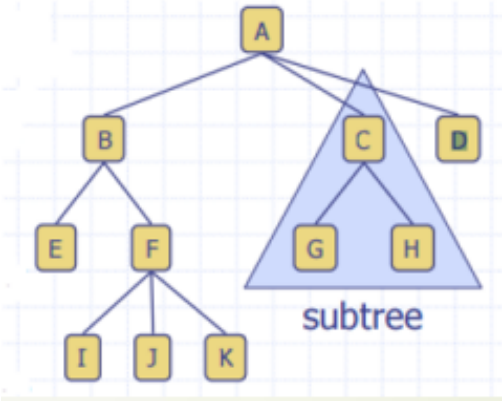
# Qishi - Tree

## Young Chen




• Root: The top node in a tree.(A)

• Internal node: A node with at least one child. (A, B, C, F)

• External node (a.k.a. leaf): A node with no children. (E, I, J, K, G, H, D)

• Ancestorsof a node: A node reachable by repeated proceeding from child to parent.

• Descendantof a node: A node reachable by repeated proceeding from parent to child. Also known as subchild. (A B C D E F G H I J K )

• Depthof a node:The depth of a node is the number of edges from the tree's root node to the node.. Depth(E) = 2

• Height:The height of a tree is the height of its root node. Height = 3

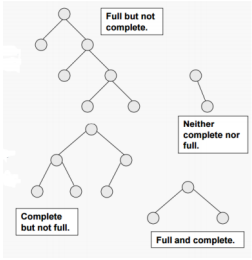
• Sibling: A group of nodes with the same parent. (C and D)

• Edgeof tree:The connection between one node and another.

• Path:A sequence of nodes and edges connecting a node with a descendant.

**A full binary tree (sometimes proper binary tree or 2-tree) is a tree in which every node other than the leaves has two children.**

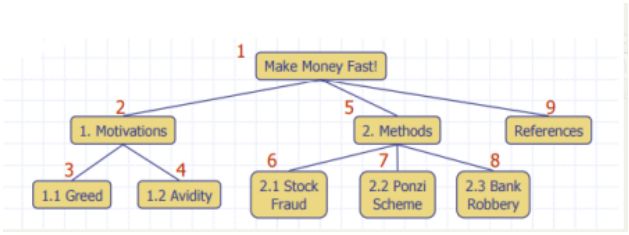
**A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.**




## Preorder Traversal:

Algorithm Preorder(tree)

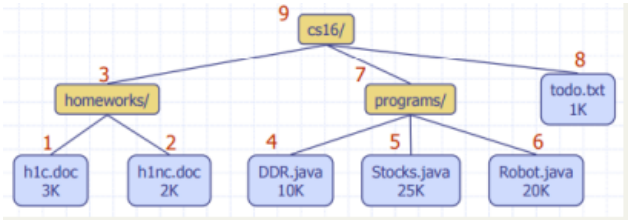
1. Visit the root.
2. Traverse the left subtree, i.e., call Preorder(left-subtree)
3. Traverse the right subtree, i.e., call Preorder(right-subtree)




## Postorder Traversal:

Algorithm Postorder(tree)

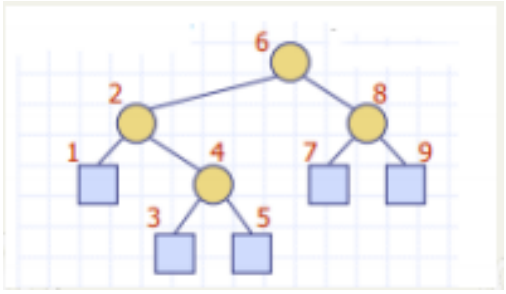
1. Traverse the left subtree, i.e., call Postorder(left-subtree)
2. Traverse the right subtree, i.e., call Postorder(right-subtree)
3. Visit the root.




## Inorder Traversal:

Algorithm Inorder(tree)

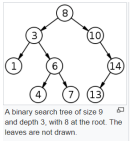
1. Traverse the left subtree, i.e., call Inorder(left-subtree)
2. Visit the root.
3. Traverse the right subtree, i.e., call Inorder(right-subtree)




## binary search trees (BST)

sometimes called ordered or sorted binary trees, are a particular type of container: data structures that store "items" (such as numbers, names etc.) in memory. They allow fast lookup, addition and removal of items, and can be used to implement either dynamic sets of items, or lookup tables that allow finding an item by its key (e.g., finding the phone number of a person by name).

Binary search trees keep their keys in sorted order, so that lookup and other operations can use the principle of binary search: when looking for a key in a tree (or a place to insert a new key), they traverse the tree from root to leaf, making comparisons to keys stored in the nodes of the tree and deciding, on the basis of the comparison, to continue searching in the left or right subtrees. On average, this means that each comparison allows the operations to skip about half of the tree, so that each lookup, insertion or deletion takes time proportional to the logarithm of the number of items stored in the tree.



A binary search tree of size 9 and depth 3, with 8 at the root. The leaves are not drawn.

| | Binary Search | ArrayList | LinkedList | BST |
|---|---|---|---|---|
| **Search** | lg N | N | N | *h* |
| **Insert** | N | N | N | *h* |
| **min/max** | 1 | N | N | *h* |
| **floor/Ceiling** | lg N | N | N | *h* |
| **Rank** | lg N | N | N | *h* |
| **Get** | 1 | 1 | N | *h* |
| **Ordered Iteration** | N | N lg N | N lg N | N |

In [1]:
```
class Node:
    __slots__ = '_item' , '_left' , '_right'

    def __init__ (self, item, left=None, right=None):
        self._item = item
        self._left = left
        self._right = right
```

In [2]:
```
class BinarySearchTree:

    def __init__ (self, root=None):
        self._root = root

    # Get methods
    def get(self, key):
        return self.__get(self._root, key);

    def __get(self, node, key):
        if (node is None):
            return None
        if (key == node._item):
            return node._item
        if (key < node._item):
            return self.__get(node._left, key)
        else:
            return self.__get(node._right, key)
    # add methods
    def add(self, value):
        self._root = self.__add(self._root, value)

    def __add(self, node, value): # return node
        if (node is None):
            return Node(value)
        if (value == node._item):
            pass
        else:
            if (value < node._item):
                node._left = self.__add(node._left, value)
            else:
                node._right = self.__add(node._right, value)
        return node
```

In [ ]:

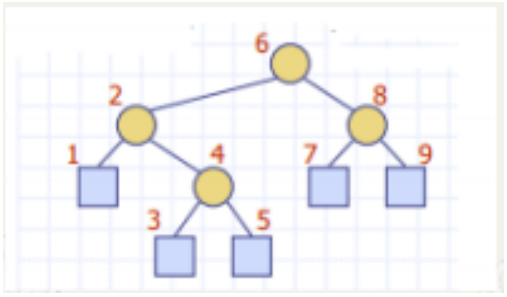
```
    # remove methods
    def remove(self, key):
        self._root = self.__remove(self._root, key)

    def __remove(self, node, key):  # helper
        if node is None:
            return None
        if (key < node._item):
            node._left = self.__remove(node._left, key)
        elif (key > node._item):
            node._right = self.__remove(node._right, key)
        else:
            if (node._left is None):
                node = node._right  # if right is None,  node = None; case 1: no child
                                    # if right is not None, node = node._right; case 2: on
            elif (node._right is None):
                node = node._left
            else:
                node._item = self.__get_max(node._left)
                node._left = self.__remove(node._left, node._item)

        return node

    # get max/min methods
    def get_max(self):
        return self.__get_max(self._root)

    def __get_max(self, node):
        if (node is None):
            return None
        while (node._right is not None):
            node = node._right
        return node._item
```



In [ ]:
```
# Traversal Methods
    def print_inorder(self):
        self._print_inorder(self._root)
        print('')

    def _print_inorder(self, node):
        if (node is None):
            return
        self._print_inorder(node._left)
        print ('[', node._item, ']', end = " ")
        self._print_inorder(node._right)

    def print_preorder(self):
        self._print_preorder(self._root)
        print('')

    def _print_preorder(self, node):
        if (node is None):
            return
        print ('[', node._item, ']', end = " ")
        self._print_preorder(node._left)
        self._print_preorder(node._right)

    def print_postorder(self):
        self._print_postorder(self._root)
        print('')

    def _print_postorder(self, node):
        if (node is None):
            return
        self._print_postorder(node._left)
        self._print_postorder(node._right)
        print ('[', node._item, ']', end = " ")
```

In [3]:
```
bst = BinarySearchTree()
numbers = [6, 4, 8, 7, 9, 2, 1, 3, 5, 13, 11, 10, 12]
for i in numbers:
    bst.add(i)
bst.print_inorder()
bst.print_postorder()
bst.print_preorder()
```

```
[ 1 ] [ 2 ] [ 3 ] [ 4 ] [ 5 ] [ 6 ] [ 7 ] [ 8 ] [ 9 ] [ 10 ] [ 11 ] [ 12 ] [ 13 ]
[ 1 ] [ 3 ] [ 2 ] [ 5 ] [ 4 ] [ 7 ] [ 10 ] [ 12 ] [ 11 ] [ 13 ] [ 9 ] [ 8 ] [ 6 ]
[ 6 ] [ 4 ] [ 2 ] [ 1 ] [ 3 ] [ 5 ] [ 8 ] [ 7 ] [ 9 ] [ 13 ] [ 11 ] [ 10 ] [ 12 ]
```

## Example.1 Tree Size

Calculate the size of the tree.

In [4]:
```
class AdvBST1(BinarySearchTree):

    def size(self):
        return self._size(self._root)

    def _size(self, node):
        if (not node):
            return 0
        return self._size(node._left) + self._size(node._right) + 1
```

In [5]:
```
bst = AdvBST1()
numbers = [6, 4, 8, 7, 9, 2, 1, 3, 5, 13, 11, 10, 12]
for i in numbers:
    bst.add(i)
bst.print_inorder()
bst.size()
```

```
[ 1 ] [ 2 ] [ 3 ] [ 4 ] [ 5 ] [ 6 ] [ 7 ] [ 8 ] [ 9 ] [ 10 ] [ 11 ] [ 12 ] [ 13 ]
```

Out[5]: 13

## Example.2 Max Depth

Calculate the max depth of a tree

In [6]:
```
class AdvBST2(AdvBST1):
    def maxDepth(self):
        return self._maxDepth(self._root)

    def _maxDepth(self, node):
        if (not node):
            return 0
        left_depth = self._maxDepth(node._left)
        right_depth = self._maxDepth(node._right)
        return max(left_depth, right_depth) + 1
```

In [7]:
```
bst = AdvBST2()
numbers = [6, 4, 8, 7, 9, 2, 1, 3, 5, 13, 11, 10, 12]
for i in numbers:
    bst.add(i)
bst.print_inorder()
bst.maxDepth()
```

```
[ 1 ] [ 2 ] [ 3 ] [ 4 ] [ 5 ] [ 6 ] [ 7 ] [ 8 ] [ 9 ] [ 10 ] [ 11 ] [ 12 ] [ 13 ]
```

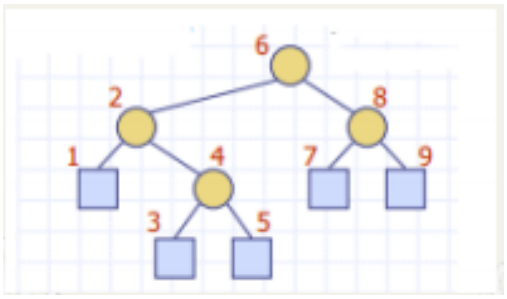
Out[7]: 6

## Example.3. Is Balance Tree

In [8]:
```
class AdvBST3(AdvBST2):
    def minDepth(self):
        return self._minDepth(self._root)

    def _minDepth(self, node):
        if (not node):
            return 0
        left_depth = self._minDepth(node._left)
        right_depth = self._minDepth(node._right)
        return min(left_depth, right_depth) + 1

    def isBalanced(self):
        return (self.maxDepth() - self.minDepth()) <= 1
```




In [9]:
```
bst = AdvBST3()
numbers = [1,2,3,4,5,6,7,8]
for i in numbers:
    bst.add(i)
bst.print_inorder()
bst.isBalanced()
```

```
[ 1 ] [ 2 ] [ 3 ] [ 4 ] [ 5 ] [ 6 ] [ 7 ] [ 8 ]
```

Out[9]: False

In [10]:
```
bst = AdvBST3()
numbers = [3,1,5]
for i in numbers:
    bst.add(i)
bst.print_inorder()
bst.isBalanced()
```

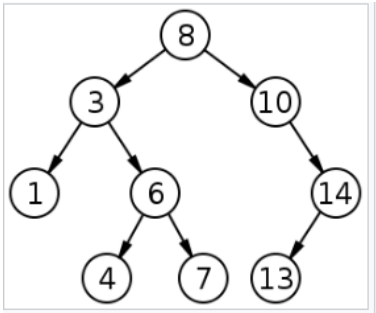
```
[ 1 ] [ 3 ] [ 5 ]
```

Out[10]: True

## Example.4 Floor and Ceiling

In [11]:
```
class AdvBST4(AdvBST3):
    def floor(self, key):
        return self._floor(self._root, key)

    def _floor(self, node, key):
        if (not node):
            return None
        if (key == node._item):
            return node
        if (key < node._item):
            return self._floor(node._left, key)
        t = self._floor(node._right, key)
        if t:
            return t
        return node
```




A binary search tree of size 9 and depth 3, with 8 at the root. The leaves are not drawn.

In [12]:
```
bst = AdvBST4()
numbers = [40,20,70,50,10,60,30,80]
for i in numbers:
    bst.add(i)
print(bst.floor(40)._item)
print(bst.floor(44)._item)
print(bst.floor(10)._item)
print(bst.floor(5))
print(bst.floor(100)._item)
```

```
40
40
10
None
80
```

## Example .5 Is Binary Search Tree

Check whether a given tree a binary search tree.

In [13]:
```
import sys
class AdvBST5(AdvBST4):
    def isBST(self):
        return self._isBST(self._root, -sys.maxsize, sys.maxsize)

    def _isBST(self, node, minval, maxval):
        if not node:
            return True
        if (node._item < minval or node._item > maxval):
            return False
        return self._isBST(node._left, minval, node._item) and self._isBST(node._right, no
```

In [14]:
```
bst = AdvBST5()
numbers = [1,2,3,4,5,6,7,8]
for i in numbers:
    bst.add(i)
bst.isBST()
```

Out[14]: True

In [15]:
```
node1 = Node(1)
node2 = Node(2)
node3 = Node(3)
node4 = Node(4)
node5 = Node(5)

node3._left = node1
node3._right = node5
node1._left = node2
node5._right = node4
bst = AdvBST5(node3)
bst.print_inorder()
bst.isBST()
```

```
[ 2 ] [ 1 ] [ 3 ] [ 5 ] [ 4 ]
```

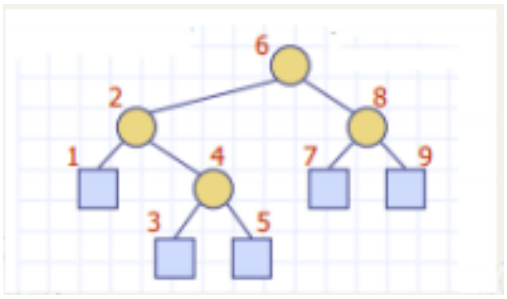
Out[15]: False

## Example.6 Is Mirror Tree ?

In [16]:
```
class AdvBST6(AdvBST5):
    def mirror(self):
        self._mirror(self._root)

    def _mirror(self, node):
        if (node is not None):
            self._mirror(node._left)
            self._mirror(node._right)

            temp = node._left
            node._left = node._right
            node._right = temp
```




In [17]:
```
bst = AdvBST6()
numbers = [6, 4, 8, 7, 9, 5, 1, 3, 2]
for i in numbers:
    bst.add(i)
bst.print_inorder()
```

```
[ 1 ] [ 2 ] [ 3 ] [ 4 ] [ 5 ] [ 6 ] [ 7 ] [ 8 ] [ 9 ]
```

In [18]:
```
bst.mirror()
bst.print_inorder()
```

```
[ 9 ] [ 8 ] [ 7 ] [ 6 ] [ 5 ] [ 4 ] [ 3 ] [ 2 ] [ 1 ]
```

## Homework

278 First Bad Version

162 Find Peak Element

153 Find Minimum in Rotated Sorted Array

272(optional) Closest Binary Search Tree Value II