6: Binary Trees

The highest priority when a programmer is organizing data is making sure that item insertions, deletion, and searches are fast. The first structure that we considered was an array. Insertion and deletion were quick but searches were not. This weakness becomes even more pronounced with larger data sizes.

The next idea was to sort the array i.e. an ordered array. That way we could employ binary searches which made the searching way faster. However, Insertions and deletions became slower as a result. This is because we would have to expand and collapse the list respectively, both of which required data movement, and that is an expensive step particularly for larger data sizes.

To speed up insertions and deletions, we turned to linked lists. Insertion and deletion in a linked list did not require any major data movement; we simply adjusted a few nodes in the list. However, the draw back for linked lists is that they must be processed sequentially i.e. we had to traverse from the head in their proper order to find or process any node. So even though the insertion and deletion didn't require much data movement, we had to begin our searches at the first node in the list every time.

The next data structure that we will discuss is the binary tree which attempts to combine the best features of arrays and linked lists i.e. it allows quick searching (like in an ordered array), but also allow fast insertion and deletion (like a linked list)

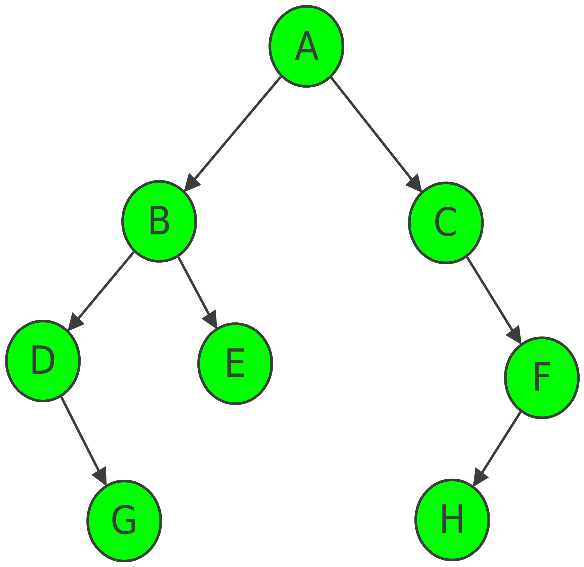
What exactly is a binary tree?

A binary tree is a data structure made up of **nodes** (kinda like linked lists) connected by **edges**. Nodes typically contain "entities" or complex objects, and the edges represent node inter-relatedness.

A binary tree T is either empty or has the following properties

- Has a special node called the **root** node.
- Has *two* (hence the binary) sets of nodes L_T and R_T called the **left subtree** and **right subtree** respectively.
- L_T and R_T are binary trees.

Binary trees can be represented diagrammatically as shown in the figure below. Nodes are represented as circles, and labeled by the node. The root node is drawn at the top (yes the root is at the top). The root of the left subtree is called a **left child**, and the root of the right subtree is called the **right child**. The child nodes are drawn below and to the left or right of the their **parent node**. The edges have a direction i.e. from the parent to the child. The edges can also be referred to as **directed edges**, **branches** or **directed branches**.



In the diagram above, *A* is the root node. Its left subtree is denoted by L_A and its right subtree by R_A. The root of L_A is *B* and the root of R_A is *C*. L_A and R_A are also binary trees themselves. *C* has no left child, and *G* has no children at all. Also recall that since it is a binary tree, each node has a maximum of two children. Any node that does not have ANY children is called a **leaf node**. Therefore *E*, *G*, and *H* are the leaves of the binary tree above.

A **path** from a node to another node is a sequence of nodes that one has to go through to move from any given node to a node in one of its subtrees. For example, the path from *A* to *G* is *A, B, D, G*. You might have noticed that there is a unique path from the root node to every node in the binary tree.

The **length** of a path in a binary tree is the number of branches or edges in that path. For example, the path from *A* to *G* has a length of 3.

The **level** of a node in a binary tree is the number of branches on the path from the root to that node. For example, node *F* is at level 2. Notice that the levels count downwards.

The **height** of a tree is the number of nodes on the longest path from the root to a leaf. In the diagram above, the height of the tree is 4. Notice that for length and level we count edges, but for height, we count nodes.

From the description thus far, it stands to reason that the implementation of a node of a binary tree is very similar to the node of a linked list. However, instead of just keeping track of the link to the next node, it must keep track of both its left and right children.

```
class TreeNode // missing accessors, mutators, constructor, etc.
{
    private int data;
    private TreeNode left;
    private TreeNode right;
}
```

The data is stored in the member called `data`. The left subtree is pointed to by `left`, and the right subtree is pointed to by `right`.

We can have a reference to the first `TreeNode` that we can call `root` to keep track of the root of the tree.

Traversing the tree

Insertion, deletion and searching all require that the binary tree is traversed, which means that traversing (or visiting) the tree is the most common operation performed on it. It makes sense to start the traversal at the root since we have a pointer to it.

For each node, we have two choices

- visit that node first (which could include any processing of its data e.g. printing, counting, etc.)
- visit the subtrees first.

The order in which these are done lead to three commonly used traversals of a binary tree.

- Inorder traversal (traverse left subtree, visit the node, then traverse right subtree) a.k.a. LNR
- Preorder traversal (visit the node, traverse the left subtree, and then the right subtree) a.k.a. NLR.
- Postorder traversal (traverse the left subtree, then the right subtree, and then visit the node) a.k.a. LRN.

It should be easy to see that these traversal algorithms are recursive. (remember recursion from Chapter 12)

Lets look at an easy example of an inorder traversal to print out the data in each node.

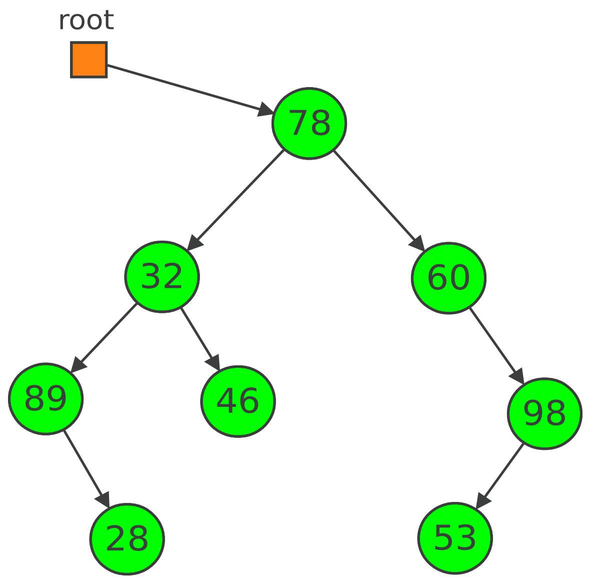
```
void InOrder(TreeNode n)
{
      if (n != null)
      {
            InOrder(n.getLeft());
            System.out.print(n.getData() + " ");
            InOrder(n.getRight());
      }
}
```

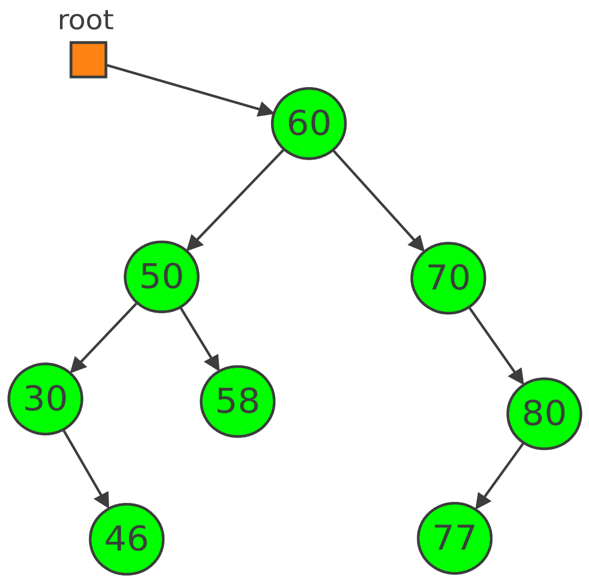
Let's see if you can come up with the `PreOrder` and `PostOrder` traversal algorithms. Given how the `InOrder` looks above, coming up with the remaining two traversals should be trivial.

What would the inorder, preorder, and postorder traversals for the tree in the diagram on page 2 look like?

Binary Search Tree

We've said that traversing the tree could potentially take a long time. The reason for this is that no criteria exists to guide our search. If we were looking for Node 54 in the tree on the left (below), we would have to traverse the entire tree before ascertaining that that node didn't exist. This kinda defeats our purpose of having the binary tree in the first place i.e. making insertion, deletion and searching quick.





However, there is a special way of arranging an arbitrary tree that makes searching way faster. Its called a binary search tree and an example is in the diagram to the right above. It has the exact same properties as a binary search tree as we discussed before but it has an extra property.

- The key in the root node is larger than every key in the left subtree, and smaller than every key in the right subtree.

This extra property means that if we are looking for a node in our tree, we can effectively reduce our search space by half depending on a comparison between a root node and the key of the node we are looking for. A natural consequence of this rule is that a binary search tree does not typically allow duplicate values. There are ways to go round this but they are beyond the scope of this class.

For example, if we are searching for *77* in the binary search tree (which is on the right), we compare *77* and *60*. Since *77* is greater than *60,* we ignore the left subtree and only consider the right subtree. We then compare *77* with *70* (which is the root of the right subtree...and is also a binary search tree in an of itself), and discard the left subtree of that node. *77* is less than *80* so we have to look at its left subtree, and we find *77*.

What about if we were looking for a number that wasn't in the tree? Let's say we were looking for *59*. Well *59* is less than *60* so we go to the left subtree. *59* is greater than *50* so we go to its right subtree. *59* is greater than *58* but we can't go to its right subtree because that tree is `null`. Therefore, *59* is not in the tree.

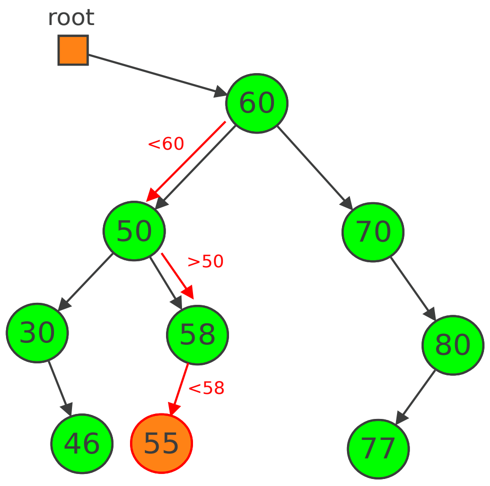
Remember complexity and big-O? What is the complexity of an algorithm that effectively halves its search space with each step? *$O(\log_2 n)$*. This is significantly better than *$O(n)$* which was the complexity of searching through a linked list.

Insertion into a binary search tree

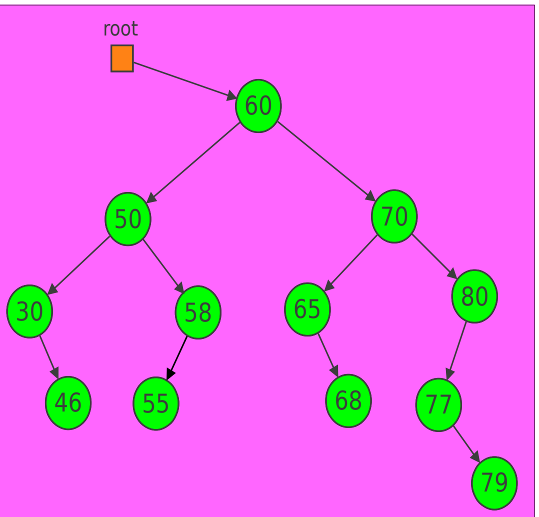
You might have realized that the order of the nodes is important in a binary search tree. Well how do we insert a node into the tree while maintaining all the rules that make it very efficient?

Nodes are ALWAYS inserted as leaves. Remember a leaf is a node that doesn't have any children. First we search the tree to find the appropriate position for our new leaf node. Compare with the root, and go to the left or right subtree based on that comparison. Keep on repeating that step until the subtree is NULL. At this point, insert the node and make sure that the node you were in is pointing to the new node using the appropriate link.

For example, suppose I wanted to add a 55 to the binary search tree we saw earlier. 55 is less than 60 so we go to the left subtree of the root. It's greater than 50 so we go to the right subtree of that node. Its less than 58 but there is no left subtree so we add the new node in that position.



```
Try and insert the following nodes to
the tree above.
65, 68, 79
```



```
/*
 this function inserts the integer x into the tree in the appropriate position
*/
TreeNode insert(int x, TreeNode root)
{
       TreeNode current, trail, newnode;    //pointers to the current node, the node before
                                            //the current (called trail), and a pointer to the
                                            //new node that we will create from int x.

       newnode = new TreeNode();            //create the new node and initialize it appropriately
       newnode.setData(x);
       newnode.setLeft(null);
       newnode.setRight(null);


       if (root == null)                    //if the root is null (i.e. no tree yet),
              root = newnode;               //just set the root to the newnode
       else
       {
              current = root;

              while (current != null)       //this while loop will set trail to the current node and
              {                             //then move "current" to the appropriate subtree until
                                            //both current and trail are at the bottom of the tree
                     trail = current;

                     if (current.getData() == x)//we don't want duplicate values in our tree
                     {
                            System.out.println("we don't want duplicate values");
                            return;
                     }
                     else if (current.getData() > x)//if x is less, move current to the left subtree
                            current = current.getLeft();
                     else                          //otherwise, move current to the right subtree
                            current = current.getRight();
              }

              //right now trail points to the last node before our insertion, and current is null.
              if (trail.getData() > x)
                     trail.setLeft(newnode);
              else
                     trail.setRight(newnode);
       }

       return root;

}
```

Finding the min and max

With a binary search tree, finding the minimum value is just a matter of following the left edges until the left most leaf node. This node is the minimum.

Finding the maximum is the opposite: follow the right edges until the right most leaf node.

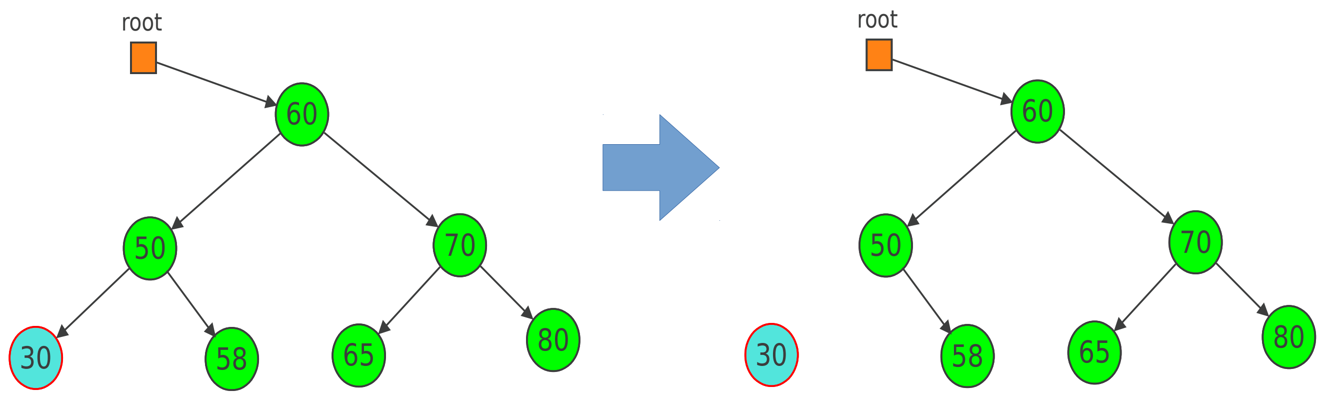
Deleting a node

We've inserted, we've searched...last thing to do is delete. Deleting also requires searching for the appropriate node. Good thing we can now do that in $O(log_2 n)$. The delete operation shouldn't change the tree from a binary search tree and this restriction brings up some different cases.

Case 1: The node to be deleted is a leaf

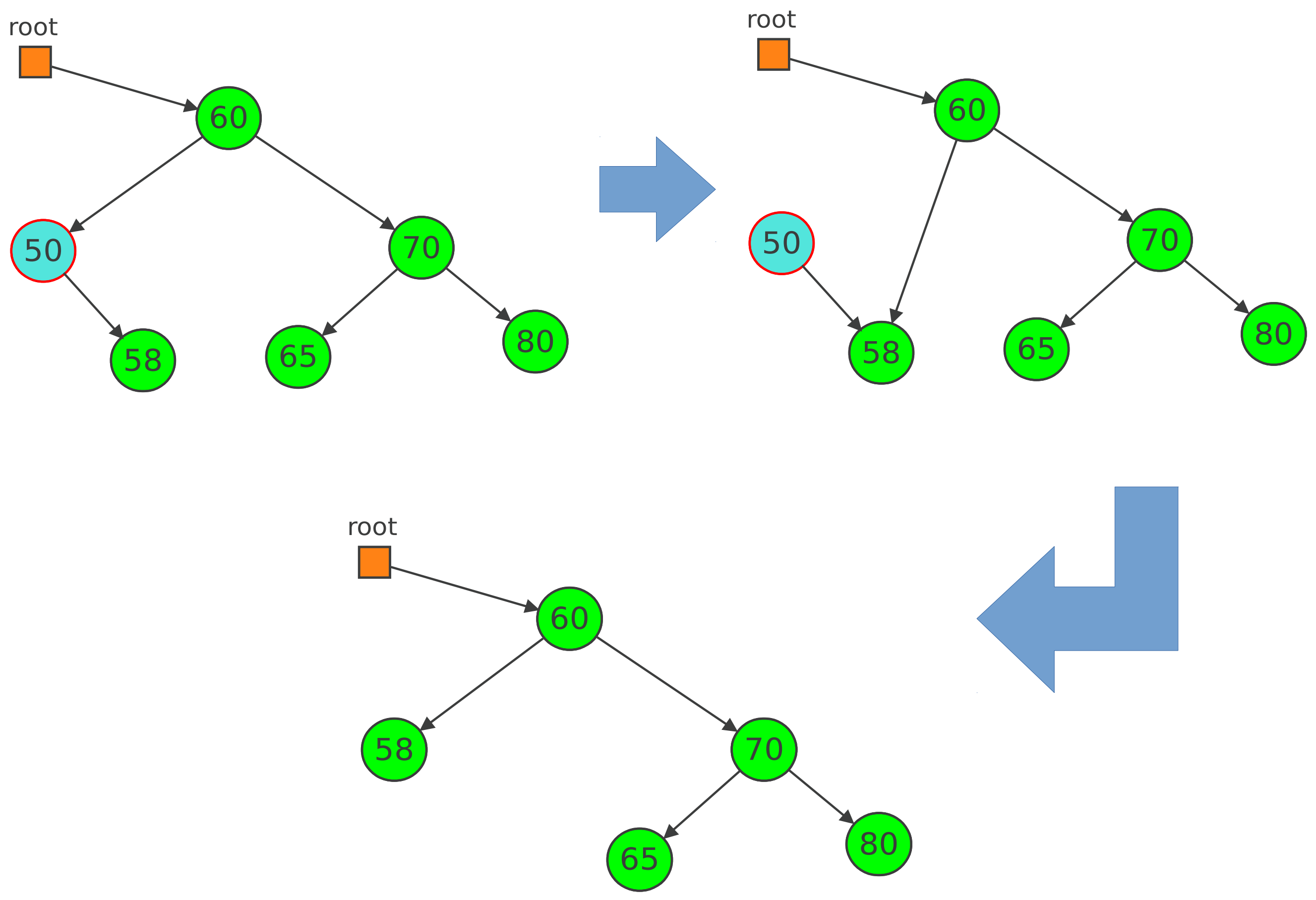
This is the most straightforward case. To delete the leaf, we modify the parent's link to this node and make it `NULL`.

For example, if we wanted to delete *30* from the tree below on the left, we would just change *50*'s left link to `NULL`.



Case 2: The node to be deleted has one child

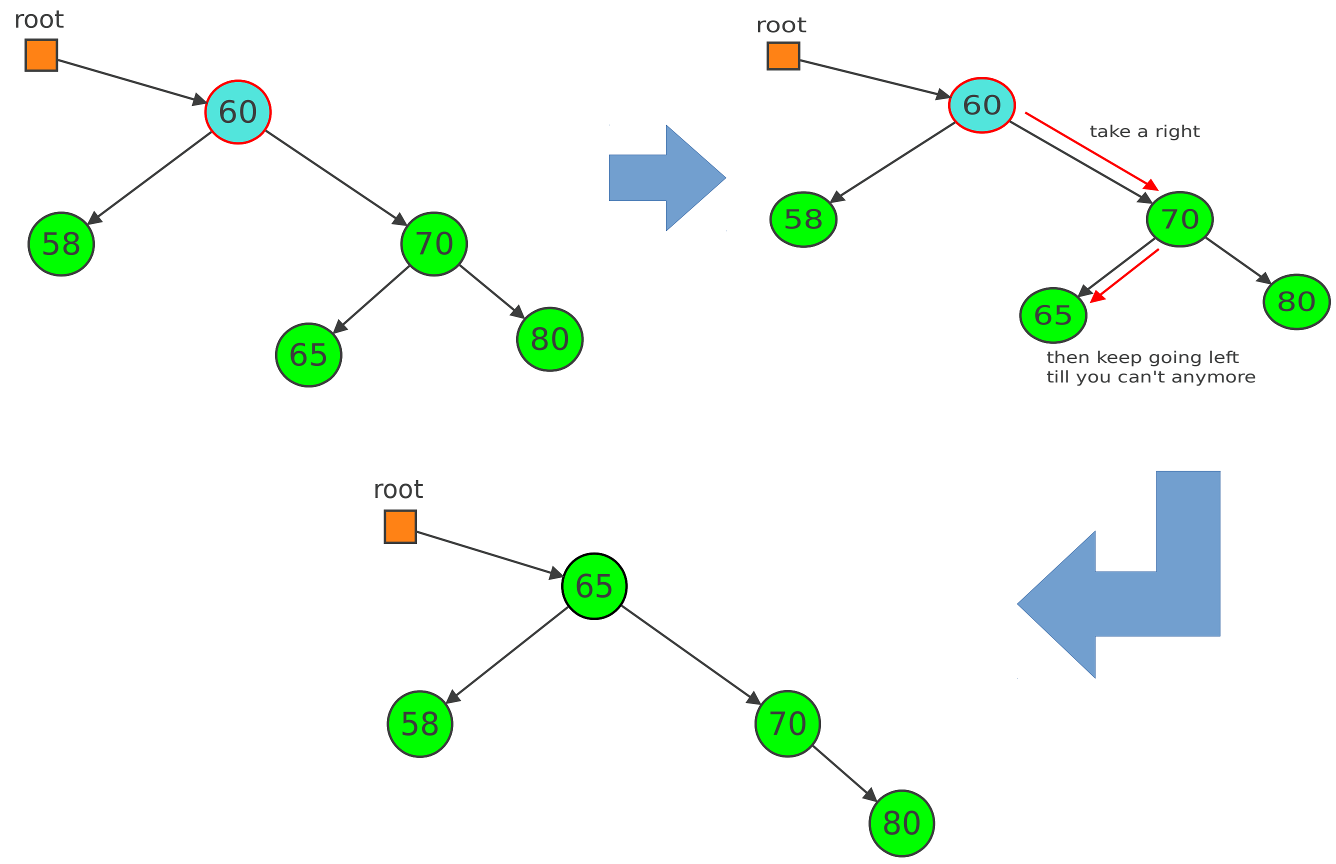
This case is also fairly straightforward. We simply promote that child to take the position of the parent. This involves making the "grandparent" point directly to the child node. For example, if we wanted to delete *50* from the tree above, we would just promote *58* to level 1 i.e. make it directly connected to *60*.



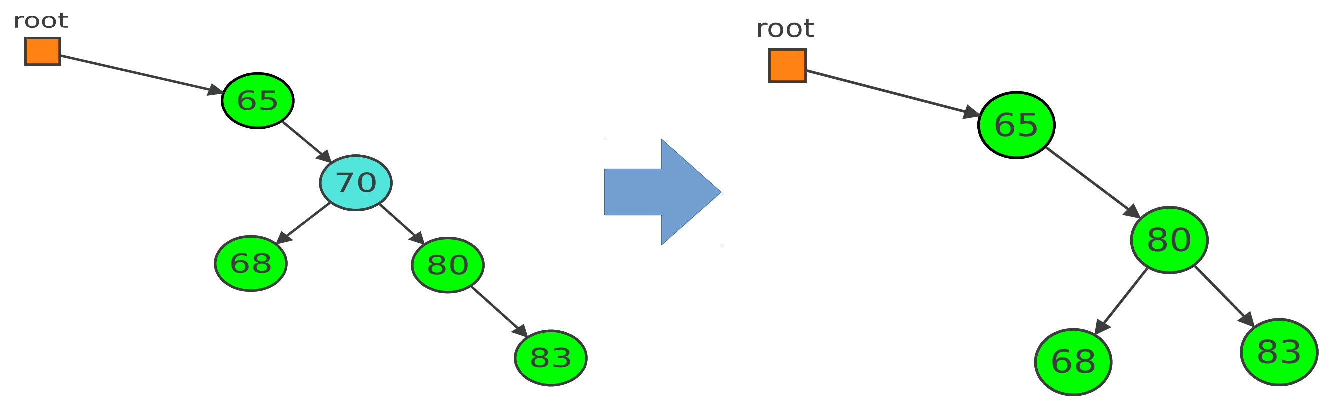
Case 3: The node to be deleted has two children.

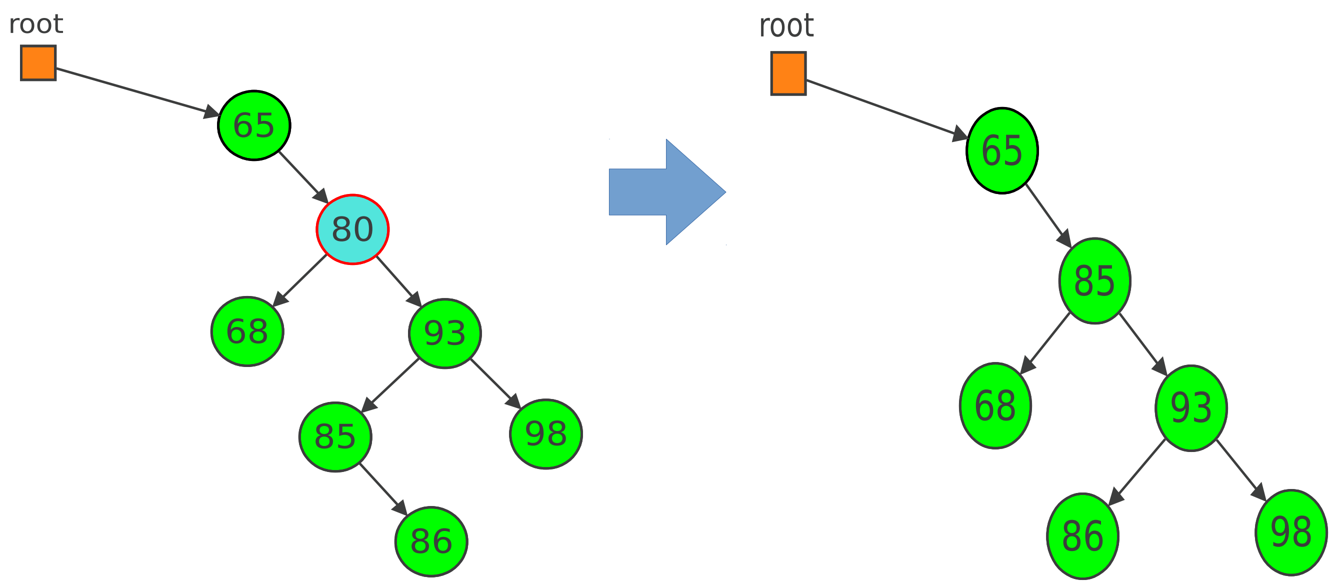
In this case, we find the node's successor and swap it with the deleted node. The node's successor is the next highest node in the binary search tree. To identify it, traverse the right edge, and then traverse left edges until you find a leaf node.

For example, if I wanted to delete *60* from the tree above, I would find *60*'s successor by traversing the right edge (to *70*) and then left edges till I couldn't find any other left edge (*65*), and then move *65* to the position that *60* originally held.



What if it was a more complicated tree?





Wrap up

Printing trees

Recursion provides a pretty nifty way of printing trees. They can be printed from right to left using the function below. In it, the variable `lev` refers to the number of tabs.

```
void print(TreeNode n, int lev)
{
      if (n != null)
      {
            print(n.getRight(), lev+1);   //go down the right subtree and increase
                                          //the number of tabs i'll have to make

            for (int i = 0; i < lev; i++)
                  System.out.print("\t"); //print out the appropriate no. of tabs
            System.out.println(n.getData());//and the data, and move to next line

            print(n.getLeft(), lev+1);    //go down the left subtree and increase
                                          //tabs
      }
}
```

Recreating a tree from its inOrder and preOrder traversals

The diagramatic representation of a tree can be recreated from its in order and preorder traversals.

Remember that inorder is LNR i.e. visit the left subtree, then the node, then the right, whereas preorder is NLR i.e. visit the node first, then the left subtree and end with the right subtree.

For example consider the two sequences below.

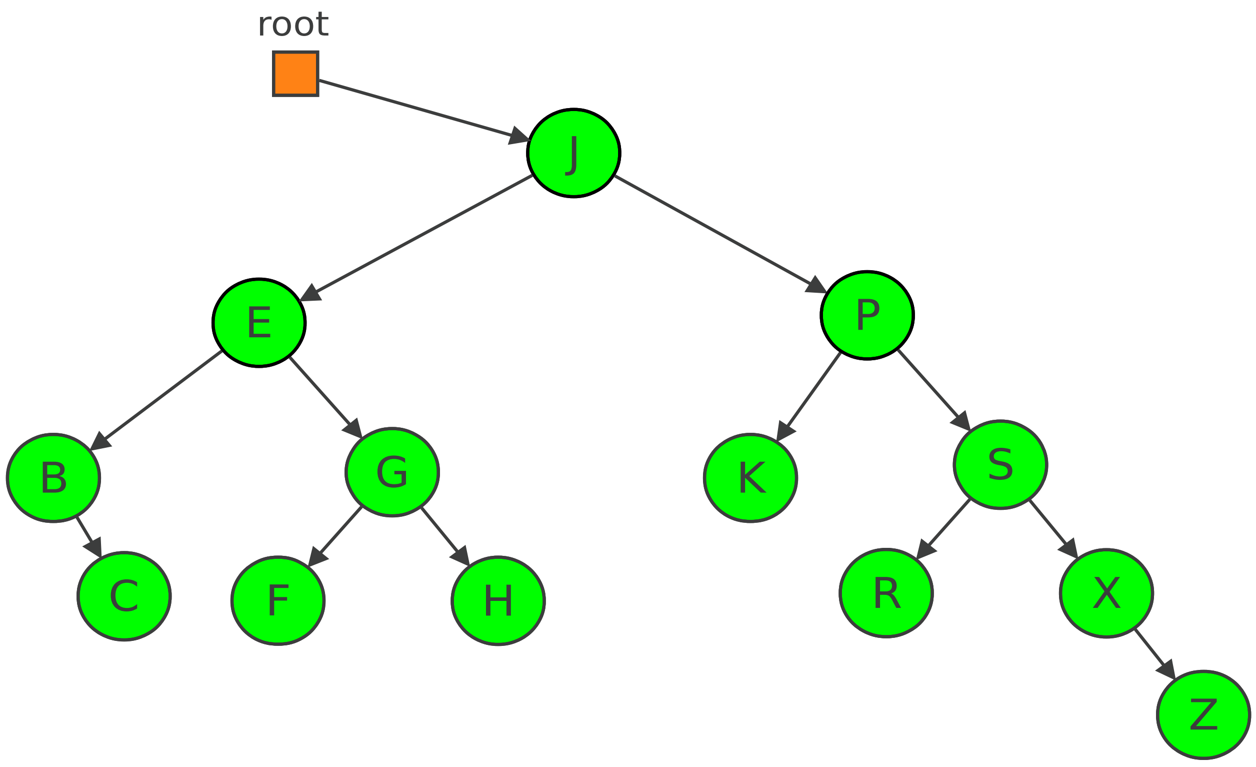
```
Inorder: B C E F G H J K P R S X Z
preorder: J E B C G F H P K S R X Z
```

Preorder starts with the node so we know that J must be the root node. However, Inorder visits the left subtree before getting to the root so we can say that all the letters to the left of J must be in its left subtree i.e. B C E F G and H. Conversely, the right subtree of the root J consists of K P R S X and Z.

Preorder says our next node (after the root J) is E. From in order, we see that the left subtree of E is B and C, while its right subtree is F, G, and H.

Preorder dictates that our next node after E is B. The left subtree of B (acc. to the inorder) has nothing, but its right subtree has C (Recall that C is also in the left subtree of E which means that it is to the right of B but left of E).

I leave it up to you to follow through with all the nodes until you get to the tree shown below.


root
J
E
P
B
G
K
S
C
F
H
R
X
Z