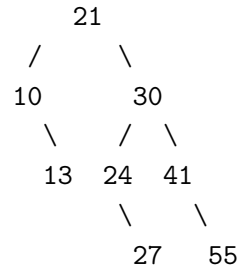
Data Structures, exercise 6, 23.-26.2.

Note: when programming do not use the following machines: melkki, melkinkari, melkinpaasi

1. (a) Draw smallest and largest AVL-trees, which have height of 3 and 4. You may choose the values of the nodes freely as long as they meet the binary search tree property.

   (b) Show that the tree below meets the AVL-tree property.



2. (a) Show how AVL insert works, when the following keys are added to an empty tree, in the following order: 41, 38, 31, 12, 19, 8, 27 and 49. Show how rotations are done in each insert.

   (b) Do as above, but the keys are inserted to an empty tree in reverse order, 49, 27, 8, 19, 12, 31, 38 and 41.

3. AVL-delete works as follows: First a node is deleted with normal binary search tree delete operation. This may cause an unbalance to the tree. If there is unbalance, the problems are in path from the deleted node to the root of the tree. So, to correct the problems, each node in that path (starting form the parent of the deleted node) are checked and if an unbalance is found, the necessary rotations are made. In AVL-insert, a single rotate or a double rotate is enough to return the balance. With AVL-delete this is not the case. The algorithm must check each node form the path to the root node of the tree. See eg. `www.cs.uga.edu/~eileen/2720/Notes/AVLTrees-II.ppt`

   Show how AVL-delete works when it removes keys 12, 49, 31 and 8 from the tree that results in 2 (a).

   (b) Remove the keys 12, 8, 49 and 41 (in this order) from the tree that results in 2 (b).

4. Implement a binary search tree with operations insert and search using Java. The keys saved to the tree are of type `long`. You don't necessarily need to implement the parents links in this exercise.

   For testing it is useful to implement an algorithm, that prints the content of the whole tree. Most simple way is to print the nodes using in-order. Question 5 of exercise 5 shows another way how a tree can be visualized.

   Bonus: implement an algorithm, that tests the height of the tree. Testing the height is done the simillarly as the counting the sum of nodes in exercise 5, question 2.

You can mark this question as done also without implementing the height testing algorithm. You'll need the algorithm anyway in the next question.

5. Testing the efficiency of a tree

   (a) The weakness of an ordinary binary tree is the fact, that in worst case tree can be high. If the keys are to be inserted many at the same time, then one solution to the problem is first to shuffle the keys to a random order.

   Let's use the binary tree of the previous question and examine the height of the tree with different amounts of keys, when the keys are shuffled before insertion. In practice you can insert random keys to he tree instead of shuffling. It is easy to generate random numbers using Java class called Random.

   Measure using several amounts of keys, how close the height of the tree is to the optimal height.

   (b) Learn how to use Java class TreeSet, which is an implementation of a balanced binary search tree.

   Compare empirically the efficiency of the TreeSet-tree and your own tree. Do three sets of measurements:

   - Add to both trees n random keys. Measure the time that went to insertion operations.
   - Add to both trees m random nodes and run n random search operations. Measure only the time that went to the search operations.
   - Add to both trees keys $1, 2, 3, \ldots, m$ in increasing order and run n random search operations. Measure only the time that went to the search operations.

   In all the cases test several values for m and n and have emphasis on large amount of input. Execution time of the algorithm can be measured the same way as week's 4 questions.

   What do we learn from the measure results? How do the results confront with O-analysis?

   Note: if you dislike Java you can use any other language for this and the previous question. When doing so, you'll need to find a library implementation that resembles Java's TreeSet in the language of your choice. In the STL of C++ there is (set), but for example in Python most likely a ready implementation is not found.

6. (a) The time requirement of the binary search tree operations *min* and *max* (finding the smallest and the largest node) is $\mathcal{O}(h)$, where h is the height of the tree. Change the tree implementation so, that the time requirement of operations *min* and *max* is only $\mathcal{O}(1)$ and the time requirement of other operations stays the same.

   (b) The time requirement of the binary search tree operations *succ* and *pred* is $\mathcal{O}(h)$, where h is the height of the tree. Change the tree implementation so, that the time requirement of operations *succ* and *pred* is only $\mathcal{O}(1)$ and the time requirement of other operations stays the same.

   (c) What are the benefits and the disadvantages of the previous changes?