Depth First Search(DFS) is another traversal algorithm, where we go to the depths of the tree/graph until we can't go further, go back up and expand other nodes. This algorithm generally uses a stack to keep track of visited nodes. As the last node seen is the next one to be visited, the rest is stored to be visited later.

There are three types of DFS:

- Pre-order - traverses the root node first, then the left and right subtrees respectively(NLR);
- In-order - traverses the left subtree, then all the way to the root node and finishes by traversing the right subtree(LNR);
- Post-order - traverses the left subtree, then moves to the right subtree and, finally, traverses the root node(LRN);

Just like in BFS, we will need a Binary Search Tree to implement all three types, so we will use the one we coded previously:

In [12]:

```
class Node():
    def __init__(self,data):
        self.data = data
        self.left = None
        self.right = None

class BST():
    def __init__(self):
        self.root = None
        self.nr_nodes = 0

    def insert(self, data):
        new_node = Node(data)
        if self.root == None:
            self.root = new_node
            self.nr_nodes += 1
            return
        else:
            current_node = self.root
            while(current_node.left != new_node) and (current_node.right !=
                if new_node.data > current_node.data:
                    if current_node.right == None:
                        current_node.right = new_node
                    else:
                        current_node = current_node.right
                elif new_node.data < current_node.data:
                    if current_node.left == None:
                        current_node.left = new_node
                    else:
                        current_node = current_node.left
            self.nr_nodes += 1
            return

    def search(self, data):
        if self.root == None:
            return 'Empty tree.'
        else:
            current_node = self.root
            while True:
                if current_node == None:
                    return 'Not found.'
                if current_node.data == data:
                    return 'Found it!'
                elif current_node.data > data:
                    current_node = current_node.left
                elif current_node.data < data:
                    current_node = current_node.right

    def remove(self, data):
        if self.root == None:
            return "Empty tree."
        current_node = self.root
        parent_node = None

        while current_node != None:
            if current_node.data > data:
                parent_node = current_node
                current_node = current_node.left
            elif current_node.data < data:
                parent_node = current_node
                current_node = current_node.right
            else:
                if current_node.right == None:
                    if parent_node == None:
                        self.root = current_node.left
                        return
                    else:
                        if parent_node.data > current_node.data:
                            parent_node.left = current_node.left
                            return
                        else:
                            parent_node.right = current_node.left
                            return
                elif current_node.left == None:
                    if parent_node == None:
                        self.root = current_node.right
                        return
                    else:
                        if parent_node.data > current_node.data:
                            parent_node.left = current_node.right
                            return
                        else:
                            parent_node.right = current_node.right
                            return
                elif current_node.left == None and current_node.right == N
                    if parent_node == None:
                        current_node = None
                        return
                    if parent_node.data > current_node.data:
                        parent_node.left = None
                        return
                    else:
                        parent_node.right = None
                        return
                elif current_node.left != None and current_node.right != N
                    rm_node = current_node.right
                    rm_parent_node = current_node.right
                    while rm_node.left != None:
                        rm_parent_node = rm_node
                        rm_node = rm_node.left
                    current_node.data = rm_node.data
                    if rm_node == rm_parent_node:
                        current_node.right = rm_node.right
                        return
                    if rm_node.right == None:
                        rm_parent_node.left = None
                        return
                    else:
                        rm_parent_node.left = rm_node.right
                        return
        return 'Not found.'

    # Implementing the three types of DFS Traversals
    def DFS_Inorder(self):
        if self.root is None:   # If the tree is empty when we the run BFS
            return 'Oops! Tree is empty.'
        else:
            return traverseInorder(self.root, [])

    def DFS_Preorder(self):
        if self.root is None:   # If the tree is empty when we the run BFS
            return 'Oops! Tree is empty.'
        else:
            return traversePreorder(self.root, [])

    def DFS_Postorder(self):
        if self.root is None:   # If the tree is empty when we the run BFS
            return 'Oops! Tree is empty.'
        else:
            return traversePostorder(self.root, [])


# Creating functions outside class that describe DFS Traversals above
def traverseInorder(node, DFS_list):
    if node.left:
        traverseInorder(node.left, DFS_list)
    DFS_list.append(node.data)
    if node.right:
        traverseInorder(node.right, DFS_list)
    return DFS_list


def traversePreorder(node,DFS_list):
    DFS_list.append(node.data)
    if node.left:
        traversePreorder(node.left, DFS_list)
    if node.right:
        traversePreorder(node.right, DFS_list)
    return DFS_list


def traversePostorder(node, DFS_list):
    if node.left:
        traversePostorder(node.left, DFS_list)
    if node.right:
        traversePostorder(node.right, DFS_list)
    DFS_list.append(node.data)
    return DFS_list
```

In [13]:

```
# Initializing a new tree with our blueprint
new_bst = BST()
```

In [14]:

```
print(new_bst.DFS_Inorder())
```

```
Oops! Tree is empty.
```

In [15]:

```
print(new_bst.DFS_Preorder())
```

```
Oops! Tree is empty.
```

In [16]:

```
print(new_bst.DFS_Postorder())
```

```
Oops! Tree is empty.
```

In [17]:

```
# Populating tree
new_bst.insert(5)
new_bst.insert(3)
new_bst.insert(7)
new_bst.insert(1)
new_bst.insert(13)
new_bst.insert(65)
new_bst.insert(0)
new_bst.insert(10)
```

In [18]:

```
print(new_bst.DFS_Inorder())
```

```
[0, 1, 3, 5, 7, 10, 13, 65]
```

In [19]:

```
print(new_bst.DFS_Preorder())
```

```
[5, 3, 1, 0, 7, 13, 10, 65]
```

In [20]:

```
print(new_bst.DFS_Postorder())
```

```
[0, 1, 3, 10, 65, 13, 7, 5]
```