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 [4]:

```
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:
                    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 == 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
        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 [5]:

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

# Populating tree
new_bst = BST()
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 [6]:

```
print(new_bst.BFS())
```

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

In [8]:

```
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]
```