In [1]:

```
#Implement Binary tree
class Node:
    def __init__(self,key):
        self.left = None
        self.right = None
        self.val = key



root = Node(1)
print(''' following is the tree after above statement
        1
      /   \
     None  None''')

root.left      = Node(2);
root.right     = Node(3);

print(''' 2 and 3 become left and right children of 1
           1
         /   \
        2      3
     /    \    /  \
   None None None None''')


root.left.left  = Node(4);

print('''4 becomes left child of 2
           1
       /       \
      2          3
    /   \       /  \
   4    None  None  None
  /  \
None None''')
```

```
 following is the tree after above statement
       1
     /        None  None
 2 and 3 become left and right children of 1
          1
        /          2      3
    /    \    /     None None None None
4 becomes left child of 2
          1
      /             2          3
   /   \       /     4    None  None  None
  /  None None
```

In [2]:

```
#Find height of a given tree
class Node:


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


def maxDepth(node):
    if node is None:
        return -1 ;

    else :


        lDepth = maxDepth(node.left)
        rDepth = maxDepth(node.right)


        if (lDepth > rDepth):
            return lDepth+1
        else:
            return rDepth+1



root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)


print ("Height of tree is %d" %(maxDepth(root)))
```

```
Height of tree is 2
```

In [3]:

```
#Perform Pre-order, Post-order, In-order traversal
class Node:
    def __init__(self, key):
        self.left = None
        self.right = None
        self.val = key



def printInorder(root):

    if root:


        printInorder(root.left)


        print(root.val)


        printInorder(root.right)



def printPostorder(root):

    if root:


        printPostorder(root.left)


        printPostorder(root.right)


        print(root.val)



def printPreorder(root):

    if root:

        print(root.val)


        printPreorder(root.left)


        printPreorder(root.right)



root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)
print("Preorder traversal of binary tree is")
printPreorder(root)

print("\nInorder traversal of binary tree is")
printInorder(root)

print("\nPostorder traversal of binary tree is")
printPostorder(root)
```

```
Preorder traversal of binary tree is
1
2
4
5
3

Inorder traversal of binary tree is
4
2
5
1
3

Postorder traversal of binary tree is
4
5
2
3
1
```

In [4]:

```
#Function to print all the leaves in a given binary tree
class Node:

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


def printLeafNodes(root: Node) -> None:


    if (not root):
        return

    if (not root.left and
        not root.right):
        print(root.data,
              end = " ")
        return


    if root.left:
        printLeafNodes(root.left)


    if root.right:
        printLeafNodes(root.right)


if __name__ == "__main__":


    root = Node(1)
    root.left = Node(2)
    root.right = Node(3)
    root.left.left = Node(4)
    root.right.left = Node(5)
    root.right.right = Node(8)
    root.right.left.left = Node(6)
    root.right.left.right = Node(7)
    root.right.right.left = Node(9)
    root.right.right.right = Node(10)


    printLeafNodes(root)
```

```
4 6 7 9 10
```

In [5]:

```
#Implement BFS (Breath First Search) and DFS (Depth First Search)
from collections import defaultdict


class Graph:


    def __init__(self):


        self.graph = defaultdict(list)


    def addEdge(self,u,v):
        self.graph[u].append(v)

    def BFS(self, s):


        visited = [False] * (max(self.graph) + 1)

        queue = []

        queue.append(s)
        visited[s] = True

        while queue:

            s = queue.pop(0)
            print (s, end = " ")


            for i in self.graph[s]:
                if visited[i] == False:
                    queue.append(i)
                    visited[i] = True


g = Graph()
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(1, 2)
g.addEdge(2, 0)
g.addEdge(2, 3)
g.addEdge(3, 3)

print ("Following is Breadth First Traversal"
                  " (starting from vertex 2)")
g.BFS(2)
```

```
Following is Breadth First Traversal (starting from vertex 2)
2 0 3 1
```

In [8]:

```
# Find sum of all left leaves in a given Binary Tree
class Node:


    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None

def leftLeavesSumRec(root, isLeft, summ):
    if root is None:
        return


    if root.left is None and root.right is None and isLeft == True:
        summ[0] += root.key


    leftLeavesSumRec(root.left, 1, summ)
    leftLeavesSumRec(root.right, 0, summ)



def leftLeavesSum(root):
    summ = [0]


    leftLeavesSumRec(root, 0, summ)

    return summ[0]


root = Node(20);
root.left= Node(9);
root.right   = Node(49);
root.right.left = Node(23);
root.right.right= Node(52);
root.right.right.left  = Node(50);
root.left.left  = Node(5);
root.left.right = Node(12);
root.left.right.right  = Node(12);

print ("Sum of left leaves is", leftLeavesSum(root))
```

```
Sum of left leaves is 78
```

In [6]:

```
# Find sum of all nodes of the given perfect binary tree
def SumNodes(l):


    leafNodeCount = pow(2, l - 1)

    vec = [[] for i in range(l)]

    for i in range(1, leafNodeCount + 1):
        vec[l - 1].append(i)


    for i in range(l - 2, -1, -1):
        k = 0


        while (k < len(vec[i + 1]) - 1):


            vec[i].append(vec[i + 1][k] +
                          vec[i + 1][k + 1])
            k += 2

    Sum = 0


    for i in range(l):
        for j in range(len(vec[i])):
            Sum += vec[i][j]

    return Sum


if __name__ == '__main__':
    l = 3

    print(SumNodes(l))
```

```
30
```

In [7]:

```
#Count subtress that sum up to a given value x in a binary tree
class Node:

    def __init__(self, data):
```

```
        self.data = data
        self.left = None
        self.right = None


def getNode(data):

    newNode = Node(data)
    return newNode

count = 0
ptr = None

def countSubtreesWithSumXUtil(root, x):

    global count, ptr

    l = 0
    r = 0

    if (root == None):
        return 0

    l += countSubtreesWithSumXUtil(root.left, x)
    r += countSubtreesWithSumXUtil(root.right, x)

    if (l + r + root.data == x):
        count += 1

    if (ptr != root):
        return l + root.data + r

    return count


if __name__=='__main__':

    ''' binary tree creation
              5
            /  \
          -10   3
          / \  / \
          9 8 -4 7
    '''

    root = getNode(5)
    root.left = getNode(-10)
    root.right = getNode(3)
    root.left.left = getNode(9)
    root.left.right = getNode(8)
    root.right.left = getNode(-4)
    root.right.right = getNode(7)

    x = 7
    ptr = root

    print("Count = " + str(countSubtreesWithSumXUtil(
        root, x)))
```

```
Count = 2
```

In [9]:

```
#Find maximum level sum in Binary Tree
from collections import deque


class Node:

    def __init__(self, key):

        self.data = key
        self.left = None
        self.right = None


def maxLevelSum(root):


    if (root == None):
        return 0


    result = root.data


    q = deque()
    q.append(root)

    while (len(q) > 0):



        count = len(q)


        sum = 0
        while (count > 0):


            temp = q.popleft()


            sum = sum + temp.data


            if (temp.left != None):
                q.append(temp.left)
            if (temp.right != None):
                q.append(temp.right)

            count -= 1


        result = max(sum, result)

    return result


if __name__ == '__main__':

    root = Node(1)
    root.left = Node(2)
    root.right = Node(3)
    root.left.left = Node(4)
    root.left.right = Node(5)
    root.right.right = Node(8)
    root.right.right.left = Node(6)
    root.right.right.right = Node(7)


    print("Maximum level sum is", maxLevelSum(root))
```

```
Maximum level sum is 17
```

In [10]:

```
# Print the nodes at odd levels of a tree
class newNode:
    def __init__(self, data):
        self.data = data
        self.left = self.right = None

def printOddNodes(root, isOdd = True):


    if (root == None):
        return


    if (isOdd):
        print(root.data, end = " ")

    printOddNodes(root.left, not isOdd)
    printOddNodes(root.right, not isOdd)

if __name__ == '__main__':
    root = newNode(1)
    root.left = newNode(2)
    root.right = newNode(3)
    root.left.left = newNode(4)
    root.left.right = newNode(5)
    printOddNodes(root)
```

```
1 4 5
```

In [ ]: