AVL Trees

Recall the operations (e.g. `find`, `insert`, `delete`) of a binary search tree. The runtime of these operations were all $O(h)$ where h represents the height of the tree, defined as the length of the longest branch. In the worst case, all the nodes of a tree could be on the same branch. In this case, $h = n$, so the runtime of these binary search tree operations are $O(n)$. However, we can maintain a much better upper bound on the height of the tree if we make efforts to balance the tree and even out the length of all branches. An AVL trees is a binary search tree that balances itself every time an element is inserted or deleted. In addition to the invariants of a BST, **each node of an AVL tree has the invariant property that the heights of the sub-tree rooted at its children differ by at most one**, i.e.:

$$|\texttt{height(node.left)} - \texttt{height(node.right)}| \leq 1 \tag{1}$$

Height Augmentation

In AVL trees, we augment each node to keep track of the node's height.

```
1  def height(node):
2      if node is None:
3          return -1
4      else:
5          return node.height
6
7  def update_height(node):
8      node.height = max(height(node.left), height(node.right)) + 1
```

Every time we insert or delete a node, we need to update the height all the way up the ancestry until the height of a node doesn't change.

AVL Insertion, Deletion and Rebalance

We can insert a node into or delete a node from a AVL tree like we do in a BST. But after this, the height invariant (1) of the AVL tree may not be satisfied any more.

For insertion, there are 2 cases where the invariant will be violated:

1. The left child of node x is heavier than the right child. Inserting into the left child may imbalance the AVL tree.

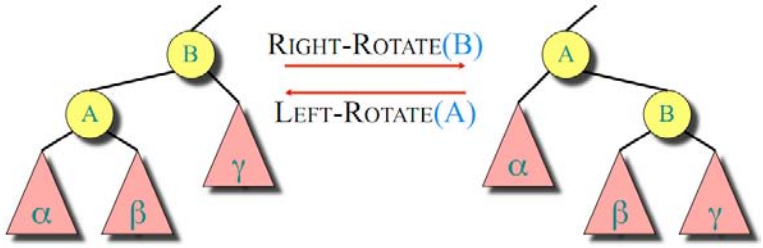
2. The right child of node x is heavier than the left child. Inserting into the right child may imbalance the AVL tree.

For deletion, the cases are analogous.

So we need to reblance the tree to maintain the invariant, starting from the node inserted or the parent of the deleted node, and continue up.

There are two operations needed to help balance an AVL tree: a left rotation and a right rotation. Rotations simply re-arrange the nodes of a tree to shift around the heights while maintaining the

order of its elements. Making a rotation requires re-assigning `left`, `right`, and `parent` of a few nodes, and **updating their heights**, but nothing more than that. Rotations are $O(1)$ time operations.




```
1  def rebalance(self, node):
2      while node is not None:
3          update_height(node)
4          if height(node.left) >= 2 + height(node.right):
5              if height(node.left.left) >= height(node.left.right):
6                  self.right_rotate(node)
7              else:
8                  self.left_rotate(node.left)
9                  self.right_rotate(node)
10         elif height(node.right) >= 2 + height(node.left):
11             if height(node.right.right) >= height(node.right.left):
12                 self.left_rotate(node)
13             else:
14                 self.right_rotate(node.right)
15                 self.left_rotate(node)
16         node = node.parent
```

Note that `rebalance` includes `upate_height` as well.

6.006 Introduction to Algorithms
Fall 2011