Project 2

For task 1 a) I have implemented nested structures which helps doing the calculations later. For the random order in this binary tree I implemented choosing random number, there are two options – 0 or 1 and of this random number it depends where – right or left the new number will be put in the tree. For finding given value I implemented ‘find’ function together with ‘search\_node’ function which is recursive function. ‘search\_node’ is looking for the given number in root, right and left branches and returns true to the find function if it is found.

For task 1 b) I have also implemented nested structures, but this time the insert function takes the whole vector in which there is sorted array. For sorting the array I used the quicksort algorithm which I implemented in the previous project.I have used the vector from the standard template library. Insert function uses the recursive function helper. It repeats the process of taking the middle element as the root and choosing middle element for left and right branch untill the whole elements from vector are put. For finding element I used function which is looking for the greater key in the right branch and for the smaller in the left branch.

For task 1 c) I have used set from standard template library which is good type for the binary tree. But instead of find method I used ‘contains’ method which also does the same but it is easier to use.

I have also included the AVL Tree which is self-balancing binary search tree. Height difference between the left and right subtrees of any node must be at most 1. This property is maintained by performing rotations on the tree whenever the balance factor exceeds 1. To achieve this I implemented for this tree „rightRotation” and „leftRotation” functions. They are changing roots with child nodes and sides are also changed. There is also height function. I implemented insert function together with recursive ‘’insert\_helper’’. It inserts key into right or left based on their value and after this the balance factor is calculated and if it is greater than 1 it means that left subtree is bigger and rotation is necessary. If it is less than -1 then it means that right subtree is bigger and rotation need to be performed. For finding element I used function which is looking for the greater key in the right branch and for the smaller in the left branch.

I tested running times for these 4 types of trees while inserting values and finding the values. I created functions for this: ‘count\_insert’ and ‘count\_find’ which takes the size and the vectors which are storing the running times for implemented binary trees. Size is 2n-1 up to 16383 elements.

Inserting into every Tree is similiar at the beginning and there aren’t big differences. The differences starts at the array of size 127. Inserting into Tree A is much slower than the other Trees. Tree Standard is slower than TreeB and TreeAVL but the difference is smaller than in the situation of TreeA. TreeB is the fastest untill the end while TreeAVL has the second place.

For find function visible are starting too at the size of 127. Tree A starts losing because it needs again much more time because it is unbalanced. Standard Tree is the second worst but difference is smaller. Again TreeB wins until end but the difference between TreeB and TreeAVL is very small.

So from what I have done I can easily say that my implementation of TreeB is the best option from here. Properly ordered binary tree with inserting ordered array is much faster. Even while finding the values the difference is not big, the difference while inserting is much more noticeable because TreeAVL is being balanced during insertions and it takes time.