CS 15 AVL Preview

Introduction

As we saw in lecture, AVL trees are just a kind of binary search tree (BST) but, in addition to the BST invariants, they have the additional invariant that the tree is AVL balanced at every node; that is, the heights of the left and right subtrees of every node differ by at most 1. This property ensures the height of the tree is at most 2 log n (where n is the number of nodes in the tree) and thus both the worst and average time complexity of insert, search, and delete are O(log n). The key task of an AVL tree implementation is to maintain the balance after each modification of a tree. Refresh your knowledge on AVL trees by answering the following questions:

Questions

- Is the first tree given below a valid AVL tree? Why or why not?
- The second drawing below depicts a binary search tree that is balanced, i. e., it is a valid AVL tree. Insert the key 18 into the tree and do any rotations needed to balance the tree. Draw the final tree.
- The previous example caused an imbalance by inserting in the right subtree of the left child. What key value could you insert in Figure 2 to cause a different type of imbalance?
- There are two more possible ways to unbalance a tree. What values could you insert to create these?
- Finally, write down the steps, preferably as bullet points, you took to ensure that the tree remains balanced after the insertion operation in the second question. How would the algorithm change in the 4 possible cases of imbalance?

Figure 1

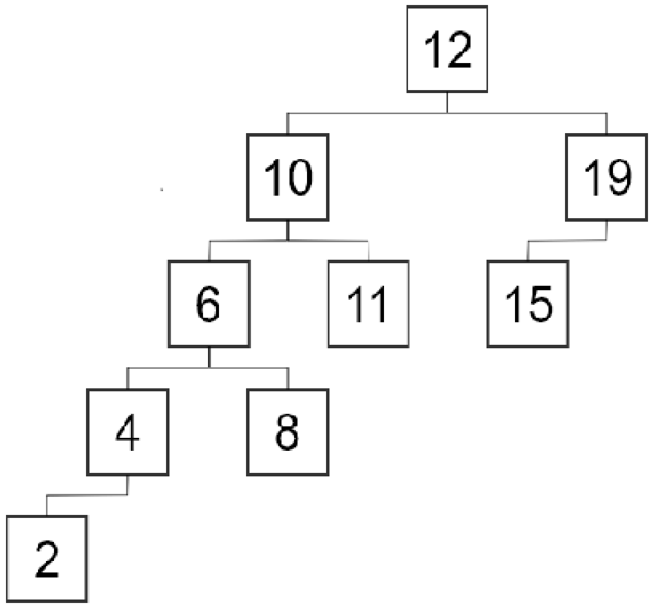
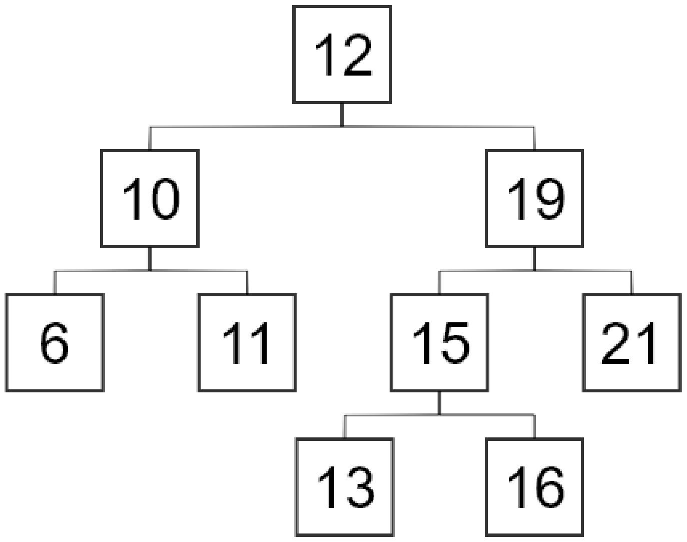


Figure 2


12
10
19
6
11
15
21
13
16