**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 № 7**

Выполнил студент группы КС-30 Лобачев Дмитрий Сергеевич

Ссылка на репозиторий: https://github.com/MUCTR-IKT-CPP/DSLobachev\_30/blob/main/Algorithms/Laba7/Laba7.cpp

Приняли: Пысин Максим Дмитриевич

Краснов Дмитрий Олегович

Лобанов Алексей Владимирович

Крашенинников Роман Сергеевич

Дата сдачи: 10.04.2023

**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 рандомизированное дерево поиска.

Для этого его потребуется реализовать и сравнить в работе с реализованным ранее AVL-деревом. Для анализа работы алгоритма понадобиться провести серии тестов:

* В одной серии тестов проводится 50 повторений
* Требуется провести серии тестов для N = 2^i элементов, при этом i от 10 до 18 включительно.

В рамках одной серии понадобится сделать следующее:

* Генерируем N случайных значений.
* Заполнить два дерева N количеством элементов в одинаковом порядке.
* Для каждого из серий тестов замерить максимальную глубину полученного деревьев.
* Для каждого дерева после заполнения провести 1000 операций вставки и замерить время.
* Для каждого дерева после заполнения провести 1000 операций удаления и замерить время.
* Для каждого дерева после заполнения провести 1000 операций поиска.
* Для каждого дерева измерить глубины всех веток дерева.

Для анализа структуры потребуется построить следующие графики:

* График зависимости среднего времени вставки от количества элементов в изначальном дереве для вашего варианта дерева и AVL дерева.
* График зависимости среднего времени удаления от количества элементов в изначальном дереве для вашего варианта дерева и AVL дерева.
* График зависимости среднего времени поиска от количества элементов в изначальном дереве для вашего варианта дерева и AVL дерева.
* График максимальной высоты полученного дерева в зависимости от N.
* Гистограмму среднего распределения максимальной высоты для последней серии тестов для AVL и для вашего варианта.
* Гистограмму среднего распределения высот веток в AVL дереве и для вашего варианта, для последней серии тестов.

# 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.

Можно подобрать такую последовательность операций с бинарным деревом поиска в наивной реализации, что его глубина будет пропорциональна количеству ключей, а следовательно операции будут выполняться за O(n). Поэтому, если поддерживать инвариант "случайности" в дереве, то можно добиться того, что математическое ожидание глубины дерева будет небольшим. Дадим рекурсивное определение рандомизированного бинарного дерева поиска.

Необходимой составляющей рандомизации является применение специальной вставки нового ключа, в результате которой новый ключ оказывается в корне дерева (полезная функция во многих отношениях, т.к. доступ к недавно вставленным ключам оказывается в результате очень быстрым). Для реализации вставки в корень нам потребуется функция поворота, которая производит локальное преобразование дерева.

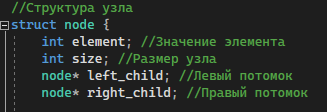
Сбалансированное бинарное дерево поиска — это бинарное дерево поиска с логарифмической высотой. Данное определение скорее идейное, чем строгое. Строгое определение оперирует разницей глубины самого глубокого и самого неглубокого листа (в AVL-деревьях) или отношением глубины самого глубокого и самого 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.

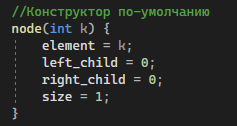
Для реализации данного метода сортировки использовался язык программирования C++.

**Структура узла**

1. Структура узла node имеет поля element– значение элемента, size– размер узла, left\_child – левый потомок, right\_child – правый потомок

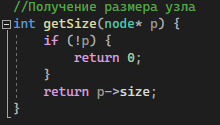


1. Структура также имеет конструктор:

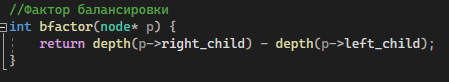


**Функции и методы**

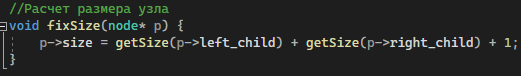
1. Функция получения размера getSize():



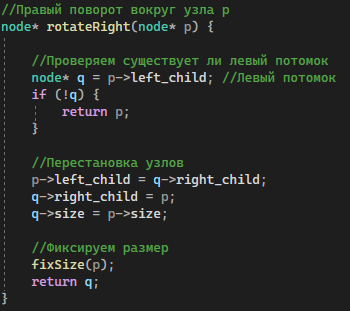
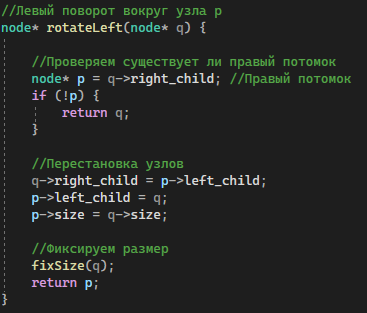
1. Функция расчета фактора балансировки bfactor(). Определяем разницу глубины между левой и правой веткой потомков:



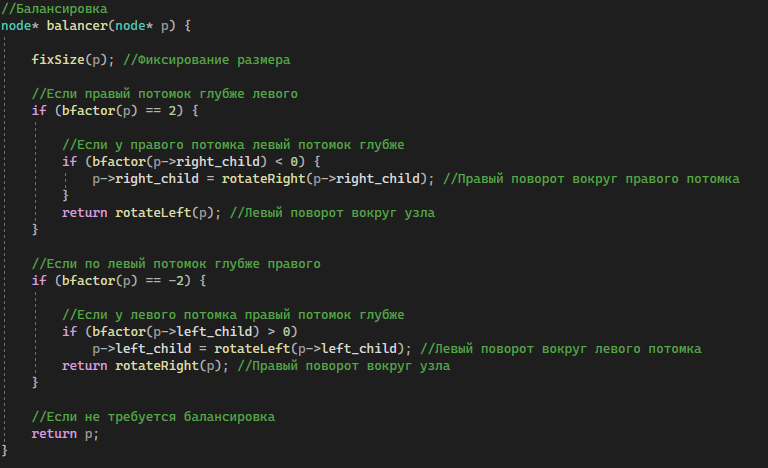
1. Функция фиксирования размера узла fixSize(). Определяем значение размера по сумме левого и правого потомка и самого узла и фиксируем её:



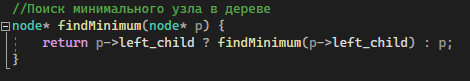
1. Функции поворота вокруг узла rotateLeft() и rotateRight(). В левом повороте мы поднимаем узел на единицу глубины и оставляем правого потомка, а левый потомок уходит в левое поддерево. Правый поворот проходит аналогично, только остается левый потом, а правый уходит в дерево:



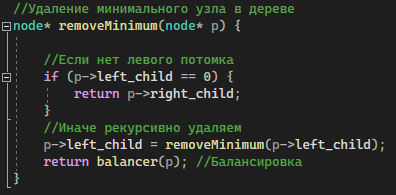
1. Функция балансировки balancer(). После определения глубины веток относительно узла, проверяется какой из поворотов необходимо сделать:



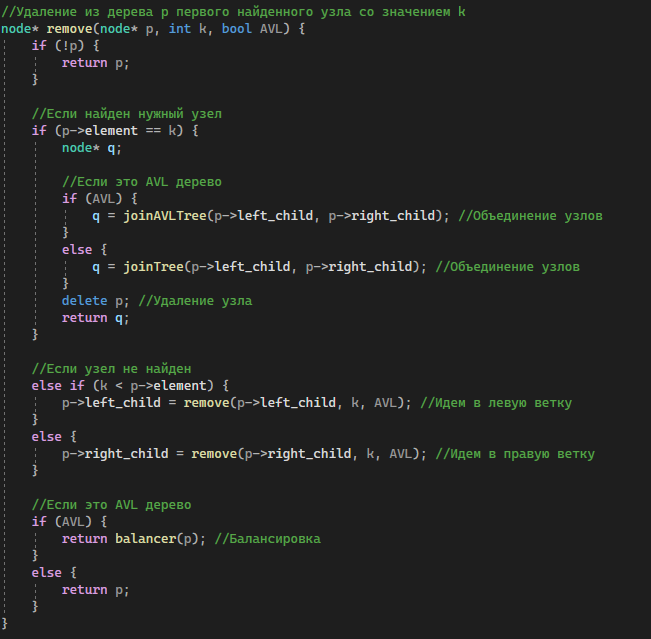
1. Рекурсивная функция поиска минимального узла findMinimum() в левом потомке:



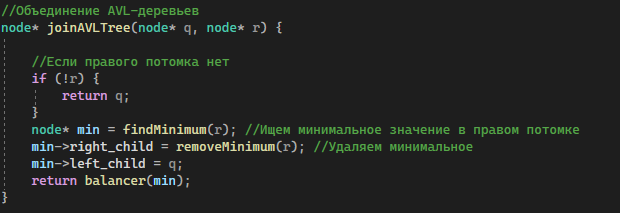
1. Функция удаления минимального узла removeMinimum(). Так как минимальный узел всегда будет в левом потомке, мы ищем его там:



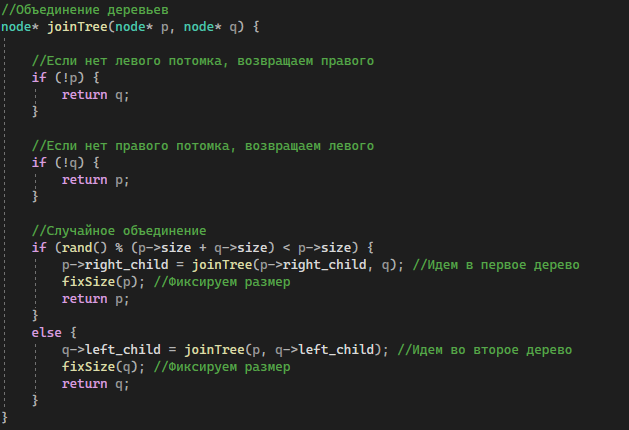
1. Функция удаления узла remove(). Мы проверяем, в каком потомке находится узел и удаляем его, распределяя его потомков по узлам:



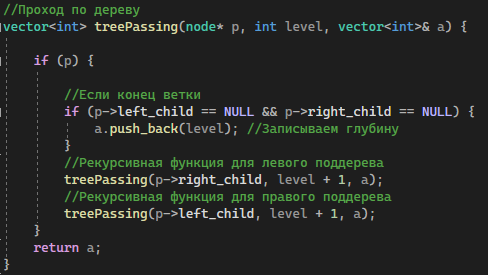
1. Функция объединения AVL-деревьев joinAVLTree(). Проверяем существование правого потомка. Ищем минимальное значение в правом потомке. Удаляем минимальный узел. Вызываем балансировку:



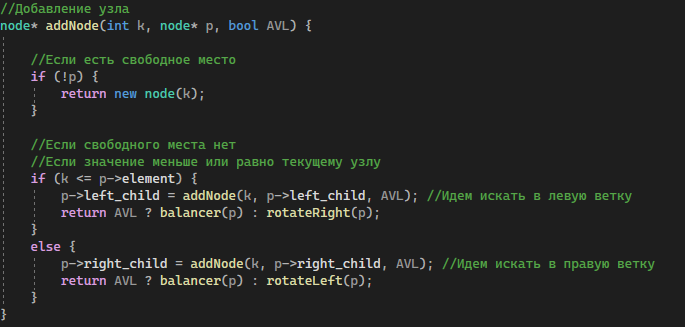
1. Функция объединения RND деревьев joinTree(). Проверяем существование обоих деревьев. С некоторой вероятностью выбираем правое или левое дерево. Фиксируем размер:



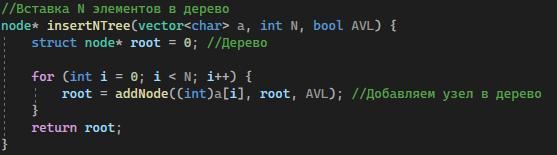
1. Функция прохода по дереву treePassing(). Пока не найдем нужный нам узел, ищем его в нужном потомке (В левом если меньше узла, в правом если больше узла):



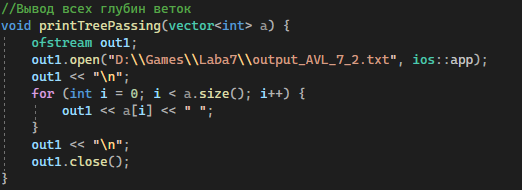
1. Функция добавления узла addNode(). Смотрим, если есть свободное место, то размещаем узел. Если место не найдено, то спускаемся по веткам и ищем его там:



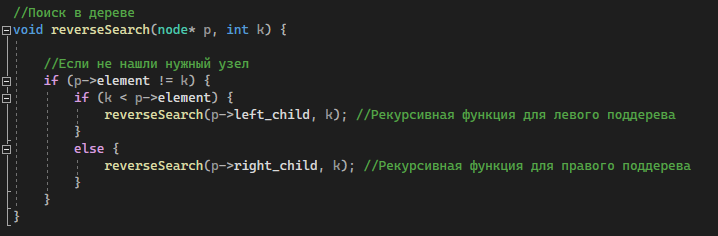
1. Функция вставки N элементов в дерево insertNTree():



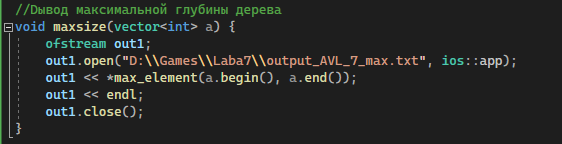
1. Функция вывода глубин всех веток дерева printTreePassing():



1. Функция поиска узла reverseSearch():



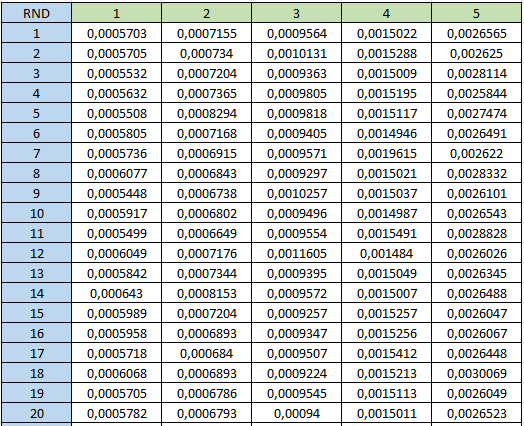
1. Функция вывода максимального значения глубины maxsize():



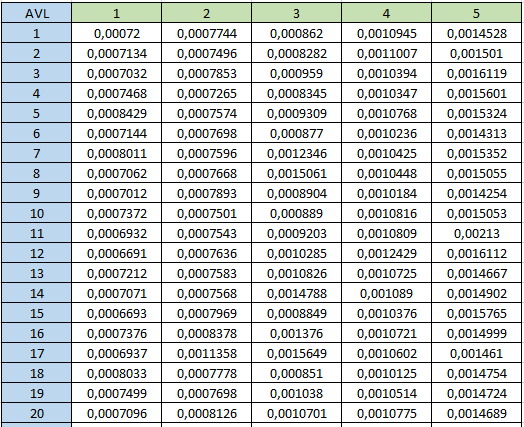
**Тестирование**

В рамках лабораторной работы было проведено тестирование алгоритма на 1000 операций удаления, вставки и поиска. Для каждой из операций было замерено время. Также необходимо было построить гистограмму среднего распределения максимальной высоты и высот всех веток в AVL и рандомизированном деревьях Полученные данные были представлены в виде таблиц:

***Вставка***



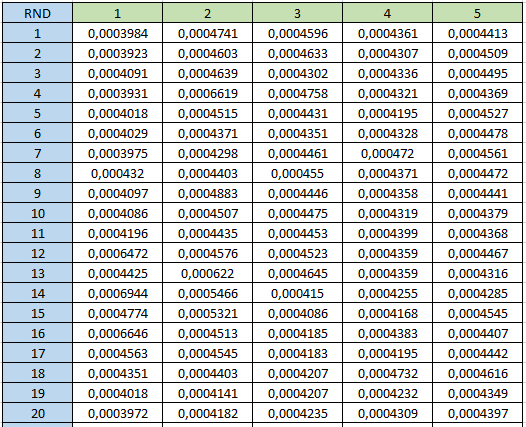
*Вставка в рандомизированном дереве*



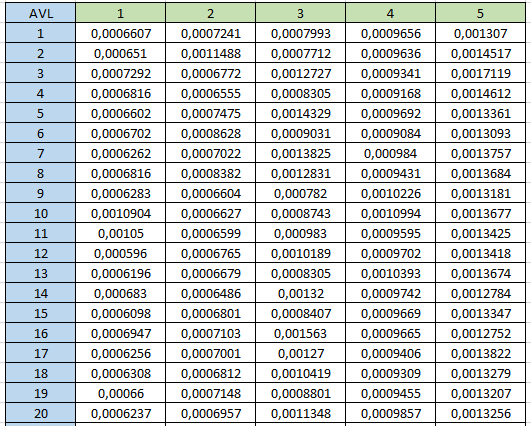
*Вставка в AVL дереве*

По результатам было решено построить график среднего времени.

***Удаление***

******

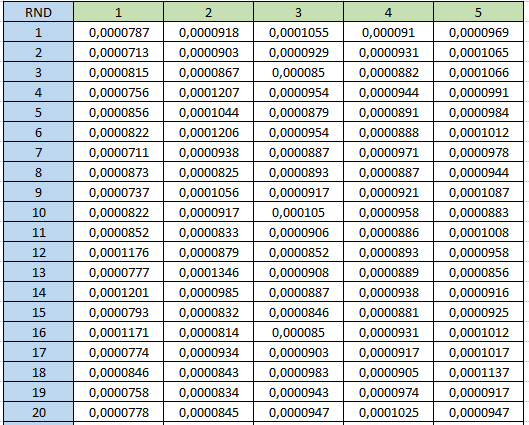
*Удаление в рандомизированном дереве*



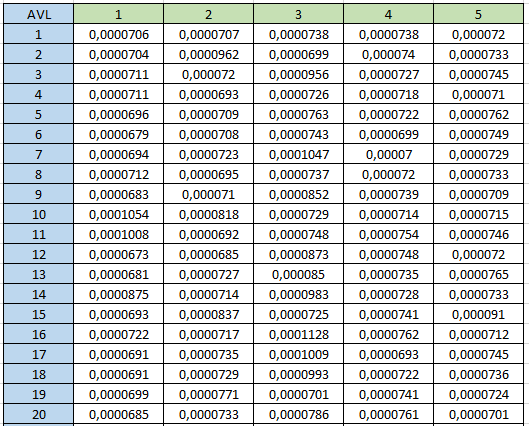
*Удаление в AVL дереве*

По результатам было решено построить график среднего времени.

***Поиск***



*Поиск в рандомизированном дереве*

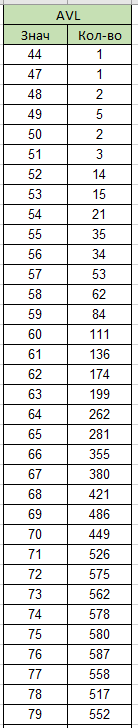


*Поиск в AVL дереве*

По результатам было решено построить график среднего времени.

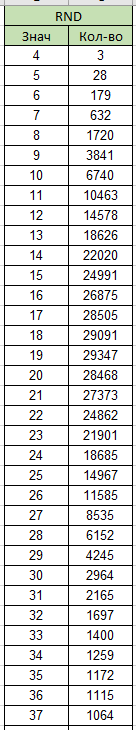
***Среднее распределения максимальной высоты***

Данные для AVL-дерева:



Гистограмма:

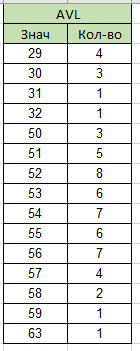
Данные для рандомизированного дерева:



Гистограмма:

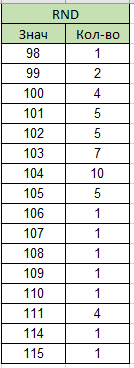
***Среднее распределения высот веток***

Данные для AVL-дерева:



Гистограмма:

Данные для рандомизированного дерева:



Гистограмма:

# 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. Однако поиск в рандомизированном дереве проходит быстрее. Это связано с тем, что при нахождении случайного значения велик шанс попадания в короткую ветку, где содержится это значение.

С помощью гистограмм можно сделать вывод, что в AVL дереве распределение веток происходит равномерно, в то время как в рандомизированном центр смещения немного смещен в сторону. Это можно объяснить тем, что в AVL дереве ветки приблизительно стараются быть равной глубины. Этим же объясняется то, что в AVL дереве максимальная глубина веток ниже, чем у рандомизированного.  
  
  
