BSTs have a height of $O(n)$ (for $n$ nodes) in the worst case. So, we need to this affects efficiency of insertion, search & deletion operations. BSTs can be "balanced" so as to maintain a height of $O(\log n)$. One variant of a balanced tree is called an AVL tree (Adelson-Velskii and Landis). An AVL tree is defined as follows: an empty tree is an AVL tree. If $T$ is a non-empty binary tree with $T_L$ & $T_R$ as left & right subtrees, then $T$ is an AVL tree iff $T_L$ & $T_R$ are both AVL trees and $|\text{height}(T_L) - \text{height}(T_R)| \leq 1$.

For any $n$, the existence of an AVL tree is always guaranteed. Searching operation in an AVL tree is $O(\log n)$ in worst case. Insertion is also an $O(\log n)$ operation. Deletion also has the same time complexity.

When a new node is inserted into an AVL tree, it becomes unbalanced. A balancing operation needs to be then performed. This is also true for the deletion operation.
Convention: Associated with each node is a "balance factor" = height (left subtree) - height (right subtree).

The permissible balance factors are -1, 1 and 0. If the tree contains any node with a balance factor of 2 or more (or -2 or less), then it needs to be balanced.

eq: 20
    /   \
  15     25
   /     /  \
12     18   30

eq: 20
    /   \
  15     20
   /     /  \
12     18   30

are not AVL trees.

eq: 20
    /   \   -2
  15     40
   /     /  \  
12     20   43

is an AVL tree
In order to balance an AVL tree during insertion, several cases need to be considered.

A) Left Rotation: To be performed when the tree becomes right heavy.

Example: When 3 is inserted in the tree 1 2, the resulting tree is:

```
   2
  / 
 1   3
```

In such a case, we make 2 as the new root. 1 becomes the parent of the left child of 2 (null in this case) and the right child of 2 remains unchanged.

(If 2 had a non-null child before 3 was inserted, it would make the tree unbalanced.)

B) Right Rotation: For left-heavy trees.
In this case, \( 2 \) becomes the new root.
\( 3 \) becomes its right child and takes
parenthood of the former right child of
\( \), ie null.

\( \) Left-right rotation (LR rotation):

Consider the tree

Now when \( 2 \) is inserted, we get:

A single left rotation will yield us:

Which is unbalanced.

So we need to do a two-step rotation. First, we will do a right rotation on the right subtree ie \( 2 \) yielding:

Which is of course unbalanced.

\( \) may take custody of the
left child of \( 2 \).