**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-36 Перминова П.А.

Приняли: Пысин Максим Дмитриевич

Краснов Дмитрий Олегович

Дата сдачи:

# Оглавление

[**Оглавление**](#_mntj2mtz0gp6) **2**

[**Описание задачи**](#_rqt1pri0r1y7) **3**

[**Описание метода/модели**](#_206yep7tue96) **4**

[**Выполнение задачи**](#_t3f4sigj5ou7) **6**

[**Заключение**](#_g8m43y2s9q2j) **8**

# Описание задачи

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

Для проверки анализа работы структуры данных требуется провести 10 серий тестов.

* 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.
* Требуется создать массив состоящий из 2^(10 + i) элементов, где i это номер серии.
* Массив должен быть помещен в оба вариант двоичных деревьев. При этому замеряется время затраченное на всю операцию вставки всего массива.
* 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.
* Провести 1000 операций поиска по массиву, замерить требуемое время на все 1000 операций и найти время на 1 операцию.
* После, требуется выполнить 1000 операций удаления значений из двоичных деревьев, и замерить время затраченное на все операции, после чего вычислить время на 1 операцию.
* 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 так же нанести для сравнения график времени поиска для обычного массива.

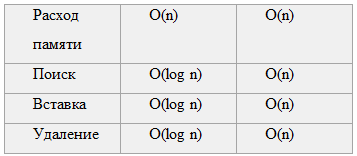
# Описание метода/модели

Двоичное дерево – это иерархическая структура, в которой каждый узел содержит не более чем двух потомков. Для каждого узла, тот узел, который стоит выше по иерархии для него называют родительским узлом, а те узлы, что стоят ниже, для которых этот узел является родительским, называются правым и левым наследниками.

Двоичное дерево поиска – это двоичное дерево придерживающееся 2х правил, согласно которым левые потомки всегда меньше или равны текущего элемента, а правые больше или равны текущего элемента.

Поисковые деревья — это решения так называемой «словарной проблемы». Предположим, что имеется большое количество ключей, каждый из которых имеет значение. В немецко–английском словаре немецкое слово является ключевым, а английские слова являются значением, которое вы ищете. Аналогично ведет себя телефонная книга с именем и адресом в качестве ключа, а номер телефона — в качестве искомого значения.

Этот подход получил в информатике название «двоичный поиск». Она воссоздана очевидным образом с помощью очень известного метода поиска «двоичный поиск в массиве». Их поведение оптимально с точки зрения информации, а именно логарифмически.



АВЛ дерево является обычным двоичным деревом поиска, следовательно, его правое поддерево всегда меньше значения корня, а правое поддерево всегда больше. При это, при построении дерева мы руководствуемся правилом балансировки или перебалансировки: для любого узла дерева высота его правого поддерева отличается от высоты левого поддерева не более чем на единицу. Является доказанным, что при соблюдении этого правила высота дерева логарифмически зависит от количества элементов, добавляемых в дерево, т.е. h = O(log(n)).

Узел AVL дерева, содержит:

⦁ Ключ

⦁ Высоту – от лепестков

⦁ Левый потомок

⦁ Правый потомок

По поводу высоты, можно учесть важный момент, классически атрибут высоты содержит не высоту, а разницу, которая может быть -1,0,1, так как в остальных случаях вызывается перебалансировка, но из за логарифмической природы высоты, можно смело хранить само значение, но тогда придется всегда рассчитывать фактор перебалансировки, который будет разницей между высотами правого и левого поддерева.

Операция вставки и удаления вызываются практически так же, как у обычного двоичного дерева. Разница только в том, что для каждой вставки и каждого удаления требуется последним действием вызывать операцию перебалансировки, каждый раз проверяя от низа к верху необходимость ребалансировать дерево в текущем узле.

Операция балансировки вызывается тогда, когда фактор балансировки становится равным 2 или -2, т.е. тогда, когда разница между правым и левым поддеревьями является больше чем заложено в правиле. В этом случае требуется выполнить операцию поворота дерева, которая переориентирует узлы так, что они изменяют свое положение решая проблему дисбаланса.

Выделяют 4 основных поворота для AVL дерева:

⦁ Простой/малый левый поворот

⦁ Простой/малый правый поворот

⦁ Сложный/большой левый поворот – это два последовательных поворота, сначала правый потом левый.

⦁ Сложный/большой правый поворот – это два последовательных поворота, сначала левый потом правый.

Все повороты выполняются относительно какого-либо узла.

# Выполнение задачи

Данная лабораторная работа была реализована на языке Python. Реализованные классы:

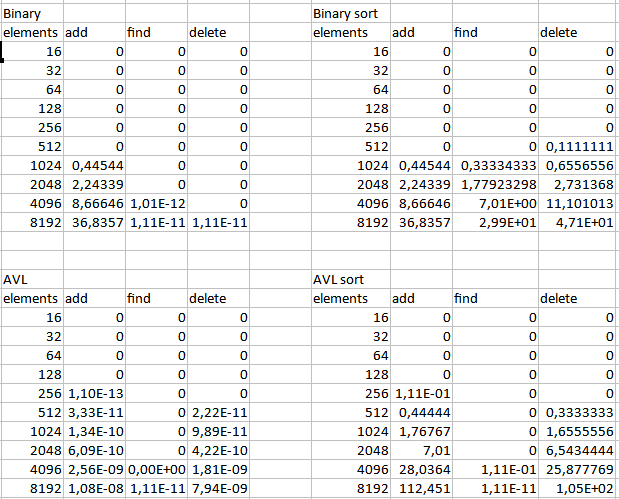
* BinaryTree
* AVL (в классе AVL реализовано самобалансирующееся дерево, в котором при помощи метода balance() при добавлении/удалении элемента в/из дерева, происходит перебалансировка, если она необходима.)

Балансировка в методе balance() происходит при помощи четырех методов:

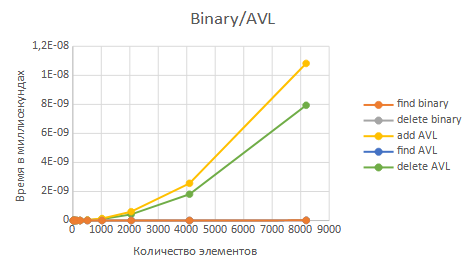
* Малый левый поворот
* Большой левый поворот (право лево)
* Малый правый поворот
* Большой правый поворот (лево право)

Метод balance() зависит от balancefactor, соответственно, если он больше единицы, то необходим левый поворот, если -1 – то правый, и, наконец, если 0 – то балансировка не нужна.

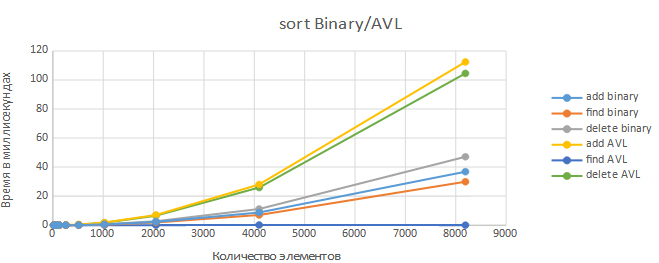
Далее произведен анализ двух деревьев, добавление, поиск и удаление массива в каждое из деревьев. На таблице представлены результаты методов деревьев в миллисекундах в зависимости от разных массивов.



На графике можно увидеть, что для неотсортированного массива поиск и удаление элементов почти одинаковы, в то время как добавление в обычное бинарное дерево занимает больше времени.



Результат для отсортированного массива выглядит следующим образом:



На графике видно, что кривая find AVL практически лежит на оси x, в то время как find binary постепенно растет, хотя даже при 8 тысячах элементов find AVL также лежит на икс, в то время как значение find binary в этой точке равно 20 миллисекундам, и постепенно растет. Однако можно увидеть, что заполнение происходит дольше у AVL дерева. Я считаю, что это связано с тем, что для каждой вставки/удаления приходится производить балансировку, так как значения будут вставляться в правое поддерево.

# Заключение

Исходя из полученных данных можно сделать вывод, что скорость поиска в AVL дереве гораздо больше, чем у бинарного дерева. Данное преимущество AVL дерева особенно заметно при заполнении дерева отсортированным массивом. Это происходит из-за того, что в бинарном дереве при поиске нам придется рассматривать всё правое поддерево, в то время как в AVL нам не придется заходить так глубоко. Однако, у AVL есть и свои недостатки. Например, мы могли заметить, что при отсортированном массиве заполнение и удаление происходит гораздо медленнее. Это связано с балансировкой: так как в балансировке мы крутим дерево для его баланса, это занимает какое-то время, в то время как в бинарном дереве идет простой поиск места, куда можно вставить значение.