Breadth First Search(BFS) is a traversal algorithm for a tree or graph, where we start from the root node(for a tree) and visit all the nodes level by level from left to right. It requires keeping track of the children of each node we visit in a queue, so that after traversal through a level is complete, our algorithm knows which node to visit next.

The time complexity is O(n), but the space complexity can become a problem in some cases, because of the queue created for tracking purposes.

To implement BFS, we will use the BST created some lessons ago:

```
In [2]:
         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 != new node):
                         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:</pre>
                             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:</pre>
                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:</pre>
            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 == None:
            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 != None:
            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 'Not found.'
```

```
# Implementing the BFS method:
   def BFS(self):
       current node = self.root # We start with the root node
       if current node is None:
                                  # If the tree is empty when we the run BFS function
           return 'Oops! Tree is empty.'
       else:
           BFS result = [] # Method will store the result of the BFS in a list
           queue = [] # Created variable to keep track of the children of each node
           queue.append(current node) # First, we add the root to the queue
           while len(queue) > 0:
               current node = queue.pop(0) # We extract the 1st element of the queue and make it current node
               BFS result.append(current node.data) # We push the data of current node to the result list, as we
               if current node.left: # If left child of current node exists, we append it to the queue
                    queue.append(current node.left)
               if current node.right: # If right child exists, we append it to the queue as well
                    queue.append(current node.right)
           return BFS result
   # We will also implement the recursive version of BFS:
   def recursive BFS(self, queue, BFS list):
       if self.root is None: # If the tree is empty when we the run BFS function
           return 'Oops! Tree is empty.'
       if len(queue) == 0:
           return BFS list
       current node = queue.pop(0)
       BFS list.append(current node.data)
       if current node.left:
           queue.append(current node.left)
       if current node.right:
           queue.append(current node.right)
       return self.recursive BFS(queue, BFS list)
```

```
In [3]:
         # Initializing a new tree with our blueprint
         new bst = BST()
         # 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 [4]:
         print(new_bst.BFS())
        [5, 3, 7, 1, 13, 0, 10, 65]
In [5]:
         print(new_bst.recursive_BFS([new_bst.root],[])) # We need to pass the root node as an array and an empty array for
        [5, 3, 7, 1, 13, 0, 10, 65]
```