Binary Trees

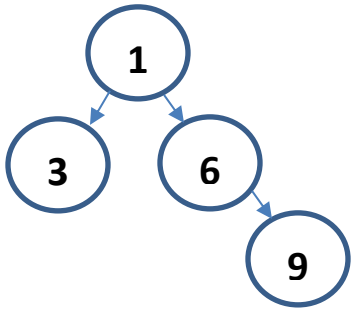
One of the most common operations on a data structure is to check if it contains a particular value. Unfortunately, arrays and linked lists store data in an otherwise *random* or *unsorted* order. This requires us to navigate the entire list structure in order to verify the existence of a value.

A binary tree is made of tree nodes. Each tree node holds a data element and has references to two other tree nodes called *left* and *right*.

To use and maintain a binary tree, we maintain a reference to the *root* of the tree (the node at the top). Similar to the *head* of a linked list, the entire binary tree is accessible through the root of the tree.

Each node in the tree can be thought of as the root of a smaller binary tree. This is a *recursive* structure since it's made up of smaller instances of binary trees.

For example, the array {1, 3, 6, 9} might be stored as the following binary tree:



In this example, the *root* of the tree contains 1.

The *left child* is a smaller binary tree with one node (3).

The *right child* is a smaller binary tree with a root of 6, a left child that is *null*, and a right child that contains 9.

The binary tree would maintain a reference called *root* to the node object containing 1.

Binary Trees

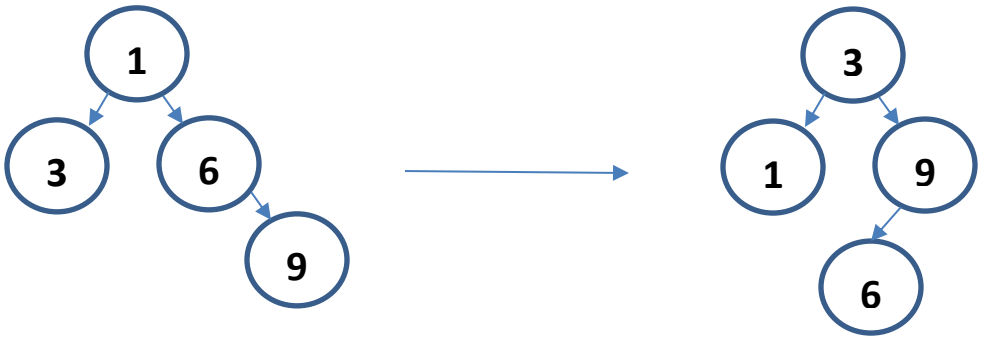
Binary Search Trees

Binary search trees are binary trees with the following *search property*:

For any node *n* in the binary tree:

- All nodes in the right subtree of *n* contain data that is strictly greater than or equal to the data stored in *n*.
- All nodes in the left subtree of *n* contain data that is strictly smaller than the stored in *n*.

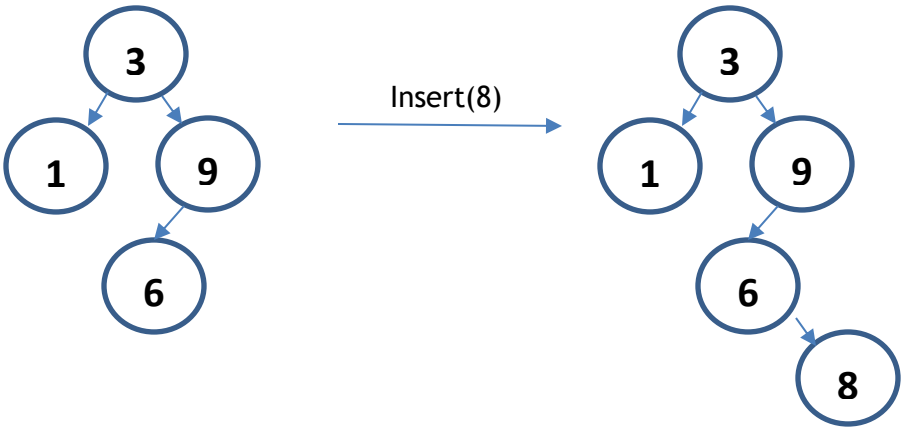
Example: The binary tree discussed in the previous section is *not* a binary search tree because the left child of 1 is *greater*. However, it can be rearranged as follows:



Binary Tree *Binary Search Tree*

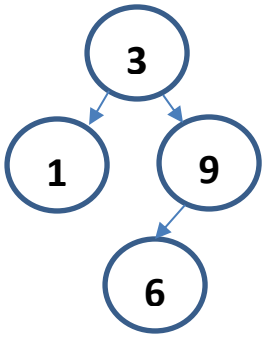
Our goal is not to transform binary trees into binary search trees. Instead, we will *build* binary search trees one node at a time. By carefully maintaining the search property with every insertion, we never have to worry about losing it!

To maintain the search property on a given binary search tree, we will consider an example. Consider inserting 8 into the above Binary Search Tree. Starting from the root, we recursively find the appropriate position for this new node by comparing it to the root and deciding whether to move into the left or right subtree. Once we find a null location, we insert the new node at that location. In the above case, since 8 is greater than 3, we move to the right subtree. Next, since 8 is less than 9, we move to its left subtree. Finally, since 8 is greater than 6, we move to its right subtree. However, since the right subtree is null, we insert 8 as the right subtree of the node containing 6. In general, inserting into a binary search tree will result in the addition of a new *leaf node* – that is, a new tree node with no children.



Binary Trees

A *leaf node* is a node in the tree with no children. For example, in the tree below, the leaf nodes are 3 and 9.



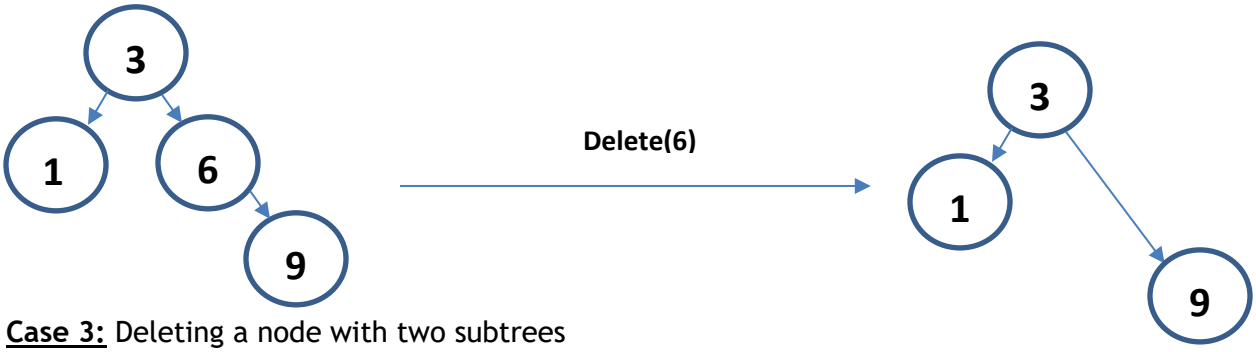
Case 1: Deleting a leaf node

Deleting a leaf node is easy – we simply tell the parent node that its child is now *null*. For example, deleting 9 in the tree above requires us to set 6's right child to *null*.

Case 2: Deleting a node with one subtree

Deleting a node with *one* subtree is also simple. We simply make the parent point to the subtree below (convince yourself that this maintains the *search property*).

For example, deleting 6 in the tree above requires us to set 1 to now point to 9 as its right child.



Case 3: Deleting a node with two subtrees

Deleting a node with *two* subtrees is more complex. For instance, deleting 3 in the original tree separates the entire structure.

Binary Trees

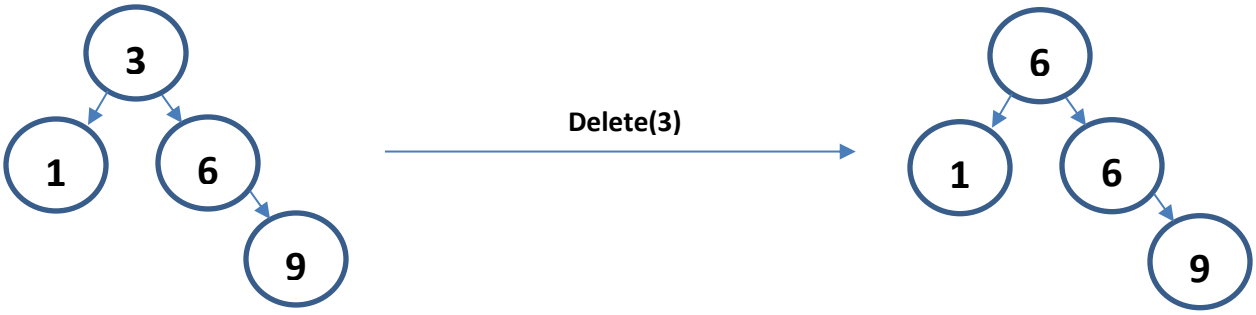
We need a way of *replacing* the node rather than removing it.

Looking back at the original tree, consider that whatever *replaces* the root of the tree must be larger than everything in the left subtree and smaller than everything in the right subtree. This suggests two possible candidates as our new roots:

1) Choose the smallest element in the right subtree
   OR
2) Choose the largest element in the left subtree

Again, stop and convince yourself that this maintains the *search property*.

For example, in the previous binary search tree, we can replace 3 with 6:



Now we have *two* instances of 6! To get around this, we delete 6 from the subtree. Since 6 has only one child, this allows us to apply *Case 2*. However, there *are* trees where we'd have to apply Case 3 over and over in a recursive fashion. But that's for you to think about…

For visualization of the insert and delete functions, visit:
https://www.cs.usfca.edu/~galles/visualization/BST.html

Exercises
Implement the `preOrder()` and `postOrder()` print methods to print the elements of the tree.

Implement the `remove(int key)` method that deletes a node from the tree.