Programming assignment 2 – Emil Collén

Problem 5

In problem 5, where we are to make an algorithm to compare and determine whether two binary trees are isomorphic or not. For my implementation, I am not using a binary tree class, I only add nodes manually by pointing to their location all the way from the root and using getLeft(), getRight(), setLeft() and setRight(). This means my implementation does not contain general methods like add, delete etc. In my implementation I have a method in my Problem5 class called isIsomorphic(), that takes the root node of two trees and recursively climb down each tree to check whether every height/depth from the root is isomorphic. If it at some level isn´t isomorphic, it returns false and stops the recursion. If it is isomorphic, it keeps on climbing all the way down to the leaves and returns true all the way back up. This means that if the trees are isomorphic, we have to check every single node in the tree, making it O(N). Should the trees not be isomorphic, we still need to check every node until the point it is no longer isomorphic. In the method, we check both the structure and values of each node in the trees. For the so called IsoNodes, I have also used Strings for values, such as in the task description, but it doesn´t really matter what we use, since we only check whether two nodes have the same value or not. If one wants to change to integers instead, it is easily done by passing other nodes as arguments (and change the argument type for the method), such as BSTNodes and use "==" instead of .equals() in the second *else if* in the isIsomorphic method.

Problem 6

For this task, I devised my program in several different ways of building the trees. Some with starting from an empty tree and adding and removing randomly until the size reached a predefined number. Some other ways were when I started by creating a tree with a certain size and then removed and added elements at random. Depending on how I did this, the heights of the trees differed a bit in the BST. The latter is the one I will present below, as well as the worst-case scenario of the BST. That is, when the elements we add are already ordered. How we add and remove have a clear impact on the height of a binary search tree, while AVL-trees are immune to this affect. In the presented result below, the tree sizes are N = 1023.

What we can say is that when we use AVL-trees, we guarantee a certain height of the trees we make (log n), where n is number of elements in the tree. And since all other operations cost on a tree are also based on the height of the tree, we reduce these by having a balanced tree. That means we get O(log(n)). On the other hand, we have some extra cost each time we add or remove an element, as we balance it afterwards. Binary search trees however, they are easy to add and insert a new element, as it doesn´t care whether the tree stays balanced or not – we just add/remove and are done with it. This makes the operation itself easier, however, other operations become more costly since we don´t balance our tree. If we only add to such a tree, using randomized values, the tree doesn´t become that unbalanced. Should we remove nodes, we start to affect this fairly even tree. The remove method favors one side over the other, making it unbalanced over time, the more we add and remove from it. If we make N^2 random operations of add/remove on the tree, we get a height of $\sqrt{N}$. This is presented below. If we however insert elements in an ordered way, the tree becomes degenerate, meaning it is basically a linked list with the linear cost of O(N).

The expected height of the AVL trees is in between the log(n) and the formula below:

$$\mathbf{1.44 * \log_2(N+2) - 1.328}$$

The expected height from the formula is 13, while log(n) is just below 10. The result for all 1000 trees, with 1 million (~N^2) randomized adds/removes, turns out to be in between these two. As shown in figure 1 below, all trees ended up having a height of 11. This basically makes it an optimal tree to operate on, as we always can find an element with 11 operations or less.

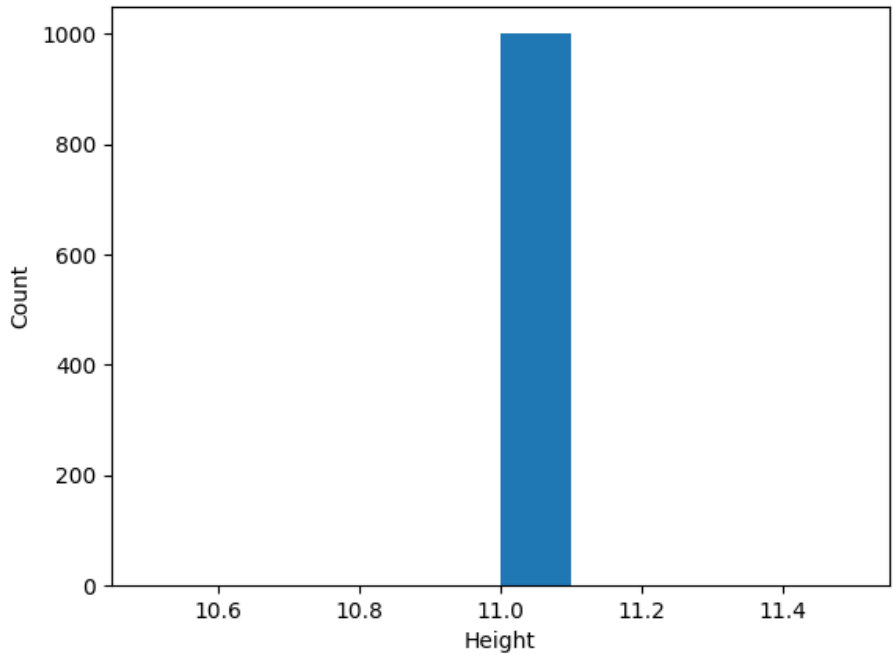

Figure 1 - Heights of 1000 AVL trees

We can compare this to the binary search tree. Below in figure 2 is the result for 1000 BST trees, after making 1 million (~N^2) randomized adds/removes. If we would have only built the tree without using removes, or even just made much fewer operations, the tree would be more balanced. However, the remove operation turns out to make the tree skewed as it favors only side when it removes. In this case, we always take the smallest subnode from the right subtree of the node to be removed, which after many repetitions will make the right subtrees smaller than the left. That is, the left side will be bigger than the right. We can see the difference in heights if we compare figure 2 with figure 3. In figure 3, we have only made adds to the tree. In this case, the tree is on average still twice as high as an AVL tree, yet much better than after doing 1 million adds/removes.

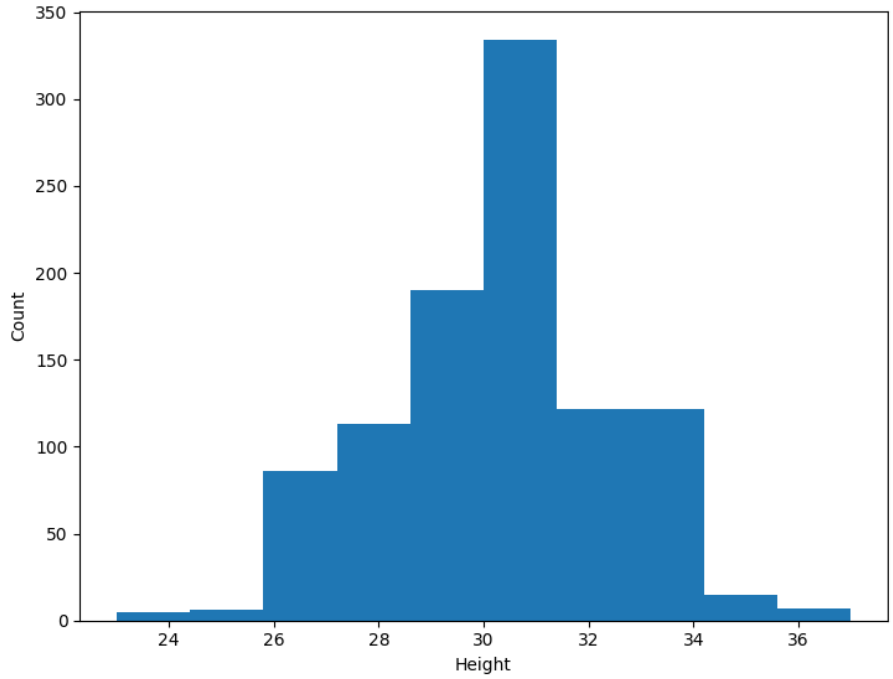

Figure 2 – Heights of 1000 BST:s

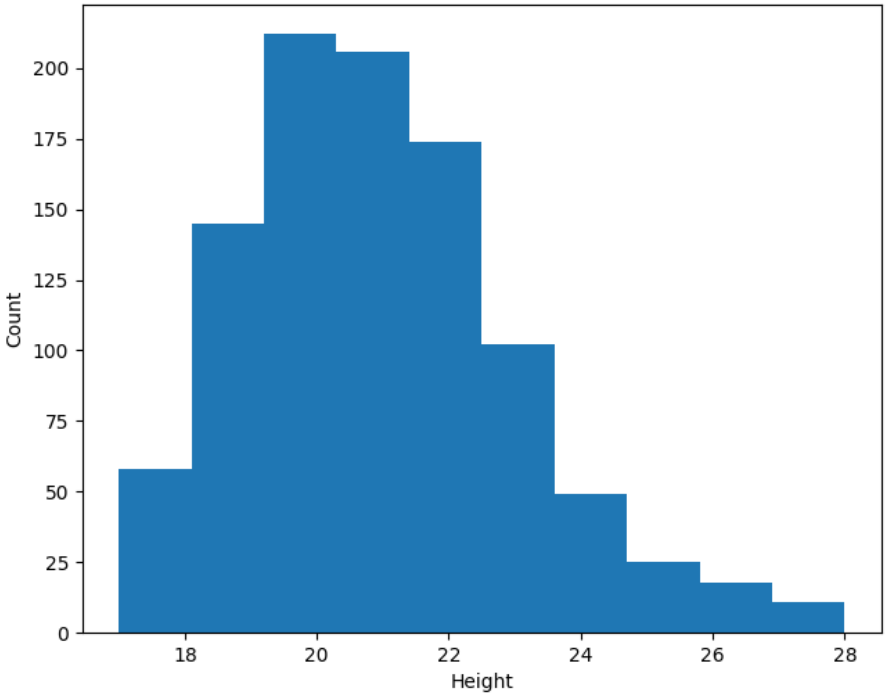

Figure 3 - Heights of 1000 BST:s, without removes

The difference between the BST and AVL becomes even more clear, the more sorted input we insert into our trees. When the entire input is sorted (either ascending or descending, doesn´t matter), then we get the worst-case scenario for the BST. In figure 4 below, the heights of degenerate BST:s is presented. When we insert sorted elements into an AVL, we get the same result as before (fig 1.). By the same logic, we also run a change of getting a perfectly balanced input, resulting in a balanced tree. This is however extremely unlikely.

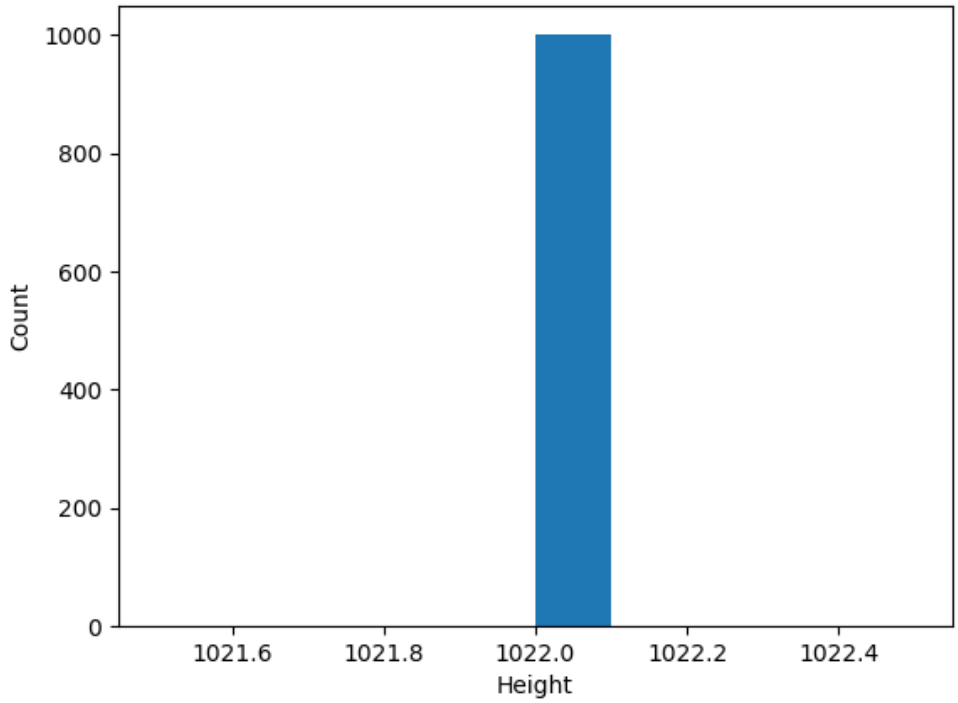

Figure 4 – Heights of degenerate BST:s

When we look at the time to perform operations on these trees, it differs to. But as previously mentioned, it is dependent on the height of the tree. This means that we can expect higher trees to take longer, and vice versa. We do however need to consider that every time we add or remove an element from the AVL, it has to balance the tree. This means we have extra cost compared to the BST. If we perform simpler operations, such as a find, we do not have the same extra cost. When perform such operations, only the height matters. The table below shows the average time it took to perform ~N^2 adds/removes as well as ~N^2 finds for each of the tree types.

	Time (ms)	
	BST	AVL
Time per add	0.0001637	0.0274616
Time per remove	0.000143	0.0276049
Time per find	0.00010035	0.0000882

Figure 5 - Time taken for different operations

From these results, we can say that the time to add and remove are very fast for the BST. We can also see that the difference in time taken between add, remove and find is not that big. When we compare this to the AVL, we can see that the add and remove operations take many times longer for the AVL, due to the balancing for each operation. The find operation however is faster. What we pay for in cost when we add and remove, we get a guaranteed short time for simple operations such as find. We have to keep in mind also that this is an average case for the BST with removes, should we have a worst-case scenario, the time for a find take many times longer. In trees where N = 1023 with heights 1022, we get that it takes 0.0032161 ms per find on average, which is more than 30x longer than the average. Even more, compared to the AVL.

What we can conclude regarding the result, is that if we use AVL trees, we pay some extra overhead cost when we add/remove elements, but we have a guaranteed balanced tree. In a regular BST, on average, we have fast adds/removes and we get an OK result for simpler and smaller applications where the insertions are fairly random, but we run a risk of a terrible worst-case scenario should it be sorted or even semi-sorted.