Ex.No:1 Implement simple ADTs as Python classes

Aim:
To Implement simple ADTs as Python classes using Stack,Queue,List using python.

Algorithm:
1.Create a Stack[ ],Queue[],List[] with MAX size as your wish.
2.Write function for all the basic operations of stack,Queue,List - PUSH(), POP() and DISPLAY(),append(),Extend().
3.Close the program

Coding :
Stack:

```
stack = []
stack.append('a')
stack.append('b')
stack.append('c')
print('Initial stack')
print(stack)
print('\nElements poped from stack:')
print(stack.pop())
print(stack.pop())
print(stack.pop())
print('\nStack after elements are poped:')
print(stack)
```

Queue:

```
queue = []
queue.append('a')
queue.append('b')
queue.append('c')
print("Initial queue")
print(queue)
print("\nElements dequeued from queue")
print(queue.pop(0))
print(queue.pop(0))
print(queue.pop(0))
print("\nQueue after removing elements")
print(queue)
```

List:

```
List = [1,2,3,4]
print("Initial List: ")
print(List)
List.extend([8, 'Geeks', 'Always'])
print("\nList after performing Extend Operation: ")
print(List)
```

Output:

Stack:

```
Initial stack
['a', 'b', 'c']
Elements poped from stack:
c
b
a
Stack after elements are poped:
[]
```

Queue:

```
['a', 'b', 'c']
Elements dequeued from queue
a
b
c
Queue after removing elements
[]
```

List:

```
Initial List:
[1, 2, 3, 4]
List after performing Extend Operation:
[1, 2, 3, 4, 8, 'Geeks', 'Always']
```

Result:

Thus the Implemention of simple ADTs as Python classes was executed successfully.

Ex.No:2 Implement recursive algorithms in Python

Aim:
To Implement a recursive algorithms in Python using Fibonacci Series

Algorithm:

Step 1: Input the 'n' value until which the Fibonacci series has to be generated

Step 2: Initialize sum = 0, a = 0, b = 1 and count = 1

Step 3:while (count <= n)

Step 4:print sum

Step 5: Increment the count variable

Step 6:swap a and b

Step 7:sum = a + b

Step 8:while (count > n)

Step 9: End the algorithm

Step 10: Else

Step 11: Repeat from steps 4 to 7

Coding:

```
No = 10
num1, num2 = 0, 1
count = 0
if No <= 0:
  print("Invalid Number")
elif No == 1:
  print("Fibonacci sequence for limit of ",No,":")
  print(num1)
else:
  print("Fibonacci sequence:")
  while count < No:
     print(num1)
     nth = num1 + num2
     num1 = num2
```

```
       num2 = nth
       count += 1
```

Output:

Fibonacci sequence:

```
0
1
1
2
3
5
8
13
21
34
```

Result:

Thus the Implemention of recursive algorithms in Python using Fibonacci series was executed successfully.

Ex.No:3 **Implement List ADT using Python arrays**

Aim:
To Implement List ADT using Python arrays
Algorithm
1.Using define function intialise the list
2.while loop to declare the elements until the condition is satisfied.
3.using convertarr function to convert the elemnts to an array
4.Stop the program
Coding:

```
class node:
    def __init__(self, data):
        self.data=data
        self.next=None
def add(data):
    nn=node(0)
    nn.data=data
    nn.next=None
    return nn
def printarray(a,n):
    i=0
    while(i<n):
        print(a[i], end = " ")
        i=i+1
def findlength(head):
    cur=head
    count=0
    while(cur!=None):
        count=count+1
        cur=cur.next
    return count
def convertarr(head):
    len=findlength(head)
    arr=[]
    index=0
    cur=head
    while(cur!=None):
        arr.append(cur.data)
        cur=cur.next
    printarray(arr, len)
head=node(0)
head=add(6)
head.next = add(4)
head.next.next = add(3)
head.next.next.next = add(4)
convertarr(head)
```

Output:

```
[6,4,3,4]
[6 4 3 4]
```

Result:

Thus the implementation of List in arrays was executed successfully.

Ex.NO:4 Linked list implementations of List

Aim:
To Implement the Linked list implementations of List using python

Algorithm:
1.Create a list[ ] with MAX size as your wish.
2. Write function for all the basic operations of list - create(), insert(), deletion(), display().
3.Using append() to extend the elements, removal() to delete the elements
4.Close the program.

Coding:
```
List = [1,2,3,4]
print("Initial List: ")
print(List)
List.extend([8, 'Geeks', 'Always'])
print("\nList after performing Extend Operation: ")
print(List)
List = []
print("Blank List: ")
print(List)
List = [10, 20, 14]
print("\nList of numbers: ")
print(List)
List = ["Geeks", "For", "Geeks"]
print("\nList Items: ")
print(List[0])
print(List[2])
  Adding the elements:
List = [1,2,3,4]
print("Initial List: ")
print(List)
List.insert(3, 12)
List.insert(0, 'Geeks')
print("\nList after performing Insert Operation: ")
print(List)
List = [1, 2, 3, 4, 5, 6,
        7, 8, 9, 10, 11, 12]
print("Intial List: ")
print(List)
```

```
List.remove(5)
List.remove(6)
print("\nList after Removal of two elements: ")
print(List)
for i in range(1, 5):
   List.remove(i)
print("\nList after Removing a range of elements: ")
print(List)
List = [['Geeks', 'For'] , ['Geeks']]
print("\nMulti-Dimensional List: ")
print(List)
```

Output:

```
Initial blank List:
[]
List after Addition of Three elements:
[1, 2, 4]
List after Addition of elements from 1-3:
[1, 2, 4, 1, 2, 3]
>>>
===================== RESTART: Z:/New folder/queue 1.py
=====================
Initial List:
[1, 2, 3, 4]
List after performing Insert Operation:
['Geeks', 1, 2, 3, 12, 4]
>>>
===================== RESTART: Z:/New folder/queue 1.py
=====================
Intial List:
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
List after Removal of two elements:
[1, 2, 3, 4, 7, 8, 9, 10, 11, 12]
List after Removing a range of elements:
[7, 8, 9, 10, 11, 12]
```

Result:

Thus the list was created,inserted,removed and extend the element was executed successfully.

Ex.No:5 Implementation of Stack and Queue ADTs

Aim:

To Implementation of Stack and Queue ADTs

Algorithm:

1.Create a Stack[ ],Queue[] with MAX size as your wish.
2. Write function for all the basic operations of stack - append(), POP()
3.Close the program.

Coding:

```
stack = []
stack.append('a')
stack.append('b')
stack.append('c')
print('Initial stack')
print(stack)
print('\nElements poped from stack:')
print(stack.pop())
print(stack.pop())
print(stack.pop())
print('\nStack after elements are poped:')
print(stack)
```

Queue:

```
queue = []
queue.append('a')
queue.append('b')
queue.append('c')
print("Initial queue")
print(queue)
print("\nElements dequeued from queue")
print(queue.pop(0))
print(queue.pop(0))
print(queue.pop(0))
print("\nQueue after removing elements")
print(queue)
```

Output:

```
Initial stack
['a', 'b', 'c']
```

```
Elements poped from stack:
c
b
a
Stack after elements are poped:
[]
```

Result:

Thus the program was executed successfully

Ex.No:6a **Application of List**

Aim:

To implement list application using Polynomial Addition in python

Algorithm:

1.Using the define function intial elements will be declared.

2.for loop gives the output of sum of the elements

3.print[poly] statement have the sum of two polynomial elements.

4.Close the program

Coding:

```
def add(A, B, m, n):
    size = max(m, n);
    sum = [0 for i in range(size)]
    for i in range(0, m, 1):
        sum[i] = A[i]
    for i in range(n):
        sum[i] += B[i]
    return sum
def printPoly(poly, n):
    for i in range(n):
        print(poly[i], end = "")
        if (i != 0):
            print("x^", i, end = "")
        if (i != n - 1):
            print(" + ", end = "")
if __name__ == '__main__':
   A = [5, 0, 10, 6]
    B = [1, 2, 4]
    m = len(A)
    n = len(B)
    print("First polynomial is")
    printPoly(A, m)
    print("\n", end = "")
    print("Second polynomial is")
    printPoly(B, n)
    print("\n", end = "")
    sum = add(A, B, m, n)
    size = max(m, n)
    print("sum polynomial is")
    printPoly(sum, size)
```

Output:

```
First polynomial is
5 + 0x^1 + 10x^2 + 6x^3
Second polynomial is
1 + 2x^1 + 4x^2
Sum polynomial is
6 + 2x^1 + 14x^2 + 6x^3
```

Result:
Thus the program was executed successfully.

Ex.No:6b **Application of Stack**

Aim:
To implement the conversion of infix to postfix in stack

Algorithm:
1.Read the given expression
2.check ifempty or not ,the stack will insert the elements.
3.Using push(),pop() to insert the element or remove the element.
4.Check the operator based on the precedence the expression will be evaluated
5.Close the program

Coding:
```
class Conversion:

    def __init__(self, capacity):
        self.top = -1
        self.capacity = capacity

        self.array = []
        self.output = []
        self.precedence = {'+':1, '-':1, '*':2, '/':2, '^':3}

    def isEmpty(self):
        return True if self.top == -1 else False

    def peek(self):
        return self.array[-1]

    def pop(self):
        if not self.isEmpty():
            self.top -= 1
            return self.array.pop()
        else:
            return "$"

    def push(self, op):
        self.top += 1
        self.array.append(op)

    def isOperand(self, ch):
        return ch.isalpha()

    def notGreater(self, i):
        try:
            a = self.precedence[i]
            b = self.precedence[self.peek()]
            return True if a  <= b else False
        except KeyError:
            return False
    def infixToPostfix(self, exp):


        for i in exp:
            if self.isOperand(i):
                self.output.append(i)

            elif i  == '(':
                self.push(i)

            elif i == ')':
                while( (not self.isEmpty()) and
                                self.peek() != '('):
                    a = self.pop()
                    self.output.append(a)
                if (not self.isEmpty() and self.peek() != '('):
                    return -1
                else:
                    self.pop()

            else:
                while(not self.isEmpty() and self.notGreater(i)):
                    self.output.append(self.pop())
                self.push(i)


        while not self.isEmpty():
            self.output.append(self.pop())
        print "".join(self.output)

exp = "a+b*(c^d-e)^(f+g*h)-i"
obj = Conversion(len(exp))
obj.infixToPostfix(exp)
```

Output:

```
abcd^e-fgh*+^*+i-
```

Result:

Thus the conversion can be successfully executed

Ex.No:6c **Application of Queue**

Aim:

To implement the application of queue using FCFS CPU Scheduling

Algorithm:

1. Input the number of processes required to be scheduled using FCFS, burst time for each process and its arrival time.
2. Calculate the Finish Time, Turn Around Time and Waiting Time for each process which in turn help to calculate Average Waiting Time and Average Turn Around Time required by CPU to schedule given set of process using FCFS.
   a. for i = 0, Finish Time $_{T\,0}$ = Arrival Time $_{T\,0}$ + Burst Time $_{T\,0}$
   b. for i >= 1, Finish Time $_{T\,i}$ = Burst Time $_{T\,i}$ + Finish Time $_{T\,i-1}$
   c. for i = 0, Turn Around Time $_{T\,0}$ = Finish Time $_{T\,0}$ - Arrival Time $_{T\,0}$
   d. for i >= 1, Turn Around Time $_{T\,i}$ = Finish Time $_{T\,i}$ - Arrival Time $_{T\,i}$
   e. for i = 0, Waiting Time $_{T\,0}$ = Turn Around Time $_{T\,0}$ - Burst Time $_{T\,0}$
   f. for i >= 1, Waiting Time $_{T\,i}$ = Turn Around Time $_{T\,i}$ - Burst Time $_{T\,i-1}$
3. Process with less arrival time comes first and gets scheduled first by the CPU.
4. Calculate the Average Waiting Time and Average Turn Around Time.
5. Stop the program

Coding:

```
def findWaitingTime(processes, n,                    bt, wt):

    wt[0] = 0
    for i in range(1, n ):
        wt[i] = bt[i - 1] + wt[i - 1]

def findTurnAroundTime(processes, n,                    bt, wt, tat):

    # calculating turnaround
    # time by adding bt[i] + wt[i]
    for i in range(n):
        tat[i] = bt[i] + wt[i]

def findavgTime( processes, n, bt):

    wt = [0] * n
    tat = [0] * n
    total_wt = 0
    total_tat = 0

    findWaitingTime(processes, n, bt, wt)

    findTurnAroundTime(processes, n,                    bt, wt, tat)
    print( "Processes Burst time " +                " Waiting time " +              " Turn around time")


    for i in range(n):
        total_wt = total_wt + wt[i]
        total_tat = total_tat + tat[i]
        print(" " + str(i + 1) + "\t\t" +                str(bt[i]) + "\t "
                    str(wt[i]) + "\t\t " +                str(tat[i]))

    print( "Average waiting time = "+
                   str(total_wt / n))
    print("Average turn around time = "+
                    str(total_tat / n))

if __name__ =="__main__":

    processes = [ 1, 2, 3]
    n = len(processes)
        burst_time = [10, 5, 8]
    findavgTime(processes, n, burst_time)
```

Output:

```
Processes  Burst time  Waiting time  Turn around time

 1       10     0         10

 2        5    10         15

 3        8    15         23

Average waiting time = 8.33333

Average turn around time = 16
```

Result:

Thus the FCFS CPU Scheduling was Executed Successfully

Ex.No:7A **Implementation of searching algorithms**

Aim:

To implement searching using Linear and Binary Search algorithm using python

Algorithm:

Linear Search:

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

Binary search :

1. Read the search element from the user
2. Find the middle element in the sorted list
3. Compare, the search element with the middle element in the sorted list.
4. If both are matching, then display "Given element found!!!" and terminate the function
5. If both are not matching, then check whether the search element is smaller or larger than middle element
. 6. If the search element is smaller than middle element, then repeat steps 2, 3, 4 and 5 for the left sublist of the middle element.
7. If the search element is larger than middle element, then repeat steps 2, 3, 4 and 5 for the right sublist of the middle element.
8. Repeat the same process until we find the search element in the list or until sublist contains only one element.
9. If that element also doesn't match with the search element, then display "Element not found in the list!!!" and terminate the function.

Binary Search Coding:

```
def BinarySearch(arr, low, high, key):
    if high >= low:
        mid = (high + low) // 2
        if (arr[mid] == key):
            return mid
        elif (arr[mid] > key):
            return BinarySearch(arr, low, mid - 1, key)
        else:
            return BinarySearch(arr, mid + 1, high, key)
    else:
        return -1

arr = [ 10, 20, 30, 40, 50, 60, 70, 80, 90, 100 ]
key = 40
result = BinarySearch(arr, 0, len(arr)-1, key)
if result != -1:
    print(key, "Found at index", str(result))
else:
    print(key, "not Found")
```

Linear Search Coding:

```
def linearsearch(arr, x):
    for i in range(len(arr)):
        if arr[i] == x:
            return i
    return -1
arr = ['t','u','t','o','r','i','a','l']
x = 'a'
print("element found at index "+str(linearsearch(arr,x)))
```

Output of Binary Search:

```
40 Found at index 3
```

Output of Linear Search:

```
 element found at index 6
```

Result:

Thus the implementation of searching using Linear and Binary Search using python was executed successfully

Ex.No:7B **Implementation of Sorting Algorithm**

Aim:

To Implement sorting Algorithm using Quick Sort and Insertion Sort algorithm using python

Algorithm:

Quick Sort:

1. Find a “pivot” item in the array. This item is the basis for comparison for a single round.
2. Start a pointer (the left pointer) at the first item in the array.
3. Start a pointer (the right pointer) at the last item in the array.
4. While the value at the left pointer in the array is less than the pivot value, move the left pointer to the right (add 1). Continue until the value at the left pointer is greater than or equal to the pivot value.
5. While the value at the right pointer in the array is greater than the pivot value, move the right pointer to the left (subtract 1). Continue until the value at the right pointer is less than or equal to the pivot value.
6. If the left pointer is less than or equal to the right pointer, then swap the values at these locations in the array.
7. Move the left pointer to the right by one and the right pointer to the left by one.

Insertion Sort:

1. Compare each element with preceding elements
2. Shift each compared element on the right
3. Place the element at the empty spot
4. Print the sorted array

Coding of Quick Sort:

```
def partition(arr,low,high):
  i = ( low-1 )
  pivot = arr[high]
  for j in range(low , high):

    if arr[j] <= pivot:

      i = i+1
      arr[i],arr[j] = arr[j],arr[i]
  arr[i+1],arr[high] = arr[high],arr[i+1]
  return ( i+1 )

def quickSort(arr,low,high):
  if low < high:
        pi = partition(arr,low,high)
        quickSort(arr, low, pi-1)
        quickSort(arr, pi+1, high)

arr = [2,5,3,8,6,5,4,7]
n = len(arr)
quickSort(arr,0,n-1)
print ("Sorted array is:")
for i in range(n):
    print (arr[i],end=" ")
```

Coding of Insertion Sort:

```
def insertionSort(arr):
    for i in range(1, len(arr)):
        key = arr[i]

        j = i-1
        while j >=0 and key < arr[j] :
            arr[j+1] = arr[j]
            j -= 1
        arr[j+1] = key

arr = ['t','u','t','o','r','i','a','l']
insertionSort(arr)
print ("The sorted array is:")
for i in range(len(arr)):
    print (arr[i])
```

Output:

```
Quick Sorted array is:
2 3 4 5 5 6 7 8
Insertion  sorted array is:
a
i
l
o
r
t
```

t
u

Result:
Thus the implementation of searching Quick and Insertion Sort algorithm using python was executed successfully

Ex.No:8 Implementation of Hash tables

Aim:

To Implement the Hash tables using python

Algorithm:

1.Create a structure, data (hash table item) with key and value as data.

2.for loops to define the range within the set of elements.

3.hashfunction(key) for the size of capacity

4.Using insert(),removal() data to be presented or removed.

5. Stop the program

Coding:

```
hashTable = [[],] * 10
def checkPrime(n):
    if n == 1 or n == 0:
        return 0
    for i in range(2, n//2):
        if n % i == 0:
            return 0
    return 1
def getPrime(n):
    if n % 2 == 0:
        n = n + 1
    while not checkPrime(n):
        n += 2
    return n
def hashFunction(key):
    capacity = getPrime(10)
    return key % capacity
def insertData(key, data):
    index = hashFunction(key)
    hashTable[index] = [key, data]
def removeData(key):
    index = hashFunction(key)
    hashTable[index] = 0
insertData(123, "apple")
insertData(432, "mango")
insertData(213, "banana")
insertData(654, "guava")
print(hashTable)
removeData(123)
```

```
print(hashTable)
```

Output:

```
[[], [], [123, 'apple'], [432, 'mango'], [213, 'banana'], [654, 'guava'], [], [], [], []]
[[], [], 0, [432, 'mango'], [213, 'banana'], [654, 'guava'], [], [], [], []]
```

Result:

Thus the Implementation of hashing was executed successfully

Ex.No:9a **Tree representation**

Aim:

To implement tree representation in binary tree format

Algorithm:

```
1.Create  a binary tree.
2.Intially all the left and right vertex are none , then declare the values using insert() function.
3.If data>right element place the element in right
4.If data<left element place the element in left
5.prin the tree
6.Stop the program
```

Coding:

```
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()
root = Node(12)
root.insert(6)
root.insert(14)
root.insert(3)
root.PrintTree()
```

Output:

```
3
6
12
14
```

Result:

Thus the binary tree was successfully created

Ex.No:9b **Tree Traversal Algorithms**

Aim: To Implement traversal using Inorder,Preorder,Postorder techniques.

Algorithm:

```
Inorder(tree)
  1. Traverse the left subtree, i.e., call Inorder(left-subtree)
  2. Visit the root.
  3. Traverse the right subtree, i.e., call Inorder(right-subtree)
Preorder(tree)
  1. Visit the root.
  2. Traverse the left subtree, i.e., call Preorder(left-subtree)
  3. Traverse the right subtree, i.e., call Preorder(right-subtree)
Postorder(tree)
  1. Traverse the left subtree, i.e., call Postorder(left-subtree)
  2. Traverse the right subtree, i.e., call Postorder(right-subtree)
      3. Visit the root
```

Coding:

```
class Node:
    def __init__(self,key):
        self.left = None
        self.right = None
        self.val = key
def printInorder(root):
    if root:
        printInorder(root.left)
        print(root.val),
        printInorder(root.right)
def printPostorder(root):
    if root:
        printPostorder(root.left)
        printPostorder(root.right)
        print(root.val),
def printPreorder(root):
    if root:
        print(root.val),
        printPreorder(root.left)
        printPreorder(root.right)
```

```
root = Node(1)
root.left      = Node(2)
root.right     = Node(3)
root.left.left  = Node(4)
root.left.right  = Node(5)
print ("\nPreorder traversal of binary tree is")
printPreorder(root)
print ("\nInorder traversal of binary tree is")
printInorder(root)
print ("\nPostorder traversal of binary tree is")
printPostorder(root)
```

Output:

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

Result:

Thus the Implementation of traversal using Inorder,Preorder,Postorder techniques was executed successfully

Ex.No:10 **Implementation of Binary Search Trees**

Aim:

To Implement the Binary Search Trees using python

Algorithm:

Step 1-Read the search element from the user.

Step 2 - Compare the search element with the value of root node in the tree.

Step 3 - If both are matched, then display "Given node is found!!!" and terminate the function

Step 4 - If both are not matched, then check whether search element is smaller or larger than that node value.

Step 5 - If search element is smaller, then continue the search process in left subtree.

Step 6- If search element is larger, then continue the search process in right subtree.

Step 7 - Repeat the same until we find the exact element or until the search element is compared with the leaf node

Step 8 - If we reach to the node having the value equal to the search value then display "Element is found" and terminate the function.

Coding:

```
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 findval(self, lkpval):
        if lkpval < self.data:
            if self.left is None:
                return str(lkpval)+" Not Found"
            return self.left.findval(lkpval)
        elif lkpval > self.data:
            if self.right is None:
                return str(lkpval)+" Not Found"
            return self.right.findval(lkpval)
        else:
            print(str(self.data) + ' is found')

    def PrintTree(self):
        if self.left:
            self.left.PrintTree()
        print( self.data),
        if self.right:
            self.right.PrintTree()
root = Node(12)
root.insert(6)
root.insert(14)
root.insert(3)
print(root.findval(7))
```

Output:

```
7 Not Found
14 is found
```

Result:

Thus the Implementation of Binary Search Trees using python was executed successfully.

Ex.NO:11 Implementation of Heaps

Aim:

To Implement the Heap algorithm using python

Algorithm:

1.Insert the heap function in the list

2.using heappush(),heappop(),heapify() to insert ,delete,display the elements.

3.Stop the program

Coding:

```
import heapq
H = [21,1,45,78,3,5]
heapq.heapify(H)
print(H)
heapq.heappush(H,8)
print(H)
heapq.heappop(H)
print(H)
```

Output:

```
1, 3, 5, 78, 21, 45
[1, 3, 5, 78, 21, 45, 8]
[3, 8, 5, 78, 21, 45]
```

Result:

Thus the Implementation of the Heap algorithm was executed succeefully.

Ex.No:12a **Graph representation**

Aim:

To implement the graph representation using python

Algorithm:

Graph Representation Coding:

```
class graph:
    def __init__(self,gdict=None):
        if gdict is None:
            gdict = []
        self.gdict = gdict
    def getVertices(self):
        return list(self.gdict.keys())
graph_elements = { "a" : ["b","c"],
                "b" : ["a", "d"],
                "c" : ["a", "d"],
                "d" : ["e"],
                "e" : ["d"]
                }
g = graph(graph_elements)
print(g.getVertices())
class graph:
    def __init__(self,gdict=None):
        if gdict is None:
            gdict = {}
        self.gdict = gdict
    def edges(self):
        return self.findedges()
    def findedges(self):
        edgename = []
        for vrtx in self.gdict:
            for nxtvrtx in self.gdict[vrtx]:
                if {nxtvrtx, vrtx} not in edgename:
                    edgename.append({vrtx, nxtvrtx})
        return edgename
graph_elements = { "a" : ["b","c"],
                "b" : ["a", "d"],
                "c" : ["a", "d"],
                "d" : ["e"],
        "e" : ["d"]
        }
g = graph(graph_elements)
print(g.edges())
```

Output:

DISPLAYING VERTICES

```
['a', 'b', 'c', 'd', 'e']
```

DISPLAYING EDGES

```
[{'a', 'b'}, {'a', 'c'}, {'d', 'b'}, {'c', 'd'}, {'d', 'e'}]
```

Result:

Thus the implementation of graphs was executed successfully.

Ex.No:12b **Graph Traversal Algorithms**

Aim:

To Implement using BFS,DFS can be traversed.

Algorithm:

DFS:

Step 1 - Define a Stack of size total number of vertices in the graph.

Step 2 - Select any vertex as **starting point** for traversal. Visit that vertex and push it on to the Stack.

Step 3 - Visit any one of the non-visited **adjacent** vertices of a vertex which is at the top of stack and push it on to the stack.

Step 4 - Repeat step 3 until there is no new vertex to be visited from the vertex which is at the top of the stack.

Step 5 - When there is no new vertex to visit then use **back tracking** and pop one vertex from the stack.

Step 6 - Repeat steps 3, 4 and 5 until stack becomes Empty.

Step 7 - When stack becomes Empty, then produce final spanning tree by removing unused edges from the graph

BFS:

Step 1 - Define a Queue of size total number of vertices in the graph.

Step 2 - Select any vertex as **starting point** for traversal. Visit that vertex and insert it into the Queue.

Step 3 - Visit all the non-visited **adjacent** vertices of the vertex which is at front of the Queue and insert them into the Queue.

Step 4 - When there is no new vertex to be visited from the vertex which is at front of the Queue then delete that vertex.

Step 5 - Repeat steps 3 and 4 until queue becomes empty.

Step 6 - When queue becomes empty, then produce final spanning tree by removing unused edges from the graph

Coding:

BFS

```
import collections
def bfs(graph, root):
    visited, queue = set(), collections.deque([root])
    visited.add(root)
    while queue:
        vertex = queue.popleft()
        print(str(vertex) + " ", end="")
        for neighbour in graph[vertex]:
            if neighbour not in visited:
                visited.add(neighbour)
                queue.append(neighbour)
if __name__ == '__main__':
    graph = {0: [1, 2], 1: [2], 2: [3], 3: [1, 2]}
    print("Following is Breadth First Traversal: ")
    bfs(graph, 0)
```

Output:

```
Following is Breadth First Traversal:
0 1 2 3
```

DFS Coding:

```
import sys
def ret_graph():
    return {
        'A': {'B':5.5, 'C':2, 'D':6},
        'B': {'A':5.5, 'E':3},
        'C': {'A':2, 'F':2.5},
        'D': {'A':6, 'F':1.5},
        'E': {'B':3, 'J':7},
        'F': {'C':2.5, 'D':1.5, 'K':1.5, 'G':3.5},
        'G': {'F':3.5, 'I':4},
        'H': {'J':2},
        'I': {'G':4, 'J':4},
        'J': {'H':2, 'I':4},
        'K': {'F':1.5}
    }
start = 'A'
dest = 'J'
visited = []
```

```
stack = []
graph = ret_graph()
path = []
stack.append(start)
visited.append(start)
while stack:
    curr = stack.pop()
    path.append(curr)
    for neigh in graph[curr]:
        if neigh not in visited:
            visited.append(neigh)
            stack.append(neigh)
            if neigh == dest :
                print("FOUND:", neigh)
                print(path)
                sys.exit(0)
print("Not found")
print(path)
```

Output:

```
FOUND: J
['A', 'D', 'F', 'G', 'I']
```

Result:

Thus the implementation of using BFS,DFS graph can be traversed.

Ex.No:13 Implementation of single source shortest path algorithm

Aim:

To Implement single source shortest path algorithm using Bellman Ford Algorithm

Algorithm:

1) This step initializes distances from source to all vertices as infinite and distance to source itself as 0. Create an array dist[] of size |V| with all values as infinite except dist[src] where src is source vertex.
2) This step calculates shortest distances. Do following |V|-1 times where |V| is the number of vertices in given graph.
a) Do following for each edge u-v
If dist[v] > dist[u] + weight of edge uv, then update dist[v]
dist[v] = dist[u] + weight of edge uv
3) This step reports if there is a negative weight cycle in graph. Do following for each edge u-v
If dist[v] > dist[u] + weight of edge uv, then “Graph contains negative weight cycle”
The idea of step 3 is, step 2 guarantees shortest distances if graph doesn’t contain negative weight cycle. If we iterate through all edges one more time and get a shorter path for any vertex, then there is a negative weight cycle

Coding:

```
from sys import maxsize
def BellmanFord(graph, V, E, src):
    dis = [maxsize] * V
    dis[src] = 0
    for i in range(V - 1):
        for j in range(E):
            if dis[graph[j][0]] + \
                    graph[j][2] < dis[graph[j][1]]:
                dis[graph[j][1]] = dis[graph[j][0]] + \
                                    graph[j][2]
    for i in range(E):
        x = graph[i][0]
        y = graph[i][1]
        weight = graph[i][2]
        if dis[x] != maxsize and dis[x] + \
                   weight < dis[y]:
            print("Graph contains negative weight cycle")
    print("Vertex Distance from Source")
    for i in range(V):
        print("%d\t\t%d" % (i, dis[i]))
```

```
if __name__ == "__main__":
    V = 5 # Number of vertices in graph
    E = 8 # Number of edges in graph
    graph = [[0, 1, -1], [0, 2, 4], [1, 2, 3],
             [1, 3, 2], [1, 4, 2], [3, 2, 5],
             [3, 1, 1], [4, 3, -3]]
    BellmanFord(graph, V, E, 0)
```

Output:

```
Vertex Distance from Source
0                                        0
1                                        -1
2                                        2
3                                        -2
```

Result:

Thus the Implementation of single source shortest path algorithm was successfully executed.

Ex.No:14 Implementation of minimum spanning tree algorithms

Aim:

To implement the minimum spanning tree algorithms using Kruskal Algorithm

Algorithm:

1.Label each vertex

2. List the edges in non-decreasing order of weight.

3. Start with the smallest weighted and beginning growing the minimum weighted spanning tree from this edge.

4. Add the next available edge that does not form a cycle to the construction of the minimum weighted spanning tree. If the addition of the next least weighted edge forms a cycle, do not use it.

5. Continue with step 4 until you have a spanning tree.

Coding:

```
class Graph:
    def __init__(self, vertices):
        self.V = vertices
        self.graph = []
    def add_edge(self, u, v, w):
        self.graph.append([u, v, w])
    def find(self, parent, i):
        return self.find(parent, parent[i])
    def apply_union(self, parent, rank, x, y):
        xroot = self.find(parent, x)
        yroot = self.find(parent, y)
        if rank[xroot] < rank[yroot]:
            parent[xroot] = yroot
        elif rank[xroot] > rank[yroot]:
            parent[yroot] = xroot
        else:
            parent[yroot] = xroot
            rank[xroot] += 1
    def kruskal_algo(self):
        result = []
        i, e = 0, 0
        self.graph = sorted(self.graph, key=lambda item: item[2])
        parent = []
        rank = []
        for node in range(self.V):
            parent.append(node)
            rank.append(0)
        while e < self.V - 1:
            u, v, w = self.graph[i]
            i = i + 1
            x = self.find(parent, u)
            y = self.find(parent, v)
            if x != y:
                e = e + 1
                result.append([u, v, w])
                self.apply_union(parent, rank, x, y)
        for u, v, weight in result:
            print("%d - %d: %d" % (u, v, weight))
g = Graph(6)
g.add_edge(0, 1, 4)
g.add_edge(0, 2, 4)
g.add_edge(1, 2, 2)
g.add_edge(1, 0, 4)
g.add_edge(2, 0, 4)
g.add_edge(2, 1, 2)
g.add_edge(2, 3, 3)
g.add_edge(2, 5, 2)
g.add_edge(2, 4, 4)
g.add_edge(3, 2, 3)
g.add_edge(3, 4, 3)
g.add_edge(4, 2, 4)
g.add_edge(4, 3, 3)
g.add_edge(5, 2, 2)
g.add_edge(5, 4, 3)
g.kruskal_algo()
```

Output:

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

Result:

Thus the program was executed successfully.