**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 № 6**

Выполнил студент группы КС-30 Лобачев Дмитрий Сергеевич

Ссылка на репозиторий: https://github.com/MUCTR-IKT-CPP/DSLobachev\_30/blob/main/Algorithms/Laba6/Laba6.cpp

Приняли: Пысин Максим Дмитриевич

Краснов Дмитрий Олегович

Лобанов Алексей Владимирович

Крашенинников Роман Сергеевич

Дата сдачи: 03.04.2023

**Оглавление**

[Описание задачи 3](#_Toc131331759)

[Описание метода/модели 4](#_Toc131331760)

[Выполнение задачи. 5](#_Toc131331761)

[Заключение 13](#_Toc131331762)

# Описание задачи

В рамках лабораторной работы необходимо изучить и реализовать бинарное дерево поиска и его самобалансирующийся вариант в лице AVL дерева.

1. Для проверки анализа работы структуры данных требуется провести 10 серий тестов.
2. В каждой серии тестов требуется выполнять 20 циклов генерации и операций. При этом первые 10 работают с массивом заполненным случайным образом, во второй половине случаев, массив заполняется в порядке возрастания значений индекса, т.е. является отсортированным по умолчанию.
3. Требуется создать массив состоящий из 2^(10 + i) элементов, где i это номер серии.
4. Массив должен быть помещен в оба вариант двоичных деревьев. При этому замеряется время затраченное на всю операцию вставки всего массива.
5. После заполнения массива, требуется выполнить 1000 операций поиска по обоим вариантам дерева, случайного числа в диапазоне генерируемых значений, замерев время на все 1000 попыток и вычислив время 1 операции поиска.
6. Провести 1000 операций поиска по массиву, замерить требуемое время на все 1000 операций и найти время на 1 операцию.
7. После, требуется выполнить 1000 операций удаления значений из двоичных деревьев, и замерить время затраченное на все операции, после чего вычислить время на 1 операцию.
8. После выполнения всех серий тестов, требуется построить графики зависимости времени затрачиваемого на операции вставки, поиска, удаления от количества элементов. При этом требуется разделить графики для отсортированного набора данных и заполненных со случайным распределением. Так же, для операции поиска, требуется также нанести для сравнения график времени поиска для обычного массива.

# Описание метода/модели

Бинарное дерево — это иерархическая структура данных, в которой каждый узел имеет значение (оно же является в данном случае и ключом) и ссылки на левого и правого потомка. Узел, находящийся на самом верхнем уровне (не являющийся чьим либо потомком) называется корнем. Узлы, не имеющие потомков (оба потомка которых равны NULL) называются листьями.

Бинарное дерево поиска — это бинарное дерево, обладающее дополнительными свойствами: значение левого потомка меньше значения родителя, а значение правого потомка больше значения родителя для каждого узла дерева. То есть, данные в бинарном дереве поиска хранятся в отсортированном виде. При каждой операции вставки нового или удаления существующего узла отсортированный порядок дерева сохраняется. При поиске элемента сравнивается искомое значение с корнем. Если искомое больше корня, то поиск продолжается в правом потомке корня, если меньше, то в левом, если равно, то значение найдено и поиск прекращается.

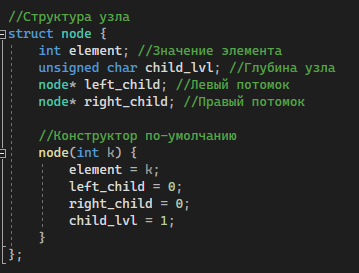
Сбалансированное бинарное дерево поиска — это бинарное дерево поиска с логарифмической высотой. Данное определение скорее идейное, чем строгое. Строгое определение оперирует разницей глубины самого глубокого и самого неглубокого листа (в AVL-деревьях) или отношением глубины самого глубокого и самого неглубокого листа (в красно-черных деревьях). В сбалансированном бинарном дереве поиска операции поиска, вставки и удаления выполняются за логарифмическое время (так как путь к любому листу от корня не более логарифма). В вырожденном случае несбалансированного бинарного дерева поиска, например, когда в пустое дерево вставлялась отсортированная последовательность, дерево превратится в линейный список, и операции поиска, вставки и удаления будут выполняться за линейное время. Поэтому балансировка дерева крайне важна. Технически балансировка осуществляется поворотами частей дерева при вставке нового элемента, если вставка данного элемента нарушила условие сбалансированности.

# Выполнение задачи.

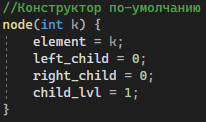
Для реализации данного метода сортировки использовался язык программирования C++.

**Структура узла**

1. Структура узла node имеет поля element– значение элемента, child\_lvl – глубина узла, left\_child – левый потомок, right\_child – правый потомок



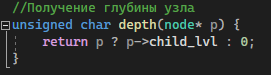
1. Структура также имеет конструктор:



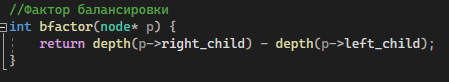
.

**Функции и методы**

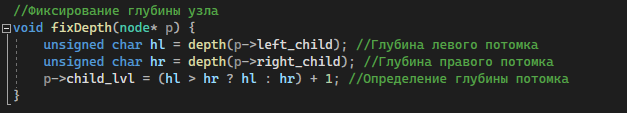
1. Функция получения глубины depth():



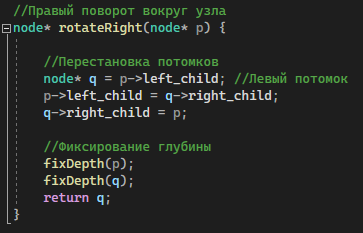
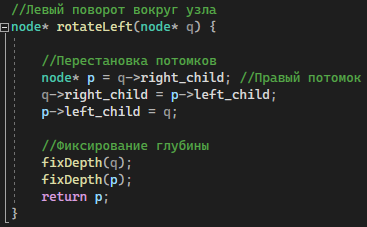
1. Функция расчета фактора балансировки bfactor(). Определяем разницу глубины между левой и правой веткой потомков:



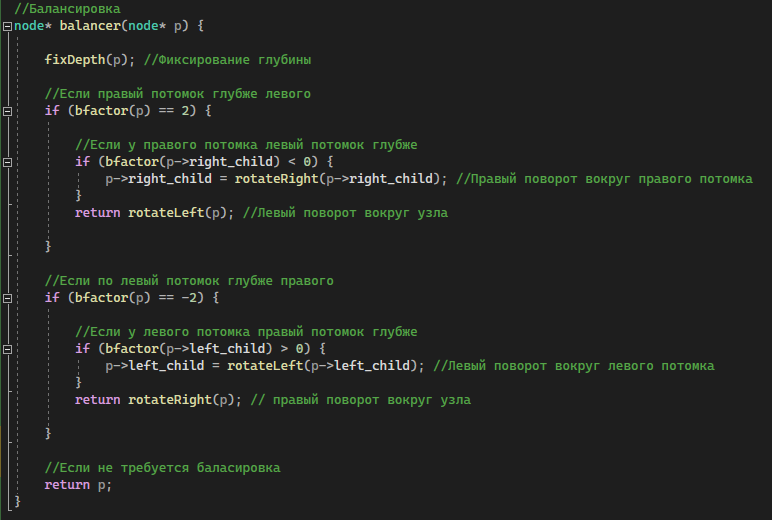
1. Функция фиксирования глубины узла fixDepth(). Определяем максимальное значение глубины по левому и правому потомку и фиксируем её:



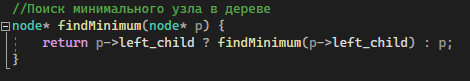
1. Функции поворота вокруг узла rotateLeft() и rotateRight(). В левом повороте мы поднимаем узел на единицу глубины и оставляем правого потомка, а левый потомок уходит в левое поддерево. Правый поворот проходит аналогично, только остается левый потом, а правый уходит в дерево:



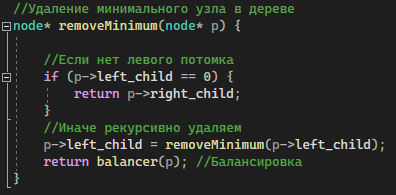
1. Функция балансировки balancer(). После определения глубины веток относительно узла, проверяется какой из поворотов необходимо сделать:



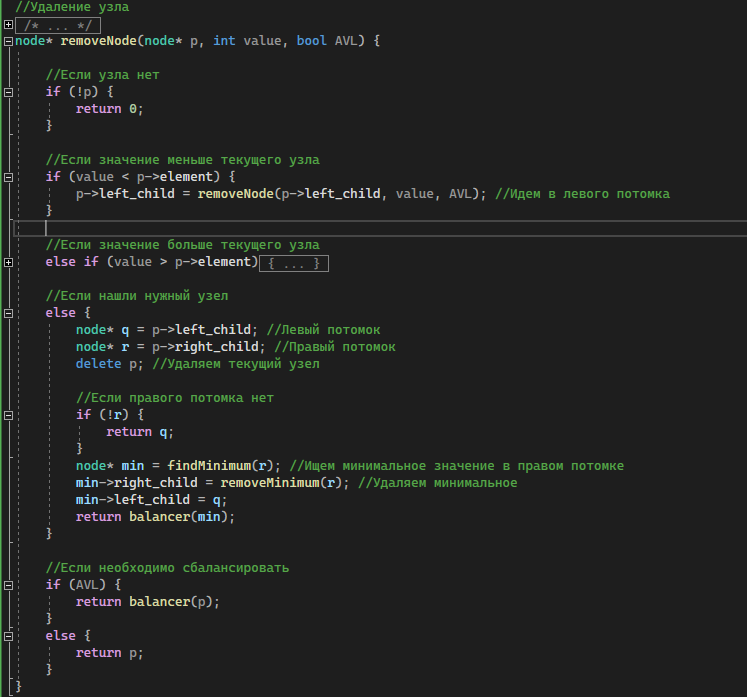
1. Рекурсивная функция поиска минимального узла findMinimum() в левом потомке:



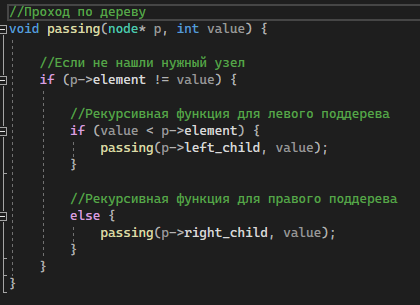
1. Функция удаления минимального узла removeMinimum(). Так как минимальный узел всегда будет в левом потомке, мы ищем его там:



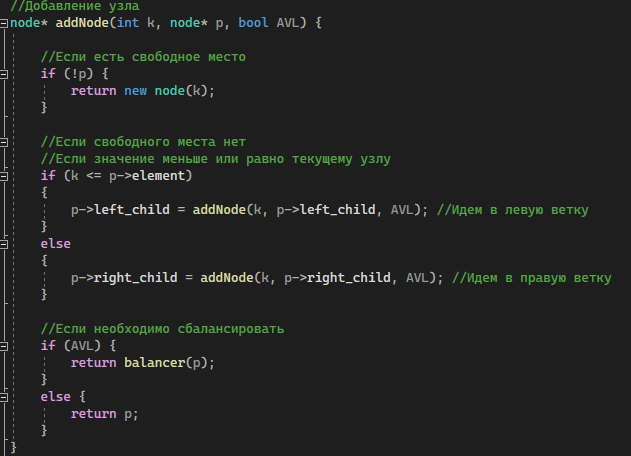
1. Функция удаления узла removeNode(). Мы проверяем, в каком потомке находится узел и удаляем его, распределяя его потомков по узлам:



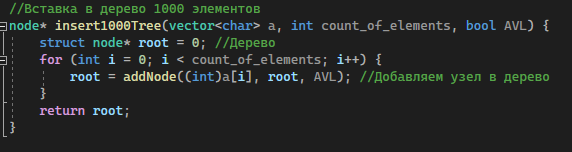
1. Функция прохода по дереву passing(). Пока не найдем нужный нам узел, ищем его в нужном потомке (В левом если меньше узла, в правом если больше узла):



1. Функция добавления узла addNode(). Смотрим, если есть свободное место, то размещаем узел. Если место не найдено, то спускаемся по веткам и ищем его там.



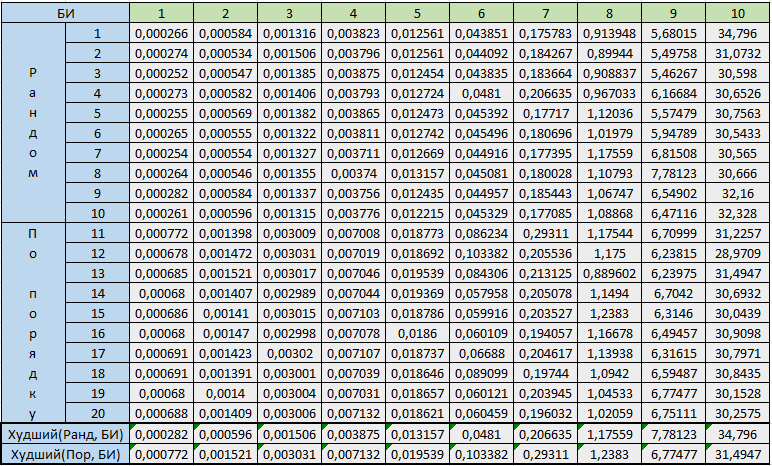
1. Функция вставки 1000 элементов в дерево insert1000Tree().



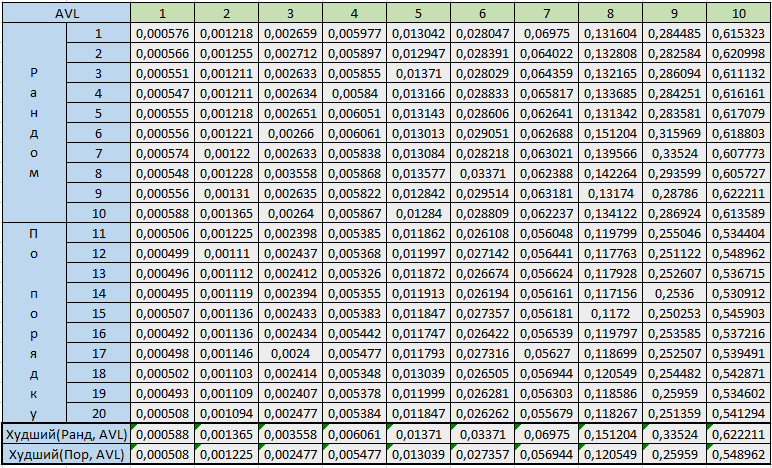
**Тестирование**

В рамках лабораторной работы было проведено тестирование алгоритма на 1000 операций удаления, вставки и поиска. Для каждой из операций было замерено время. Полученные данные были представлены в виде таблиц:

***Вставка***



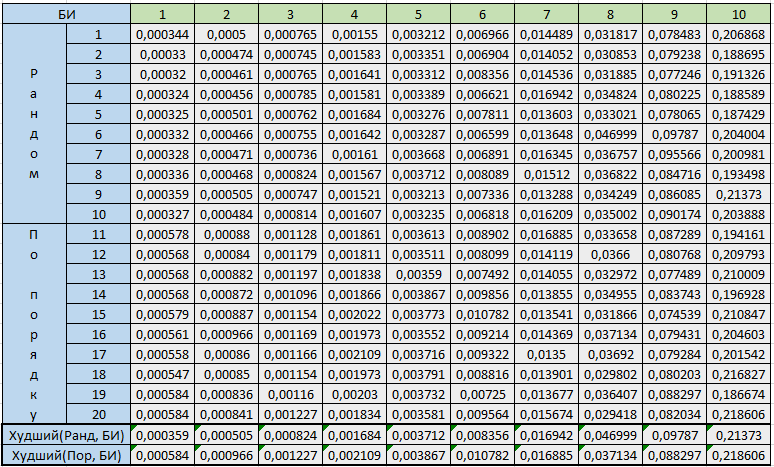
*Вставка в бинарном дереве*



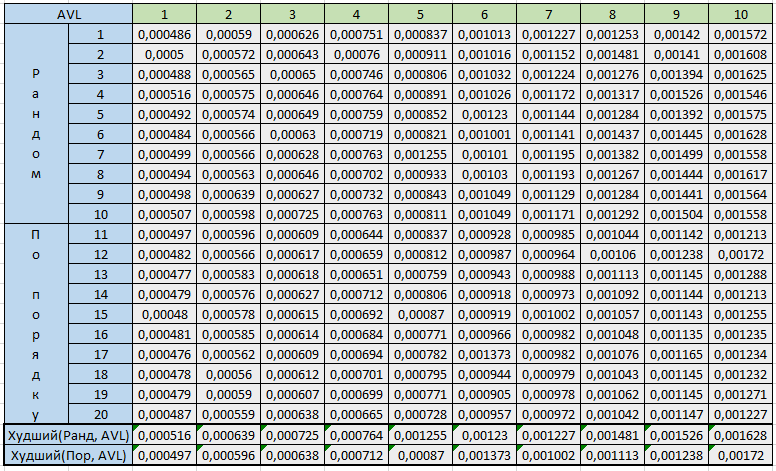
*Вставка в AVL дереве*

По результатам было решено построить график худших случаев.

***Удаление***

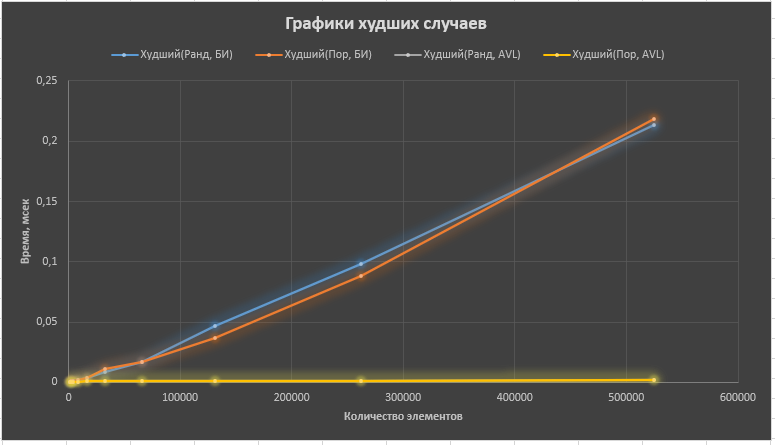
******

*Удаление в бинарном дереве*

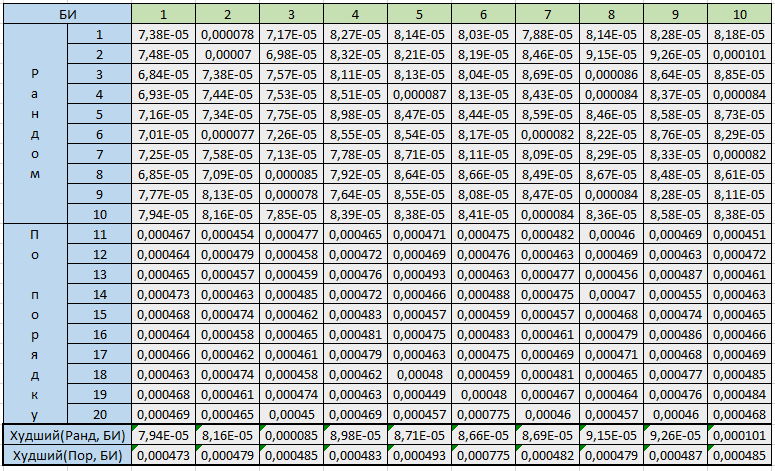
******

*Удаление в AVL дереве*

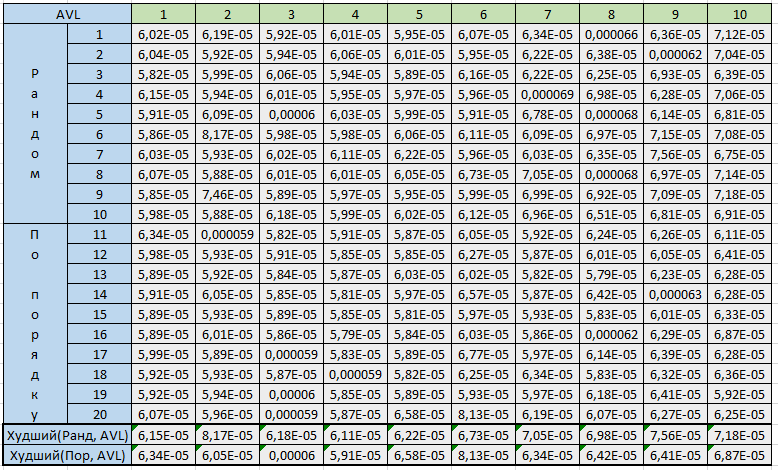
По результатам было решено построить график худших случаев.



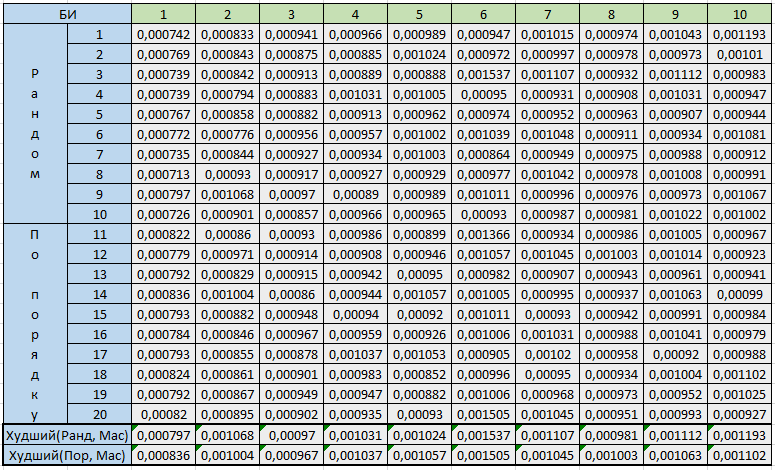
***Поиск***



*Поиск в бинарном дереве*



*Поиск в AVL дереве*



*Поиск в массиве*

По результатам было решено построить график худших случаев.

# Заключение

Проанализировав графики, можно заметить, что скорость операций в AVL дереве в большинстве случаев превосходит скорость операций в обычном дереве. Это объясняется тем, что AVL (или сбалансированное) дерево представляет собой дерево, в котором элементы расставлены равномерно, что и влияет на скорость каждой операции.

Проанализировав график скорости поиска, можно заметить, что любое дерево выигрывает в скорости выполнения у массива, но из всех деревьев, ближе к массиву находится бинарное дерево из последовательной расстановки чисел. Данное дерево представляет из себя единую ветку, где каждое значение встает в правого потомка. Таким образом, поиск по такому дереву будет занимать наибольшее время, так как это дерево является экстремально несбалансированным.

