PART-1

Stack Implementation

Stack is a linear Data Structure with last in first out where it can be implemented in 2 different ways such as

-lists

-linked lists

In [7]:
```
##stack through linkedlists.
class Node:                                          #creating Node Class which contains data and next address
    def __init__(self,data):
        self.data=data
        self.next=None



class Stack:                                         #creating a stack class to link the nodes and do the operations
    def __init__(self):                              # constructor with head Node as None and length=0
        self.head=None
        self.length=0



    def push(self,data):                             #pushing the element at the top
        new_node=Node(data)                          # creating node object
        new_node.next=self.head                      # new node->next=head
        self.head=new_node                           #head=newnode
        self.length+=1                               #length is increased by 1 when pushing an element



    def pop(self):                                   # delete the element which is at the top
        if self.length==0:                           # if stack is empty return nothing
            return None
        top=self.head                                # the top most element is head
        self.head=self.head.next                     # head=head->next
        self.length-=1                               #length is decreased by 1 when poping an element
        top.next=None                                # top most element top->next=None
        return self.head.data                        # returning the head->data


    def peek(self):                                 # to finding the top most element if stack is empty returns None
         if self.length==0:                         # else returns head->data
            return None
         return self.head.data


    def isEmpty(self):                              # checks stack is empty or not
        return self.length==0


if __name__=='__main__':
    stk=Stack()
    print('Stack is empty or not',stk.isEmpty())
    stk.push(1)
    stk.push(2)
    stk.push(3)
    print('Top Most element in stack',stk.peek())
    stk.pop()
    print('Top Most element in stack after pop',stk.peek())
    print('Stack is empty or not',stk.isEmpty())
```

```
Stack is empty or not True
Top Most element in stack 3
Top Most element in stack after pop 2
Stack is empty or not False
```

Queue Implentation

Queue is a linear data structure which follows first in last out where it can be implemented through lists and linked lists

In [14]:
```
class Node:                                          #creating Node Class which contains data and next address
    def __init__(self,data):
        self.data=data
        self.next=None
class Queue:
    def __init__(self):
        self.head=None
        self.tail=None
        self.length=0
    def enqueue(self,data):
        new_node=Node(data)
        self.length+=1
        temp=self.head
        if self.head is None:
            self.head=new_node
            self.tail=new_node
        else:
            self.tail.next=new_node
            self.tail=new_node

    def dequeue(self):
        if self.length==0:
            return None
        temp=self.head
        self.head=self.head.next
        temp.next=None
        self.length-=1
        if self.length==0:
            return None
        temp.next=None
        return self.head.data



    def peek(self):                                 # if queue is empty returns None
         if self.length==0:                         # else returns head->data
            return None
         return self.head.data


    def isEmpty(self):                              # checks stack is empty or not
        return self.length==0

if __name__=='__main__':
    que=Queue()
    que.enqueue(9)
    que.enqueue(22)
    que.enqueue(33)
    print(que.dequeue())
    que.enqueue(9)
    que.enqueue(22)
    que.enqueue(33)
    print(que.dequeue())
    print(que.peek())
```

```
22
33
33
```

Binary Search Tree Implementation

In [15]:

```
Inorder traversal after insertion: 2 4 5 6 8 9 10
```

In [23]:
```
class Node:                             # Constructor to create a new node
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None

def inorder(root):
    if root is not None:
        inorder(root.left)
        print(root.key, end=" ")
        inorder(root.right)

def insert(node, key):                  # If the tree is empty,returns new node
    if node is None:
        return Node(key)
    if key < node.key:
        node.left = insert(node.left, key)
    else:
        node.right = insert(node.right, key)

    return node

def deleteNode(root, key):
# Base Case
    if root is None:
        return root
    if key < root.key:                      # Recursive calls for ancestors of node to be deleted
        root.left = deleteNode(root.left, key)
        return root

    elif(key > root.key):
        root.right = deleteNode(root.right, key)
        return root


    if root.left is None and root.right is None:
        return None

                                         # If one of the children is empty
    if root.left is None:
        temp = root.right
        root = None
        return temp

    elif root.right is None:
        temp = root.left
        root = None
        return temp

    # If both children exist

    succParent = root
    # Find Successor

    succ = root.right
    while succ.left != None:
        succParent = succ
        succ = succ.left

    if succParent != root:
        succParent.left = succ.right
    else:
        succParent.right = succ.right
                                             # Copy Successor Data to root

    root.key = succ.key

    return root
def search(root,key):
    if root is None or root.key == key:
        return root.key
    if root.key < key:
        return search(root.right,key)
    return search(root.left,key)
def totalNodes(root):
  # Base case
    if(root == None):
        return 0

    # Find the left height and the
    # right heights
    l = totalNodes(root.left)
    r = totalNodes(root.right)

    return 1 + l + r

root = None
root = insert(root, 50)
root = insert(root, 30)
root = insert(root, 20)
root = insert(root, 40)
root = insert(root, 70)
root = insert(root, 60)
root = insert(root, 80)

print("Inorder traversal of the given tree")
inorder(root)
print('\n\nsearch element 50')
print(search(root,50))
print("\n\nDelete 20")
root = deleteNode(root, 20)
print("Inorder traversal of the modified tree")
inorder(root)

print("\n\nDelete 30")
root = deleteNode(root, 30)
print("Inorder traversal of the modified tree")
inorder(root)

print("\n\nDelete 50")
root = deleteNode(root, 50)
print("Inorder traversal of the modified tree")
inorder(root)
print('numberr of nodes')
print(totalNodes(root))
```

```
Inorder traversal of the given tree
20 30 40 50 60 70 80

search element 50
50


Delete 20
Inorder traversal of the modified tree
30 40 50 60 70 80

Delete 30
Inorder traversal of the modified tree
40 50 60 70 80

Delete 50
Inorder traversal of the modified tree
40 60 70 80 numberr of nodes
4
```

PART-2

Problem 1: Anagram Checker

Write a Python function that takes in two strings and returns True if they are anagrams of each other, else False. An anagram is a word or phrase formed by rearranging the letters of a different word or phrase, typically using all the original letters exactly once.

In [ ]:
```
def anagrams(s1,s2):
    if len(s1)!=len(s2):
        return False
    counts={}
    for i,j in zip(s1,s2):
        if i in counts.keys():                                    ##counts the alphabets in s1
            counts[i]+=1
        else:
            counts[i]=1
        if j in counts.keys():                                     ##
            counts[j]-=1
        else:
            counts[j]=-1
    for x in counts.values():
        if x!=0:
            return False
    return True
if __name__=='__main__':
    s1=input('enter the first string: ')
    s2=input('enter the secod string: ')
    print(anagrams(s1,s2))
```

PROBLEM -2

Problem 2: FizzBuzz Write a Python function that takes in an integer n and prints the numbersfrom 1 to n. For multiples of 3, print "Fizz" instead of the number. For multiples of 5, print "Buzz" instead of the number. For multiples of both 3 and 5, print "FizzBuzz" instead of the number.

In [27]:
```
def FizzBuzz(n):
    for i in range(1,n+1):
        if i%3==0 and i%5==0:
            print('FizzBuzz')

        elif i%3==0:
            print('Fizz')
        elif i%5==0:
            print('Buzz')
        else:
            print(i)
n=30
FizzBuzz(n)
```

```
1
2
Fizz
4
Buzz
Fizz
7
8
Fizz
Buzz
11
Fizz
13
14
FizzBuzz
16
17
Fizz
19
Buzz
Fizz
22
23
Fizz
Buzz
26
Fizz
28
29
FizzBuzz
```

Problem 3: Fibonacci Sequence Write a Python function that takes in an integer n and returns the nth number in the Fibonacci sequence. The Fibonacci sequence is a series of numbers in which each number after the first two is the sum of the two preceding ones.

In [31]:
```
def FibonnaciSequence(n):
    if n==0 or n==1:
        return n
    else:
        return FibonnaciSequence(n-1)+FibonnaciSequence(n-2)
if __name__=='__main__':
    n=int(input())
    FibonnaciSequence(n)
```

```
5
```

In [ ]: