Binary Search Tree

Binary Search Tree is a node-based binary tree data structure which has the following properties:

- The left subtree of a node contains only nodes with keys lesser than the node’s key.
- The right subtree of a node contains only nodes with keys greater than the node’s key.
- The left and right subtree each must also be a binary search tree.

Construction of BST given pre-order sequence

- **Recursive approach (pre-order input)**
  Time complexity: O(n)

The trick is to set a range {min .. max} for every node. Initialize the range as {INT_MIN .. INT_MAX}. The first node will definitely be in range, so create root node. To construct the left subtree, set the range as {INT_MIN …root->data}. If a values is in the range {INT_MIN .. root->data}, the values is part part of left subtree. To construct the right subtree, set the range as {root->data .. max .. INT_MAX}

- **Using stack (pre-order input)**
  Time Complexity: O(n)

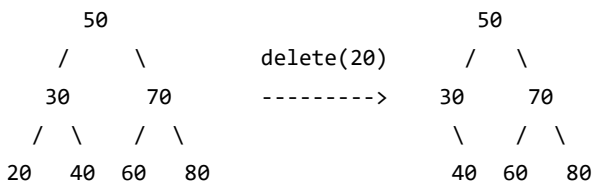
```
Algorithm:
1. Create an empty stack.
2. take the first value as root. Push it to the stack.
3. Keep on popping while the stack is not empty and the next value is greater than stack's top value.
   Make this value as the right child of the last popped node.
   Push the new node to the stack.
4. If the next value is less than the stack's top value, make this value as the left child of the stack's top noc
5. Repeat steps 2 and 3 until there are items remaining in pre[].
```

We can observe that every item is pushed and popped only once. So at most 2n push/pop operations are performed in the main loops of getBSTUsingStackFromPreOrder(). Therefore, time complexity is O(n).

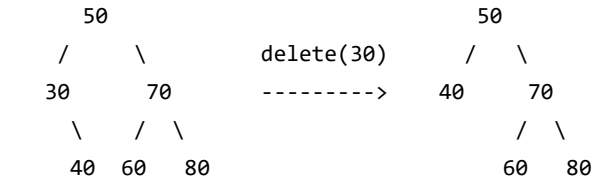
Search, Insertion and Delete in BST

1. *Searching a key*
   To search a given key in Binary Search Tree, we first compare it with root, if the key is present at root, we return root. If key is greater than root's key, we recur for right subtree of root node. Otherwise we recur for left subtree.
2. *Insertion a key*
   A new key is **always** inserted at leaf. We start searching a key from root till we hit a leaf node. Once a leaf node is found, the new node is added as a child of the leaf node. No duplicates allowed.
   Time Complexity: The worst case time complexity of search and insert operations is O(h) where h is height of Binary Search Tree. In worst case, we may have to travel from root to the deepest leaf node. The height of a skewed tree may become n and the time complexity of search and insert operation may become O(n).
3. *Deletion a key*
   When we delete a node, three possibilities arise

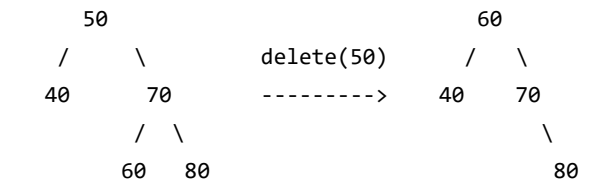
a) **Node to be deleted is leaf:** Simply remove from the tree.



b) **Node to be deleted has only one child:** Copy the child to the node and delete the child.



c) **Node to be deleted has two children:** Find in-order successor of the node. Copy contents of the in-order successor to the node and delete the in-order successor. Note that in-order predecessor can also be used.



Time Complexity: same explanation as insertion in BST.

Optimization:
In delete method, we recursively call delete() for successor. We can avoid recursive call by keeping track of parent node of successor so that we can simply remove the successor by making child of parent as NULL. We know that successor would always be a leaf node.
(Refer code)

Further reads(optional)

- construction of BST from post-order and in-order input
- Insert in BST in level-order