Binary trees

In [1]:

```
#creating node
class TreeNode:
    def __init__(self,val):
        self.val=val
        self.left=None
        self.right=None

# creating binary tree
def create_binary_tree(arr,root,i,n):
    if i<n:
        root=tem
        root.left=create_binary_tree(arr,root.left,2*i,n)
        root.right=create_binary_tree(arr,root.right,2*i+1,n)
    return root
arr=[3,6,9,12,5,7,4,8]
root=None
binary_tree=create_binary_tree(arr,root,1,len(arr)+1)
print(binary_tree)
```

```
<__main__.TreeNode object at 0x00000001D1AB9370>
```

DFS

In [11]:

```
#DFS
def preorder(root):
    if not root:
        return
    print(root.val,end=' ')
    preorder(root.left)
    preorder(root.right)

def inorder(root):
    if not root:
        return
    inorder(root.left)
    print(root.val,end=' ')
    inorder(root.right)

def postorder(root):
    if not root:
        return
    postorder(root.left)
    postorder(root.right)
    print(root.val,end=' ')

#binary_tree
preorder(binary_tree)

print('\n')
inorder(binary_tree)

print('\n')
postorder(binary_tree)
```

```
3 6 12 8 5 9 7 4

8 12 6 5 3 7 9 4

8 12 5 6 7 4 9 3
```

BFS

In [12]:

```
#BFS
def levelorder(root):
    if not root:
        return []
    result=[]
    queue=[]
    queue.append(root)
    while queue:
        result.append(queue[0].val)
        node=queue.pop(0)
        if node.left:
            queue.append(node.left)
        if node.right:
            queue.append(node.right)
    print(result)
levelorder(binary_tree)
```

```
[3, 6, 9, 12, 5, 7, 4, 8]
```

SEARCH ELEMENT

In [ ]:

```
def searchElement(root,ele):
    if not root:
        return False
    if root.val==ele:
        return True
    return searchElement(root.left,ele) or searchElement(root.right,ele)
searchElement(binary_tree,12)
```

PRINT TREE

In [17]:

```
def printTree(root):
    if not root:
        return
    print(root.val,end=' ')
    printTree(root.left)
    printTree(root.right)

printTree(binary_tree)
```

```
3 6 12 8 5 9 7 4
```

LEAF NODES

In [18]:

```
def leafNodes(root,leaf):
    if not root:
        return
    ele=root.val
    if not root.left and not root.right:
        leaf+=[ele]
    leafNodes(root.left,leaf)
    leafNodes(root.right,leaf)
    return leaf
leafNodes(binary_tree,leaf=[])
```

Out[18]:

```
[8, 5, 7, 4]
```

In [19]:

```
class TreeNode:
    def __init__(self,val=0):
        self.val=val
        self.left=None
        self.right=None
class Solution():
    def buildTree(self,preorder,inorder):
        if not preorder or not inorder:
            return None

        root=TreeNode(preorder[0])
        mid=inorder.index(preorder[0])

        root.left=self.buildTree(preorder[1:mid+1],inorder[:mid])
        root.right=self.buildTree(preorder[mid+1:],inorder[mid+1:])

        print(root.val,end=' ')

obj=Solution()
obj.buildTree([3,6,12,8,5,9,7,4],[8,12,6,5,3,7,9,4])
print('\n')
obj.buildTree([3,9,20,15,7],[9,3,15,20,7])
```

```
8 12 5 6 7 4 9 3

9 15 7 20 3
```

binary search tree (BST)

In [1]:

```
class TreeNode:
    def __init__(self,val,left=None,right=None):
        self.val=val
        self.left=left
        self.right=right
def BST(root,key):
    if not root: return TreeNode(key)
    elif root.val<key:
        root.right=BST(root.right,key)
    else:
        root.left=BST(root.left,key)
    return root
arr=[8,3,10,1,6,14,4,7]
root=TreeNode(arr[0])
for i in range(1,len(arr)):
    tree=BST(root,arr[i])
tree
```

Out[1]:

```
<__main__.TreeNode at 0x2dc994310>
```

DFS (preorder,inorder,postorder)

In [6]:

```
def preorder(root,order):
    if not root: return
    order+=[root.val]
    preorder(root.left,order)
    preorder(root.right,order)
    return order
preorder(tree,order=[])
```

Out[6]:

```
[8, 3, 1, 6, 4, 7, 10, 14]
```

In [7]:

```
def inorder(root,order):
    if not root: return
    inorder(root.left,order)
    order+=[root.val]
    inorder(root.right,order)
    return order
inorder(tree,order=[])
```

Out[7]:

```
[1, 3, 4, 6, 7, 8, 10, 14]
```

In [8]:

```
def postorder(root,order):
    if not root: return
    postorder(root.left,order)
    postorder(root.right,order)
    order+=[root.val]
    return order
postorder(tree,order=[])
```

Out[8]:

```
[1, 4, 7, 6, 3, 14, 10, 8]
```

searching an element

In [9]:

```
def searchElement(root,ele):
    if not root: return False
    if root.val==ele: return True
    return searchElement(root.left,ele) or searchElement(root.right,ele)
searchElement(tree,101)
```

Out[9]:

```
False
```

In [13]:

```
class Stack:
    def __init__(self):
        self.container=[]

    def push(self,data):
        return self.container.append(data)

    def pop(self):
        #underflow condition
        if self.length()==0:
            print("stack is empty")
            return
        return self.container.pop()

    def length(self):
        return len(self.container)

    def peek(self):   #top of the element
        if self.length()==0:
            print("stack is empty")
            return
        return self.container[-1]
obj=Stack()
obj.push(7)
obj.push(11)
obj.push(70)
obj.peek()
obj.length()
```

Out[13]:

```
3
```

In [1]:

```
from collections import deque
class Queue:
    def __init__(self):
        self.container=deque()

    def enqueue(self,data):
        return self.container.append(data)

    def dequeue(self):
        return self.container.popleft()

    def front(self):
        return self.container[0]

    def rear(self):
        return self.container[-1]

    def length(self):
        return len(self.container)
    def print(self):
        return self.container

obj=Queue()
obj.enqueue(7)
obj.enqueue(17)
obj.enqueue(27)
obj.front()
obj.rear()
obj.dequeue()
obj.print()
```

Out[1]:

```
deque([17, 27])
```

In [1]:

```
#creating a node
class Node:
    def __init__(self,data):
        self.data=data
        self.next=None
#creating sll
class LinkedList:
    def __init__(self):
        self.head=None

    def print(self):
        if self.head is None:
            print("sll is empty")
            return
        itr=self.head
        lstr=''
        while itr:
            lstr+=str(itr.data)+'--->' if itr.next else str(itr.data)
            itr=itr.next
        return lstr

    def length(self):
        l=0
        itr=self.head
        while itr:
            l+=1
            itr=itr.next
        return l
    def insert_beg(self,data):
        temp=self.head
        node=Node(data)
        self.head=node
        node.next=temp
    def insert_end(self,data):
        node=Node(data)
        if self.head is None:
            self.head=node
        itr=self.head
        while itr.next:
            itr=itr.next
        itr.next=node
    def insert_at(self,pos,data):
        if pos < 0 or pos>=self.length():
            raise Exception("invalid positon")
            return
        if pos==0:
            temp=self.head
            node=Node(data)
            self.head=node
            node.next=temp
            return
        elif pos==self.length():
            node=Node(data)
            if self.head is None:
                self.head=node
            itr=self.head
            while itr.next:
                itr=itr.next
            itr.next=node
```

```
            return
        c=0
        itr=self.head
        while itr:
            if c==pos-1:
                node=Node(data)
                temp=itr.next
                itr.next=node
                node.next=temp
                return
            itr=itr.next
            c+=1
    def delete_beg(self):
        if not self.head:
            return None
        temp=self.head
        self.head=self.head.next
        temp=None
    def delete_end(self):
        if not self.head:
            return None
        if self.head.next==None:
            self.head=None
        itr=self.head
        while itr.next.next:
            itr=itr.next
        itr.next=None
    def delete_at(self,pos):
        if pos<0 or pos>self.length():
            raise Exception('invalid index')
            return
        if pos==0:
            temp=self.head
            self.head=self.head.next
            temp=None
            return
        c=0
        itr=self.head
        while(itr):
            if c==pos-1:
                itr.next=itr.next.next
                return
            itr=itr.next
            c+=1

    def search_ele(self,ele):
        if not self.head:
            return False
        itr=self.head
        while(itr):
            if ele==itr.data:
                return True
            itr=itr.next
        return False

    def reverse(self):
        if self.head is None:
            return
        curr=self.head
        prev=None
        while(curr):
```

```
            next=curr.next
            curr.next=prev
            prev=curr
            curr=next
        self.head=prev
```

In [3]:

```
obj=LinkedList()
obj.head=Node(1)
second=Node(2)
third=Node(3)
fourth=Node(400)
fifth=Node(50)
sixth=Node(6)
obj.head.next=second
second.next=third
third.next=fourth
fourth.next=fifth
fifth.next=sixth
obj.insert_beg(1000)
```

In [4]:

```
obj.insert_end(500)
```

In [5]:

```
obj.insert_at(3,999)
```

In [6]:

```
obj.print()
```

Out[6]:

```
'1000--->1--->2--->999--->3--->400--->50--->6--->500'
```

In [7]:

```
obj.reverse()
```

In [8]:

```
obj.print()
```

Out[8]:

```
'500--->6--->50--->400--->3--->999--->2--->1--->1000'
```

In [10]:

```
obj.search_ele(10)
```

Out[10]:

```
False
```

In [11]:

```
obj.delete_beg()
```

In [12]:

```
obj.print()
```

Out[12]:

```
'6--->50--->400--->3--->999--->2--->1--->1000'
```

In [13]:

```
obj.delete_end()
```

In [14]:

```
obj.print()
```

Out[14]:

```
'6--->50--->400--->3--->999--->2--->1'
```

In [15]:

```
obj.delete_at(6)
```

In [16]:

```
obj.print()
```

Out[16]:

```
'6--->50--->400--->3--->999--->2'
```

In [18]:

```
obj.length()
```

Out[18]:

```
6
```

In [15]:

```
def add_node(v,nodes,adj):
    global node_count
    if v in nodes:
        print(v,"alreadt presented")
    else:
        node_count+=1
        nodes+=[v]

        for n in adj:
            n+=[0]
        adj+=[[0]*node_count]

nodes=[]
adj=[]
node_count=0
print("before adding nodes")
print(nodes)
print(adj)
add_node("A",nodes=[],adj=[])
add_node("B",nodes=[],adj=[])
print("after adding nodes")
print("nodes:",nodes)
print("graph",adj)
```

```
before adding nodes
[]
[]
after adding nodes
nodes: []
graph []
```

In [1]:

```
num_nodes=5
edges=[(0,1),(0,4),(1,2),(1,3),(1,4),(2,3),(3,4)]

class Graph:
    def __init__(self,num_nodes,edges):
        self.num_nodes=num_nodes
        self.data=[[] for _ in range(num_nodes)]

        for n1,n2 in edges:
            self.data[n1].append(n2)
            self.data[n2].append(n1)

    def __repr__(self):
        return '\n'.join(["{}:{}".format(n,neighbours) for n,neighbours in enumerate(graph.

    def __str__(self):
        return self.__repr__()
```

In [2]:

```
graph=Graph(num_nodes,edges)
```

In [3]:

```
graph
```

Out[3]:

```
0:[1, 4]
1:[0, 2, 3, 4]
2:[1, 3]
3:[1, 2, 4]
4:[0, 1, 3]
```

In [4]:

```
["{}:{}".format(n,neighbours) for n,neighbours in enumerate(graph.data)]
```

Out[4]:

```
['0:[1, 4]', '1:[0, 2, 3, 4]', '2:[1, 3]', '3:[1, 2, 4]', '4:[0, 1, 3]']
```

In [9]:

```
graph.data
```

Out[9]:

```
[[1, 4], [0, 2, 3, 4], [1, 3], [1, 2, 4], [0, 1, 3]]
```

In [ ]:

```
def BFS(graph,root):
    queue=[]
    discovered=[False]*len(graph.data)

    discovered[root]=True
    queue.append(root)
    idx=0

    while idx<len(queue):
        current=queue[idx]
        idx+=1
```

DLL

In [79]:

```
class Node:
    def __init__(self,data):
        self.data=data
        self.nref=None
        self.pref=None
class DLL:
    def __init__(self):
        self.head=None
        self.length=0
    def ft(self):
        if not self.head: return 'empty'
        itr=self.head
        s=''
        while(itr):
            self.length+=1
            s+=str(itr.data)+'--->' if itr.nref else str(itr.data)
            itr=itr.nref
        return s
    def rt(self):
        if not self.head: return 'empty'
        temp=itr=self.head
        s=''
        while(itr.nref):
            itr=itr.nref
        while(itr):
            s+=str(itr.data)+'--->' if itr.nref is not temp else str(itr.data)
            itr=itr.pref
        return s
    def insert_beg(self,data):
        node=Node(data)
        if not self.head:
            self.head=node
            return
        node.nref=self.head
        self.head.pref=node
        self.head=node
    def insert_end(self,data):
        node=Node(data)
        if not self.head:
            self.head=node
            return
        itr=self.head
        while(itr.nref):
            itr=itr.nref
        node.pref=itr
        itr.nref=node
        node.nref=None
    def insert_at(self,data,pos):
        node=Node(data)
        if not self.head:
            self.head=node
        if pos<0 or pos>self.length:
            return 'not possible'
        if pos==0: return insert_beg(data)
        if pos==self.length: return insert_end(data)
        c=0
        itr=self.head
        while(itr):
            c+=1
```

```
            if c==pos:
                temp=itr.nref
                node.pref=itr
                itr.nref=node
                node.nref=temp
                temp.pref=node
                return
            itr=itr.nref
```

In [80]:

```
dll=DLL()
```

In [81]:

```
dll.ft()
```

Out[81]:

```
'empty'
```

In [82]:

```
dll.rt()
```

Out[82]:

```
'empty'
```

In [83]:

```
dll.insert_beg(2)
dll.insert_beg(4)
dll.insert_beg(6)
```

In [84]:

```
dll.ft()
```

Out[84]:

```
'6--->4--->2'
```

In [85]:

```
dll.rt()
```

Out[85]:

```
'2--->4--->6--->'
```

In [86]:

```
dll.insert_end(8)
```

In [87]:

```
dll.ft()
```

Out[87]:

```
'6--->4--->2--->8'
```

In [88]:

```
dll.rt()
```

Out[88]:

```
'8--->2--->4--->6--->'
```

In [91]:

```
dll.insert_at(10,4)
dll.ft()
```

Out[91]:

```
'6--->4--->2--->8--->10--->10'
```

CLL

In [1]:

```
class Node:
    def __init__(self,val):
        self.val=val
        self.next=None
```

In [8]:

```
class CLL:
    def __init__(self):
        self.head=None
        self.tail=None
    def display(self):
        if not self.head: return 'empty cll'
        itr=self.head
        s=''
        while(itr.next!=self.head):
            s+=str(itr.val)+'-->'
            itr=itr.next
        s+=str(itr.val)
        return s
```

In [9]:

```
c=CLL()
```

In [14]:

```
n1=Node(10)
n2=Node(2)
n3=Node(6)
c.head=n1
c.tail=n1
c.tail.ext=c.head
n1.next=n2
n2.next=n3
n3.next=n1
```

In [15]:

```
c.display()
```

Out[15]:

```
'10-->2-->6'
```

In [17]:

```
l=[1,6,4]
l==sorted(l)
```

Out[17]:

```
False
```

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]: