Adelson-Velskii Landis Trees and Balancing

Ordered Map Implementation #3: Adelson-Velskii Landis Trees
The AVL Tree is somewhat of an upgrade from the usual BST. An AVL Tree's principle is to keep itself height-balanced via rotations, such that all operations that depend on O(h) results in O(log n). We will be referring to the tree as both AVL and BST, they mean the same thing.

To do so, we need to maintain two more attributes with each vertex – its size and height.

- Size: Number of vertices of the subtree rooted at this vertex
    - Need to update after every insertion and deletion
- Height: Number of edges on the path from this vertex to the deepest leaf
    - Need to update after every insertion and deletion

These attributes are recursively defined, with the following:

- Height of empty tree = -1
- Height of vertex = max(left subtree height, right subtree height) + 1
- Size of empty tree = 0
- Size of vertex = left subtree size + right subtree size + 1

Therefore the height of any BST tree is root.height, and the size is root.size.

The Importance of Being Balanced
Most operations take O(h) time. The lower bound of h is $h \geq \lfloor \log_2 N \rfloor$
Proof:

$$N = \sum_{i=0}^{h} 2^i$$

$$N = 2^{h+1} - 1 < 2^{h+1}$$

$$\log_2 N < \log_2 2^{h+1}$$

$$\log_2 N < (h+1) \times \log_2 2$$

$$h > \log_2 N - 1$$

Therefore,

$$h \geq \lfloor \log_2 N \rfloor$$

We say a BST is balanced if h=O(log N), i.e. c*log N, allowing all operations to run in O(log N) time.

How to Get a Balanced Tree

- Define a good property of a tree
- Show that if the good property holds, then the tree is balanced

- After every insert/delete, make sure the good property still holds
    - If not, fix it!

AVL Tree Property
A vertex x is said to be height-balanced if |x.left.height - x.right.height| ≤ 1.
A binary search tree is said to be height balanced if *every vertex* in the tree is height-balanced

From this point onwards, there will be many proofing questions.

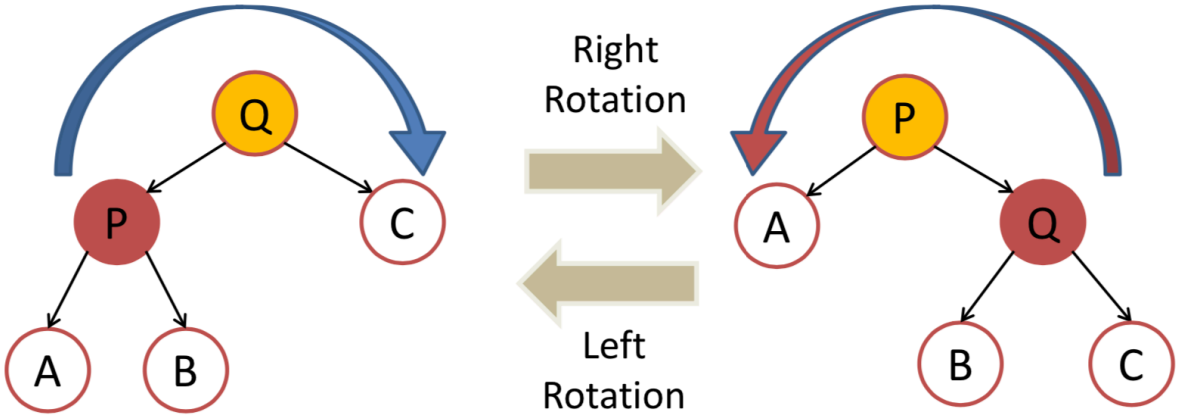
Claim: A height balanced tree with N vertices has height $h < 2\log_2 N$
Step 1:
Let N_h be the minimum number of vertices in a height-balanced tree of height h
The actual number of vertices $N \geq N_h$

Step 2:
$N_h = 1 + N_{h-1} + N_{h-2}$
$N_h > 1 + 2N_{h-2}$ (as $N_{h-1} > N_{h-2}$) $N_h > 2N_{h-2}$ (obvious)
$= 4N_{h-4}$ (recursive)
$= 8N_{h-6}$
= …
In other words,
$N_h > 2N_{h-2}$
$> 4N_{h-4}$
$> 8N_{h-6}$
> ...
$N_h > 2^{h/2} N_0$
As $N_0 = 1$, $N_h > 2^{h/2}$

Step 3:
Since $N \geq N_h$ and $N_h > 2^{h/2}$,
$N \geq N_h > 2^{h/2}$
$N > 2^{h/2}$
$\log_2(N) > \log_2(2^{h/2})$ (log2 on both side)
$\log_2(N) > h/2$ (formula simplification)
$2 * \log_2(N) > h$ or $h < 2 * \log_2(N)$
$h = O(\log(N))$

Balancing the Tree
Every time we update the tree through insertion or deletion, we will need to check if the tree is balanced from the node that was changed all the way up to the root node recursively. This is because there is the possibility that balancing may not be required at where the node was removed/inserted, but required at the root.

Balance Factor
The balance factor of x, bf(x) = x.left.height - x.right.height
If we get a balance factor of +2 (left subtree is taller) or -2 (right subtree is taller), we will need to rebalance it. Once again, this is checked from the insertion point/deletion point all the way up to the root node.



Right Rotation
The assignments and operations must be in the correct order to prevent accidental dereferencing.

- First, we create a vertex P to point to Q.left
- P.parent = Q.parent
- Q.parent = P
- At this point, we have swapped their parents. Now, we need to swap their children.
- Q.left = P.right (passing of B from P to Q)
- If P.right is not null, set P.right.parent = Q
- P.right = Q
- Update height of Q, then update height of P.
- **RETURN P – this is for the recursive assignment**

Left Rotation

- First, we create a vertex Q to point to P.right
- Q.parent = P.parent
- P.parent = Q (done with parents at this point)
- P.right = Q.left (passing of B from Q to P)
- If Q.left is not null, set Q.left.parent = P
- Q.left = P
- Update height of P, then update height of Q.
- **RETURN Q – this is for the recursive assignment**

Although we have two different rotations, there are actually four different cases where rotations are required, with two of those cases requiring two rotations. This is because we require the subtree to be balanced in a similar manner (aka if we need to rotate right because the left is heavier, the left subtree should also be balanced or left heavy, and the opposite is also true). If not, when we balance it, we will end up passing the heavier sub-subtree to the other side (B, with reference to the image above), which causes the other side to be now heavier by 2, and will require balancing again.

As such, we need to check the balance factor of the child vertex at the side with the greater height, and decide on what rotations to use.

Let the vertex to be rotated be x.

Left-Left Rotation
Happens when the left subtree of x is heavier, and the left subtree itself has a heavier left side. In other words, **bf(x) = +2 and 0 <= bf(x.left) <= 1**
We simply need to rightRotate(x).

Left-Right Rotation
Happens when the left subtree of x is heavier, and the left subtree itself has a heavier right side. In other words, **bf(x) = +2 and bf(x.left) = -1**
We need to first leftRotate(x.left), then rightRotate(x).

Right-Right Rotation
Happens when the right subtree of x is heavier, and the right subtree itself has a heavier right side. In other words, **bf(x) = -2 and -1 <= bf(x.right) <= 0**
We simply need to leftRotate(x).

Right-Left Rotation
Happens when the right subtree of x is heavier, and the right subtree itself has a heavier left side. In other words, **bf(x) = -2 and bf(x.right) = 1**
We need to first rightRotate(x.right), then leftRotate(x).

Summary of bBST
With a bBST (balanced BST), we now have the following time complexities:

- Search – O(log N)
- Insert – O(log N)
- Delete – O(log N)
- GetMax – O(log N)
- GetMin – O(log N)
- ListSorted – O(N)
- Predecessor – O(log N)
- Successor – O(log N)
- Select – O(log N)
- Rank – O(log N)

Proving Questions from Tutorial

Claim: Given a BST of size n, for any vertex v in the tree, we can compare at most O(lg n) times to find its successor.

Proof:
False. For a very imbalanced BST, we may potentially take O(n) comparisons to find the successor. This may occur if we are trying to find the successor of the bottom-most node or the successor of the root node.

Claim: The smallest and largest keys in an AVL tree must always be found on either the last (bottom) level or the next-to-last level.

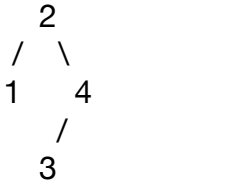
Proof:
False. Basically a tree with increasingly more nodes towards the centre, with the leftmost and rightmost (smallest and biggest) values being 2 levels above the bottom level.

Claim: BST insert operation is always not commutative, in the sense that inserting x then inserting y always produces structurally different BST as inserting y then inserting x (y != x).

Proof:
False. It is not “always not commutative”. We can easily draw a tree where inserting x and y in a different order results in the same tree.

Claim: BST delete operation is always commutative, in the sense that deleting x then deleting y always produces structurally same BST as deleting y then deleting x (y != x, both y and x exist in BST).

Proof:
False.
Imagine a tree having



Removing 1 then 2 results in:

While removing 2 then 1 results in: