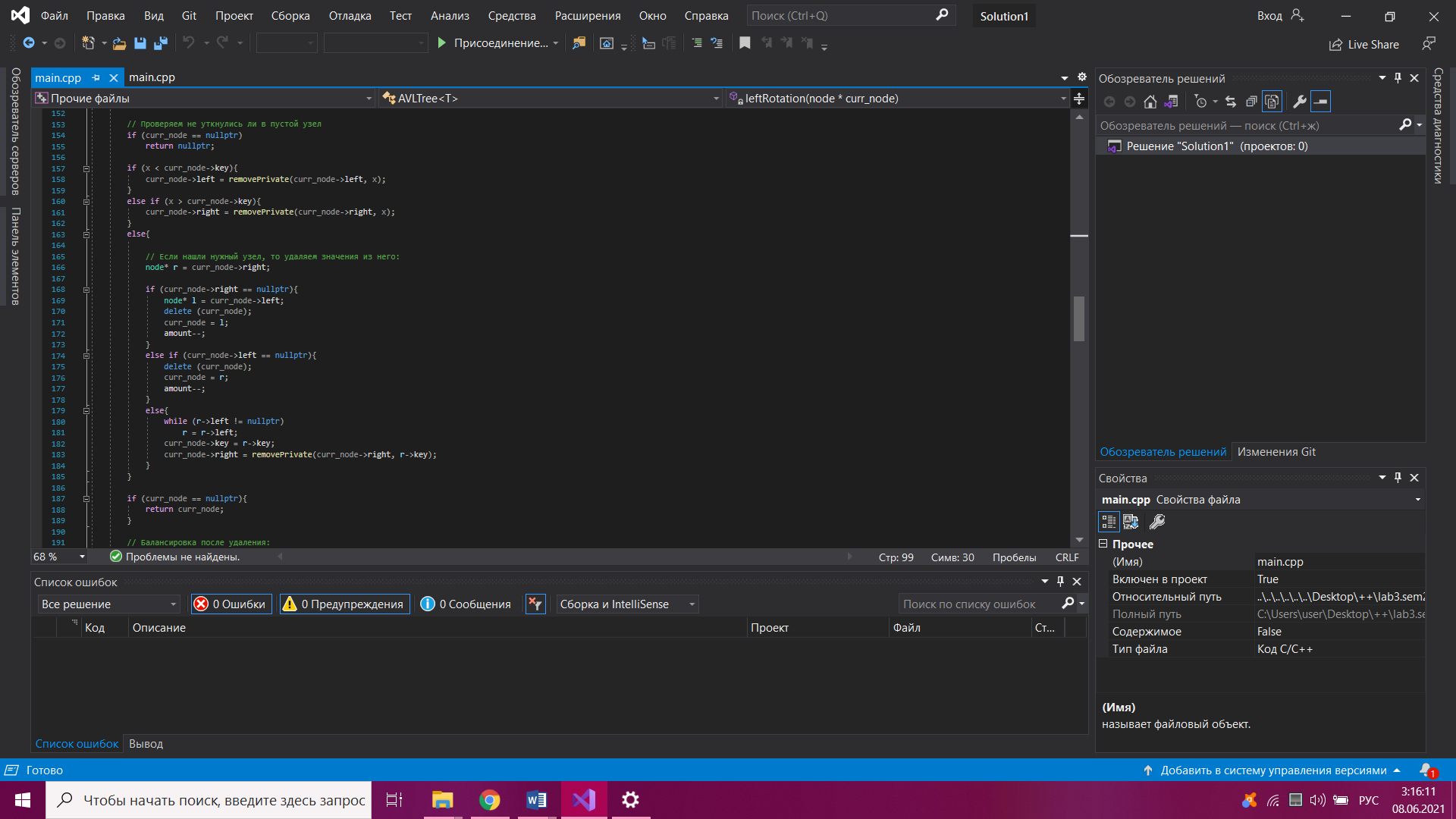
внутренний метод вставки и проверку на то, дошли ли до места для вставки узла

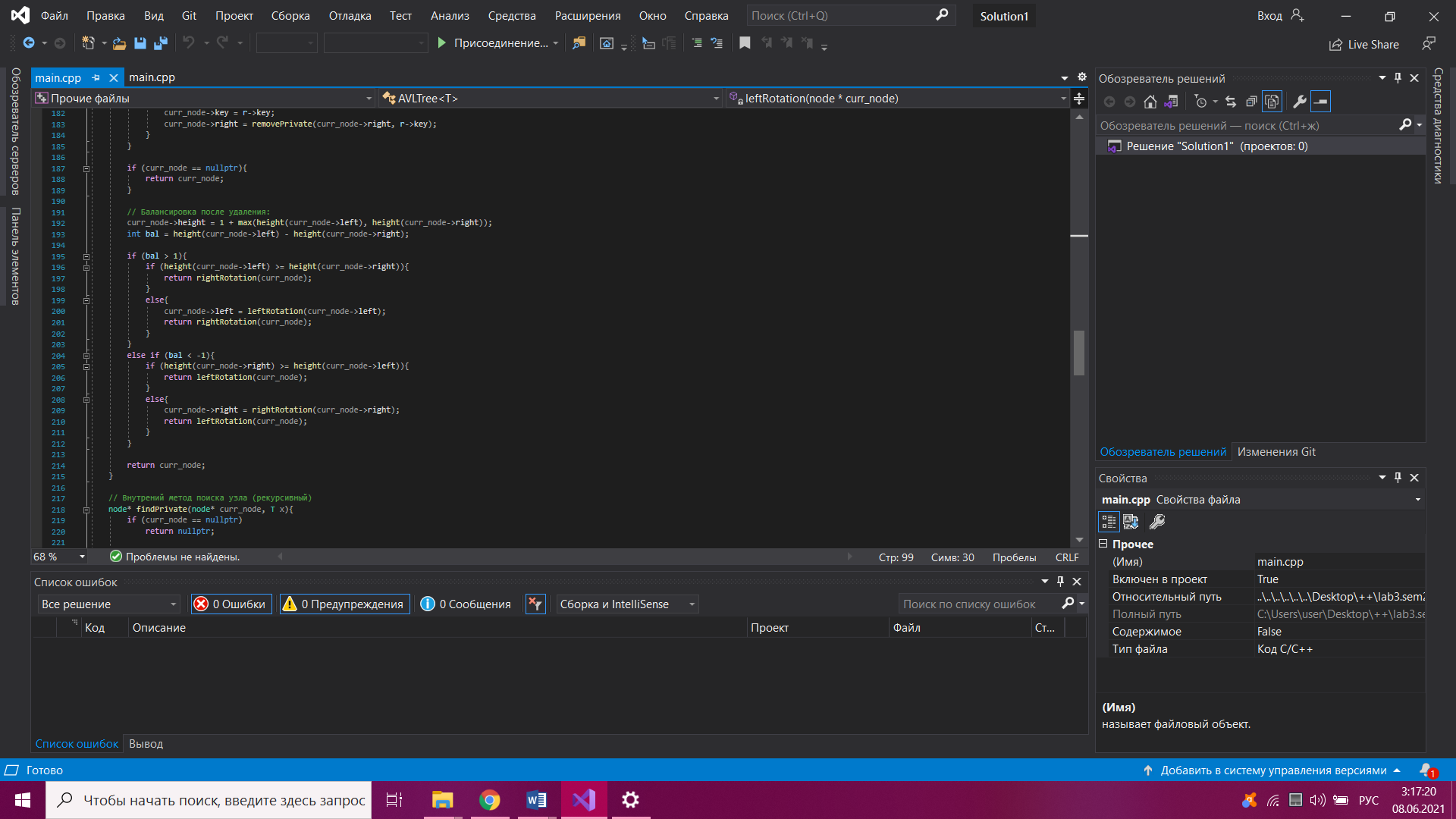
балансировку, которую мы делаем после вставки, где если баланс дерева вышел за данный отрезок, то делаем балансировку. В конце возвращаем новый узел.

,

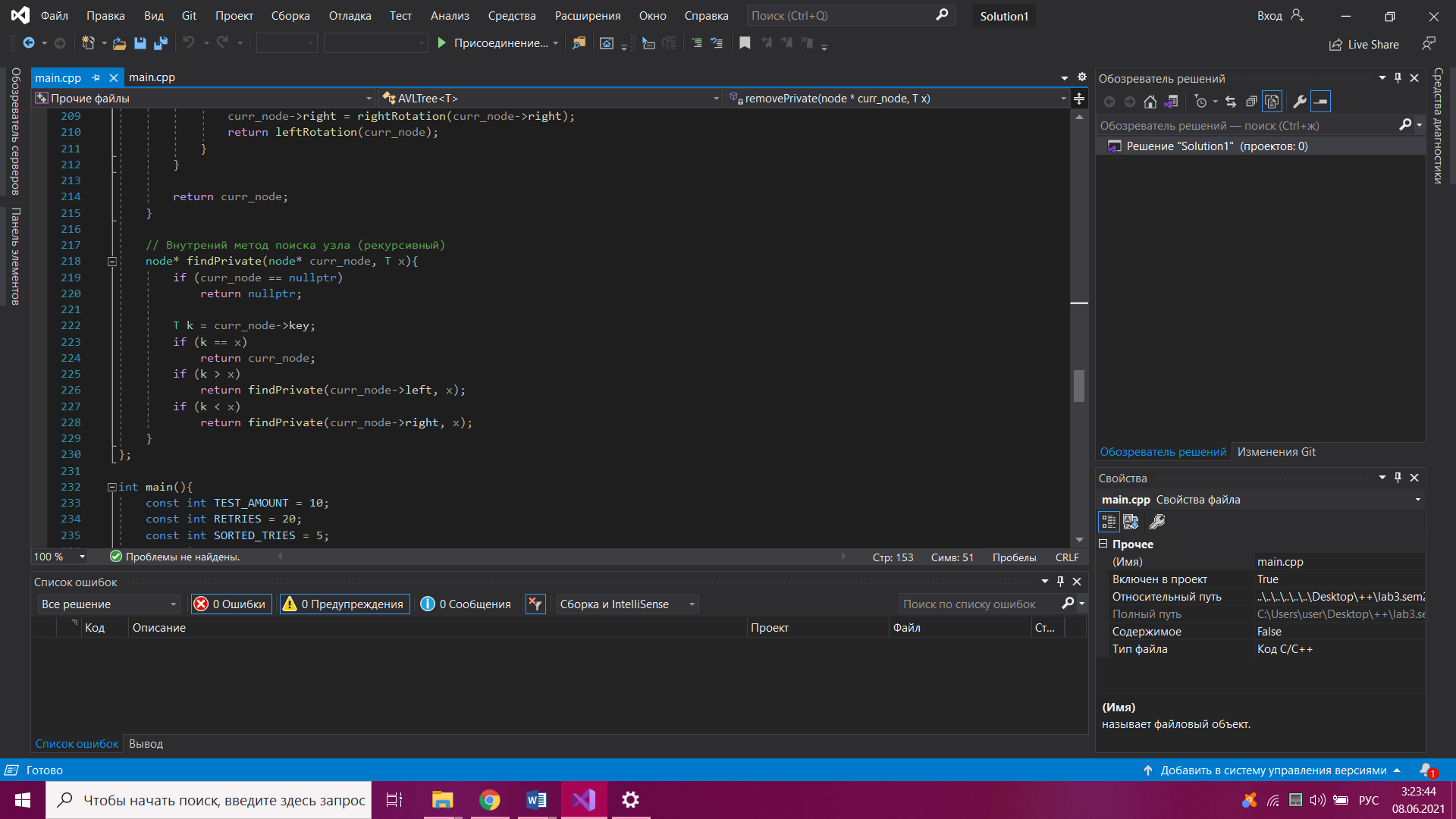
* Следующий операция – удаление. Он очень схожен с предыщим.



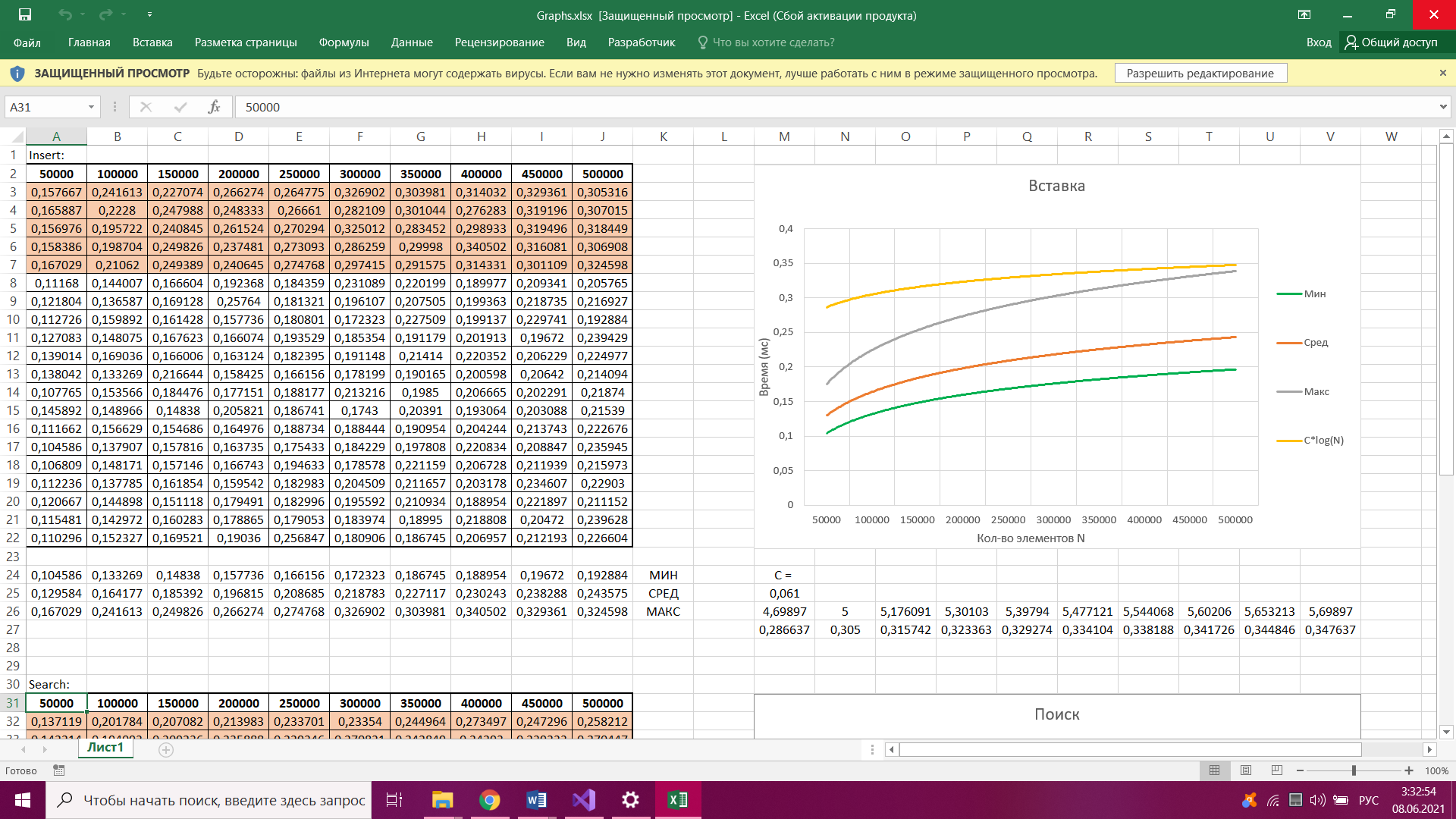




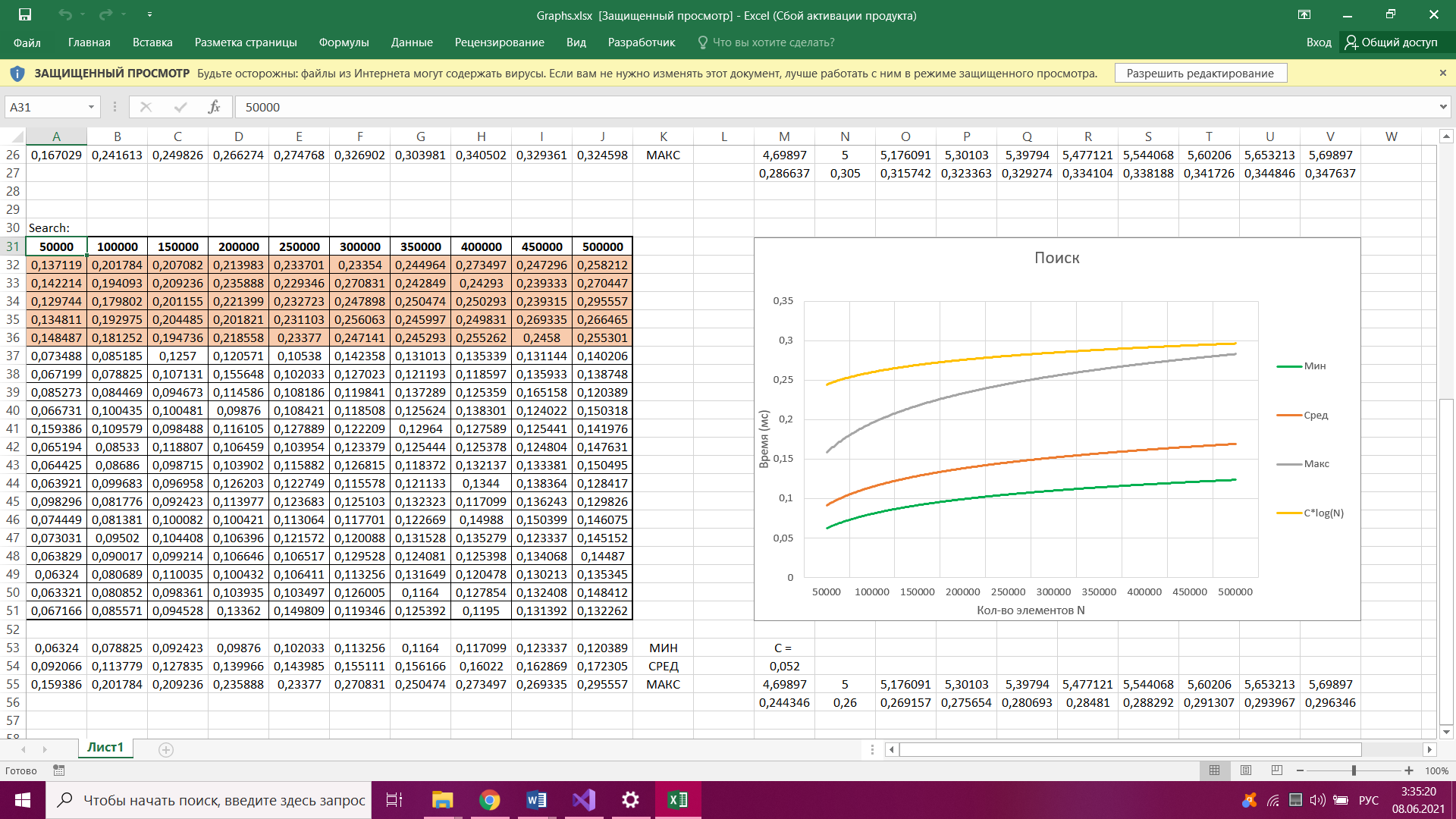
* Метод поиска в дереве. Пробегаемся по дереву, пока не найдем нужный элемент, если нашли искомый элемент, то возвращаем его, иначе вернем нулевой указатель.



Данные по времени на вставку 100 элементов (миллисекунд)



Данные по времени на поиск 100 элементов (миллисекунд)



Так же, приведены графики лучшего, среднего, худшего случаев и график c\*log(N) для обоих операций:

C = 0,061

C = 0,052

# Заключение.

В рамках лабораторной работы изучили и реализовали AVL дерево: класс, операцию вставки и удаления элемента с последующей балансировкой.

Замер времени показал, что операция поиска быстрее, чем вставки.