DSA – Zadanie 1

Hredzhev Oleksandr

ID: 123560

**Binary Tree** – is a data structure based on a binary tree in which individual elements (root, nodes, vertices) organized in such a way that the set value can be quickly found in this tree. Every peak in a binary tree it has at most two children, left and right, where the smaller value is on the left and the larger one on the right.

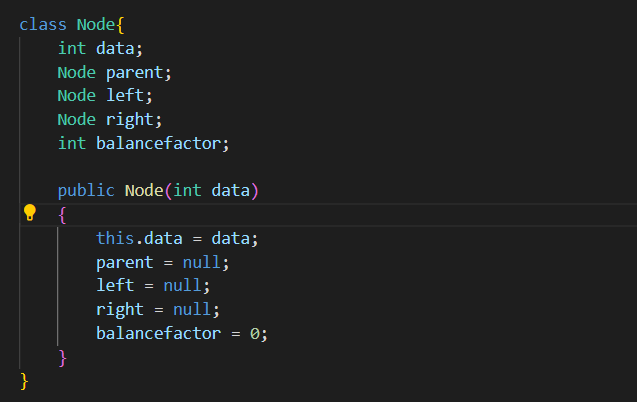
Balancing the tree basically means ensuring an even distribution of the tree. This will improve the complexity of the individual operations (left and right rotations) that we will perform. If we did not balance the graph, the worst case would have a complexity of O(n), which would mean that the program must perform n iterations. Using balancing, the complexity of the operation should look like O(log n).

**AVL Tree**

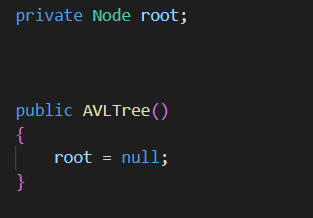
As the first implementation of balanced trees, I chose the *AVL tree*. This is a tree that maintains its balance thanks to the rotations that occur when the height of the left and right subtrees differs (for each node of the tree) > 2 | -2, the maximum allowable height difference is -1 | 1 or no difference: 0.

As the implementation of the node in the tree, the "Node" structure was chosen, which, in addition to the numeric *key*, has pointers to the *right* and *left children* (since each node can have one or two children), the current *balance factor* and also a pointer to the *parent* (which greatly simplifies the execution some operations). (Picture №1.1)

In addition, there is a pointer to the root of the tree – “root” (Picture №1.2)

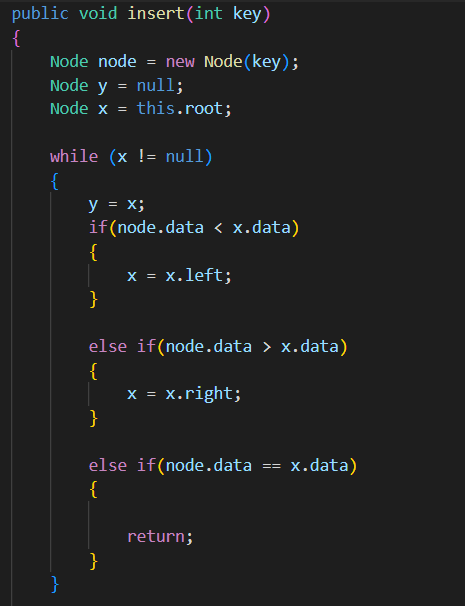


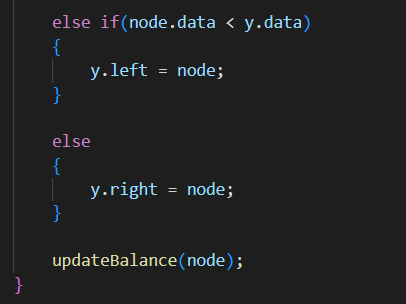
Picture №1.1

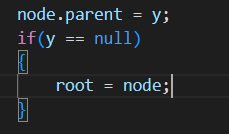


Picture №1.2

This picture shows the implementation of inserting a new node into the tree (if the root already exists):

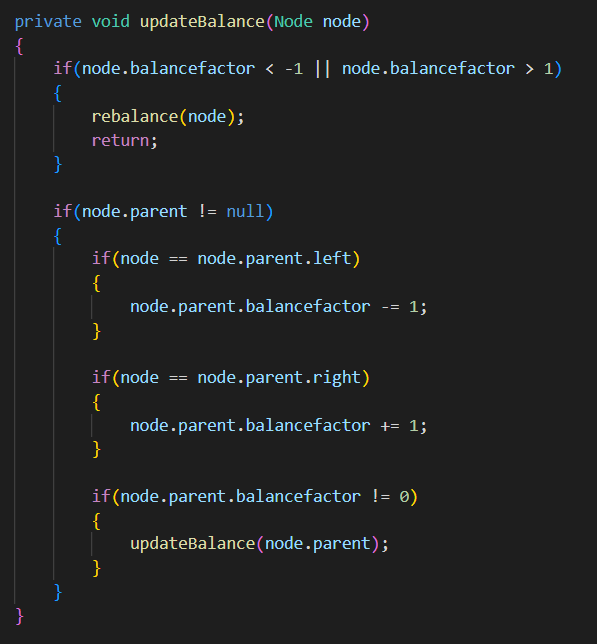






This code defines the root

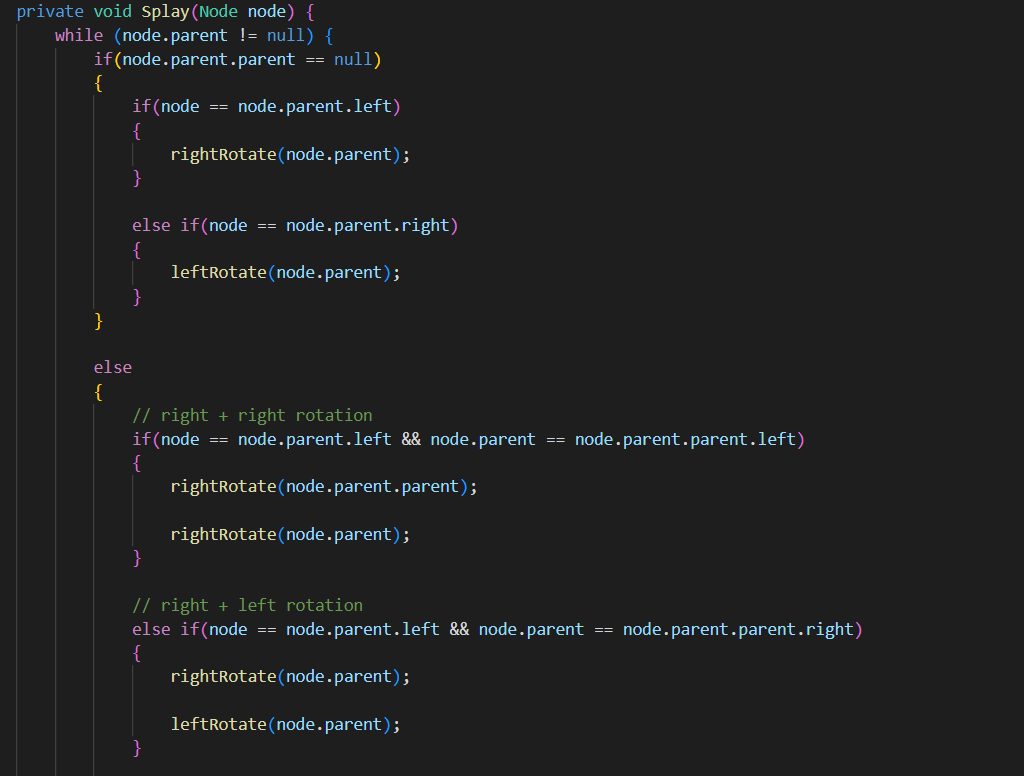
Also in the code there is a function for balancing the tree, which is called every time an element is added. This the function checks the parent of the added element, then the parent of that parent, and so on until it reaches root. If the function detects a balance coefficient that does not correspond to the norm, it will call the rebalancing function, which will correct the balance of this node using rotations (left and right). *After rebalancing* **current node**, an unsatisfactory balance may occur in the nodes of the tree that are higher in the tree (i.e. are parents, parents of parents, etc. up to the root of the tree). This requires a loop that will balance all nodes along the way to the root of the tree as needed. This algorithm is shown in the picture:

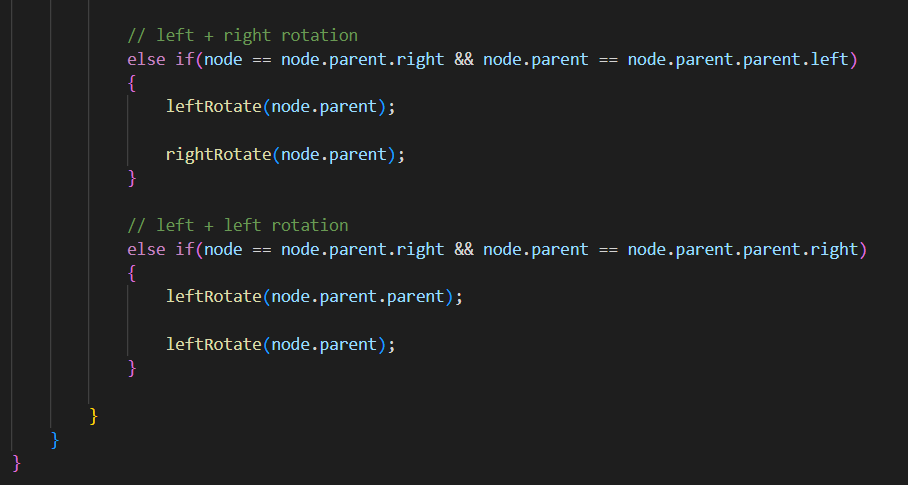


**Splay Tree**

As the second implementation of balanced trees, I chose the *Splay tree*.

The *Splay tree* has a specific way of balancing, since each new node inserted or found node(by “search” operation) becomes the root of the tree (such a tree is often not perfectly balanced, but it is extremely effective for finding periodically repeated nodes) This operation called “*Splay*”(Shown on the next picture).





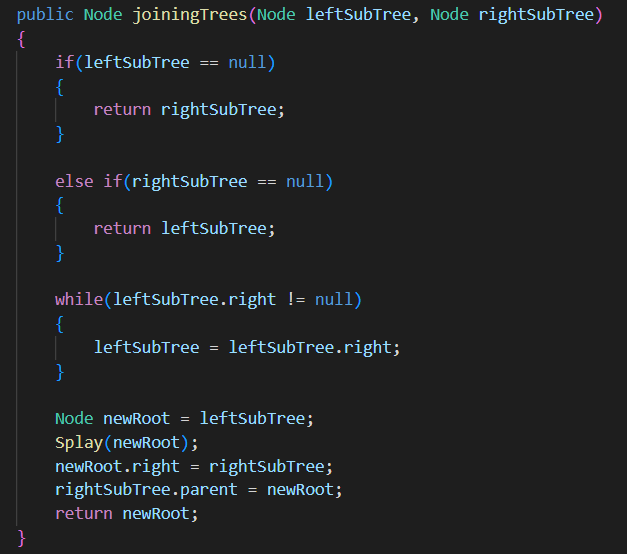
Another feature of this tree is the “*Join”* operation, which is used when deleting a node. Once the node to be removed is determined, it becomes the new root of the tree using the *Splay* operation, and then the *Join* operation is triggered.

Join:

\* If the root being removed does not have a left or right subtree, then the Join operation declares the right or left tree as a new tree, respectively.

\* If the root to be removed has both subtrees, then the maximum value of the left subtree is declared as the new root, having previously used the Splay operation on it, and the right subtree remains in the same place.

(Shown on the next picture)



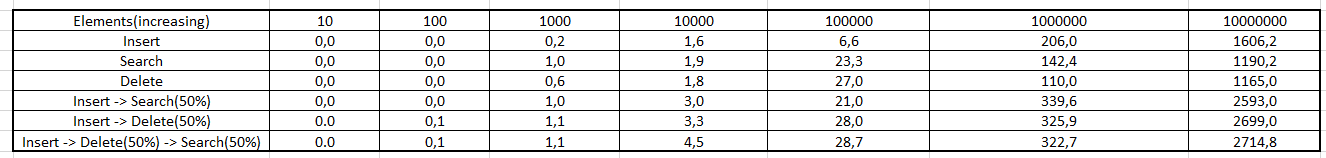
**Testing file**

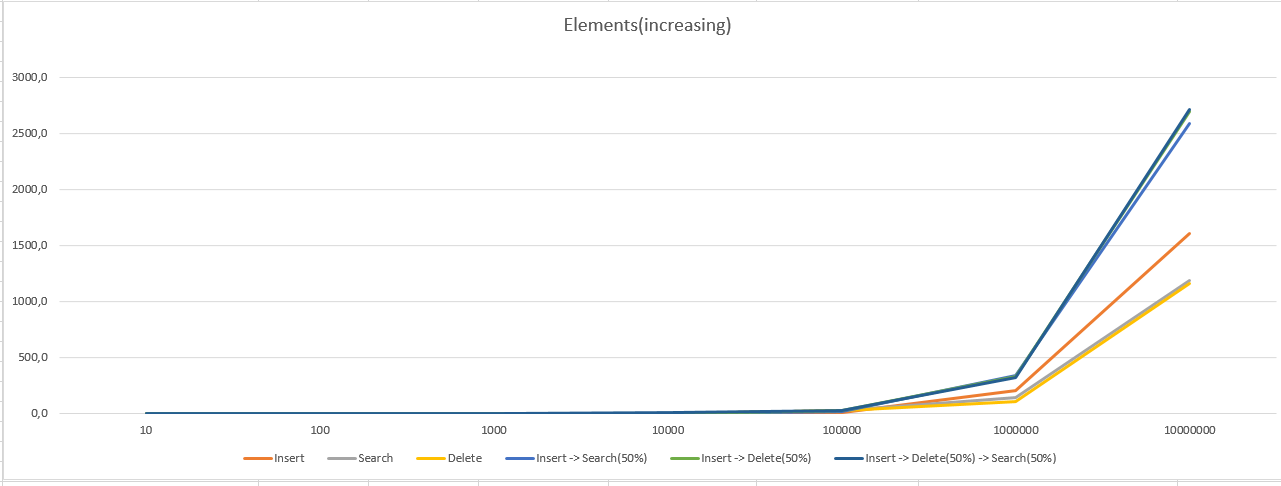
In the test file, I check the effectiveness of the trees. Since testing is fair, identical operations are involved in all operations and it is enough not to change the contents of the file (but there is such a possibility). Visualization and delete / search notifications are there, but I will turn off them with a comment (" // ") as they will significantly affect the test results. I will demonstrate these features during the presentation. All user instructions will be printed to the console, so I won't write about them here.

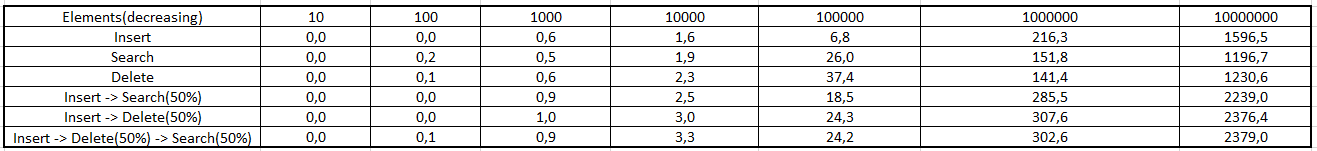
**TEST:**

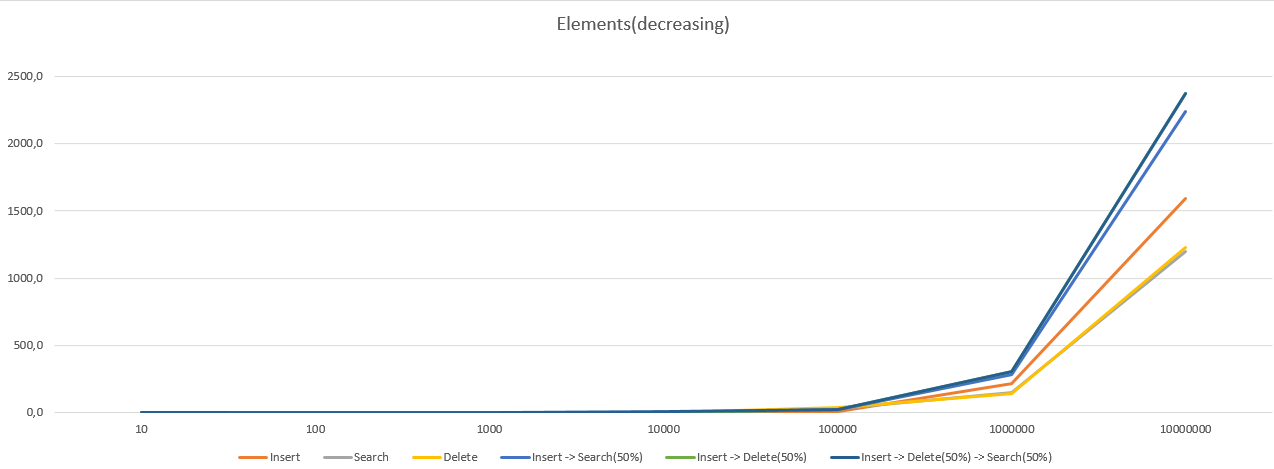
As planned by my test, I compare different operations with different amounts of data within the same tree, and then provide an average value and compare two different trees with each other.

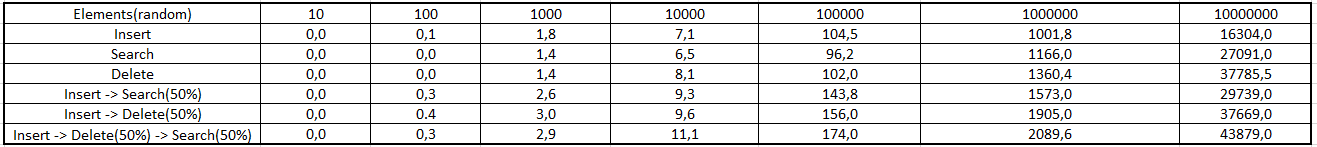
**Splay Tree**

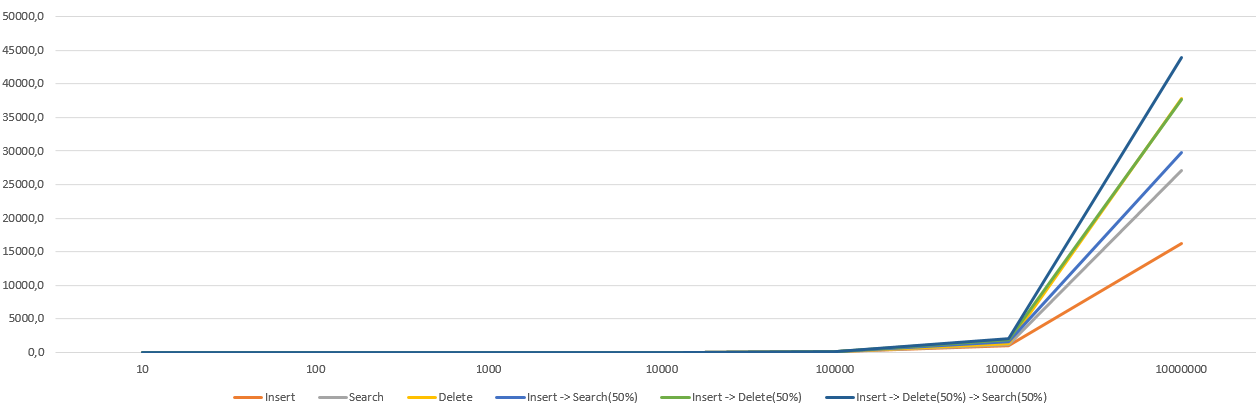




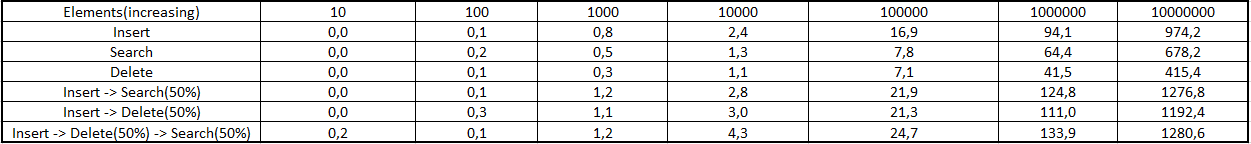


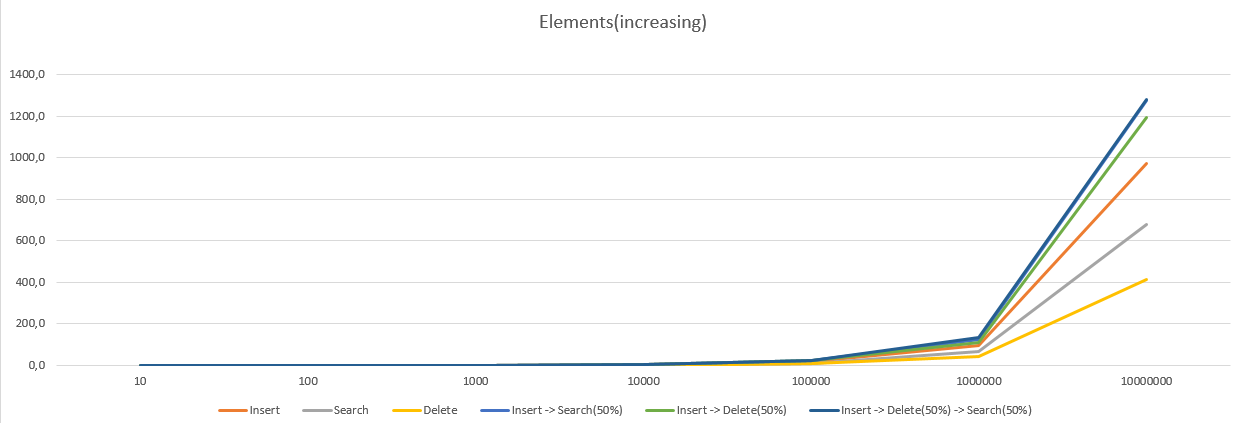


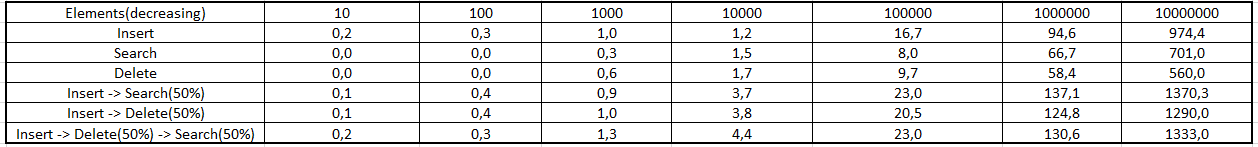


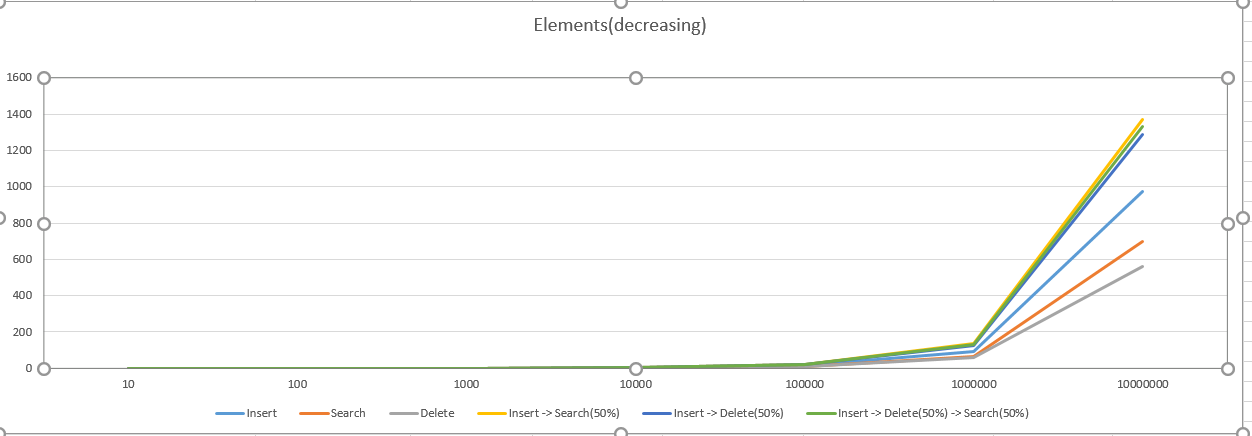


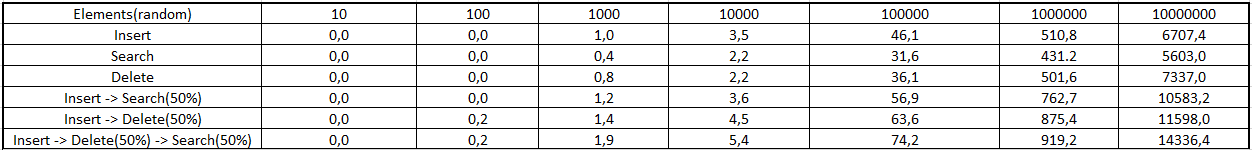
AVL Tree

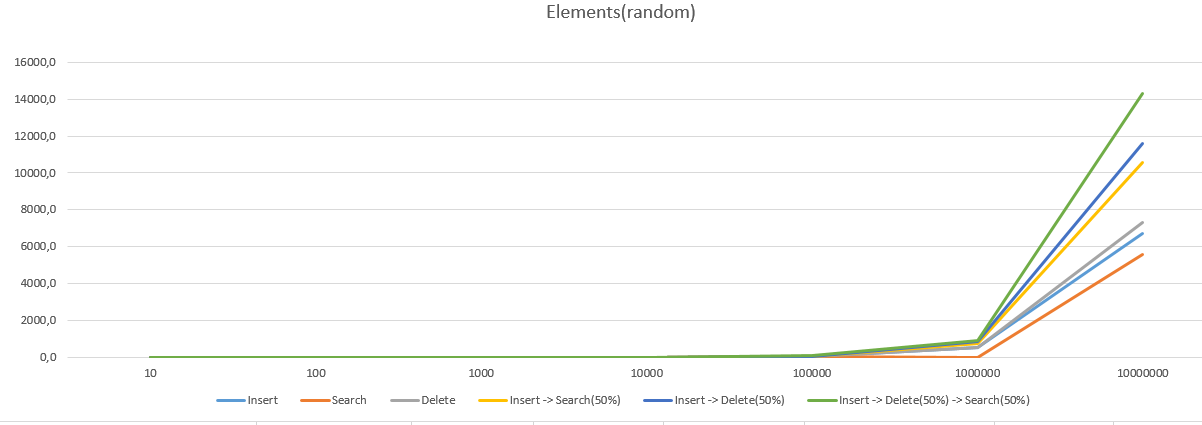








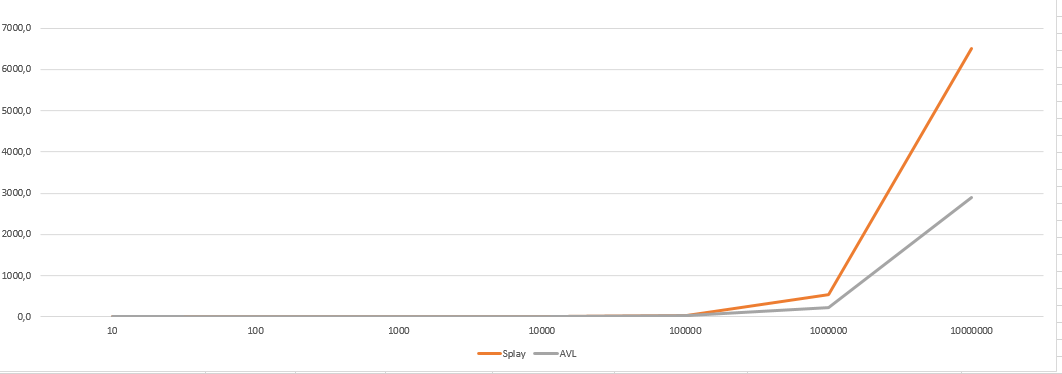




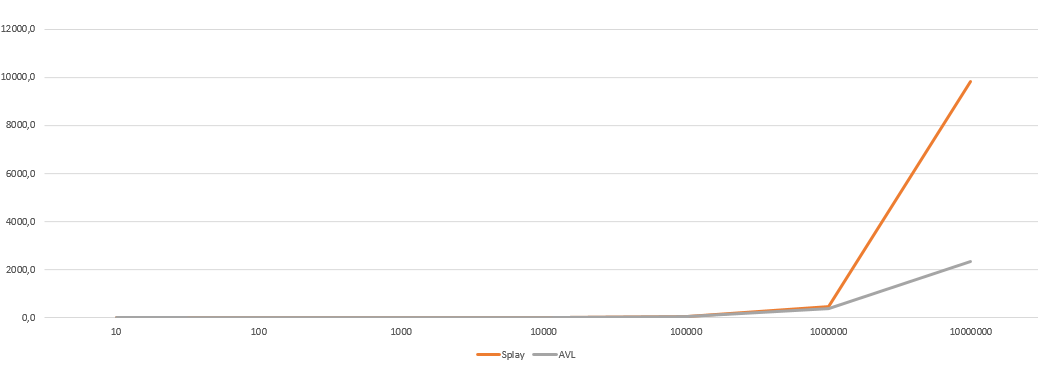
Judging by these diagrams, we can see that they practically do not differ.

Let's move on to comparing trees with each other:









It can be unequivocally said that the AVL tree is much more efficient than the Splay tree, since the Splay tree needs special conditions that facilitate its effective use (my test was not such)

When deleting, the same thing is traced, I will demonstrate this at the presentation.