EX NO: DATE:	**1.ADTS AS PYTHON CLASSES**

Aim:

To write a program to implement sum of all items, smallest number, reverse a string, basic queue operation in ADTS as python classes

Algorithm:

1a) sum of all items

1. Define a function and store the list of values
2. Initialize and set sum =0
3. Iterate each values in a list
4. Add sum and each value in the list
5. Return the sum

1b) smallest number in list

1. Define a function and store the list of values
2. Initialize and set minvalue
3. For each value in a list, check whether the value less than minvalue
4. If yes, then minvalue = value
5. Else, return minvalue

1c) Reverse a string

1. Initialize stack and check whether stack is empty
2. Find the length of the string
3. If empty, then push the string of each character into the stack
4. Then pop each character from stack
5. Finally,Store each character in another string

1d) queue

1. Initialize queue and check whether stack is empty
2. If empty, then enqueue the value into the queue
3. Then deque from stack
4. Find the length of the string
5. Display the results

Program:

1a)

```
def  sum_list(items):
    sum_numbers = 0
    for x in items:
        sum_numbers += x
    return sum_numbers
print('sum is',sum_list([1,2,-8]))
```

1b)

```
def smallest_num_in_list( list ):
    min = list[ 0 ]
    for a in list:
        if a < min:
            min = a
    return min
print(“smallest number”, smallest_num_in_list([1, 2, -8, 0]))
```

1c)

```
def createStack():
    stack=[]
    return stack
def size(stack):
    return len(stack)
def isEmpty(stack):
    if size(stack) == 0:
        return true
def push(stack,item):
    stack.append(item)
def pop(stack):
    if isEmpty(stack):
        return
    return stack.pop()
def reverse(string):
    n = len(string)
    stack = createStack()
    for i in range(0,n,1):
```

```
        push(stack,string[i])
    string=""
    for i in range(0,n,1):
        string+=pop(stack)
    return string
string="data structures"
string = reverse(string)
print("Reversed string is " + string)
```

1d) queue

```
class Queue:
    def_init_(self):
       self.items = []
    def isEmpty(self):
        return self.items == []
    def enqueue(self, item):
       self.items.insert(0,item)
    def dequeue(self):
        return self.items.pop()
    def size(self):
        return len(self.items)
q=Queue()
q.enqueue(4)
q.enqueue('dog')
q.enqueue(True)
print(q.size())
print(q.isEmpty())
print(q.dequeue())
print(q.size())
```

Output:

```
sum is -7
```

Output:

```
smallest number -8
```

Output:
```
Reversed string is serutcurts atad
```

Output:
```
3
False
4
2
```

Result
Thus the program is successfully executed

EX NO:- DATE:-	**2.RECURSIVE ALGORITHM**

Aim:

Write a program to perform tower of hanoi , factorial of a number, fibbonaci series, pascal triangle using recursive algorithm

2a.Tower of Hanoi:

Algorithm:

1. Create a **tower_of_hanoi** recursive function and pass two arguments: the number of disks n and the name of the rods such as **source, auxiliary,** and **target**.
2. define the base case when the number of disks is 1. In this case, simply move the one disk from the **source** to **target** and return.
3. Now, move remaining n-1 disks from **source** to **auxiliary** using the target as the **auxiliary**.
4. Then, the remaining 1 disk move on the **source** to **target**.
5. Move the n-1 disks on the auxiliary to the target using the source as the auxiliary.

Program:

```
def tower_of_hanoi(disks, source, auxiliary, target):
    if(disks == 1):
        print('Move disk 1 from rod {} to rod {}.'.format(source, target))
        return
    tower_of_hanoi(disks - 1, source, target, auxiliary)
    print('Move disk {} from rod {} to rod {}.'.format(disks, source, target))
    tower_of_hanoi(disks - 1, auxiliary, source, target)
disks = int(input('Enter the number of disks: '))
tower_of_hanoi(disks, 'A', 'B', 'C')
```

output

```
Enter the number of disks: 3
Move disk 1 from rod A to rod C.
Move disk 2 from rod A to rod B.
Move disk 1 from rod C to rod B.
Move disk 3 from rod A to rod C.
Move disk 1 from rod B to rod A.
Move disk 2 from rod B to rod C.
Move disk 1 from rod A to rod C.
```

2b. Factorial of a number:

Algorithm:

1. Create a factorial recursive function and give the number as argument
2. Declare a variable with value(number)
3. If the number is 1, return number
4. Else , return (number* recur_factorial(n-1))

Program:

```
def recur_factorial(n):
   if n == 1:
       return n
   else:
       return n*recur_factorial(n-1)
num = 5
if num < 0:
   print(" factorial does not exist for negative numbers")
elif num == 0:
   print("The factorial of 0 is 1")
else:
   print("The factorial of", num, "is", recur_factorial(num))
```

output:

```
The factorial of 5 is 120
```

2c. Fibbonaci series:

Algorithm:

1. Create a fibbonaci recursive function and give the number as argument
2. Declare a variable with value(number)
3. Check whether the number is less than and equal to zero, then print enter a positive number
4. Else, set a loop for the given number and call the recursive function
5. Check whether the number is less than and equal to 1, return number
6. Else, return (recur_fib(n-1)+recur_fib(n-2))

Program:

```
def recur_fibo(n):
  if n <= 1:
      return n
  else:
      return(recur_fibo(n-1) + recur_fibo(n-2))
nterms = 5
if nterms <= 0:
  print(" enter a positive integer")
else:
  print("Fibonacci sequence:")
  for i in range(nterms):
      print(recur_fibo(i))
```

output

```
Fibonacci sequence:
0
1
1
2
3
```

2d. Pascal triangle

Algorithm:

1. Create two function: pascal() and pascal triangle(), give the number as argument
2. Declare variables r,c, num=1
3. Take input (num) for number of rows and iterate *loop1* for 'num' times
4. Take input (rows) for number of columns and iterate *loop2 inside loop1* for '(row+1)' times
5. Then, call the pascal() and give (c,r) as arguments
6. Check whether the column is zero or equal to row, if it satisfies then return 1
7. Else, return pascal(col-1,row-1) + pascal(col,row-1)
8. Print new line under *loop1*
9. End

Program:

```
import sys
def pascal(col,row):
    if(col == 0) or (col == row):
        return 1
    else:
        return pascal(col-1,row-1) + pascal(col,row-1)
def PascalTriangle(num):
    if (num <= 0):
        print('Number must be greater than zero')
    for r in range(num):
        for c in range(r+1):
            sys.stdout.write(str(pascal(c,r))+' ')
        sys.stdout.write('\n')
PascalTriangle(4)
```

Output

```
1
1 1
1 2 1
1 3 3 1
```

Result:

Thus the program is successfully executed

EX NO:- DATE:-	**3.LIST ADT USING PYTHON ARRAYS**

Aim:

To write a program to implement list ADT using array

Algorithm:

1. Initialize class variables
2. Define the insert function and pass the class variables as arguments
3. Display the list of values
4. Define the search function and it iterate all values present in the list
5. Check whether the element is present, if it is found then print element is found
6. Else, print element is not found
7. Define the delete function and find the element which is to be deleted
8. Delete the element from the list
9. Define the update function and set the index position then update the element
10. Display the elements in list

Program:

```
class Student:

 def_init_(self, name, rollno, m1, m2):

 self.name = name

  self.rollno = rollno

  self.m1 = m1

  self.m2 = m2

 def accept(self, Name, Rollno, marks1, marks2 ):

  ob = Student(Name, Rollno, marks1, marks2 )

  ls.append(ob)

 def display(self, ob):

   print("Name : ", ob.name)

   print("RollNo : ", ob.rollno)

   print("Marks1 : ", ob.m1)

   print("Marks2 : ", ob.m2)

   print("\n")

 def search(self, rn):
```

```
        for i in range(ls. len ()):
            if(ls[i].rollno == rn):
                return i
    def delete(self, rn):
        i = obj.search(rn)
        del ls[i]
    def update(self, rn, No):
        i = obj.search(rn)
        roll = No
        ls[i].rollno = roll;
ls =[]
obj = Student('', 0, 0, 0)
print("\nOperations used, ")
print("\n1.Accept Student details\n2.Display Student Details\n"
      "3.Search Details of a Student\n4.Delete Details of Student"
      "\n5.Update Student Details\n6.Exit")
# ch = int(input("Enter choice:"))
# if(ch == 1):
obj.accept("A", 1, 100, 100)
obj.accept("B", 2, 90, 90)
obj.accept("C", 3, 80, 80)
# elif(ch == 2):
print("\n")
print("\nList of Students\n")
for i in range(ls. len ()):
    obj.display(ls[i])
# elif(ch == 3):
print("\n Student Found, ")
```

```
s = obj.search(2)

obj.display(ls[s])

# elif(ch == 4):

obj.delete(2)

print(ls._len_())

print("List after deletion")

for i in range(ls._len_()):

    obj.display(ls[i])

# elif(ch == 5):

obj.update(3, 2)

print(ls._len_())

print("List after updation")

for i in range(ls._len_()):

    obj.display(ls[i])

# else:

print("")
```

output:

```
Operations used,

1.Accept Student details
2.Display Student Details
3.Search Details of a Student
4.Delete Details of Student
5.Update Student Details
6.Exit



List of Students

Name   :  A
RollNo :  1
Marks1 :  100
Marks2 :  100
```

```
Name   :  B
RollNo :  2
Marks1 :  90
Marks2 :  90


Name   :  C
RollNo :  3
Marks1 :  80
Marks2 :  80



 Student Found,
Name   :  B
RollNo :  2
Marks1 :  90
Marks2 :  90


2
List after deletion
 Name   :  A
 RollNo :  1
 Marks1 :  100
 Marks2 :  100


 Name   :  C
 RollNo :  3
 Marks1 :  80
 Marks2 :  80


2
List after updation
Name   :  A
RollNo :  1
Marks1 :  100
Marks2 :  100


Name   :  C
RollNo :  2
Marks1 :  80
Marks2 :  8
```

RESULT

Thus the program is successfully executed

EX NO:- DATE:-	**4.LINKED LIST IMPLEMENTATION**

Aim:

To write a program for linked list implementation of single, double and circular linked list

Algorithm:

1. Defining the **Node class** which actually holds the data as well as the next element link
2. Defining the **Linked List class**
3. Initializing the Linked List constructor with head variable
4. Defining the **insert()** method which is used to add elements to the Circular Singly Linked List
    a. Checking whether or not the Linked List empty
    b. Adding a Node to the beginning of the Linked List
    c. Adding a Node to the end of the Linked List
    d. Adding a Node in the middle of the Linked List
5. Defining the **delete()** method which is used to delete elements from the Circular Singly Linked List
    a. Checking whether or not the Linked List is empty or not, or deleting the last element in the Linked List
    b. Deleting the first element of the Linked List
    c. Deleting the last element of the Linked List
    d. Deleting an element by position or by value
6. Defining the **display()** method which is used to present the Circular Singly Linked List in a user-comprehendible form

Program:

4a. singly linked list

```
class Node:

  def_init_(self, dataval=None):

    self.dataval = dataval

    self.nextval = None

class SLinkedList:

  def_init_(self):

    self.headval = None

  def AtBegining(self,newdata):

    NewNode = Node(newdata)

    NewNode.nextval = self.headval

    self.headval = NewNode
```

```
    def AtEnd(self, newdata):
        NewNode = Node(newdata)
        if self.headval is None:
            self.headval = NewNode
            return
        laste = self.headval
        while(laste.nextval):
            laste = laste.nextval
        laste.nextval=NewNode
def Inbetween(self,middle_node,newdata):
        if middle_node is None:
            print("The mentioned node is absent")
            return
        NewNode = Node(newdata)
        NewNode.nextval = middle_node.nextval
        middle_node.nextval = NewNode
    def search_item(self, x):
        if self.headval is None:
            print("List has no elements")
            return
        n = self.headval
        while n is not None:
            if n.dataval == x:
                print("Item found")
                return True
            n = n.nextval
        print("item not found")
        return False
```

```
def getCount(self):
    temp = self.headval # Initialise temp
    count = 0 # Initialise count
    while (temp):
        count += 1
        temp = temp.nextval
    return count
def RemoveNode(self, Removekey):
    HeadVal = self.headval
    if (HeadVal is not None):
        if (HeadVal.dataval == Removekey):
            self.headval = HeadVal.nextval
            HeadVal = None
            return
    while (HeadVal is not None):
        if HeadVal.dataval == Removekey:
            break
        prev = HeadVal
        HeadVal = HeadVal.nextval
    if (HeadVal == None):
        return
    prev.nextval = HeadVal.nextval
    HeadVal = None
def listprint(self):
    printval = self.headval
    while printval is not None:
        print (printval.dataval)
        printval = printval.nextval
```

```
list = SLinkedList()

list.headval = Node("1")

e2 = Node("2")

e3 = Node("3")

list.headval.nextval = e2

e2.nextval = e3

list.AtBegining("4")

list.AtEnd("5")

list.Inbetween(list.headval.nextval,"6")

list.search_item("3")

print ("Count of nodes is :",list.getCount())

list.RemoveNode("2")

list.listprint()
```

output:

```
Item found
Count of nodes is : 6
4
1
6
2
3
5

after removing
4
1
6
3
5
```

4b. Doubly linked list

```
class Node:

    def_init_(self, data):

        self.item = data

        self.nref = None
```

```
        self.pref = None
class DoublyLinkedList:
    def init (self):
        self.start_node = None
    def insert_in_emptylist(self, data):
        if self.start_node is None:
            new_node = Node(data)
            self.start_node = new_node
        else:
            print("list is not empty")
    def insert_at_start(self, data):
        if self.start_node is None:
            new_node = Node(data)
            self.start_node = new_node
            print("node inserted")
            return
        new_node = Node(data)
        new_node.nref = self.start_node
        self.start_node.pref = new_node
        self.start_node = new_node
    def insert_at_end(self, data):
        if self.start_node is None:
            new_node = Node(data)
            self.start_node = new_node
            return
        n =  self.start_node
        while n.nref is not None:
            n = n.nref
```

```
        new_node = Node(data)
        n.nref = new_node
        new_node.pref = n
    def insert_after_item(self, x, data):
        if self.start_node is None:
            print("List is empty")
            return
        else:
            n = self.start_node
            while n is not None:
                if n.item == x:
                    break
                n = n.nref
            if n is None:
                print("item not in the list")
            else:
                new_node = Node(data)
                new_node.pref = n
                new_node.nref = n.nref
                if n.nref is not None:
                    n.nref.prev = new_node
                n.nref = new_node
    def insert_before_item(self, x, data):
        if self.start_node is None:
            print("List is empty")
            return
        else:
            n = self.start_node
```

```
            while n is not None:
                if n.item == x:
                    break
                n = n.nref
            if n is None:
                print("item not in the list")
            else:
                new_node = Node(data)
                new_node.nref = n
                new_node.pref = n.pref
                if n.pref is not None:
                    n.pref.nref = new_node
                n.pref = new_node
    def traverse_list(self):
        if self.start_node is None:
            print("List has no element")
            return
        else:
            n = self.start_node
            while n is not None:
                print(n.item , " ")
                n = n.nref
    def delete_at_start(self):
        if self.start_node is None:
            print("The list has no element to delete")
            return
        if self.start_node.nref is None:
            self.start_node = None
```

```
            return
        self.start_node = self.start_node.nref
        self.start_prev = None;
    def delete_at_end(self):
        if self.start_node is None:
            print("The list has no element to delete")
            return
        if self.start_node.nref is None:
            self.start_node = None
            return
        n =  self.start_node
        while n.nref is not None:
            n = n.nref
        n.pref.nref = None
    def delete_element_by_value(self, x):
        if self.start_node is None:
            print("The list has no element to delete")
            return
        if self.start_node.nref is None:
            if self.start_node.item == x:
                self.start_node = None
            else:
                print("Item not found")
            return
        if self.start_node.item == x:
            self.start_node = self.start_node.nref
            self.start_node.pref = None
            return
```

```
        n = self.start_node
        while n.nref is not None:
            if n.item == x:
                break;
            n = n.nref
        if n.nref is not None:
            n.pref.nref = n.nref
            n.nref.pref = n.pref
        else:
            if n.item == x:
                n.pref.nref = None
            else:
                print("Element not found")
new_linked_list = DoublyLinkedList()
new_linked_list.insert_in_emptylist(50)
new_linked_list.insert_at_start(10)
new_linked_list.insert_at_start(5)
print(new_linked_list.traverse_list())
```

output:

```
5
10
50
None
```

4c.Circular singly linked list

```
class Node:
    def_init_(self, data):
        self.data = data
        self.next = None
class CircularLinkedList:
    def_init_(self):
        self.head = None
    def get_node(self, index):
        if self.head is None:
            return None
        current = self.head
        for i in range(index):
            current = current.next
            if current == self.head:
                return None
        return current
    def get_prev_node(self, ref_node):
        if self.head is None:
            return None
        current = self.head
        while current.next != ref_node:
            current = current.next
        return current
    def insert_after(self, ref_node, new_node):
        new_node.next = ref_node.next
        ref_node.next = new_node
    def insert_before(self, ref_node, new_node):
```

```
        prev_node = self.get_prev_node(ref_node)
        self.insert_after(prev_node, new_node)
    def insert_at_end(self, new_node):
        if self.head is None:
            self.head = new_node
            new_node.next = new_node
        else:
            self.insert_before(self.head, new_node)
    def insert_at_beg(self, new_node):
        self.insert_at_end(new_node)
        self.head = new_node
    def remove(self, node):
        if self.head.next == self.head:
            self.head = None
        else:
            prev_node = self.get_prev_node(node)
            prev_node.next = node.next
            if self.head == node:
                self.head = node.next
    def display(self):
        if self.head is None:
            return
        current = self.head
        while True:
            print(current.data, end = ' ')
            current = current.next
            if current == self.head:
                break
```

```
a_cllist = CircularLinkedList()
print('Menu')
print('insert <data> after <index>')
print('insert <data> before <index>')
print('insert <data> at beg')
print('insert <data> at end')
print('remove <index>')
print('quit')

while True:
    print('The list: ', end = '')
    a_cllist.display()
    print()
    do = input('What would you like to do? ').split()
    operation = do[0].strip().lower()
    if operation == 'insert':
        data = int(do[1])
        position = do[3].strip().lower()
        new_node = Node(data)
        suboperation = do[2].strip().lower()
        if suboperation == 'at':
            if position == 'beg':
                a_cllist.insert_at_beg(new_node)
            elif position == 'end':
                a_cllist.insert_at_end(new_node)
        else:
            index = int(position)
            ref_node = a_cllist.get_node(index)
```

```
            if ref_node is None:
                print('No such index.')
                continue
            if suboperation == 'after':
                a_cllist.insert_after(ref_node, new_node)
            elif suboperation == 'before':
                a_cllist.insert_before(ref_node, new_node)

    elif operation == 'remove':
        index = int(do[1])
        node = a_cllist.get_node(index)
        if node is None:
            print('No such index.')
            continue
        a_cllist.remove(node)
    elif operation == 'quit':
        break
```

output:

```
Menu
insert <data> after <index>
insert <data> before <index>
insert <data> at beg
insert <data> at end
remove <index>
quit
The list:
What would you like to do? insert 5 at beg
The list: 5
What would you like to do? insert 4 at beg
The list: 4 5
What would you like to do? insert 9 at end
The list: 4 5 9
What would you like to do? insert 6 after 1
The list: 4 5 6 9
What would you like to do? insert 7 after 6
No such index.
```

```
The list: 4 5 6 9
What would you like to do? insert 8 before 2
The list: 4 5 8 6 9
What would you like to do? remove 4
The list: 4 5 8 6
What would you like to do? remove 7
No such index.
The list: 4 5 8 6
What would you like to do? remove 0
The list: 5 8 6
What would you like to do? remove 1
The list: 5 6
What would you like to do? quit
```

4d. Circular doubly linked list

```
class Node:
    def__init_(self, data):
        self.data = data
        self.next = None
        self.prev = None
class CircularDoublyLinkedList:
    def init (self):
        self.first = None
    def get_node(self, index):
        current = self.first
        for i in range(index):
            current = current.next
            if current == self.first:
                return None
        return current
def insert_after(self, ref_node, new_node):
        new_node.prev = ref_node
        new_node.next = ref_node.next
        new_node.next.prev = new_node
        ref_node.next = new_node
    def insert_before(self, ref_node, new_node):
        self.insert_after(ref_node.prev, new_node)
    def insert_at_end(self, new_node):
        if self.first is None:
            self.first = new_node
            new_node.next = new_node
            new_node.prev = new_node
```

```
        else:
            self.insert_after(self.first.prev, new_node)
    def insert_at_beg(self, new_node):
        self.insert_at_end(new_node)
        self.first = new_node
    def remove(self, node):
        if self.first.next == self.first:
            self.first = None
        else:
            node.prev.next = node.next
            node.next.prev = node.prev
            if self.first == node:
                self.first = node.next
    def display(self):
        if self.first is None:
            return
        current = self.first
        while True:
            print(current.data, end = ' ')
            current = current.next
            if current == self.first:
                break
a_cdllist = CircularDoublyLinkedList()
print('Menu')
print('insert <data> after <index>')
print('insert <data> before <index>')
print('insert <data> at beg')
print('insert <data> at end')
```

```
print('remove <index>')
print('quit')
while True:
    print('The list: ', end = '')
    a_cdllist.display()
    print()
    do = input('What would you like to do? ').split()
    operation = do[0].strip().lower()
    if operation == 'insert':
        data = int(do[1])
        position = do[3].strip().lower()
        new_node = Node(data)
        suboperation = do[2].strip().lower()
        if suboperation == 'at':
            if position == 'beg':
                a_cdllist.insert_at_beg(new_node)
            elif position == 'end':
                a_cdllist.insert_at_end(new_node)
        else:
            index = int(position)
            ref_node = a_cdllist.get_node(index)
            if ref_node is None:
                print('No such index.')
                continue
            if suboperation == 'after':
                a_cdllist.insert_after(ref_node, new_node)
            elif suboperation == 'before':
                a_cdllist.insert_before(ref_node, new_node)
```

```
    elif operation == 'remove':
        index = int(do[1])
        node = a_cdllist.get_node(index)
        if node is None:
            print('No such index.')
            continue
        a_cdllist.remove(node)
    elif operation == 'quit':
        break
```

output:

```
Menu
insert <data> after <index>
insert <data> before <index>
insert <data> at beg
insert <data> at end
remove <index>
quit
The list:
What would you like to do? insert 1 at beg
The list: 1
What would you like to do? insert    3 at end
The list: 1 3
What would you like to do? insert 2 before 1
The list: 1 2 3
What would you like to do? insert 4 after 2
The list: 1 2 3 4
What would you like to do? insert 0 at beg
The list: 0 1 2 3 4
What would you like to do? remove 2
The list: 0 1 3 4
What would you like to do? quit
```

Result:

thus the program is successfully executed

EX NO:-	
DATE:-	**5.STACK AND QUEUE IMPLEMENTATION**

Aim:

To write a program to implement stack and queue ADTs using linked list

5a.Algorithm

1. Create a node with value Null.
2. Then push an element new
3. Check if new is not null otherwise insertion cannot be done.
4. Then put Item into data part of new and assign top to its link
5. Top is updated to the new value. And element is inserted.
6. For popping, check if the item is not null, otherwise stack is empty i.e underflow condition.
7. The pointer is then made to point the next element andset_head link is assigned to the pointer's value.
8. Remove the element from the link list.
9. Peek value is stored or printed by returning node at which top is pointing
10. Stop

Program:

```
class stack():
    def___init_(self):
        self.elements=[]
    def push(self,data):
        self.elements.append(data)
    def pop(self):
        return self.elements.pop()
stack=stack()
print("\n The initial view of stack",stack._dict_)
stack.push(3)
stack.push('test')
print("\n The current view of the stack",stack._dict__)
print("\n The popped out element from the stack",stack.pop())
print("\n The current view of the stack",stack._dict__)
print("\n The popped out element from the stack",stack.pop())
print("\n The final view of the stack",stack._dict_)
```

Output:

The initial view of stack {'elements': []}

The current view of the stack {'elements': [3, 'test']}

The popped out element from the stack test

The current view of the stack {'elements': [3]}

The popped out element from the stack 3

The final view of the stack {'elements': []}

5b.Algorithm:

1. Create a **newNode** with the given value and set the node's pointer to **NULL**.
2. Check whether queue is **EMPTY.**
3. If it is **EMPTY**, set FRONT and REAR to newNode.
4. Else, set the pointer of REAR to newNode and make REAR as the newNode.
5. Check if the queue is **EMPTY.**
6. If it is **EMPTY**, then display **"EMPTY QUEUE"** and exit.
7. Else, create a temporary node and set it to FRONT.
8. Make the next node as FRONT and delete the temporary node.
9. Display the nodes in queue

Program:

```
class Queue:
   def__init_(self):
      self.items=[]
   def isempty(self):
      return self.items==[]
   def enqueue(self,items):
      self.items.insert(0,items)
   def  dequeue(self):
      return self.items.pop()
   def size(self):
      return len(self.items)
q=Queue()
q.enqueue(4)
q.enqueue('dog')
q.enqueue(True)
print("The size of the queue after inserting some elements",q.size())
```

Output:

The size of the queue after inserting some elements 3

Result:

Thus the program is executed successfully

EX NO:- DATE:-	**6.APPLICATIONS OF LIST, STACK AND QUEUE ADT's**

Aim:

To write a program to implement application of list, stack and queue

a) Matrix manipulation(list)

Algorithm:

1. Initialize N*N matrix
2. Enter the row and column of the first (a) matrix.
3. Enter the row and column of the second (b) matrix.
4. Enter the elements of the first (a) and second (b) matrix.
5. Print the elements of the first (a) and second(b) matrix in matrix form.
6. Set a loop up to row.
7. Set an inner loop up to the column.
8. Set another inner loop up to the column.
9. Add the first (a) and second (b) matrix and store the element in the third matrix (c)
10. subtract the first (a) and second (b) matrix and store the element in the third matrix (c)
11. Multiply the first (a) and second (b) matrix and store the element in the third matrix (c)
12. divide the first (a) and second (b) matrix and store the element in the third matrix (c)
13. Print the final matrix.

Program:

```
A=[]

n=int(input("Enter N for N x N matrix : "))

print("Enter the element ::>")

for i in range(n):

    row=[]

    for j in range(n):

        row.append(int(input()))

    A. append(row)

print(A)

print("Display Array In Matrix Form")
```

```
for i in range(n):
    for j in range(n):
        print(A[i][j], end=" ")
    print()
B=[]
n=int(input("Enter N for N x N matrix : "))
print("Enter the element ::>")
for i in range(n):
    row=[]
    for j in range(n):
        row.append(int(input()))
    B. append(row)
print(B)
print("Display Array In Matrix Form")
for i in range(n):
    for j in range(n):
        print(B[i][j], end=" ")
    print()
result = [[0,0,0], [0,0,0], [0,0,0]]
for i in range(n):
    for j in range(len(A[0])):
        result[i][j] = A[i][j] + B[i][j]
print("Resultant Matrix is ::>")
for r in result:
    print("Resultant Matrix is ::>",r)
for i in range(n):
    for j in range(len(A[0])):
        result[i][j] = A[i][j] - B[i][j]
```

```
print("Resultant Matrix is ::>")
for r in result:
    print("Resultant Matrix is ::>",r)
for i in range(n):
    for j in range(len(A[0])):
        result[i][j] = A[i][j] * B[i][j]
print("Resultant Matrix is ::>")
for r in result:
    print("Resultant Matrix is ::>",r)
for i in range(n):
        for j in range(n):
            result[i][j] = result[j][i]
for r in result:
    print("Resultant Matrix is ::>",r)
for i in range(n):
    for j in range(len(A[0])):
        result[i][j] = A[i][j] % B[i][j]
print("Resultant Matrix is ::>")
for r in result:
    print("Resultant Matrix is ::>",r)
```

output

```
Enter N for N x N matrix : 3
Enter the element ::>
1
2
3
4
5
6
7
8
9
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
Display Array In Matrix Form
```

```
1 2 3
4 5 6
7 8 9
Enter N for N x N matrix : 3
Enter the element ::>
1
2
3
4
5
6
7
8
9
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
Display Array In Matrix Form
1 2 3
4 5 6
7 8 9
Resultant Matrix is ::>
Resultant Matrix is ::> [2, 4, 6]
Resultant Matrix is ::> [8, 10, 12]
Resultant Matrix is ::> [14, 16, 18]
Resultant Matrix is ::>
Resultant Matrix is ::> [0, 0, 0]
Resultant Matrix is ::> [0, 0, 0]
Resultant Matrix is ::> [0, 0, 0]
Resultant Matrix is ::>
Resultant Matrix is ::> [1, 4, 9]
Resultant Matrix is ::> [16, 25, 36]
Resultant Matrix is ::> [49, 64, 81]
Resultant Matrix is ::> [1, 16, 49]
Resultant Matrix is ::> [16, 25, 64]
Resultant Matrix is ::> [49, 64, 81]
Resultant Matrix is ::>
Resultant Matrix is ::> [0, 0, 0]
Resultant Matrix is ::> [0, 0, 0]
Resultant Matrix is ::> [0, 0, 0]
```

b) Applications of stack

Infix to postfix

Algorithm:

1. Scan the infix expression from left to right.
2. If the scanned character is an operand, output it.
3. Else,
    a. If the precedence of the scanned operator is greater than the precedence of the operator in the stack(or the stack is empty or the stack contains a '('), push it.
    b. Else, Pop all the operators from the stack which are greater than or equal to in precedence than that of the scanned operator. After doing that Push the scanned operator to the stack.
4. If the scanned character is an '(', push it to the stack.

5. If the scanned character is an ‘)’, pop the stack and output it until a ‘(‘ is encountered, and discard both the parenthesis.
6. Repeat steps 2-6 until infix expression is scanned.
7. Print the output
8. Pop and output from the stack until it is not empty.

Program:

```
OPERATORS = set(['+', '-', '*', '/', '(', ')', '^'])

PRIORITY = {'+':1, '-':1, '*':2, '/':2, '^':3}

def infix_to_postfix(expression):

    stack = [] # initially stack empty

    output = '' # initially output empty

    for ch in expression:

        if ch not in OPERATORS:

            output+= ch

        elif ch=='(':

            stack.append('(')

        elif ch==')':

            while stack and stack[-1]!= '(':

                output+=stack.pop()

            stack.pop()

        else:

            while stack and stack[-1]!='(' and PRIORITY[ch]<=PRIORITY[stack[-1]]:

                output+=stack.pop()

            stack.append(ch)

    while stack:

        output+=stack.pop()

    return output

expression = input('Enter infix expression')

print('infix expression: ',expression)
```

```
print('postfix expression: ',infix_to_postfix(expression))
```

output:

```
Enter infix expressiona*(b+c)/d
infix expression: a*(b+c)/d
postfix expression: abc+*d/
```

c) queue using stack

Algorithm:

1. Initialize with two stacks
2. Add an item to the queue
3. Move all elements from the first stack to the second stack
4. push item into the first stack
5. Move all elements back to the first stack from the second stack
6. Remove an item from the queue
7. if the first stack is empty, return underflow
8. Else, return the top item from the first stack

Program:

```
from collections import deque

class Queue:

    def_init_(self):

        self.s1 = deque()

        self.s2 = deque()

   def enqueue(self, data):

        while len(self.s1):

            self.s2.append(self.s1.pop())

        self.s1.append(data)

                while len(self.s2):

            self.s1.append(self.s2.pop())

    def dequeue(self):

        if not self.s1:

            print("Underflow!!")
```

```
            exit(0)
        return self.s1.pop()
if___name_==  '__main__':
    keys = [1, 2, 3, 4, 5]
    q    =    Queue()
    for key in keys:
        q.enqueue(key)
    print(q.dequeue())
    print(q.dequeue())
    print(q.dequeue())
```

output:

```
1
2
3
4
5
```

Result:

Thus the program is executed successfully

EX NO :- DATE:-	**7.SORTING AND SEARCHING ALGORITHM**

Aim:

To write a program to implement sorting and searching algorithm

1. Bubble sort

Algorithm

1. Get the total number of items in the given list
2. Determine the number of outer passes (n - 1) to be done. Its length is list minus one
3. Perform inner passes (n - 1) times for outer pass 1. Get the first element value and compare it with the second value. If the second value is less than the first value, then swap the positions
4. Repeat step 3 passes until you reach the outer pass (n - 1). Get the next element in the list then repeat the process that was performed in step 3 until all the values have been placed in their correct ascending order.
5. Return the result when all passes have been done. Return the results of the sorted list

Program

```
def bubbleSort(arr):
  n = len(arr)
for i in range(n-1):
    for j in range(0, n-i-1):
    if arr[j] > arr[j+1] :
        arr[j], arr[j+1] = arr[j+1], arr[j]
  print(arr)
arr = [5,3,8,6,7,2]
bubbleSort(arr)
print ("Sorted array is:")
for i in range(len(arr)):
  print ("%d" %arr[i]),
```

output:

```
Sorted array is:
[2, 3, 5, 6, 7, 8]
```

2) selection sort

Algorithm:

1. Get the value of n which is the total size of the array
2. Partition the list into sorted and unsorted sections. The sorted section is initially empty while the unsorted section contains the entire list
3. Pick the minimum value from the unpartitioned section and placed it into the sorted section.
4. Repeat the process (n – 1) times until all of the elements in the list have been sorted.

Program:

```
import sys

A = [5,3,8,6,7,2]

for i in range(len(A)):

    min_idx = i

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

        if A[min_idx] > A[j]:

            min_idx = j

    A[i], A[min_idx] = A[min_idx], A[i]

print(A)

 print ("Sorted array")

 for i in range(len(A)):

    print("%d" %A[i]),
```

output:

```
Sorted array
[2, 3, 5, 6, 7, 8]
```

3) insertion sort

Algorithm

1. Spilt a list in two parts - sorted and unsorted.
2. Iterate from arr[1] to arr[n] over the given array.
3. Compare the current element to the next element.
4. If the current element is smaller than the next element, compare to the element before, Move to the greater elements one position up to make space for the swapped element.

Program:

```
def insertionSort(arr):
  for i in range(1, len(arr)):
        key = arr[i]
        j=i
        j = i-1
        while j >=0 and key < arr[j] :
            arr[j+1] = arr[j]
            j -= 1
            arr[j+1] = key
        print(arr)
arr = [12, 11, 13, 5, 6]
insertionSort(arr)
print ("Sorted array is:")
for i in range(len(arr)):
     print ("%d" %arr[i])
```

output:

```
Sorted array is:
5
6
11
12
13
[5, 6, 11, 12, 13]
```

4) **quick sort**

Algorithm:

1. Select the Pivot Element as first, last, random and median element
2. Rearrange the Array
    a. A pointer is fixed at the pivot element. The pivot element is compared with the elements beginning from the first index.
    b. If the element is greater than the pivot element, a second pointer is set for that element.
    c. Now, pivot is compared with other elements. If an element smaller than the pivot element is reached, the smaller element is swapped with the greater element found earlier.
    d. Again, the process is repeated to set the next greater element as the second pointer. And, swap it with another smaller element.
    e. The process goes on until the second last element is reached.
    f. Finally, the pivot element is swapped with the second pointer
3. Divide Subarrays
    a. Pivot elements are again chosen for the left and the right sub-parts separately. And, **step 2** is repeated.
    b. The subarrays are divided until each subarray is formed of a single element. At this point, the array is sorted.

Program:

```
def Quick(arr,low,high):
    if(low<high):
       m=partition(arr,low,high)
       Quick(arr,low,m-1)
       Quick(arr,m+1,high)
def partition(arr,low,high):
    pivot=arr[low]
    i=low+1
    j=high
    flag=False
    while(not flag):
        while(i<=j and arr[i]<=pivot):
            i=i+1
        while(i<=j and arr[j]>=pivot):
            j=j-1
        if(j<i):
          flag=True
        else:
          temp=arr[i]
          arr[i]=arr[j]
          arr[j]=temp
    temp=arr[low]
    arr[low]=arr[j]
    arr[j]=temp
    return j
print("\n program for quick sort")
print("\n How many elements are there in array?")
n=int(input())
array=[]
i=0
```

```
for i in range(n):
    print("\n Enter element in array")
    item=int(input())
    array.append(item)
print("original array is \n")
print(array)
Quick(array,0,n-1)
print("\n Sorted array is")
print(array)
```

Output:

```
program for quick sort

 How many elements are there in array?
3

 Enter element in array
100

 Enter element in array
1

 Enter element in array
34
original array is

[100, 1, 34]

 Sorted array is
[1, 34, 100]
```

5) **merge sort**

Algorithm:

1. Create a merge sort function and declare list of values in array
2. Calculate the length for the given array
3. If the length is less than and equal one the list is already sorted, return the value.
4. Else, divide the list recursively into two halves: L and R, until it can no more be divided.
5. Then, merge the smaller lists into new list in sorted order.

Program:

```
def mergeSort(arr):

    if len(arr) > 1:

       mid = len(arr)//2

        L = arr[:mid]

      R  =  arr[mid:]

       mergeSort(L)

       mergeSort(R)

       i = j = k = 0

while i < len(L) and j < len(R):

            if L[i] < R[j]:

                arr[k] = L[i]

                i += 1

            else:

                arr[k] = R[j]

                j += 1

            k += 1

       while i < len(L):

            arr[k] = L[i]i

            += 1

            k += 1

        while j < len(R):
```

```
            arr[k] = R[j]

            j += 1

            k += 1

def printList(arr):

    for i in range(len(arr)):

        print(arr[i], end=" ")

    print()

if__name__== '_main_':

    arr = [38, 27, 43, 3, 9, 82, 10]

    print("Given array is", end="\n")

    printList(arr)

    mergeSort(arr)

    print("Sorted array is: ", end="\n")

    printList(arr)
```

output:

```
Given array is
38 27 43 3 9 82 10
Sorted array  is:
3 9 10 27 38 43 82
```

6) **LINEAR SEARCH**

Algorithm:

1. Read the search element from the user
2. Compare the search element with the first element in the list.
3. If both are matched, then display "Given element is found!!!" and terminate the function
4. If both are not matched, then compare search element with the next element in the list.
5. Repeat steps 3 and 4 until search element is compared with last element in the list.
6. If last element in the list also doesn't match, then display "Element is not found!!!" and terminate the function.

Program:

```
def linear_search(obj, item, start=0):

    for i in range(start, len(obj)):

        if obj[i] == item:

            return i

    return -1

arr=[1,2,3,4,5,6,7,8]

x=4

result=linear_search(arr,x)

if result==-1:

    print ("element does not exist")

else:

    print ("element exist in position %d" %result)
```

Output:

```
element exist in position 3
```

7) BINARY SEARCH

Algorithm:

1. Given an input array that is supposed to be sorted in ascending order.
2. Take two variables which act as a pointer i.e, beg, and end.
3. Beg assigned with 0 and the end assigned to the last index of the array.
4. No, introduce another variable mid which is the middle of the current array. Then computed as (low+high)/2.
5. If the element present at the mid index is equal to the element to be searched, then just return the mid index.
6. If the element to be searched is smaller than the element present at the mid index, move end to mid-1, and all RHS get discarded.
7. If the element to be searched is greater than the element present at the mid index, move beg to mid+1, and all LHS get discarded.

Program:

```
def binary_search(arr, low, high, x):

    if high >= low:

        mid = (high + low) // 2

        if arr[mid] == x:

            return  mid

        elif arr[mid] > x:

            return binary_search(arr, low, mid - 1, x)

        else:

            return binary_search(arr, mid + 1, high, x)

    else:

        return -1

arr = [ 2, 3, 4, 10, 40 ]

x = 40

result = binary_search(arr, 0, len(arr)-1, x)

if result != -1:

    print("Element is present at index", str(result))

else:
```

```
    print("Element is not present in array")
```

output:

```
Element is present at index 4
```

Result

Thus the program is executed successfully

EX NO:- DATE:-	**8.IMPLEMENTATION OF HASH TABLES**

Aim:

To write a program to implement hash tables

1. # Program to implement Hashing with Linear Probing

Algorithm:

1. Initialize hash Table with all elements 0
2. Create a method that checks if the hash table is full or not
3. Create a method that returns position for a given element
4. Create a method that inserts element inside the hash table
    a. checking if the table is full
    b. checking if the position is empty
    c. collision occurred , do linear probing
5. Create a method that searches for an element in the table
6. Returns position of element if found
7. Else returns False
    a. If element is not found at position returned hash function
    b. Then first we search element from position+1 to end
    c. If not found then we search element from position-1 to 0
        a. Check if the element is stored between position+1 to size
        b. Now checking if the element is stored between position-1 to 0
8. Create a method to remove an element from the table
9. Create a method to display the hash table
    a. Displaying the Table

Program:

```
class  hashTable:

   def___init_(self):

      self.size = int(input("Enter the Size of the hash table : "))

      self.table = list(0 for i in range(self.size))

      self.elementCount = 0

      self.comparisons = 0

   def isFull(self):

      if self.elementCount == self.size:

         return True

      else:
```

```
            return False

    def hashFunction(self, element):

        return element % self.size

    def insert(self, element):

        # checking if the table is full

        if self.isFull():

            print("Hash Table Full")

            return False

        isStored = False

        position = self.hashFunction(element)

        if self.table[position] == 0:

            self.table[position] = element

            print("Element " + str(element) + " at position " + str(position))

            isStored = True

            self.elementCount += 1

       else:

            print("Collision has occured for element " + str(element) + " at position " + str(position) + "
finding new Position.")

            while self.table[position] != 0:

                position += 1

                if position >= self.size:

                    position = 0

            self.table[position] = element

            isStored = True

            self.elementCount += 1

        return isStored

    def search(self, element):

        found = False
```

```
    position = self.hashFunction(element)
    self.comparisons += 1
    if(self.table[position] == element):
        return position
        isFound = True
    else:
        temp = position - 1
        while position < self.size:
            if self.table[position] != element:
                position += 1
                self.comparisons += 1
            else:
                return position
        position = temp
        while position >= 0:
            if self.table[position] != element:
                position -= 1
                self.comparisons += 1
            else:
                return position
    if not found:
        print("Element not found")
        return False
def remove(self, element):
    position = self.search(element)
    if position is not False:
        self.table[position] = 0
```

```
            print("Element " + str(element) + " is Deleted")
            self.elementCount -= 1
        else:
            print("Element is not present in the Hash Table")
        return
    def display(self):
        print("\n")
        for i in range(self.size):
            print(str(i) + " = " + str(self.table[i]))
        print("The number of element is the Table are : " + str(self.elementCount))
table1 = hashTable()
table1.insert(89)
table1.insert(18)
table1.insert(49)
table1.insert(58)
table1.insert(9)
table1.display()
print()
print("The position of element 9 is : " + str(table1.search(9)))
print("The position of element 58 is : " + str(table1.search(58)))
print("\nTotal number of comaprisons done for searching = " + str(table1.comparisons))
print()
table1.remove(18)
table1.display()
```

output

```
Enter the Size of the hash table : 10
Element 89 at position 9
Element 18 at position 8
```

```
Collision has occured for element 49 at position 9 finding new Position
.
Collision has occured for element 58 at position 8 finding new Position
.
Collision has occured for element 9 at position 9 finding new Position.


0 = 49
1 = 58
2 = 9
3 = 0
4 = 0
5 = 0
6 = 0
7 = 0
8 = 18
9 = 89
The number of element is the Table are : 5

The position of element 9 is : 2
The position of element 58 is : 1

Total number of comaprisons done for searching = 17

Element 18 is Deleted


0 = 49
1 = 58
2 = 9
3 = 0
4 = 0
5 = 0
6 = 0
7 = 0
8 = 0
9 = 89
The number of element is the Table are : 4
```

2. # Program to implement Hashing with Quadratic Probing

Algorithm:

1. Initialize hash Table with all elements 0
2. Create a method that checks if the hash table is full or not
3. Create a method that returns position for a given element
4. Replace with hash function
5. Create a method to resolve collision by quadratic probing method
    a. Start a loop to find the position and calculate new position by quadratic probing
    b. If new position is empty then break out of loop and return new Position
    c. Else, as the position is not empty increase i
6. Create a method that inserts element inside the hash table
    a. Checking if the table is full
    b. Checking if the position is empty, if empty position found , store the element
    c. Collision occurred, do quadratic probing

7. Create a method that searches for an element in the table
8. Returns position of element if found
9. Else if element is not found at position returned hash function and then search element using quadratic probing
    a. Start a loop to find the position and calculate new position by quadratic probing
    b. If element at new position is equal to the required element
    c. Else, as the position is not empty increase i

10. Create a method to remove an element from the table
11. Create a method to display the hash table

Program:

```
class hashTable:

 def___init_(self):

        self.size = int(input("Enter the Size of the hash table : "))

        self.table = list(0 for i in range(self.size))

        self.elementCount = 0

        self.comparisons = 0

    def isFull(self):

        if self.elementCount == self.size:

            return True

        else:
```

```
        return False
    def hashFunction(self, element):
        return element % self.size
    def quadraticProbing(self, element, position):
        posFound = False
        limit = 50
        i = 1
        while i <= limit:
            newPosition = position + (i**2)
            newPosition = newPosition % self.size
            if self.table[newPosition] == 0:
                posFound = True
                break
            else:
                i += 1
        return posFound, newPosition
    def insert(self, element):
        if self.isFull():
            print("Hash Table Full")
            return False
        isStored = False
        position = self.hashFunction(element)
        if self.table[position] == 0:
                    self.table[position] = element
            print("Element " + str(element) + " at position " + str(position))
            isStored = True
            self.elementCount += 1
        else:
```

```
            print("Collision has occured for element " + str(element) + " at position " + str(position) + " finding new Position.")
            isStored, position = self.quadraticProbing(element, position)
            if isStored:
                self.table[position] = element
                self.elementCount += 1
        return isStored
    def search(self, element):
        found = False
        position = self.hashFunction(element)
        self.comparisons += 1
        if(self.table[position] == element):
            return position
        else:
            limit = 50
            i = 1
            newPosition = position
                    while i <= limit:
                newPosition = position + (i**2)
                newPosition = newPosition % self.size
                self.comparisons += 1
                if self.table[newPosition] == element:
                    found = True
                    break
                elif self.table[newPosition] == 0:
                    found = False
                    break
                else:
```

```
                i += 1
            if found:
                return newPosition
            else:
                print("Element not Found")
                return found
    def remove(self, element):
        position = self.search(element)
        if position is not False:
            self.table[position] = 0
            print("Element " + str(element) + " is Deleted")
            self.elementCount -= 1
        else:
            print("Element is not present in the Hash Table")
        return
    def display(self):
        print("\n")
        for i in range(self.size):
            print(str(i) + " = " + str(self.table[i]))
        print("The number of element is the Table are : " + str(self.elementCount))
table1 = hashTable()
table1.insert(89)
table1.insert(18)
table1.insert(49)
table1.insert(58)
table1.insert(9)
table1.display()
print()
```

```
print("The position of element 9 is : " + str(table1.search(9)))

print("The position of element 58 is : " + str(table1.search(58)))

print("\nTotal number of comaprisons done for searching = " + str(table1.comparisons))

print()

#table1.remove(90)

table1.remove(18)

table1.display()
```

output:

```
Enter the Size of the hash table : 10
Element 89 at position 9
Element 18 at position 8
Collision has occured for element 49 at position 9 finding new Position
.
Collision has occured for element 58 at position 8 finding new Position
.
Collision has occured for element 9 at position 9 finding new Position.


0 = 49
1 = 0
2 = 58
3 = 9
4 = 0
5 = 0
6 = 0
7 = 0
8 = 18
9 = 89
The number of element is the Table are : 5

The position of element 9 is : 3
The position of element 58 is : 2

Total number of comaprisons done for searching = 6

Element 18 is Deleted


0 = 49
1 = 0
2 = 58
3 = 9
4 = 0
5 = 0
6 = 0
7 = 0
```

```
8 = 0
9 = 89
The number of element is the Table are : 4
```

3. # Program to implement Double Hashing

Algorithm:

1. Initialize hash Table with all elements 0
2. Create a method that checks if the hash table is full or not
3. Create a method that returns position for a given element
4. Replace with hash function
5. Create another method that returns position for a given element
6. Create a method to resolve collision by double hashing method
    a. Start a loop to find the position and calculate new position by double hashing
    b. If new position is empty then break out of loop and return new Position
    c. Else, as the position is not empty increase i
7. Create a method that inserts element inside the hash table
    a. Checking if the table is full
    b. Checking if the position is empty, if empty position found , store the element
    c. Collision occurred, do double hashing
8. Create a method that searches for an element in the table
9. Returns position of element if found
10. Else if element is not found at position returned hash function and then search element using double hashing
    a. Start a loop to find the position and calculate new position by double hashing
    b. If element at new position is equal to the required element
    c. Else, as the position is not empty increase i
11. Create a method to remove an element from the table
12. Create a method to display the hash table

Program:

```
class doubleHashTable:

  def_init_(self):

    self.size = int(input("Enter the Size of the hash table : "))

    self.num = 7

    self.table = list(0 for i in range(self.size))

    self.elementCount = 0

    self.comparisons = 0

  def isFull(self):

    if self.elementCount == self.size:

      return True

    else:
```

```
        return False
    def h1(self, element):
        return element % self.size
    def h2(self, element):
        return (self.num-(element % self.num))
    def doubleHashing(self, element, position):
        posFound = False
        limit = 50
        i = 1
        while i <= limit:
            newPosition = (self.h1(element) + i*self.h2(element)) % self.size
                    if self.table[newPosition] == 0:
                posFound = True
                break
            else:
                i += 1
        return posFound, newPosition
    def insert(self, element):
        if self.isFull():
            print("Hash Table Full")
            return False
        posFound = False
        position = self.h1(element)
        if self.table[position] == 0:
            self.table[position] = element
            print("Element " + str(element) + " at position " + str(position))
            isStored = True
            self.elementCount += 1
```

```
        else:
            while not posFound:
                print("Collision has occured for element " + str(element) + " at position " + str(position) + "
finding new Position.")
                posFound, position = self.doubleHashing(element, position)
                if posFound:
                    self.table[position] = element
                    self.elementCount += 1
        return posFound
    def search(self, element):
        found = False
        position = self.h1(element)
        self.comparisons += 1

        if(self.table[position] == element):
            return position
        else:
            limit = 50
            i = 1
            newPosition = position
                 while i <= limit:
                     position = (self.h1(element) + i*self.h2(element)) % self.size
                self.comparisons += 1
                if self.table[position] == element
                    found = True
                    break
                elif self.table[position] == 0:
                    found = False
```

```
                break
            else:
                # as the position is not empty increase i
                i += 1
        if found:
            return position
        else:
            print("Element not Found")
            return found
    def remove(self, element):
        position = self.search(element)
        if position is not False:
            self.table[position] = 0
            print("Element " + str(element) + " is Deleted")
            self.elementCount -= 1
        else:
            print("Element is not present in the Hash Table")
        return
    def display(self):
        print("\n")
        for i in range(self.size):
            print(str(i) + " = " + str(self.table[i]))
        print("The number of element is the Table are : " + str(self.elementCount))
table1 = doubleHashTable()
table1.insert(89)
table1.insert(18)
table1.insert(49)
table1.insert(58)
```

table1.insert(9)

table1.display()

print()

print("The position of element 9 is : " + str(table1.search(9)))

print("The position of element 58 is : " + str(table1.search(58)))

print("\nTotal number of comaprisons done for searching = " + str(table1.comparisons))

print()

table1.remove(18)

table1.display()

output:

```
Enter the Size of the hash table : 10
Element 89 at position 9
Element 18 at position 8
Collision has occured for element 49 at position 9 finding new Position.
Collision has occured for element 58 at position 8 finding new Position.
Collision has occured for element 9 at position 9 finding new Position.


0 = 0
1 = 0
2 = 0
3 = 58
4 = 9
5 = 0
6 = 49
7 = 0
8 = 18
9 = 89
The number of element is the Table are : 5

The position of element 9 is : 4
The position of element 58 is : 3

Total number of comaprisons done for searching = 4

Element 18 is Deleted


0 = 0
```

```
1 = 0
2 = 0
3 = 58
4 = 9
5 = 0
6 = 49
7 = 0
8 = 0
9 = 89
The number of element is the Table are : 4
```

Result:

Thus the program executed successfully

EX NO:- DATE:-	**9.TREE REPRESENTATION AND TRAVERSAL ALGORITHM**

Aim:

To write a program for to implement tree representation and tree traversal algorithm

9a.Array representation

Algorithm:

1. Initialize the array with size numbering starting from 0 to n-1.
2. Create a root method to set a root value
3. if the array is not none, then print tree already had root
4. Else, set the element as root
5. Create a left method for to create left child
6. If the array is none, then print no parent node is found
7. Else, create a node as left child
8. Create a Right method for to create right child
9. If the array is none, then print no parent node is found
10. Else, create a node as right child
11. Display the nodes

Program:

```
tree = [None] * 20

def root(key):

    if tree[0] != None:

        print("Tree already had root")

    else:

        tree[0] = key

def set_left(key, parent):

    if tree[parent] == None:

        print("Can't set child at", (parent * 2) + 1, ", no parent found")

    else:

        tree[(parent * 2) + 1] = key

def set_right(key, parent):

    if tree[parent] == None:

        print("Can't set child at", (parent * 2) + 2, ", no parent found")

    else:
```

```
        tree[(parent * 2) + 2] = key
def print_tree():
    for i in range(20):
        if tree[i] != None:
            print(tree[i], end="")
        else:
            print("-", end="")
    print()
root('3')
set_left('5', 0)
set_right('9', 0)
set_left('6', 1)
set_right('8', 1)
set_left('20', 2)
set_right('10', 2)
set_left('9', 5)
print_tree()
```

output

```
Array representation
359682010-----9-------
```

9b.linkedlist representation

Algorithm:

1. Create a class and represent a node of binary tree
2. Assign data to the new node, set left and right children to None
3. Create a class and represent the root of binary tree
4. Create a new node and check whether tree is empty ,add root to the queue
5. If node has both left and right child, add both the child to queue
6. If node has no left child, make newNode as left child
7. If node has left child but no right child, make newNode as right child
8. Display the list of nodes in binary tree

Program:

```
class Node:
    def_init_(self,data):
        self.data  =  data;
        self.left  =  None;
        self.right = None;
class    BinaryTree:
    def_init_(self):
        self.root = None;
    def  insertNode(self,  data):
        newNode = Node(data);
        if(self.root == None):
            self.root = newNode;
            return;
        else:
            queue = [];
            queue.append(self.root);
            while(True):
                node = queue.pop(0);
                if(node.left != None and node.right != None):
```

```
                queue.append(node.left);
                queue.append(node.right);
            else:
                if(node.left == None):
                    node.left = newNode;
                    queue.append(node.left);
                else:
                    node.right = newNode;
                    queue.append(node.right);
                break;
    def inorderTraversal(self, node):
        #Check whether tree is empty
        if(self.root == None):
            print("Tree is empty");
            return;
        else:
            if(node.left != None):
                self.inorderTraversal(node.left);
            print(node.data),
            if(node.right!= None):
                self.inorderTraversal(node.right);
bt = BinaryTree();
bt.insertNode(1);
print("Binary tree after insertion");
bt.inorderTraversal(bt.root);
bt.insertNode(2);
bt.insertNode(3);
print("\nBinary tree after insertion");
```

bt.inorderTraversal(bt.root);

bt.insertNode(4);

bt.insertNode(5);

print("\nBinary tree after insertion");

bt.inorderTraversal(bt.root);

bt.insertNode(6);

bt.insertNode(7);

print("\nBinary tree after insertion");

bt.inorderTraversal(bt.root);

output:

```
Binary tree after insertion
1

Binary tree after insertion
2
1
3

Binary tree after insertion
4
2
5
1
3

Binary tree after insertion
4
2
5
1
6
3
7
```

9c. traversal algorithm

Algorithm:

1. Create a class and represent a node
2. Assign data to the new node, set left and right children to None
3. If tree is non empty, create a new node and compare to current node
4. If new node less than current node, then make it as left child of current node
5. If new node greater than current node, then make it as right child of current node
6. Else, if tree is empty ,add node to the root node
7. Create inorder method, then recursively traverse left,root, right order
8. Create preorder method, then recursively traverse root, left,right order
9. Create postorder method, then recursively traverse left,right,root order
10. Display all nodes

Program:

```
class Node:

    def_init_(self, data):

        self.left = None

        self.right = None

        self.data = data

    def insert(self, data):

        if self.data:

            if data < self.data:

                if self.left is None:

                    self.left = Node(data)

                else:

                    self.left.insert(data)

            elif data > self.data:

                if self.right is None:

                    self.right = Node(data)

                else:

                    self.right.insert(data)

        else:
```

```
        self.data = data
  def PrintTree(self):
    if self.left:
        self.left.PrintTree()
    print( self.data),
    if self.right:
        self.right.PrintTree()
  def inorderTraversal(self, root):
    res = []
    if root:
        res = self.inorderTraversal(root.left)
        res.append(root.data)
        res = res + self.inorderTraversal(root.right)
    return res
  def PreorderTraversal(self, root):
    res = []
    if root:
        res.append(root.data)
        res = res + self.PreorderTraversal(root.left)
        res = res + self.PreorderTraversal(root.right)
    return res
  def PostorderTraversal(self, root):
    res = []
    if root:
        res = self.PostorderTraversal(root.left)
        res = res + self.PostorderTraversal(root.right)
        res.append(root.data)
    return res
```

```
root = Node(27)

root.insert(14)

root.insert(35)

root.insert(10)

root.insert(19)

root.insert(31)

root.insert(42)

print(root.inorderTraversal(root))

print(root.PreorderTraversal(root))

print(root.PostorderTraversal(root))
```

output

```
Inorder traversal
[10, 14, 19, 27, 31,   35, 42]
preorder traversal
[27, 14, 10, 19, 35,  31, 42]
postorder traversal
[10, 19, 14, 31, 42,   35, 27]
```

Result

Thus the program is executed successfully

EX NO:- DATE:-	**10.IMPLEMENTATION OF BINARY SEARCH TREE**

Aim:

To write a program to implement binary search tree

10 a. binary search tree

Algorithm:

1. Insert node into the tree as the root of the tree.
2. Read the next element, if it is lesser than the root node element, insert it as the root of the left sub-tree.
3. Otherwise, insert it as the root of the right of the right sub-tree
4. Compare the element with the root of the tree.
5. If the item is matched then return the location of the node.
6. Otherwise check if item is less than the element present on root, if so then move to the left sub-tree.
7. If not, then move to the right sub-tree.
8. Repeat this procedure recursively until match found.
9. If element is not found then return NULL.
10. Find the data of the node to be deleted.
11. If the node is a leaf node, delete the node directly.
12. Else if the node has one child, copy the child to the node to be deleted and delete the child node.
13. Else if the node has two children, find the inorder successor of the node.
14. Copy the contents of the inorder successor to the node to be deleted and delete the inorder successor.

Program:

```
class 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(str(root.key) + "->", end=' ')
```

```
        inorder(root.right)
def insert(node, key):
    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 minValueNode(node):
    current = node
    while(current.left is not None):
        current = current.left
    return current
def deleteNode(root, key):
    if root is None:
        return root
    if key < root.key:
        root.left = deleteNode(root.left, key)
    elif(key > root.key):
        root.right = deleteNode(root.right, key)
    else:
        if root.left is None:
            temp = root.right
            root = None
            return temp

        elif root.right is None:
```

```
            temp = root.left
            root = None
            return temp
        temp = minValueNode(root.right)
        root.key = temp.key
        root.right = deleteNode(root.right, temp.key)
    return root
root = None
root = insert(root, 8)
root = insert(root, 3)
root = insert(root, 1)
root = insert(root, 6)
root = insert(root, 7)
root = insert(root, 10)
root = insert(root, 14)
root = insert(root, 4)
print("Inorder traversal: ", end=' ')
inorder(root)
print("\nDelete 4")
root = deleteNode(root, 4)
print("Inorder traversal: ", end=' ')
inorder(root)
print("\nDelete 6")
root = deleteNode(root, 6)
print("Inorder traversal: ", end=' ')
inorder(root)
print("\nDelete 3")
root = deleteNode(root, 3)
```

print("Inorder traversal: ", end=' ')

inorder(root)

output:

```
Inorder traversal:  1-> 3-> 4-> 6-> 7-> 8-> 10-> 14->
Delete 4
Inorder traversal:  1-> 3-> 6-> 7-> 8-> 10-> 14->
Delete 6
Inorder traversal:  1-> 3-> 7-> 8-> 10-> 14->
Delete 3
Inorder traversal:  1-> 7-> 8-> 10-> 14->
```

10b. AVL tree

Algorithm:

1. Create a class and represent a node
2. Assign data to the new node, set left and right children to None and set height is 1
3. If tree is non empty, create a new node and compare to current node
4. If new node less than current node, then make it as left child of current node
5. If new node greater than current node, then make it as right child of current node
6. Else, if tree is empty ,add node to the root node
7. After inserting the elements check the Balance Factor of each node.
8. If the balance factor of every node found like 0 or 1 or -1 then the algorithm perform for the next operation.
9. If the balance factor of any node comes other than the above three values then the tree is imbalanced. Then perform rotation to make it balanced and then the algorithm perform for the next operation.
10. Create a method to delete a node and find where the node is stored
11. If the node is a leaf node, delete the node directly.
12. Else if the node has one child, copy the child to the node to be deleted and delete the child node.
13. Else if the node has two children, find the inorder successor of the node.
14. Copy the contents of the inorder successor to the node to be deleted and delete the inorder successor.
15. Again do step 8 and 9
16. Display the preorder traversal

Program:

```
import sys

class TreeNode(object):

    def_init_(self, key):

        self.key = key

        self.left = None

        self.right = None

        self.height = 1

class AVLTree(object):

    def insert_node(self, root, key):

        if not root:

            return TreeNode(key)
```

```
        elif key < root.key:
            root.left = self.insert_node(root.left, key)
        else:
            root.right = self.insert_node(root.right, key)
        root.height = 1 + max(self.getHeight(root.left),
                              self.getHeight(root.right))
        balanceFactor = self.getBalance(root)
        if balanceFactor > 1:
            if key < root.left.key:
                return self.rightRotate(root)
            else:
                root.left = self.leftRotate(root.left)
                return self.rightRotate(root)
        if balanceFactor < -1:
            if key > root.right.key:
                return self.leftRotate(root)
            else:
                root.right = self.rightRotate(root.right)
                return self.leftRotate(root)
        return root
    def delete_node(self, root, key):
        if not root:
            return root
        elif key < root.key:
            root.left = self.delete_node(root.left, key)
        elif key > root.key:
            root.right = self.delete_node(root.right, key)
        else:
```

```
    if root.left is None:
        temp = root.right
        root = None
        return temp
    elif root.right is None:
        temp = root.left
        root = None
        return temp
    temp = self.getMinValueNode(root.right)
    root.key = temp.key
    root.right = self.delete_node(root.right,
                                  temp.key)
if root is None:
    return root
root.height = 1 + max(self.getHeight(root.left),
                      self.getHeight(root.right))
balanceFactor = self.getBalance(root)
if balanceFactor > 1:
    if self.getBalance(root.left) >= 0:
        return self.rightRotate(root)
    else:
        root.left = self.leftRotate(root.left)
        return self.rightRotate(root)
if balanceFactor < -1:
    if self.getBalance(root.right) <= 0:
        return self.leftRotate(root)
    else:
        root.right = self.rightRotate(root.right)
```

```
            return self.leftRotate(root)
        return root
    def leftRotate(self, z):
        y = z.right
        T2 = y.left
        y.left = z
        z.right = T2
        z.height = 1 + max(self.getHeight(z.left),
                           self.getHeight(z.right))
        y.height = 1 + max(self.getHeight(y.left),
                           self.getHeight(y.right))
        return y
    def rightRotate(self, z):
        y = z.left
        T3 = y.right
        y.right = z
        z.left = T3
        z.height = 1 + max(self.getHeight(z.left),
                           self.getHeight(z.right))
        y.height = 1 + max(self.getHeight(y.left),
                           self.getHeight(y.right))
        return y
    def getHeight(self, root):
        if not root:
            return 0
        return root.height
    def getBalance(self, root):
        if not root:
```

```
            return 0
        return self.getHeight(root.left) - self.getHeight(root.right)
    def getMinValueNode(self, root):
        if root is None or root.left is None:
            return root
        return self.getMinValueNode(root.left)
    def preOrder(self, root):
        if not root:
            return
        print("{0} ".format(root.key), end="")
        self.preOrder(root.left)
        self.preOrder(root.right)
    def printHelper(self, currPtr, indent, last):
        if currPtr != None:
            sys.stdout.write(indent)
            if last:
                sys.stdout.write("R---- ")
                indent += "     "
            else:
                sys.stdout.write("L---- ")
                indent += "|    "
            print(currPtr.key)
            self.printHelper(currPtr.left, indent, False)
            self.printHelper(currPtr.right, indent, True)
myTree = AVLTree()
root = None
nums = [15,20,24,10,13,7,30,36,25]
for num in nums:
```

```
    root = myTree.insert_node(root, num)

myTree.printHelper(root, "", True)

key = 24

root = myTree.delete_node(root, key)

print("After Deletion: ")

myTree.printHelper(root, "", True)

key = 20

root = myTree.delete_node(root, key)

print("After Deletion: ")

myTree.printHelper(root, "", True)

key = 15

root = myTree.delete_node(root, key)

print("After Deletion: ")

myTree.printHelper(root, "", True)
```

output:

```
R----13
     L--- 10
     |    L--- 7
     R----24
          L--- 20
          |    L--- 15
          R----30
               L--- 25
               R----36
After Deletion:
R----13
     L--- 10
     |    L--- 7
     R----25
          L--- 20
          |    L--- 15
          R----30
               R--- 36
After Deletion:
R----13
     L--- 10
     |    L--- 7
     R----25
          L--- 15
```

```
          R---30
               R--- 36
After Deletion:
R---13
     L--- 10
     |    L--- 7
     R---30
          L--- 25
          R---36
```

Result:

Thus the program is executed successfully

EX NO:- DATE:-	**11.Implementation of heaps**

Aim:

To write an program to implement binary heap—min and max heap

11a. Algorithm: MAX heap

1. Use an array to store the data.
2. Start storing from index 1, not 0.
3. Create a insert method and find the size of the array
4. If the size is 0, then append the value
5. Else, append the value and set the range for heapify function
6. For any given node at position i:
7. If it is **Left Child** then l= ***[2*i+1]***
8. If it is **Right Child** then r= ***[2*i+2]***
9. Then find the maximum value and swap the position
10. Create delete method and delete the root node
11. Swap the position

Program:

```
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 insert(array, newNum):

    size = len(array)

    if size == 0:

        array.append(newNum)
```

```
    else:
        array.append(newNum);
        for i in range((size//2)-1, -1, -1):
            heapify(array, size, i)
def deleteNode(array, num):
    size = len(array)
    i = 0
    for i in range(0, size):
        if num == array[i]:
            break
    array[i], array[size-1] = array[size-1], array[i]
    array.remove(num)
    for i in range((len(array)//2)-1, -1, -1):
        heapify(array, len(array), i)
arr = []
insert(arr, 35)
insert(arr, 33)
insert(arr, 42)
insert(arr, 10)
insert(arr, 14)
insert(arr, 19)
insert(arr, 27)
insert(arr, 44)
insert(arr, 26)
print ("Max-Heap array: " + str(arr))
deleteNode(arr, 44)
print("After deleting an element: " + str(arr))
deleteNode(arr, 33)
```

```
print("After deleting an element: " + str(arr))
```

output:

```
Max-Heap array: [44, 42, 35, 33, 14, 19, 27, 10, 26]
After deleting an element: [42, 33, 35, 26, 14, 19, 27, 10]
After deleting an element: [42, 26, 35, 10, 14, 19, 27]
```

11b. Algorithm: min heap

1. Use an array to store the data.
2. Start storing from index 1, not 0.
3. Create a insert method and find the size of the array
4. If the size is 0, then append the value
5. Else, append the value and set the range for heapify function
6. For any given node at position i:
7. If it is **Left Child** then l= ***[2*i+1]***
8. If it is **Right Child** then r= ***[2*i+2]***
9. Then find the minimum value and swap the position
10. Create delete method and delete the root node
11. Swap the position

program:

```
def min_heapify(A,k):
    l = left(k)
    r = right(k)
    if l < len(A) and A[l] < A[k]:
        smallest = l
    else:
        smallest = k
    if r < len(A) and A[r] < A[smallest]:
        smallest = r
    if smallest != k:
        A[k], A[smallest] = A[smallest], A[k]
        min_heapify(A, smallest)
def left(k):
    return 2 * k + 1
def right(k):
    return 2 * k + 2
def build_min_heap(A):
    n = int((len(A)//2)-1)
```

```
    for k in range(n, -1, -1):

        min_heapify(A,k)

A = [3,9,2,1,4,5]

build_min_heap(A)

print(A)
```

output:

```
Min heap:
[1, 3, 2, 9, 4, 5]
```

Result:

Thus the program is executed successfully

EX NO:- DATE:-	**12.Graph representation and Traversal algorithm**

Aim:

To write a program to implement graph representation and traversal algorithm

12a. GRAPH REPRESENTATION—ADJACENCY LIST

Algorithm:

1. Create an array of size and type of array must be list of vertices
2. Each array element is initialize with empty list
3. Create a add edge method to store v in list
4. Iterate each given edge of the form (u,v)
5. Append v to the uth list of array

Program:

```
class AdjNode:

    def_init_(self, data):

        self.vertex = data

        self.next = None

 class Graph:

    def_init_(self, vertices):

        self.V = vertices

        self.graph = [None] * self.V

   def add_edge(self, src, dest):

        node = AdjNode(dest)

        node.next = self.graph[src]

        self.graph[src] = node

      node = AdjNode(src)

        node.next = self.graph[dest]

        self.graph[dest] = node

    def print_graph(self):

        for i in range(self.V):
```

```
            print("Adjacency list of vertex {}\n head".format(i), end="")
            temp = self.graph[i]
            while temp:
                print(" -> {}".format(temp.vertex), end="")
                temp = temp.next
            print(" \n")
if __name__ == "__main__":
    V = 5
    graph = Graph(V)
    graph.add_edge(0, 1)
    graph.add_edge(0, 4)
    graph.add_edge(1, 2)
    graph.add_edge(1, 3)
    graph.add_edge(1, 4)
    graph.add_edge(2, 3)
    graph.add_edge(3, 4)
    graph.print_graph()
```

output

```
Adjacency list of  vertex 0
 head -> 4 -> 1

Adjacency list of  vertex 1
 head -> 4 -> 3 -> 2 -> 0

Adjacency list of vertex 2
 head -> 3 -> 1

Adjacency list of vertex 3
 head -> 4 -> 2 -> 1

Adjacency list of vertex 4
 head -> 3 -> 1 -> 0
```

12b. GRAPH REPRESNTATION—ADJACENCY MATRIX

Algorithm:

1. Create a matrix of size N*N
2. Initialise it with zero
3. Iterate over each edge of the form(u,v)
4. Assign 1 to matix[u][v]
5. Checks if the vertex exists in the graph
6. Checks if the vertex is connecting to itself
7. Else, connecting the vertices
8. Define a remove function
9. Check the vertex is already present
10. Else, remove the vertex

Program:

```
class Graph:

    __n = 0

    __g =[[0 for x in range(10)] for y in range(10)]

    def_init_(self, x):

      self._n = x

      for i in range(0, self._n):

          for j in range(0, self._n):

              self___g[i][j]= 0

    def displayAdjacencyMatrix(self):

      print("\n\n Adjacency Matrix:", end ="")

      for i in range(0, self._n):

          print()

          for j in range(0, self._n):

              print("", self._g[i][j], end ="")

    def addEdge(self, x, y):

      if(x>= self._n) or (y >= self._n):

          print("Vertex does not exists !")

      if(x == y):
```

```
            print("Same Vertex !")
        else:
            self___g[y][x]= 1
            self___g[x][y]= 1
    def addVertex(self):
        self._n = self._n + 1;
        for i in range(0, self._n):
            self.__g[i][self.__n-1]= 0
            self._g[self._n-1][i]= 0
    def removeVertex(self, x):
        if(x>self._n):
            print("Vertex not present !")
        else:
            while(x<self._n):
                for i in range(0, self._n):
                    self._g[i][x]= self._g[i][x + 1]
                for i in range(0, self._n):
                    self._g[x][i]= self._g[x + 1][i]
                x = x + 1
            self._n = self.__n - 1
obj = Graph(4);
obj.addEdge(0, 1);
obj.addEdge(0, 2);
obj.addEdge(1, 2);
obj.addEdge(2, 3);
obj.displayAdjacencyMatrix();
#obj.addVertex();
#obj.addEdge(4, 1);
```

```
#obj.addEdge(4, 3);

obj.displayAdjacencyMatrix();

obj.removeVertex(1);

obj.displayAdjacencyMatrix();
```

output

```
Adjacency Matrix:
 0  1  1  0
 1  0  1  0
 1  1  0  1
 0  0  1  0

 Adjacency Matrix:
 0  1  1  0
 1  0  1  0
 1  1  0  1
 0  0  1  0

 Adjacency Matrix:
 0  1  0
 1  0  1
 0  1  0
```

12c. TRAVERSAL ALGORITHM—BFS

Algorithm:

1. Start by putting any one of the graph's vertices at the back of a queue.
2. Take the front item of the queue and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the back of the queue.
4. Keep repeating steps 2 and 3 until the queue is empty.

Program 1:

```
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)
print("Following is the Breadth-First Search")
bfs(visited, graph, '5')
```

output:

```
Following is the Breadth-First Search
5 3 7 2 4 8
```

program 2:

```
graph = {'A': ['B', 'C', 'E'],
         'B': ['A','D', 'E'],
         'C': ['A', 'F', 'G'],
         'D': ['B'],
         'E': ['A', 'B','D'],
         'F': ['C'],
```

```
          'G': ['C']}
def bfs_connected_component(graph, start):
        explored = []
        queue = [start]
    while queue:
            node = queue.pop(0)
        if node not in explored:
                explored.append(node)
            neighbours = graph[node]
            for neighbour in neighbours:
                queue.append(neighbour)
    return explored
bfs_connected_component(graph,'A')
```

output:

```
['A', 'B', 'C', 'E', 'D', 'F', 'G']
```

12d. TRAVERSAL ALGORITHM—DFS

Algorithm:

1. Create a recursive function that takes the index of the node and a visited array.
2. Mark the current node as visited and print the node.
3. Traverse all the adjacent and unmarked nodes and call the recursive function with the index of the adjacent node.
4. Run a loop from 0 to the number of vertices and check if the node is unvisited in the previous DFS, call the recursive function with the current node.

Program:

```
def recursive_dfs(graph, source,path = []):
       if source not in path:
           path.append(source)
           if source not in graph:
               return path
           for neighbour in graph[source]:
               path = recursive_dfs(graph, neighbour, path)
       return path
graph = {"A":["B","C", "D"],
           "B":["E"],
           "C":["F","G"],
           "D":["H"],
           "E":["I"],
           "F":["J"]}
path = recursive_dfs(graph, "A")
print(" ".join(path))
```

output:

```
DFS
A B E I C F J G D H
```

Result:

Thus the program is executed succesfully

EX NO:- DATE:-	**13.Single source shortest path algorithm**

Aim:

To write a program to implement single source shortest path algorithm

Algorithm:

1. Start with a weighted graph
2. Choose a starting vertex and assign infinity path values to all other vertices
3. Go to each vertex and update its path length
4. If the path length of the adjacent vertex is lesser than new path length, don’t update it
5. Avoid updating path length of already visited vertices
6. After each iteration pick the unvisited vertiex with the least path length
7. Repeat until all verties have been visited

Program:

```
import heapq
def calculate_distances(graph, starting_vertex):
    distances = {vertex: float('infinity') for vertex in graph}
    distances[starting_vertex] = 0
    pq = [(0, starting_vertex)]
    while len(pq) > 0:
        current_distance, current_vertex = heapq.heappop(pq)
        if current_distance > distances[current_vertex]:
            continue
        for neighbor, weight in graph[current_vertex].items():
            distance = current_distance + weight
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(pq, (distance, neighbor))
    return distances
example_graph = {
    'v1': {'v2': 2, 'v4': 1,},
    'v2': {'v4': 3, 'v5': 10,},
    'v3': {'v1': 4,},
    'v4': {'v3': 2, 'v6': 8, 'v7': 4, 'v5': 2},
    'v5': {'v7': 6,},
    'v6': {'v3': 5,},
    'v7': {'v6': 1,},
}
print(calculate_distances(example_graph, 'v1'))
```

output:

```
shortest path
```

```
{'v1': 0, 'v2': 2, 'v3': 3, 'v4': 1, 'v5': 3, 'v6': 6, 'v7': 5}
```

Result:

Thus the program is executed succesfully

EX NO:- DATE:-	**14.Minimum spanning tree algorithm**

Algorithm:

1. *Sort all the edges in non-decreasing order of their weight.*
2. *Pick the smallest edge.*
3. *Check if it forms a cycle with the spanning tree formed so far.*
4. *If cycle is not formed, include this edge. Else, discard it.*
5. ***3.** Repeat step2 until there are (V-1) edges in the spanning tree.*

Program for KRUSKAL'S ALGORITHM

```
class Edge :

   def_init_(self, arg_src, arg_dst, arg_weight) :
       self.src = arg_src
       self.dst = arg_dst
       self.weight = arg_weight
class Graph :
    def_init_(self, arg_num_nodes, arg_edgelist) :
        self.num_nodes = arg_num_nodes
        self.edgelist  = arg_edgelist
        self.parent    = []
        self.rank      = []
        self.mst       = []
    def FindParent (self, node) :.
        if node != self.parent[node] :
            self.parent[node] = self.FindParent(self.parent[node])
        return self.parent[node]
    def KruskalMST (self) :
        self.edgelist.sort(key = lambda Edge : Edge.weight)
        self.parent = [None] * self.num_nodes
        self.rank = [None] * self.num_nodes
        for n in range(self.num_nodes) :
            self.parent[n] = n
            self.rank[n] = 0
        for edge in self.edgelist :
            root1 = self.FindParent(edge.src)
            root2 = self.FindParent(edge.dst)
            if root1 != root2 :
               self.mst.append(edge)
               if self.rank[root1] < self.rank[root2] :
                  self.parent[root1] = root2
                  self.rank[root2] += 1
               else :
```

```
                self.parent[root2] = root1
                self.rank[root1] += 1
        print("\nEdges of minimum spanning tree in graph :", end=' ')
        cost = 0
        for edge in self.mst :
            print("[" + str(edge.src) + "-" + str(edge.dst) + "](" + str(edge.weight) + ")", end = ' ')
            cost += edge.weight
        print("\nCost of minimum spanning tree : " +str(cost))
def main() :
    num_nodes = 5
    e1 = Edge(0, 1, 5)
    e2 = Edge(0, 3, 6)
    e3 = Edge(1, 2, 1)
    e4 = Edge(1, 3, 3)
    e5 = Edge(2, 3, 4)
    e6 = Edge(2, 4, 6)
    e7 = Edge(3, 4, 2)
    g1 = Graph(num_nodes, [e1, e2, e3, e4, e5, e6, e7])
    g1.KruskalMST()
if__name___== "_main_" :
    main()
```

output:

```
Edges of minimum spanning tree in graph : [1-2](1) [3-4](2) [1-3](3) [0
-1](5)
Cost of minimum spanning tree : 11
```

PRIM’S ALGORITHM

Algorithm:

1. Start at any node in the list
2. Choose a starting vertex is visited and assign all other vertices are unvisited
3. Find an edges e with minimum cost
4. Add the edge e found in previous to the spanning tree and change the edge as visited
5. Repeat the step 2 and 3 until all nodes become visited
6.

Program:

```
INF = 9999999
V = 7
G = [[0, 2, 4, 1, 0, 0, 0],
     [2, 0, 0, 3, 7, 0, 0],
     [4, 0, 0, 2, 0, 5, 0],
     [1, 3, 2, 0, 7, 8, 4],
     [0, 7, 0, 7, 0, 0, 6],
     [0, 0, 5, 8, 0, 0, 1],
```

```
     [0, 0, 0, 4, 6, 1, 0]]
selected = [0, 0, 0, 0, 0, 0, 0]
no_edge = 0
selected[0] = True
print("Edge : Weight\n")
while (no_edge < V - 1):
    minimum = INF
    x = 0
    y = 0
    for i in range(V):
        if selected[i]:
            for j in range(V):
                if ((not selected[j]) and G[i][j]):
                    if minimum > G[i][j]:
                        minimum = G[i][j]
                        x = i
                        y = j
    print(str(x) + "-" + str(y) + ":" + str(G[x][y]))
    selected[y] = True
    no_edge += 1
```

output:

```
Edge : Weight

0-3:1
0-1:2
3-2:2
3-6:4
6-5:1
6-4:6
```

Result:

thus the program is executed successfully