Exercises: Balanced Search Trees

Questions

1. If you are rusty on binary search trees, then see exercises on this topic in my COMP 250 course public web page.

2. Give a recurrence for the number of possible binary search trees with n keys. You may assume the keys are {1,2, …, n}. Hint: the root of the binary search tree can be any one of the n keys. You need to consider each of these n possiblities.

3. Consider any two binary search trees T1 and T2 with the same set of keys, {1,..,n}. We can always transform T1 to T2 using rotations. Sketch a general method for doing this.

4. In class I claimed that the rotateRight(root) algorithm maintains the binary search tree property, even if root is not the root of the BST, but rather it is an internal node of the tree i.e. it's the root of a subtree. Why is this true?

5. In class we saw an algorithm for rotating Right. Give the corresponding algorithm for rotating left.

6. What is the maximum number of nodes in an AVL tree of a given height h?

7. Draw an AVL tree of height 3 that has the minimum number of nodes, that is, the minimum number of nodes of all AVL trees of height h. Note that this tree need not be unique. Draw an AVL tree of height 4 that has the minimum number of nodes. Think how you would draw an AVL tree of height h that has the minimum number of nodes.

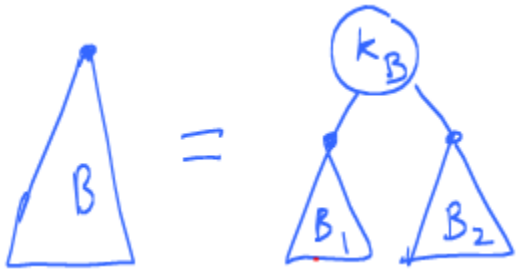
8. Give a recurrence for the minimum number of nodes in a valid AVL tree, as a function of the tree height. Hints: First, check out the solution for the previous question. Let h be the height and let f(h) be this minimum number of nodes. Then f(0) = 1 i.e a tree with just a root, f(1) = 2 i.e. a root node and one child.

9. Suppose we have an AVL tree and we insert a node that causes the tree to become unbalanced (i.e. so it no longer satisfies the AVL balance condition). In fact there could be multiple nodes in this tree that now fail to satisfy the AVL balance condition.

    a. Given an example in which this happens.
    b. Show why such unbalanced nodes must lie on a common path from the root to a leaf.

c. In the example given in the solution of (a), one of the unbalanced nodes is an outside case and the other is an inside case. The method that I gave in class did not address this situation in (c) and so you don't yet know which one of the imbalances to handle first. The way the algorithm actually works is that once it inserts the new node, it needs to update the heights and check if the tree is balanced. It does so by climbing up the tree from the inserted node to the root, and as it climbs, it increments the heights of the nodes (note that it needs to store a height variable at each node) and it checks if a node is unbalanced - just by applying the definition, i.e. comparing the heights of the children. Thus it rebalances the lowest node first. Argue why rebalancing the lower node automatically rebalances any imbalanced upper node(s).

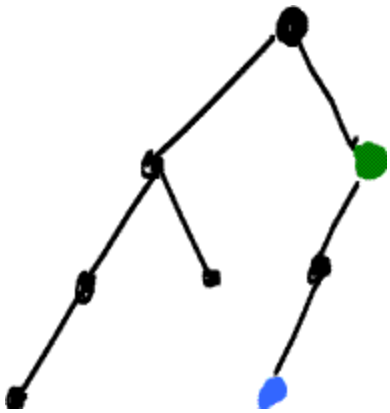
10. At the end of the discussion of AVL trees in class, we saw two ways in which insertion could lead to a node becoming unbalanced: namely inside and outside. The outside case was handled easily with a single rotation, as shown in class. The inside case requires two rotations (sometimes called a double rotation). Using the example in class, show what these two rotations are. Hint: write tree B as:



11. Starting with an empty tree, construct an AVL tree by inserting the following keys in the order given: 2, 3, 5, 6, 9, 8, 7, 4, 1. If an insertion causes the tree to become unbalanced, then perform the necessary rotations to maintain the balance. State where the rotations were done.

12. Use the above sequence to construct a 2-3 tree.

13. Here is another example that is related to Question 9. Suppose you insert a node (blue) into a valid AVL tree, making it imbalanced. Note that the root of the tree below IS balanced. Rebalance the tree.



14. The defining property of binary search trees is that, for each node in the tree, any keys in the left subtree are less than the key in that node and any keys in the right subtree are greater than the key at that node. Wikipedia http://en.wikipedia.org/wiki/Binary_search_tree also throws in the condition that the left and right subtrees are binary search trees, but that "recursion condition" should not be part of the definition because it is redundant i.e. the recursion property follows from the other definition properties that are listed.

    Now consider the following alternative attempt at a recursive definition for a binary search tree with unique keys): a tree is a BST if (1) the key at the left child of the root is less than the key at the root which in turn is less than the key at the right child of the root and (2) the left subtree is a binary search tree and the right subtree is a binary search tree. Show that this definition is inadequate i.e. there are trees that satisfy this definition but are not binary search trees.

Answers

1. Solutions are posted there too.

2. See my COMP 250 lecture notes, lecture 20 page 2.

3. Yes we can. We can transform any binary tree into a "left branch only" tree e.g. by applying left rotations to any node that has a right child. So, suppose we transform transform T1 into a "left branch only" tree. Now note that there must a sequence of rotations that can perform the opposite transformation, namely transform the "left branch only tree back to T1 since, as we saw in class, rotations are invertible. (Careful: the inverse of a rotateLeft is a rotateRight, but the root parameters will be different in the two cases.)

    Similarly, consider the transformation of T2 to a left branch only tree, which is the same left branch only tree that we obtained from T1. (It has to be the same left-branch-only tree, since the keys are ordered and so there can be only one left-branch only tree for a given set of keys.) Again, there must be a sequence of rotations that can take the left branch only tree to T2.

Finally, to transform from T1 to T2, we can apply the sequence of transformations that take T1 to the left branch only tree, and then apply the sequence of transformations that takes the left branch only tree to T2. Together, this gives a transformation that takes T1 to T2.

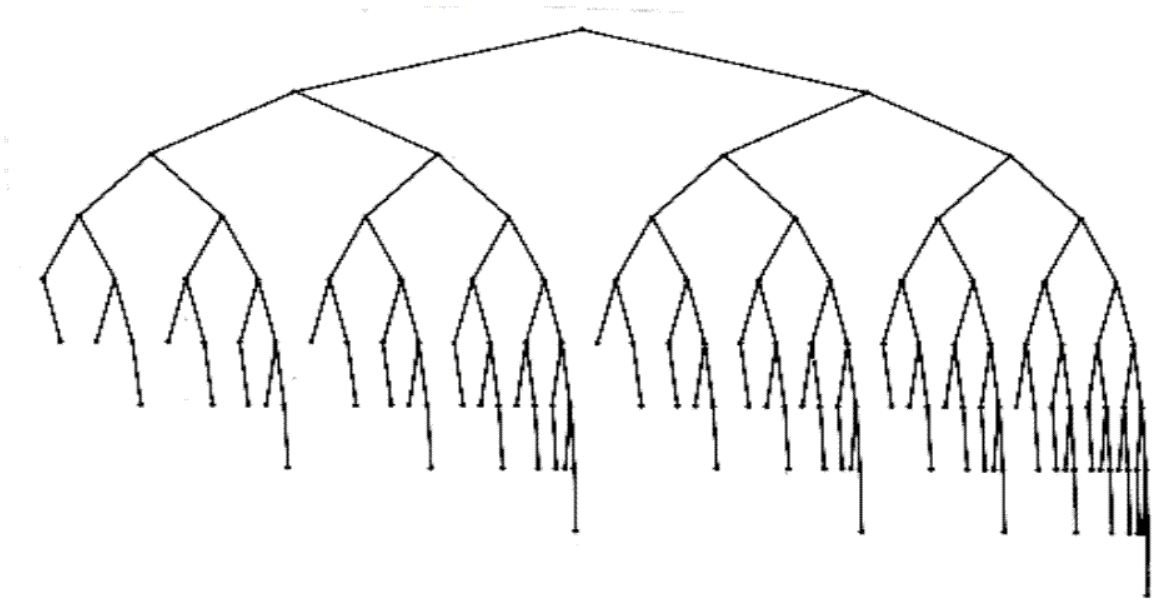
4. The smallest and largest key within this subtree define an interval of keys within the ordering of all the keys in the tree. The only keys in the tree that fall in this interval are those keys within this subtree. Changing the parent-child links of this subtree by rotating doesn't affect the interval of keys defined by the subtree. And since the rotation doesn't affect the parent-child links that outside of the subtree, it doesn't affect the two intervals outside of the interval defined by the subtree, that is, the interval less than that of the subtree and the interval greater than that of the subtree. (If this is not clear conceptually, review what how an in order traversal of a binary search tree works and note that any subtree defines an interval in the ordering of all the nodes in the tree.)

5. rotateLeft(root){

```
        newRoot =  root.right
        root.right =  newRoot.left
        newRoot.left = root
        return newRoot
}
```

6. This would be a complete binary tree of height h. Such a tree has 2^{h+1} – 1 nodes. This is COMP 250 stuff…

7. Here is an AVL tree of height 9 that has the minimum number of nodes. An AVL tree of height 8 that has the minimum number of nodes (of all AVL trees of height 8) is the tree found in the right subchild of the root node. Similarly, an AVL tree of height 7 that has the minimum number of nodes is the tree found in the left subtree of the root.

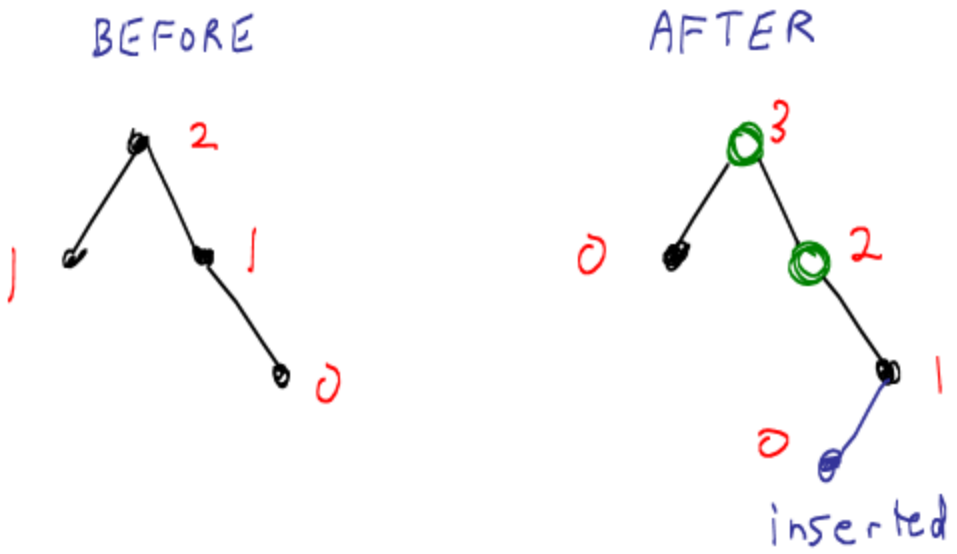


Note that the shape of the tree is nice for visualization, but there are many AVL trees that have this distribution of pathlengths and heights. For any node in the above tree, we can swap its left and right children to give the tree a different shape. Of course, we would need to reorder the keys within this new shape so that the keys are in order (i.e. satisfy the properties of a binary search tree). But we can always do this.

8. In a valid AVL tree, the heights of left and right subtrees of any node can differ by at most 1. Indeed, that is the definition of a valid AVL tree. This balance condition is true in particular for the heights of the left and right subtrees of the root node. Now if an AVL tree of height h has the minimum number of nodes of all AVL trees of height h, then its left and right subtrees must themselves satisfy this minimum condition as well. Thus, we can write f(h) in terms of the heights of the subtrees. To meet the minimum condition, the subtrees must differ in height, in particular, they must be of height h-1 and h-2 . (If they were both height h-1, then we could reduce the number of nodes by removing some of the leaf nodes of one of the subtrees, while still maintaining the AVL property.) Thus, the recurrence is f(h) = f(h-1) + f(h-2) + 1. We need the "+1" because we have to count the root.

9.

   a) Here is an example. On the left is "before" and on the right is "after" (the blue node was inserted). The two green nodes are unbalanced in that the heights of the left and right subtrees of each differ by 2. NEED TO EDIT – THE BEFORE TREE ON THE LEFT BRANCH SHOULD HAVE A 0.



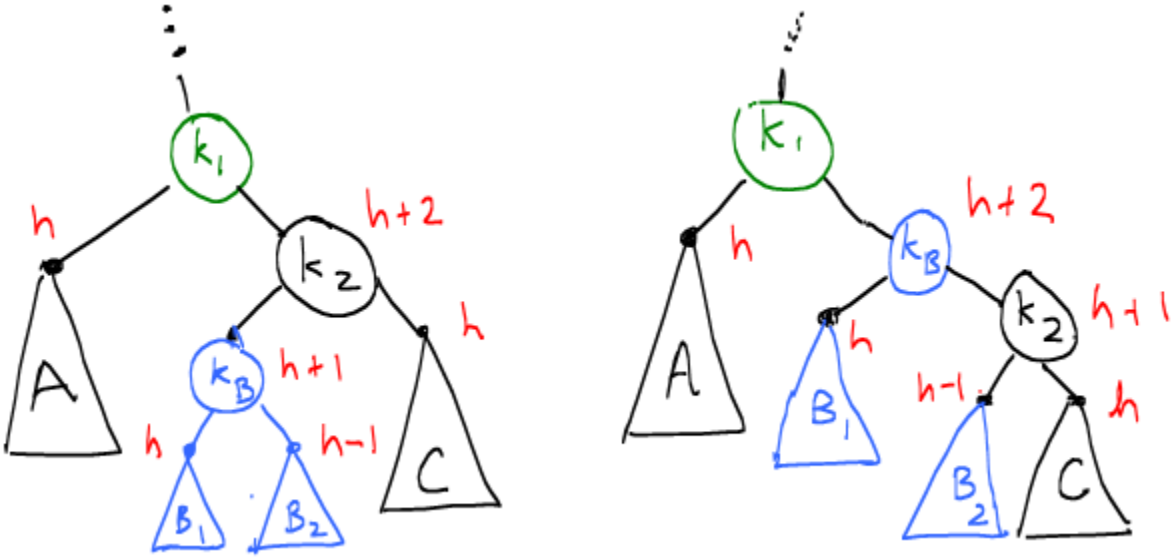
b) The second part of the question concerns the ancestor relationship between unbalanced (green) nodes. The argument you give should hold for all cases of course, not just for the example above.

Before inserting the node, we had a valid AVL tree. We now insert a new node and two (or more) nodes become unbalanced. For any two unbalanced nodes, you are asked to show that one must be an ancestor of the other. To see why this true, assume the opposite namely that the claim is NOT true: namely, assume the inserted node is a descendent of only one of these two newly unbalanced nodes (since if it was a descendent of both, then one of the nodes would have to be a descendent of the other). But for the newly unbalanced node that is (allegedly) not an ancestor of the inserted node, the heights of the left and right subtrees would not have been affected by the insertion. So the insertion could not have caused that node to become imbalanced, which contradicts our assumption. Hence, the claim holds.

c) The tree was balanced before the insertion. Observe that when you rebalance the lower imbalanced node, you ALWAYS reduce the height of this node by 1 (verify why) which makes this node have the same height that it had before the new node was inserted. Since all ancestors were balanced prior to the insertion, they must be balanced now too.

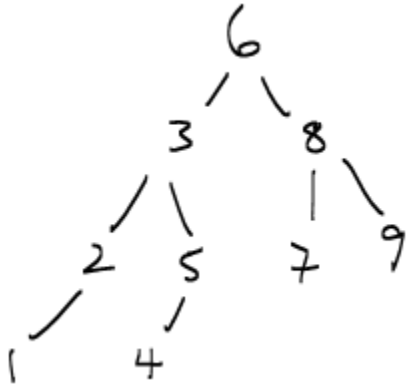
10. Using the hint, suppose rewrite the tree as shown on the left below and label the heights as done there. Then rotateRight(k2) gives us the tree on the right below. This now reduces the outside case that was discussed in class. So we can balance the subtree rooted at k1 by

rotateLeft(k1).

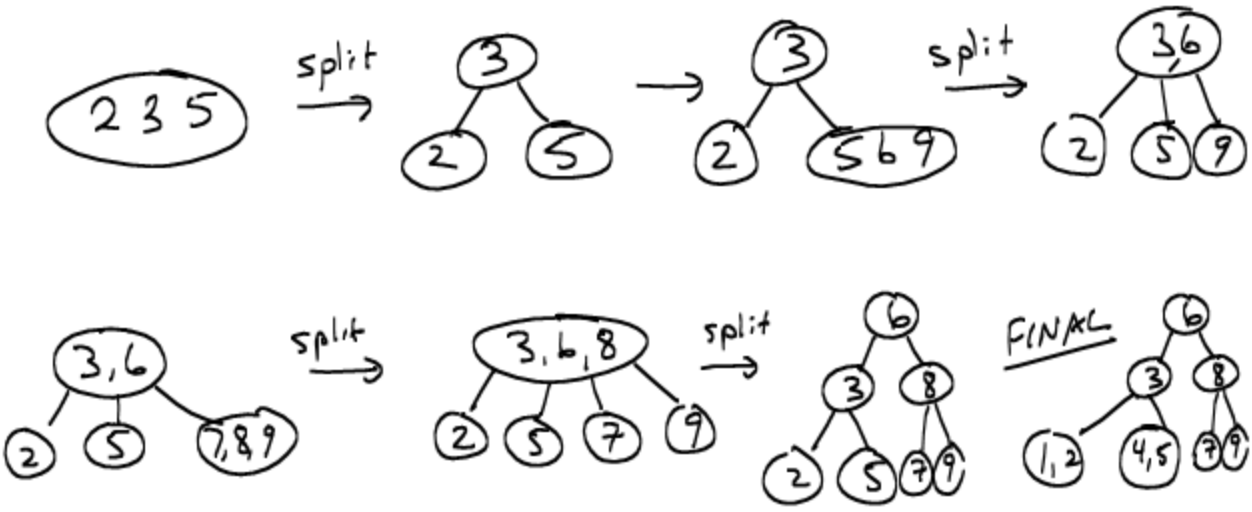


[Addendum: Jan. 31] Notice that I could have labelled B1 as having height h-1 and B2 as having height h, rather than the other way around i.e. it would depend on which of these subtrees contains the newly inserted key. If I labelled B1 and B2 as I just said, then the rotateRight(k2) now would cause the kB tree to become unbalanced but fortunately the subsequent rotateLeft(k1) would fix that. In this sense, it doesn't matter which of the B1 or B2 took the newly inserted key.

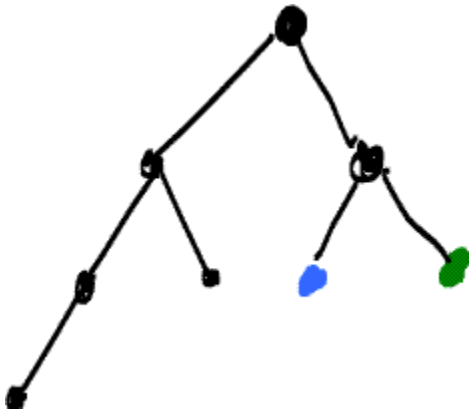
11. Add(2), add(3), add(5), rotateLeft(2), add(6), add(9) , rotateLeft(5) and note that this solves the imbalance at 3 also, add(8), rotateLeft(3), add(7), rotateRight(9), add(4), add(1). The final AVL tree is



12.



13. One possible confusion that you may have had is thinking that this was an inside case. In fact, it is an outside case because "inside/outside" is defined relative to the (lowest – in this case there is only one) node that becomes imbalanced. After rebalancing we have the tree below. I have left the colors in just so its clear which node is which.



14. The tree below satisfies the recursive definition, but it is not a binary search tree. The fail happens because of the red node, namely the is in the left subtree of the root but it is greater than the key at the root.