A tree is a hierarchical data structure defined as a collection of nodes. They can have zero or more child nodes, as opposed to arrays, linked lists, stacks and queues. The structure of a tree is similar to the one used commonly to organise a family or a company. There's grandpa followed by his children, and each children will be followed by either no kids at all or more children, and so on. The structure of the DOM (Document Object Model) in HTML is a tree, as well.

In [1]:

```
# [Binary Trees] - Binary Search Tree
```

This is a type of tree where each node can only have either 0, 1 or 2 nodes as children, and each child can only have one parent. Each node has three attributes: root, left_child and right_child.

In most cases, the time complexity of operations for a Binary Search Tree is of O(log n), including operations like lookups, insertions and removals. However, in worse cases where the tree is quite unbalanced(vast majority of nodes stored on one side of the tree), it turns itself into a linked list. In this case, the time complexity of operations increase to O(n).

The best way to understand these concepts is by creating our own tree. We can start by implementing a Binary Search Tree that will end up unbalanced.

In [2]:

```
class Node():
    def __init__(self,data):
        self.data = data
        self.left = None
        self.right = None
# Because we need a place to store info about each node (data, and left and right pointers),
# we start by creating the Node class
# Now, we can start building our BST.
# It will have a constructor with root node initialised in None, along with 3 methods: insert, search and remove.

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
# The first step in creating insert() is checking if the root node is None. If so, we make it point to the new_nod
# If not, a temporary node is created that refers firstly the root node.
# After that, the data is compared between both the new_node and the node that the temporary node is referring.
# If the data on new_node is greater than the other one,
# the method checks if the child node on the right of the temporary node exists and,
# if so, updates the child on the right of the temporary node with new_node.
# Otherwise, the method will update the temporary node to refer the new_node as the child on the right.
# If the data on new_node is less than the data on the temporary node,
# the methods performs the same operation as above, only for the child on the left.
# On an average case, the time complexity of this operation is of O(log n). On worst case though, it is of O(n).

    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
# Next up, we create search(). This method operates in the same logic as insert()
# to find the node we want to look for.
# However, instead of inserting a new node, it will return a message displaying whether or not our node was found.
# So, if the node referred by the temporary node contains the same value that we are looking for,
# the method will output "Found it!".

    def remove(self, data):
        if self.root == None:
            return "Empty tree."
        current_node = self.root
        parent_node = None
# To reach the node we want or the end of our BST, the method needs to traverse it
        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
# If nothing prior occurs, it means the method found a match. So, we check some scenarios:
            else:
                if current_node.right == None: # If node has only a child on the left
                    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 node has only a child on the right
                    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 node has no children
                if parent_node == None: # Node to be deleted is the root node
                    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: # If node has both children
                rm_node = current_node.right
                rm_parent_node = current_node.right
                while rm_node.left != None:
# Method loops to reach the node that is at the very left end of the child on the right of the current node.
                    rm_parent_node = rm_node
                    rm_node = rm_node.left
                current_node.data = rm_node.data
        # Method copies the value that is going the replace the one removed
                if rm_node == rm_parent_node:
# If the node to be deleted is the direct child of the current node on its right side
                    current_node.right = rm_node.right
                    return
                if rm_node.right == None:
# If the node at the very left end of the child on the right of the current node does not have a child on the righ
                    rm_parent_node.left = None
                    return
                else: # If it does, the method connects it to the parent of rm_node.
                    rm_parent_node.left = rm_node.right
                    return
        return 'Not found.'

# At last, we reached the end of remove(). This is the most complicated to build of all 3.
# As this one is quite long, I left comments on the steps the method takes to remove a node,
# along with all the verifcations it performs during this operation.
```

After creating the BST class, all that is left to do is testing:

In [3]:
```
new_bst = BST()
# Used the BST blueprint to build a new object (new_bst) and inserted a few values
# to make a considerable-sized tree for the test.

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)

print(new_bst)
```

```
<__main__.BST object at 0x7fe589268ee0>
```

In [4]:
```
# It will look like this:
#
#                5
#            3       7
#         1              13
#       0            10      65


new_bst.remove(13)
```

In [5]:
```
print(new_bst.root.data)
```

```
5
```

In [6]:
```
#
#                5
#            3       7
#         1             65
#       0            10

new_bst.remove(5)
```

In [7]:
```
print(new_bst.root.data)
```

```
7
```

In [8]:
```
#
#                7
#            3       65
#         1       10
#       0


new_bst.search(8)
```

Out[8]:
```
'Not found.'
```

In [9]:
```
new_bst.search(1)
```

Out[9]:
```
'Found it!'
```

In [10]:
```
new_bst.remove(3)
```

In [11]:
```
print(new_bst.root.data)
```

```
7
```

In [12]:

```
#
#                7
#            1       65
#         0       10

new_bst.remove(7)
```

In [13]:

```
#
#                10
#            1       65
#         0

new_bst.remove(1)
```

In [14]:

```
#
#                10
#            0       65

new_bst.remove(0)
new_bst.remove(10)
new_bst.remove(65)
```

In [15]:

```
new_bst.insert(10)
print(new_bst.root.data) # 10
```

```
10
```

In [16]:

```
new_bst.remove(10)
```

In [17]:

```
new_bst.remove(5)
```

Out[17]:

```
'Empty tree.'
```