University of Illinois @ Urbana-Champaign

CI 487: Data Structures for CS Teachers

Implementation #5:
Generic AVL Tree

---

1 Objectives

The objectives of this assignment are as follows:

1. Understand how to transform a BST into an AVL tree and the utility of doing so.
2. Learn how to construct test cases and use the debugger in order to verify the AVL tree is being implemented correctly.
3. Implement a novel recursive search algorithm that will report the number of steps to find a node in the tree.

2 Structures and Specifications

2.1 Step 0: Understanding the Code You're Given

The code you are given is quite similar to the BST starter code you were given last week with a few notable exceptions:

1. `TreeNode<E>`: The tree node class now contains a height attribute which you will use later in the assignment to keep track of that nodes height within the tree.
2. `inorderTraversal`: You are given a method that prints an in order traversal of the tree. This can be used to display the output of the tree in addition to verifying that your rotations aren't reordering the nodes in such a way that the BST rule is being broken.
3. `remove/findMinNode`: You are provided the BST remove method ad the findMinNode method. You will be modifying the former in step 1 in order to transform it into an AVL removal method.
4. `insert`: Your are also provided a working BST insertion method and, as was the case with remove, you will be transforming it into an AVL insert.

As with all past assignments, it his recommended that you spend some time reading the existing code in order to orient yourself before moving on to the next step(s).

2.2 Step 1: Modifying the BST Remove and Balance Methods

The first step of our process of transforming a traditional BST into an AVL tree is modifying the Insert and remove methods to include two things: 1) the ability to update the heights of nodes as we recursively unwrap and 2) perform rebalancing as the recursion unwraps. Below is the annotated psudeo code which identifies the original portions from the BST class as well as the lines you will add in order to accomplish the two tasks just described.

Algorithm AVL: Insert

```
1: procedure INSERT(CurrNode, NewNode)
2:     if CurrNode is Null then                          (1)
3:         Return NewNode
4:     else if NewNode < CurrNode then
5:         CurrNode.Left = Insert(CurrNode.Left, NewNode)
6:     else
(2)        CurrNode.Right = Insert(CurrNode.Right, NewNode)
8:     UpdateHeight(CurrNode.Height)
9:     Return Balance(CurrNode);                         (3)
```

1. The first block is our usual insertion step.
2. If we have reached this line, we are performing the recursive unwrapping after inserting the node. Given we just inserted a node it is possible that the heights of the intermediary nodes we traversed over may need to have their height updated.
3. Inserting a node may have caused the subtrees under the node we're on to come unbalanced so we should use the Balance method to rebalance if needed.

Algorithm AVL: Remove

```
1:  procedure REMOVE(CurrNode, RemovalNode)
2:
3:      if CurrNode is Null then
4:          return Null
5:
6:      if CurrNode < RemovalNode then
7:          CurrNode.Right = Remove(CurrNode.Right, RemovalNode)
8:      else if CurrNode > RemovalNode then
9:          CurrNode.Left = Remove(CurrNode.Left, RemovalNode)
10:     else
11:         if Curr.Left is Null then
12:             Return CurrNode.Right
13:         else if Curr.Right is Null then
14:             Return CurrNode.Left
15:         else
16:             MinNode = FindMinNode(CurrNode.Right)
17:             CopyData(MinNode, CurrNode)
18:             Curr.Right = Remove(CurrNode.Right, MinNode)
19:
20:     UpdateHeight(CurrNode)
21:     Return Balance(CurrNode)
```

1. The first block is our usual insertion step.
2. After removing a node, the actual height of the node might have changed so we should update the current node's height.
3. We should also call the balance method as removing a node might have caused a portion of the tree that was previously balanced to become unbalanced.

Your Task: Modify the existing recursive `insert` and `remove` methods to preform the height update and rebalancing steps detailed above. The equation for calculating the height of a node is as follows:

$$\text{Node.Height} = \text{Max}(\text{Node.Left.Height}, \text{Node.Right.Height}) + 1$$

Recall that, if a node isn't to the left or the right of a node, then we treat it's height as -1. Additionally, procedure for implementing the balance method is detailed in the next section.

Hint: At this point, it may be useful to create a method that takes a reference to a node and returns -1 if it's null and it's height attribute if its not.

2.3 Step 2: The Balance Method

Algorithm 1 AVL: Balance

```
1:  procedure BALANCE(N)
2:      if BF(N) > 1 then
3:          if BF(N.left) < 0 then
4:              N = RotateLeftRight(N)
5:          else
6:              N = Right(N)
7:      else if BF(N) < -1 then
8:          if BF(N.right) > 0 then
9:              N = RotateRightLeft(N)
10:         else
11:             N = Left(N)
12:     Return N
```

Your Task: Implement the balance method shown to the left. This method will be the control center which will determine: 1) if the tree is unbalanced with respect to N and 2) select the correct rotation if rebalancing is needed. It will do this using the balance factor which in turn uses height. The equation for calculating balance factor is as follows:

$$BF(N) = N.Left.Height - N.Right.Height$$

Recall that, if a node isn't to the left or the right of a node then we treat it's height as -1.

Hint: At this point, it may be useful to create a method that can be used to calculate the balance factor of a node.

2.4 Step 3: Rotations

Algorithm 2 AVL: Right Rotation

```
1:  procedure RIGHT(N)
2:      Tmp = N.Left
3:      N.Left = Tmp.Right;
4:      Tmp.Right = N
5:
6:      UpdateHeight(N)
7:      UpdateHeight(Tmp)
8:
9:      Return Tmp
```

Algorithm 3 AVL: Left Rotation

```
1:  procedure LEFT(N)
2:      Tmp = N.Right
3:      N.Right = Tmp.Left;
4:      Tmp.Left = N
5:
6:      UpdateHeight(N)
7:      UpdateHeight(Tmp)
8:
9:      Return Tmp
```

Algorithm 4 AVL: LeftRightRotate

```
1: procedure LeftRightRotate(N)
2:     N.Left = Left(N.Left)
3:     Return Right(N)
```

Algorithm 5 AVL: RightLeftRotate

```
1: procedure RightLeftRotate(N)
2:     N.Right = Right(N.Right)
3:     Return Left(N)
```

Your Task: Implement each of the rotations using their respective psudeocode presented above.

1. `private TreeNode<E> leftRotate(TreeNode<E> node)`
2. `private TreeNode<E> rightRotate(TreeNode<E> node)`
3. `private TreeNode<E> leftRightRotate(TreeNode<E> node)`
4. `private TreeNode<E> rightLeftRotate(TreeNode<E> node)`

It is recommended that you develop insertion test cases that you know will trigger certain rotations and then investigate each method in the debugger in order to: 1) verify that method is being called and 2) that the tree is being properly updated.

2.5 Step 4: Modified Search

It can be tricky to verify whether or not an AVL tree is working correctly. One way is to implement a method that calculates the number of steps it takes to find a node in the tree. If you try to find one of the extreme values (i.e., leftmost/rightmost nodes) and that operation takes around $log_2(N)$ steps to complete.

Your Task: You will be modifying your BST search method from last week. Rather than returning the node that we found you will be returning the number of steps it took to find that node. For this exercise I **highly** recommend Googling around to find similar problems and other solutions. The ability to break a problem down to several keywords that yield existing answers that can be either directly used or modified is a highly useful skill and developing that skill is a portion of the goal of this exercise.

For the provided test cases the numbers 1-10000 are inserted into an AVL tree. As such, your search operation should take 14 steps for both 1 and 10000 since they should be at the bottom of the tree and to the far right and far left, respectively..

3 Checklist

- ☐ Read through all existing code in the files.
- ☐ Modify the following methods to update height and call balance:
  - ☐ Insert
  - ☐ Remove
- ☐ Implement the Balance Method
- ☐ Implement the four rotations
  - ☐ Left
  - ☐ Right
  - ☐ Left-Right
  - ☐ Right-Left
- ☐ Implement the modified search method.
- ☐ Construct test cases for each rotation (doesn't need to be turned in)