#linear search

def linear(list,element):

flag=0

for i in range(0,len(list)):

if list[i]==element:

return i

flag+=1

break

if flag==0:

return 0

list=[2,5,34,179,41,59,57,10]

element=int(input())

result=linear(list,element)

if result==0:

print("element not found")

else:

print("element found at",result)

#binary search

def binary(list,element):

list.sort()

flag=0

low=0

high=len(list)-1

#printing the sorted list

print(list)

while high>=low:

mid=(low+high)//2

if list[mid]==element:

return mid

flag+=1

break

elif element>list[mid]:

low=mid+1

high=high

elif element<list[mid]:

low=0

high=mid-1

if flag==0:

return -1

element=int(input())

list=[3,8,66,143,9,8,10,56,2,9,0,1,12]

print(binary(list,element))

#bubble sort

def bubble(list):

for i in range(0,len(list)):

for j in range(0, len(list)-i-1):

if list[j]>list[j+1]:

temp=list[j]

list[j]=list[j+1]

list[j+1]=temp

return list

list=[3,6,1,7,8,99,14,5]

print("Sorted List : ", bubble(list))

#selection sort

def selection(list):

for j in range(0,len(list)):

min=j

for i in range(j+1,len(list)):

if list[i]<list[min]:

temp=list[min]

list[min]=list[i]

list[i]=temp

return list

list=[7,5,34,179,60,59,57,1]

print("Sorted list is : ", selection(list))

#insertion sort

def insertion(list):

for i in range(1,len(list)):

value=list[i]

j=i-1

while j>=0 and value<list[j]:

list[j+1]=list[j]

list[j]=value

j-=1

list[j+1]=value

return list

list=[7,1,5,12,46,33,2,6,3,8]

print("Sorted list : ",insertion(list))

# Quick sort in Python

def partition(array, low, high):

pivot = array[high]

i = low - 1

for j in range(low, high):

if array[j] <= pivot:

i = i + 1

(array[i], array[j]) = (array[j], array[i])

(array[i + 1], array[high]) = (array[high], array[i + 1])

return i + 1

def quickSort(array, low, high):

if low < high:

pi = partition(array, low, high)

quickSort(array, low, pi - 1)

quickSort(array, pi + 1, high)

data = [8, 7, 2, 1, 0, 9, 6]

print("Unsorted Array")

print(data)

size = len(data)

quickSort(data, 0, size - 1)

print('Sorted Array in Ascending Order:')

print(data)

#merge sort

def merge\_sort(list):

if len(list)>1:

mid=len(list)//2

left=list[:mid]

right=list[mid:]

merge\_sort(left)

merge\_sort(right)

i=0

j=0

k=0

while i<len(left) and j<len(right):

if left[i]<=right[j]:

list[k]=left[i]

i+=1

else:

list[k]=right[j]

j+=1

k+=1

while i<len(left):

list[k]=left[i]

i+=1

k+=1

while j<len(right):

list[k]=right[j]

j+=1

k+=1

list=[7,4,6,2,5,13,8,1,10,6]

merge\_sort(list)

print(list)

#Single Linked List

#menu

print("""choiceoose an option:

1:Insert at beginning

2:Insert at position

3:Insert at end

4:Delete at beginning

5:Delete at position

6:Delete at end

7:Searchoice

8:Count Nodes

9:Display

0:Exit

""")

#creation of node

class node :

def \_\_init\_\_(self,val):

self.val=val

self.next=None

#creation of single linked list

class linkedlist:

def \_\_init\_\_(self):

self.head=None

#inserting at the beginning of a list

def insert\_begin(self,data):

new\_node=node(data)

new\_node.next=self.head

self.head=new\_node

#inserting at the end of a list

def insert\_end(self,data):

new\_node = node(data)

if self.head is None:

self.head=new\_node

else:

pos=self.head

while pos.next!=None:

pos=pos.next

pos.next=new\_node

#inserting at position

def insert\_pos(self,pos,data):

temp=self.head

i=1

if pos<1:

print("Invalid Position")

else:

new\_node=node(data)

while i<pos-1 and temp.next!=None:

i+=1

temp=temp.next

if i<pos-1:

print("Invalid Position")

else:

new\_node.next = temp.next

temp.next = new\_node

#displaying the list

def display(self):

pos=self.head

while(pos):

print(pos.val, end="-->")

pos=pos.next

#deleting the first element

def delete\_begin(self):

if self.head==None:

print("Empty List")

else:

self.head=self.head.next

def delete\_end(self):

temp=self.head

while temp.next.next!=None:

temp=temp.next

temp.next=None

def delete\_pos(self,pos):

if self.head == None:

print("Empty list")

else:

temp = self.head

i = 1

while temp.next!=None and i<pos-1:

i+=1

temp = temp.next

if i==pos-1:

temp.next = temp.next.next

else:

print("Given position is out of range")

#counting nodes

def count(self):

i=1

temp=self.head

while temp.next!=None:

temp=temp.next

i=i+1

print("No of Nodes: ", i)

#searchoiceing

def searchoice(self,pivot):

temp=self.head

if self.head==None:

print("Empty List")

else:

while temp.next!=None:

if temp.val!=pivot:

temp=temp.next

else:

print("Found")

break

if temp.val==pivot:

print("Found")

llist = linkedlist()

choice=1

while choice!=0:

choice=int(input("choiceoose an option"))

if choice==1:

data=input()

llist.insert\_begin(data)

if choice==2:

data=input()

pos=int(input())

llist.insert\_begin(pos,data)

if choice==3:

data=input()

llist.insert\_end(data)

if choice==4:

llist.delete\_begin()

if choice==5:

pos=int(input())

llist.delete\_pos(pos)

if choice==6:

llist.delete\_end()

if choice==7:

llist.display()

if choice==8:

llist.count()

if choice==0:

break

#double linked lists

print("""choiceoose an option:

1:Insert at beginning

2:Insert at position

3:Insert at end

4:Delete at beginning

5:Delete at position

6:Delete at end

7:Searchoice

8:Count Nodes

9:Display

0:Exit

""")

class Node:

def \_\_init\_\_(self,data):

self.data=data

self.prev=None

self.next=None

class DoubleLinkedList:

def \_\_init\_\_(self):

self.head=None

def insert\_begin(self,data):

new\_node=Node(data)

if self.head==None:

self.head=new\_node

else:

new\_node.next=self.head

self.head.prev=new\_node

self.head=new\_node

def insert\_end(self,data):

new\_node=Node(data)

if self.head==None:

self.head=new\_node

else:

temp=self.head

while temp.next!=None:

temp=temp.next

temp.next=new\_node

new\_node.prev=temp

def insert\_pos(self,pos,data):

new\_node=Node(data)

temp=self.head

i=1

while i<pos-1 and temp.next!=None:

temp=temp.next

i+=1

if i<pos-1:

print("Invalid Position")

elif i==pos-1:

new\_node.next=temp.next

temp.next.prev=new\_node

temp.next=new\_node

new\_node.prev=temp

def delete\_begin(self):

if self.head==None:

print("Empty List")

else:

self.head=self.head.next

self.head.prev=None

def delete\_end(self):

if self.head==None:

print("Empty List")

elif self.head.next==None:

self.head=None

else:

temp=self.head

while temp.next!=None:

temp=temp.next

temp.prev.next=None

def delete\_pos(self,pos):

if self.head==None:

print("Empty List")

else:

temp=self.head

i=1

while temp.next!=None and i<pos-1:

i+=1

temp=temp.next

if i<pos-1:

print("Invalid Position")

elif i==pos-1:

temp.next=temp.next.next

temp.next.prev=temp

def searchoice(self,pivot):

temp=self.head

if self.head==None:

print("Empty List")

else:

while temp.next!=None:

if temp.data!=pivot:

temp=temp.next

else:

print("Found")

break

if temp.data==pivot:

print("Found")

def count(self):

i=1

temp=self.head

while temp.next!=None:

temp=temp.next

i=i+1

print("No of Nodes: ", i)

def display(self):

pos=self.head

while(pos):

print(pos.data, end="-->")

pos=pos.next

dll=DoubleLinkedList()

choice=1

while choice!=0:

choice=int(input("choiceoose an option"))

if choice==1:

data=input()

dll.insert\_begin(data)

if choice==2:

data=input()

pos=int(input())

dll.insert\_begin(pos,data)

if choice==3:

data=input()

dll.insert\_end(data)

if choice==4:

dll.delete\_begin()

if choice==5:

pos=int(input())

dll.delete\_pos(pos)

if choice==6:

dll.delete\_end()

if choice==7:

dll.display()

if choice==8:

dll.count()

if choice==0:

break

#Single Circular Linked Lists

print("""Menu:

1:Insert Begin

2:Insert End

3:Delete Begin

4:Delete End

5:Display

0:Exit

""")

class Node:

def \_\_init\_\_(self, data):

self.data = data

self.next = None

class csll:

def \_\_init\_\_(self):

self.head = None

def insertBegin(self, data):

newnode = Node(data)

if self.head is None:

self.head = newnode

newnode.next = self.head

else:

temp = self.head

while temp.next != self.head:

temp = temp.next

newnode.next = self.head

temp.next = newnode

self.head = newnode

def insertEnd(self, data):

newnode = Node(data)

if self.head is None:

self.head = newnode

newnode.next = self.head

else:

temp = self.head

while temp.next != self.head:

temp = temp.next

newnode.next = self.head

temp.next = newnode

def deleteEnd(self):

if self.head is None:

print('no elements present')

else:

temp = self.head

while temp.next.next != self.head:

temp = temp.next

print(temp.next.data, 'deleted successfully')

temp.next = self.head

def deleteBegin(self):

if self.head is None:

print('no elements present')

else:

temp = self.head

while temp.next != self.head:

temp = temp.next

print(self.head.data, 'deleted successfully')

temp.next = self.head.next

self.head = self.head.next

def display(self):

temp = self.head

while temp.next is not self.head:

print(temp.data, end="-->")

temp = temp.next

print(temp.data)

#driver code

csll = csll()

choice=1

while choice != 0:

choice=int(input())

if choice == 1:

ele=int(input('enter element to insert:'))

csll.insertBegin(ele)

elif choice == 2:

ele = int(input('enter element to insert:'))

csll.insertEnd(ele)

elif choice == 3:

csll.deleteBegin()

elif choice == 4:

csll.deleteEnd()

elif choice == 5:

csll.display()

#double circular linked list

print("""Choose an option:

1:Insert at Beginning

2:Insert at End

3:Delete at Beginning

4:Delete at End

5:Display

0:Exit""")

class Node:

def \_\_init\_\_(self,data):

self.data=data

self.next=None

self.prev=None

class CDLL:

def \_\_init\_\_(self):

self.head=None

def insert\_begin(self,data):

new\_node=Node(data)

if self.head is None:

self.head=new\_node

new\_node.next=self.head

new\_node.prev=self.head

else:

temp=self.head

while temp.next!=self.head:

temp=temp.next

temp.next=new\_node

new\_node.next=self.head

self.head=new\_node

new\_node.prev=temp

def insert\_end(self,data):

new\_node=Node(data)

if self.head is None:

self.head=new\_node

new\_node.next=self.head

new\_node.prev=self.head

else:

temp=self.head

while temp.next!=self.head:

temp=temp.next

temp.next=new\_node

new\_node.next=self.head

new\_node.prev=temp

self.head=new\_node

def delete\_begin(self):

if self.head is None:

print("No elements found")

else:

temp=self.head

while temp.next!=self.head:

temp=temp.next

temp.next=self.head.next

self.head=temp.next

self.head.prev=temp

def delete\_end(self):

if self.head is None:

print("No elements found")

else:

temp=self.head

while temp.next.next!=self.head:

temp=temp.next

temp.next=self.head

self.head.prev=temp

def display(self):

temp = self.head

while temp.next is not self.head:

print(temp.data, end="-->")

temp = temp.next

print(temp.data)

cdll=CDLL()

choice=1

while choice!=0:

choice=int(input())

if choice == 1:

ele=int(input('enter element to insert:'))

cdll.insert\_begin(ele)

elif choice == 2:

ele = int(input('enter element to insert:'))

cdll.insert\_end(ele)

elif choice == 3:

cdll.delete\_begin()

elif choice == 4:

cdll.delete\_end()

elif choice == 5:

cdll.display()

elif choice == 0:

break

#stack implementation

print("""Choose an option:

1:Insert at end

2:Delete at end

3:To check if stack is empty

4:To check if stack is full

5:To display

0:TO Exit""")

class Stack:

def \_\_init\_\_(self):

self.stack=[]

self.length=0

self.mai=10

self.top=-1

def push(self,data):

if self.length<self.mai:

self.stack.append(data)

self.length+=1

self.top+=1

elif self.length>=self.mai:

print("STACK OVERFLOW")

choice=0

def pop(self):

if self.length==0:

print("STACK UNDERFLOW")

else:

self.length-=1

self.top-=1

return self.stack.pop()

def isEmpty(self):

if self.length==0:

print("Stack is empty")

else:

print("Stack is not empty")

def isFull(self):

if self.length==self.mai:

print("Stack is Full")

else:

print("Stack is not Full")

def display(self):

for items in self.stack[::-1]:

print(items)

def peak(self):

return self.stack[-1]

def reverse(self,string):

stack=Stack()

n=len(string)

for i in range(0,n):

stack.push(string[i])

string=""

for i in range(0,n):

string+=stack.pop()

print(string)

def eval\_postfii(self,string):

stack1=Stack()

for i in range(0,len(string)):

if string[i].isdigit():

stack1.push((int(string[i])))

else:

val2 = stack1.pop()

val1 = stack1.pop()

if string[i]=='\*':

stack1.push(val1\*val2)

if string[i]=='+':

stack1.push(val1+val2)

if string[i]=='-':

stack1.push(val1-val2)

if string[i]=='/':

stack1.push(val1/val2)

print("Postfii Eipression :", string)

print("Answer :",stack1.pop())

def infix\_to\_postfix(self,data):

priority={"^":0,

"\*":1,

"/":1,

"+":2,

"-":2,

}

result=""

for i in data:

if i=="+" or i=="-" or i=="\*" or i=="/" or i=="^":

if len(stack.stack)>=1:

last=stack.stack[-1]

if last!="(" and priority[last]<=priority[i]:

result+=last

stack.pop()

stack.push(i)

else:

stack.push(i)

else:

stack.push(i)

elif i=="(":

stack.push(i)

elif i==")":

while len(stack.stack)>=1:

i=stack.stack[-1]

stack.pop()

if i=="(":

break

result+=i

else:

result+=i

while len(stack.stack)>=1:

i=stack.stack[-1]

stack.pop()

result+=i

print("Infix Expression :",data)

print("Postfix Expression :",result)

stack=Stack()

choice=1

while choice!=0:

choice=int(input())

if choice==1:

data=int(input())

stack.push(data)

elif choice==2:

stack.pop()

elif choice==3:

stack.isEmpty()

elif choice==4:

stack.isFull()

elif choice==5:

stack.display()

elif choice==7:

string=input("Enter postfix expression")

stack.eval\_postfii(string)

elif choice==8:

string=input("Enter infix expression")

stack.infix\_to\_postfix(string)

elif choice==0:

break

else:

print("Wrong Input")

#stack implementation

print("""Choose an option:

1:Insert at end

2:Delete at end

3:To check if stack is empty

4:To check if stack is full

5:To display

0:TO Exit""")

class Stack:

def \_\_init\_\_(self):

self.stack=[]

self.length=0

self.mai=10

self.top=-1

def push(self,data):

if self.length<self.mai:

self.stack.append(data)

self.length+=1

self.top+=1

elif self.length>=self.mai:

print("STACK OVERFLOW")

choice=0

def pop(self):

if self.length==0:

print("STACK UNDERFLOW")

else:

self.length-=1

self.top-=1

return self.stack.pop()

def isEmpty(self):

if self.length==0:

print("Stack is empty")

else:

print("Stack is not empty")

def isFull(self):

if self.length==self.mai:

print("Stack is Full")

else:

print("Stack is not Full")

def display(self):

for items in self.stack[::-1]:

print(items)

def peak(self):

return self.stack[-1]

def reverse(self,string):

stack=Stack()

n=len(string)

for i in range(0,n):

stack.push(string[i])

string=""

for i in range(0,n):

string+=stack.pop()

print(string)

def eval\_postfii(self,string):

stack1=Stack()

for i in range(0,len(string)):

if string[i].isdigit():

stack1.push((int(string[i])))

else:

val2 = stack1.pop()

val1 = stack1.pop()

if string[i]=='\*':

stack1.push(val1\*val2)

if string[i]=='+':

stack1.push(val1+val2)

if string[i]=='-':

stack1.push(val1-val2)

if string[i]=='/':

stack1.push(val1/val2)

print("Postfii Eipression :", string)

print("Answer :",stack1.pop())

def infix\_to\_postfix(self,data):

priority={"^":0,

"\*":1,

"/":1,

"+":2,

"-":2,

}

result=""

for i in data:

if i=="+" or i=="-" or i=="\*" or i=="/" or i=="^":

if len(stack.stack)>=1:

last=stack.stack[-1]

if last!="(" and priority[last]<=priority[i]:

result+=last

stack.pop()

stack.push(i)

else:

stack.push(i)

else:

stack.push(i)

elif i=="(":

stack.push(i)

elif i==")":

while len(stack.stack)>=1:

i=stack.stack[-1]

stack.pop()

if i=="(":

break

result+=i

else:

result+=i

while len(stack.stack)>=1:

i=stack.stack[-1]

stack.pop()

result+=i

print("Infix Expression :",data)

print("Postfix Expression :",result)

stack=Stack()

choice=1

while choice!=0:

choice=int(input())

if choice==1:

data=int(input())

stack.push(data)

elif choice==2:

stack.pop()

elif choice==3:

stack.isEmpty()

elif choice==4:

stack.isFull()

elif choice==5:

stack.display()

elif choice==7:

string=input("Enter postfix expression")

stack.eval\_postfii(string)

elif choice==8:

string=input("Enter infix expression")

stack.infix\_to\_postfix(string)

elif choice==0:

break

else:

print("Wrong Input")

#stack implementation

print("""Choose an option:

1:Insert at end

2:Delete at end

3:To check if stack is empty

4:To check if stack is full

5:To display

0:TO Exit""")

class Stack:

def \_\_init\_\_(self):

self.stack=[]

self.length=0

self.mai=10

self.top=-1

def push(self,data):

if self.length<self.mai:

self.stack.append(data)

self.length+=1

self.top+=1

elif self.length>=self.mai:

print("STACK OVERFLOW")

choice=0

def pop(self):

if self.length==0:

print("STACK UNDERFLOW")

else:

self.length-=1

self.top-=1

return self.stack.pop()

def isEmpty(self):

if self.length==0:

print("Stack is empty")

else:

print("Stack is not empty")

def isFull(self):

if self.length==self.mai:

print("Stack is Full")

else:

print("Stack is not Full")

def display(self):

for items in self.stack[::-1]:

print(items)

def peak(self):

return self.stack[-1]

def reverse(self,string):

stack=Stack()

n=len(string)

for i in range(0,n):

stack.push(string[i])

string=""

for i in range(0,n):

string+=stack.pop()

print(string)

def eval\_postfii(self,string):

stack1=Stack()

for i in range(0,len(string)):

if string[i].isdigit():

stack1.push((int(string[i])))

else:

val2 = stack1.pop()

val1 = stack1.pop()

if string[i]=='\*':

stack1.push(val1\*val2)

if string[i]=='+':

stack1.push(val1+val2)

if string[i]=='-':

stack1.push(val1-val2)

if string[i]=='/':

stack1.push(val1/val2)

print("Postfii Eipression :", string)

print("Answer :",stack1.pop())

def infix\_to\_postfix(self,data):

priority={"^":0,

"\*":1,

"/":1,

"+":2,

"-":2,

}

result=""

for i in data:

if i=="+" or i=="-" or i=="\*" or i=="/" or i=="^":

if len(stack.stack)>=1:

last=stack.stack[-1]

if last!="(" and priority[last]<=priority[i]:

result+=last

stack.pop()

stack.push(i)

else:

stack.push(i)

else:

stack.push(i)

elif i=="(":

stack.push(i)

elif i==")":

while len(stack.stack)>=1:

i=stack.stack[-1]

stack.pop()

if i=="(":

break

result+=i

else:

result+=i

while len(stack.stack)>=1:

i=stack.stack[-1]

stack.pop()

result+=i

print("Infix Expression :",data)

print("Postfix Expression :",result)

stack=Stack()

choice=1

while choice!=0:

choice=int(input())

if choice==1:

data=int(input())

stack.push(data)

elif choice==2:

stack.pop()

elif choice==3:

stack.isEmpty()

elif choice==4:

stack.isFull()

elif choice==5:

stack.display()

elif choice==7:

string=input("Enter postfix expression")

stack.eval\_postfii(string)

elif choice==8:

string=input("Enter infix expression")

stack.infix\_to\_postfix(string)

elif choice==0:

break

else:

print("Wrong Input")

#stack implementation

print("""Choose an option:

1:Insert at end

2:Delete at end

3:To check if stack is empty

4:To check if stack is full

5:To display

0:TO Exit""")

class Stack:

def \_\_init\_\_(self):

self.stack=[]

self.length=0

self.mai=10

self.top=-1

def push(self,data):

if self.length<self.mai:

self.stack.append(data)

self.length+=1

self.top+=1

elif self.length>=self.mai:

print("STACK OVERFLOW")

choice=0

def pop(self):

if self.length==0:

print("STACK UNDERFLOW")

else:

self.length-=1

self.top-=1

return self.stack.pop()

def isEmpty(self):

if self.length==0:

print("Stack is empty")

else:

print("Stack is not empty")

def isFull(self):

if self.length==self.mai:

print("Stack is Full")

else:

print("Stack is not Full")

def display(self):

for items in self.stack[::-1]:

print(items)

def peak(self):

return self.stack[-1]

def reverse(self,string):

stack=Stack()

n=len(string)

for i in range(0,n):

stack.push(string[i])

string=""

for i in range(0,n):

string+=stack.pop()

print(string)

def eval\_postfii(self,string):

stack1=Stack()

for i in range(0,len(string)):

if string[i].isdigit():

stack1.push((int(string[i])))

else:

val2 = stack1.pop()

val1 = stack1.pop()

if string[i]=='\*':

stack1.push(val1\*val2)

if string[i]=='+':

stack1.push(val1+val2)

if string[i]=='-':

stack1.push(val1-val2)

if string[i]=='/':

stack1.push(val1/val2)

print("Postfii Eipression :", string)

print("Answer :",stack1.pop())

def infix\_to\_postfix(self,data):

priority={"^":0,

"\*":1,

"/":1,

"+":2,

"-":2,

}

result=""

for i in data:

if i=="+" or i=="-" or i=="\*" or i=="/" or i=="^":

if len(stack.stack)>=1:

last=stack.stack[-1]

if last!="(" and priority[last]<=priority[i]:

result+=last

stack.pop()

stack.push(i)

else:

stack.push(i)

else:

stack.push(i)

elif i=="(":

stack.push(i)

elif i==")":

while len(stack.stack)>=1:

i=stack.stack[-1]

stack.pop()

if i=="(":

break

result+=i

else:

result+=i

while len(stack.stack)>=1:

i=stack.stack[-1]

stack.pop()

result+=i

print("Infix Expression :",data)

print("Postfix Expression :",result)

stack=Stack()

choice=1

while choice!=0:

choice=int(input())

if choice==1:

data=int(input())

stack.push(data)

elif choice==2:

stack.pop()

elif choice==3:

stack.isEmpty()

elif choice==4:

stack.isFull()

elif choice==5:

stack.display()

elif choice==7:

string=input("Enter postfix expression")

stack.eval\_postfii(string)

elif choice==8:

string=input("Enter infix expression")

stack.infix\_to\_postfix(string)

elif choice==0:

break

else:

print("Wrong Input")

#stack implementation

print("""Choose an option:

1:Insert at end

2:Delete at end

3:To check if stack is empty

4:To check if stack is full

5:To display

0:TO Exit""")

class Stack:

def \_\_init\_\_(self):

self.stack=[]

self.length=0

self.mai=10

self.top=-1

def push(self,data):

if self.length<self.mai:

self.stack.append(data)

self.length+=1

self.top+=1

elif self.length>=self.mai:

print("STACK OVERFLOW")

choice=0

def pop(self):

if self.length==0:

print("STACK UNDERFLOW")

else:

self.length-=1

self.top-=1

return self.stack.pop()

def isEmpty(self):

if self.length==0:

print("Stack is empty")

else:

print("Stack is not empty")

def isFull(self):

if self.length==self.mai:

print("Stack is Full")

else:

print("Stack is not Full")

def display(self):

for items in self.stack[::-1]:

print(items)

def peak(self):

return self.stack[-1]

def reverse(self,string):

stack=Stack()

n=len(string)

for i in range(0,n):

stack.push(string[i])

string=""

for i in range(0,n):

string+=stack.pop()

print(string)

def eval\_postfii(self,string):

stack1=Stack()

for i in range(0,len(string)):

if string[i].isdigit():

stack1.push((int(string[i])))

else:

val2 = stack1.pop()

val1 = stack1.pop()

if string[i]=='\*':

stack1.push(val1\*val2)

if string[i]=='+':

stack1.push(val1+val2)

if string[i]=='-':

stack1.push(val1-val2)

if string[i]=='/':

stack1.push(val1/val2)

print("Postfii Eipression :", string)

print("Answer :",stack1.pop())

def infix\_to\_postfix(self,data):

priority={"^":0,

"\*":1,

"/":1,

"+":2,

"-":2,

}

result=""

for i in data:

if i=="+" or i=="-" or i=="\*" or i=="/" or i=="^":

if len(stack.stack)>=1:

last=stack.stack[-1]

if last!="(" and priority[last]<=priority[i]:

result+=last

stack.pop()

stack.push(i)

else:

stack.push(i)

else:

stack.push(i)

elif i=="(":

stack.push(i)

elif i==")":

while len(stack.stack)>=1:

i=stack.stack[-1]

stack.pop()

if i=="(":

break

result+=i

else:

result+=i

while len(stack.stack)>=1:

i=stack.stack[-1]

stack.pop()

result+=i

print("Infix Expression :",data)

print("Postfix Expression :",result)

stack=Stack()

choice=1

while choice!=0:

choice=int(input())

if choice==1:

data=int(input())

stack.push(data)

elif choice==2:

stack.pop()

elif choice==3:

stack.isEmpty()

elif choice==4:

stack.isFull()

elif choice==5:

stack.display()

elif choice==7:

string=input("Enter postfix expression")

stack.eval\_postfii(string)

elif choice==8:

string=input("Enter infix expression")

stack.infix\_to\_postfix(string)

elif choice==0:

break

else:

print("Wrong Input")

class Queue:

def \_\_init\_\_(self,):

self.max=5

self.queue=[]

self.front=0

self.rear= -1

def enqueue(self,data):

if self.rear<self.max:

self.rear+=1

self.queue[self.rear]=data

else:

print("Queue is full")

def dequeue(self):

if self.front<=self.rear:

self.queue[self.front]=None

else:

print("Queue is empty")

def display(self):

for i in range(self.front,self.rear+1):

print(self.queue[i])

queue1=Queue()

queue1.enqueue(3)

queue1.enqueue(5)

queue1.enqueue(6)

queue1.dequeue()

queue1.enqueue(0)

queue1.display()

#binary search tree and searching

class Node:

def \_\_init\_\_(self,data):

self.data=data

self.left=None

self.right=None

class binarysearchtree:

def \_\_init\_\_(self):

self.root=None

def add(self,data):

tree\_node=Node(data)

if self.root==None:

self.root=tree\_node

else:

if data>root.data:

if root.right:

self.add(root.right)

else:

root.right=tree\_node

#traverse right

else:

if root.left:

self.add(root.left)

else:

root.left=tree\_node

#traverse left

self.add(self.root)

def search(self,pivot,root):

if root==None:

return "not found"

if root.data==pivot:

print(root.data)

return "found"

if pivot<root.data:

return self.search(pivot,root.left)

elif pivot>root.data:

return self.search(pivot,root.right)

tree=binarysearchtree()

tree.add(3)

tree.add(8)

tree.add(6)

tree.add(7)

tree.add(14)

tree.add(9)

print(tree.search(14,tree.root))

#dfs

graph = {

'5' : ['3','7'],

'3' : ['2', '4'],

'7' : ['8'],

'2' : [],

'4' : ['8'],

'8' : []

}

visited = set()

def dfs(visited, graph, vertex):

if vertex not in visited:

print (vertex)

visited.add(vertex)

for neighbour in graph[vertex]:

dfs(visited, graph, neighbour)

# Driver Code

print("Following is the Depth-First Search")

dfs(visited, graph, '5')

#breadth first search

graph = {

'5' : ['3','7'],

'3' : ['2', '4'],

'7' : ['8'],

'2' : [],

'4' : ['8'],

'8' : []

}

visited = []

queue = []

def bfs(visited, graph, node): #

visited.append(node)

queue.append(node)

while queue:

m = queue.pop(0)

print (m, end = " ")

for neighbour in graph[m]:

if neighbour not in visited:

visited.append(neighbour)

queue.append(neighbour)

# Driver Code

print("Following is the Breadth-First Search")

bfs(visited, graph, '5') # function calling

# Dijkstra's Algorithm in Python

import sys

vertices = [[0, 0, 1, 1, 0, 0, 0],

[0, 0, 1, 0, 0, 1, 0],

[1, 1, 0, 1, 1, 0, 0],

[1, 0, 1, 0, 0, 0, 1],

[0, 0, 1, 0, 0, 1, 0],

[0, 1, 0, 0, 1, 0, 1],

[0, 0, 0, 1, 0, 1, 0]]

edges = [[0, 0, 1, 2, 0, 0, 0],

[0, 0, 2, 0, 0, 3, 0],

[1, 2, 0, 1, 3, 0, 0],

[2, 0, 1, 0, 0, 0, 1],

[0, 0, 3, 0, 0, 2, 0],

[0, 3, 0, 0, 2, 0, 1],

[0, 0, 0, 1, 0, 1, 0]]

def to\_be\_visited():

global visited\_and\_distance

v = -10

for index in range(num\_of\_vertices):

if visited\_and\_distance[index][0] == 0 and (v < 0 or visited\_and\_distance[index][1] <= visited\_and\_distance[v][1]):

v = index

return v

num\_of\_vertices = len(vertices[0])

visited\_and\_distance = [[0, 0]]

for i in range(num\_of\_vertices-1):

visited\_and\_distance.append([0, sys.maxsize])

for vertex in range(num\_of\_vertices):

to\_visit = to\_be\_visited()

for neighbor\_index in range(num\_of\_vertices):

if vertices[to\_visit][neighbor\_index] == 1 and visited\_and\_distance[neighbor\_index][0] == 0:

new\_distance = visited\_and\_distance[to\_visit][1] \

+ edges[to\_visit][neighbor\_index]

if visited\_and\_distance[neighbor\_index][1] > new\_distance:

visited\_and\_distance[neighbor\_index][1] = new\_distance

visited\_and\_distance[to\_visit][0] = 1

i = 0

for distance in visited\_and\_distance:

print("Distance of ", chr(ord('a') + i),

" from source vertex: ", distance[1])

#heap sort

def heapify(arr, n, i):

largest = i

l = 2 \* i + 1

r = 2 \* i + 2

if l < n and arr[i] < arr[l]:

largest = l

if r < n and arr[largest] < arr[r]:

largest = r

if largest != i:

(arr[i], arr[largest]) = (arr[largest], arr[i])

heapify(arr, n, largest)

def heapSort(arr):

n = len(arr)

for i in range(n // 2 - 1, -1, -1):

heapify(arr, n, i)

for i in range(n - 1, 0, -1):

(arr[i], arr[0]) = (arr[0], arr[i]) # swap

heapify(arr, i, 0)

arr = [12, 11, 13, 5, 6, 7, ]

heapSort(arr)

n = len(arr)

print('Sorted array is')

for i in range(n):

print(arr[i])