MIDDLE EAST TECHNICAL UNIVERSITY, NORTHERN CYPRUS CAMPUS
CNG213 Data Structures – Worksheet 11

AVL Trees

Purpose:
This worksheet is concerned with AVL trees. An AVL tree is a self-balancing binary search tree. In an AVL tree, the heights of the two child subtrees of any node differ by at most one; therefore, it is also said to be height-balanced. Lookup, insertion, and deletion all take *O(log n)* time in both the average and worst cases, where n is the number of nodes in the tree prior to the operation. Insertions and deletions may require the tree to be rebalanced by one or more tree rotations. You are not expected to complete the entire worksheet in class. Work on as many problems as you can; the remaining problems you can use for practice and to test your understanding of AVL trees.

1) Print the tree structure of a binary tree – implement `PrintTree()` function. You can either print it out with the root at the topmost or with the root at the leftmost position. For the former, you need to perform a breadth-first traversal of the binary tree, for the latter a normal in order traversal will work.

2) Use the above C function to show how the structure of a tree grows as you insert new values into a binary search tree.

3) Implement a recursive C function `TreeHeight()` for finding the height of a binary search tree. The height of a binary search tree is the length of the longest path from the root to a leaf. **Hint:** Modify your function `PrintTree()` to determine the height of each node's left and right subtree.

4) The balance factor of a node is the height of its left subtree minus the height of its right subtree, and a node with balance factor 1, 0, or -1 is considered balanced. A node with any other balance factor is considered unbalanced and requires rebalancing the tree. The balance factor is either stored directly at each node or computed from the heights of the subtrees. Write a C function that computes the balance factor for each subtree and prints it out along with the tree structure.

5) Write a C function which checks whether a tree is an AVL tree.

6) Write a C function to implement insertion of a node with tree rebalancing. After inserting a node, it is necessary to check each of the node's ancestors for consistency with the rules of AVL. For each node checked, if the balance factor remains -1, 0, or 1 then no rotations are necessary. However, if the balance factor becomes 2 or -2 then the subtree rooted at this node is unbalanced. If insertions are performed serially, after each insertion, at most two tree rotations are needed to restore the entire tree to the rules of AVL.

   There are four cases which need to be considered, of which two are symmetric to the other two. Let P be the root of the unbalanced subtree. Let R be the right child of P. Let L be the left child of P.

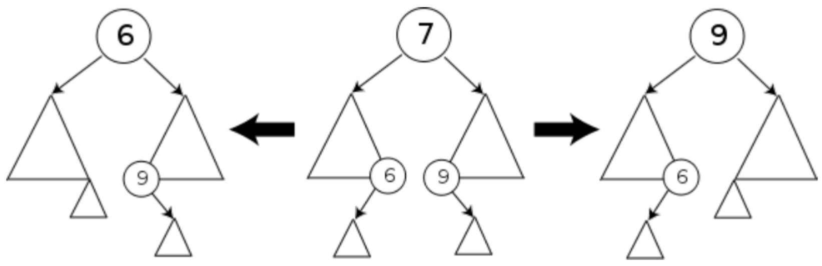
   Right-Right case and Right-Left case: If the balance factor of P is -2, then the right subtree outweighs the left subtree of the given node, and the balance factor of the right child (R) must be checked. If the balance factor of R is -1, a left rotation is needed with P as the root. If the balance factor of R is +1, a double left rotation is needed. The first rotation is a right rotation with R as the root. The second is a left rotation with P as the root.

   Left-Left case and Left-Right case: If the balance factor of P is +2, then the left subtree outweighs the right subtree of the given node, and the balance factor of the left child (L) must then checked. If the balance factor of L is +1, a right rotation is needed with P as the root. If the balance factor of L is -1, a double right rotation is needed. The first rotation is a left rotation with L as the root. The second is a right rotation with P as the root.

7) Write a C function to implement deletion of a node with tree rebalancing. The node is a leaf, remove it. If the node is not a leaf, replace it with either the largest in its left subtree (inorder predecessor) or the smallest in its right subtree (inorder successor), and remove that node. The node that was found as

replacement has at most one subtree. After deletion, retrace the path back up the tree (parent of the replacement) to the root, adjusting the balance factors as needed.

As with all binary trees, a node's in-order successor is the left-most child of its right subtree, and a node's in-order predecessor is the right-most child of its left subtree. In either case, this node will have zero or one children. Delete it according to one of the two simpler cases above.



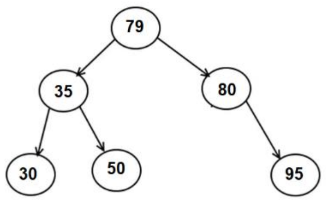
In addition to the balancing described above for insertions, if the balance factor for the tree is 2 and that of the right subtree is 0, a left rotation must be performed on P. The mirror of this case is also necessary.

The retracing can stop if the balance factor becomes -1 or 1 indicating that the height of that subtree has remained unchanged. If the balance factor becomes 0 then the height of the subtree has decreased by one and the retracing needs to continue. If the balance factor becomes -2 or 2 then the subtree is unbalanced and needs to be rotated to fix it. If the rotation leaves the subtree's balance factor at 0 then the retracing towards the root must continue since the height of this subtree has decreased by one. This is in contrast to an insertion where a rotation resulting in a balance factor of 0 indicated that the subtree's height has remained unchanged.

Please note that from this worksheet, you need to submit the solution of the following question (8)

8) Write a program which will do following;

a) A function `CreateAvlTree()` that generates 6 random numbers between 0 and 100 and creates corresponding AVL tree. For instance; if the generated random numbers are 50,30,35,80,79,95 then the following AVL tree will be created:



b) A function `checkGuess()` that takes the created tree as an input, and then asks user to guess a number between 0 and 100. Subsequently, this function will check if that guessed number is existing in the AVL tree or not. If it exists, it will return 1. Otherwise, it will return 0.

c) `PrintTree()` function which will print the Tree structure.

Please note that you can write helper functions.

Possible sample run:

```
Guess a number: 80
You win!
Tree structure is: 30,35,50,79,80,95
```

Possible sample run:

```
Guess a number: 2

You lose!

Tree structure is: 30,35,50,79,80,95
```