MALAD KANDIVALI EDUCATION SOCIETY'S

NAGINDAS KHANDWALA COLLEGE OF COMMERCE, ARTS & MANAGEMENT STUDIES & SHANTABEN NAGINDAS KHANDWALA COLLEGE OF SCIENCE

MALAD [W], MUMBAI – 64

AUTONOMOUS INSTITUTION

(Affiliated To University Of Mumbai)

Reaccredited 'A' Grade by NAAC | ISO 9001:2015 Certified

CERTIFICATE

Name: Mr./Ms. **Shyamal Bhatt**

Roll No: **311** Programme: BSc CS Semester: III

This is certified to be a bonafide record of practical works done by the above student in the college laboratory for the course **Data Structures (Course Code: 2032UISPR)** for the partial fulfilment of Third Semester of BSc IT during the academic year 2020-21.

The journal work is the original study work that has been duly approved in the year 2020-21 by the undersigned.

_____ _____

External Examiner Mr. Gangashankar Singh
(Subject-In-Charge)

Date of Examination: (College Stamp)

Class: S.Y. B.Sc. CS Sem- III **Roll No: 311**

Subject: Data Structures

INDEX

Sr No	Date	Topic	Sign
1	04/09/2020	Implement the following for Array: a) Write a program to store the elements in 1-D array and provide an option to perform the operations like searching, sorting, merging, reversing the elements. b) Write a program to perform the Matrix addition, Multiplication and Transpose Operation.	
2	11/09/2020	Implement Linked List. Include options for insertion, deletion and search of a number, reverse the list and concatenate two linked lists.	
3	18/09/2020	Implement the following for Stack: a) Perform Stack operations using Array implementation. b. b) Implement Tower of Hanoi. c) WAP to scan a polynomial using linked list and add two polynomials. d) WAP to calculate factorial and to compute the factors of a given no. (i) using recursion, (ii) using iteration	
4	25/09/2020	Perform Queues operations using Circular Array implementation.	
5	01/10/2020	Write a program to search an element from a list. Give user the option to perform Linear or Binary search.	
6	09/10/2020	WAP to sort a list of elements. Give user the option to perform sorting using Insertion sort, Bubble sort or Selection sort.	
7	16/10/2020	Implement the following for Hashing: a) Write a program to implement the collision technique. b) Write a program to implement the concept of linear probing.	
8	23/10/2020	Write a program for inorder, postorder and preorder traversal of tree.	

SYCS ***Shyamal Bhatt*** **Old Roll no.: 4024, New Roll no.: 311**

Practical 1

Aim: Implement the following for Array:

Theory: An array is a special variable, which can hold more than one value at a time. An array can hold many values under a single name, and you can access the values by referring to an index number. You can use the for in loop to loop through all the elements of an array. You can use the append() method to add an element to an array. You can use the pop() method to remove an element from the array. You can also use the remove() method to remove an element from the array.

(A)Aim: Write a program to store the elements in 1-D array and provide an option to perform the operations like searching, sorting, merging, reversing the elements.

Code:

```
'''
Write a program to store the elements in 1-D array and provide an option to perform
the operatons like searching, sorting, merging, reversing the elements.
'''
class OneD:
    def __init__(self, a):
        self.array = a

    def search(self, e):
        if e in self.array:
            return True
        return False

    def sort(self):
        for i in range(len(self.array)):
            lowest_value_index = i
            for j in range(i+1, len(self.array)):
                if self.array[j] < self.array[lowest_value_index]:
                    lowest_value_index = j
            self.array[i], self.array[lowest_value_index] = self.array[lowest_value_index], self.array[i]
        return self.array

    def merg(self,l):
        self.array = self.array + l
        return self.array

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

a = [5,6,7,89,2,5,6,1]
o = OneD(a)
print(o.sort())
print(o.search(89))
print(o.merg([8,5,7,9,3]))
print(o.reverse())
```

Output:

```
[1, 2, 5, 5, 6, 6, 7, 89]
True
[1, 2, 5, 5, 6, 6, 7, 89, 8, 5, 7, 9, 3]
[3, 9, 7, 5, 8, 89, 7, 6, 6, 5, 5, 2, 1]
```

Aim: Write a program to perform the Matrix addition, Multiplication and Transpose Operation.

Code:

```
import numpy as np
M1 = [[8, 14, -6],
      [12,7,4],
      [-11,3,21]]

M2 = [[3, 16, -6],
           [9,7,-4],
           [-1,3,13]]

M3  = [[0,0,0],
       [0,0,0],
       [0,0,0]]
matrix_length = len(M1)
for i in range(len(M1)):
    for k in range(len(M2)):
            M3[i][k] = M1[i][k] + M2[i][k]

print("The sum of Matrix M1 and M2 = ", M3)
for i in range(len(M1)):
    for k in range(len(M2)):
            M3[i][k] = M1[i][k] * M2[i][k]

print("The multiplication of Matrix M1 and M2 = ", M3)
M1 = np.array([[3, 6, 9], [5, -10, 15], [4,8,12]])
M2 = M1.transpose()

print(M2)
```

Output:

```
The sum of Matrix M1 and M2 =  [[11, 30, -12], [21, 14, 0], [-12, 6, 34]]
The multiplication of Matrix M1 and M2 =  [[24, 224, 36], [108, 49, -16], [11, 9, 273]]
[[  3   5   4]
 [  6 -10   8]
 [  9  15  12]]
```

Practical 2

Aim: Implement Linked List. Include options for insertion, deletion and search of a number, reverse the list and concatenate two linked lists

Theory:

A linked list is a sequence of data elements, which are connected together via links. Each data element contains a connection to another data element in form of a pointer. Python does not have linked lists in its standard library. We are going to study the types of linked lists known as singly linked lists. In this type of data structure there is only one link between any two data elements. We create such a list and create additional methods to insert, update and remove elements from the list. A linked list is created by using the node class we studied in the last chapter. We create a Node object and create another class to use this ode object. Singly linked lists can be traversed in only forward direction starting form the first data element. We simply print the value of the next data element by assigning the pointer of the next node to the current data element. Inserting an element in the linked list involves reassigning the pointers from the existing nodes to the newly inserted node. You can remove an existing node using the key for that node.

Code:

```
class Node:

    def __init__ (self, element, next = None ):
        self.element = element
        self.next = next
        self.previous = None
    def display(self):
        print(self.element)

class LinkedList:

    def __init__(self):
        self.head = None
        self.size = 0



    def _len_(self):
        return self.size

    def get_head(self):
        return self.head


    def is_empty(self):
        return self.size == 0

    def display(self):
        if self.size == 0:
            print("No element")
            return
        first = self.head
        print(first.element.element)
        first = first.next
        while first:
            if type(first.element) == type(my_list.head.element):
                print(first.element.element)
                first = first.next
            print(first.element)
            first = first.next
```

```
def reverse_display(self):
    if self.size == 0:
        print("No element")
        return None
    last = my_list.get_tail()
    print(last.element)
    while last.previous:
        if type(last.previous.element) == type(my_list.head):
            print(last.previous.element.element)
            if last.previous == self.head:
                return None
            else:
                last = last.previous
        print(last.previous.element)
        last = last.previous



def add_head(self,e):
    #temp = self.head
    self.head = Node(e)
    #self.head.next = temp
    self.size += 1

def get_tail(self):
    last_object = self.head
    while (last_object.next != None):
        last_object = last_object.next
    return last_object


def remove_head(self):
    if self.is_empty():
        print("Empty Singly linked list")
    else:
        print("Removing")
        self.head = self.head.next
        self.head.previous = None
        self.size -= 1

def add_tail(self,e):
    new_value = Node(e)
    new_value.previous = self.get_tail()
    self.get_tail().next = new_value
    self.size += 1
```

```
def find_second_last_element(self):
    #second_last_element = None


    if self.size >= 2:
        first = self.head
        temp_counter = self.size -2
        while temp_counter > 0:
            first = first.next
            temp_counter -= 1
        return first


    else:
        print("Size not sufficient")

    return None



def remove_tail(self):
    if self.is_empty():
        print("Empty Singly linked list")
    elif self.size == 1:
        self.head == None
        self.size -= 1
    else:
        Node = self.find_second_last_element()
        if Node:
            Node.next = None
            self.size -= 1

def get_node_at(self,index):
    element_node = self.head
    counter = 0
    if index == 0:
        return element_node.element
    if index > self.size-1:
        print("Index out of bound")
        return None
    while(counter < index):
        element_node = element_node.next
        counter += 1
    return element_node
```

```
def get_previous_node_at(self,index):
    if index == 0:
        print('No previous value')
        return None
    return my_list.get_node_at(index).previous

def remove_between_list(self,position):
    if position > self.size-1:
        print("Index out of bound")
    elif position == self.size-1:
        self.remove_tail()
    elif position == 0:
        self.remove_head()
    else:
        prev_node = self.get_node_at(position-1)
        next_node = self.get_node_at(position+1)
        prev_node.next = next_node
        next_node.previous = prev_node
        self.size -= 1

def add_between_list(self,position,element):
    element_node = Node(element)
    if position > self.size:
        print("Index out of bound")
    elif position == self.size:
        self.add_tail(element)
    elif position == 0:
        self.add_head(element)
    else:
        prev_node = self.get_node_at(position-1)
        current_node = self.get_node_at(position)
        prev_node.next = element_node
        element_node.previous = prev_node
        element_node.next = current_node
        current_node.previous = element_node
        self.size += 1
```

Output:

```
[1, 2, 5, 5, 6, 6, 7, 89]
True
[1, 2, 5, 5, 6, 6, 7, 89, 8, 5, 7, 9, 3]
[3, 9, 7, 5, 8, 89, 7, 6, 6, 5, 5, 2, 1]
```

Practical 3

Aim: Implement the following for Stack:

Theory:

In English dictionary the word stack means arranging objects on over another. It is the same way memory is allocated in this data structure. It stores the data elements in a similar fashion as a bunch of plates are stored one above another in the kitchen. So, stack data structure allows operations at one end which can be called top of the stack. We can add elements or remove elements only form this end of the stack. In a stack the element inserted last in sequence will come out first as we can remove only from the top of the stack. Such feature is known as Last in First Out(LIFO) feature. The operations of adding and removing the elements is known as PUSH and POP. In the following program we implement it as add and remove functions. We declare an empty list and use the append() and pop() methods to add and remove the data elements. The remove function in the following program returns the top most element.

A)Aim: Perform Stack operations using Array implementation.

Code:

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

    def __len__(self):
        return len(self.data)

    def is_empty(self):
        return len(self.data) == 0

    def push(self, e):
        self.data.append(e)

    def top(self):
        if self.is_empty():
            print('Stack is Empty')
        return self.data[-1]

    def pop(self):
        if self.is_empty():
            print('Stack is Empty')
        return self.data.pop()

s = Stack()
print(s.is_empty())
s.push(5)
s.push(7)
s.push(9)
s.push(6)
s.push(2)
print(s.top())
print(s.__len__())
s.pop()
```

Output:

```
True
2
5
```

C) Aim: WAP to scan a polynomial using linked list and add two polynomials.

Code and Output:

```
class Node:

    def __init__ (self, element, next = None ):
        self.element = element
        self.next = next
        self.previous = None
    def display(self):
        print(self.element)

class LinkedList:

    def __init__(self):
        self.head = None
        self.size = 0



    def _len_(self):
        return self.size

    def get_head(self):
        return self.head


    def is_empty(self):
        return self.size == 0

    def display(self):
        if self.size == 0:
            print("No element")
            return
        first = self.head
        print(first.element.element)
        first = first.next
        while first:
            if type(first.element) == type(my_list.head.element):
                print(first.element.element)
                first = first.next
            print(first.element)
            first = first.next

    def reverse_display(self):
        if self.size == 0:
            print("No element")
            return None
        last = my_list.get_tail()
        print(last.element)
        while last.previous:
            if type(last.previous.element) == type(my_list.head):
                print(last.previous.element.element)
                if last.previous == self.head:
                    return None
                else:
                    last = last.previous
            print(last.previous.element)
            last = last.previous
```

```
def add_head(self,e):
    #temp = self.head
    self.head = Node(e)
    #self.head.next = temp
    self.size += 1

def get_tail(self):
    last_object = self.head
    while (last_object.next != None):
        last_object = last_object.next
    return last_object


def remove_head(self):
    if self.is_empty():
        print("Empty Singly linked list")
    else:
        print("Removing")
        self.head = self.head.next
        self.head.previous = None
        self.size -= 1

def add_tail(self,e):
    new_value = Node(e)
    new_value.previous = self.get_tail()
    self.get_tail().next = new_value
    self.size += 1

def find_second_last_element(self):
    #second_last_element = None


    if self.size >= 2:
        first = self.head
        temp_counter = self.size -2
        while temp_counter > 0:
            first = first.next
            temp_counter -= 1
        return first


    else:
        print("Size not sufficient")

    return None



def remove_tail(self):
    if self.is_empty():
        print("Empty Singly linked list")
    elif self.size == 1:
        self.head == None
        self.size -= 1
    else:
        Node = self.find_second_last_element()
        if Node:
            Node.next = None
            self.size -= 1
```

```
def get_node_at(self,index):
    element_node = self.head
    counter = 0
    if index == 0:
        return element_node.element
    if index > self.size-1:
        print("Index out of bound")
        return None
    while(counter < index):
        element_node = element_node.next
        counter += 1
    return element_node

def get_previous_node_at(self,index):
    if index == 0:
        print('No previous value')
        return None
    return my_list.get_node_at(index).previous

def remove_between_list(self,position):
    if position > self.size-1:
        print("Index out of bound")
    elif position == self.size-1:
        self.remove_tail()
    elif position == 0:
        self.remove_head()
    else:
        prev_node = self.get_node_at(position-1)
        next_node = self.get_node_at(position+1)
        prev_node.next = next_node
        next_node.previous = prev_node
        self.size -= 1

def add_between_list(self,position,element):
    element_node = Node(element)
    if position > self.size:
        print("Index out of bound")
    elif position == self.size:
        self.add_tail(element)
    elif position == 0:
        self.add_head(element)
    else:
        prev_node = self.get_node_at(position-1)
        current_node = self.get_node_at(position)
        prev_node.next = element_node
        element_node.previous = prev_node
        element_node.next = current_node
        current_node.previous = element_node
        self.size += 1

def search (self,search_value):
    index = 0
    while (index < self.size):
        value = self.get_node_at(index)
        if value.element == search_value:
            return value.element
        index += 1
    print("Not Found")
    return False
```

```
    def merge(self,linkedlist_value):
        if self.size > 0:
            last_node = self.get_node_at(self.size-1)
            last_node.next = linkedlist_value.head
            linkedlist_value.head.previous = last_node
            self.size = self.size + linkedlist_value.size

        else:
            self.head = linkedlist_value.head
            self.size = linkedlist_value.size


my_list = LinkedList()
order = int(input('Enter the order for polynomial : '))
my_list.add_head(Node(int(input(f"Enter coefficient for power {order} : "))))
for i in reversed(range(order)):
    my_list.add_tail(int(input(f"Enter coefficient for power {i} : ")))

my_list2 = LinkedList()
my_list2.add_head(Node(int(input(f"Enter coefficient for power {order} : "))))
for i in reversed(range(order)):
    my_list2.add_tail(int(input(f"Enter coefficient for power {i} : ")))

for i in range(order + 1):
    print(my_list.get_node_at(i).element + my_list2.get_node_at(i).element)
```

```
First polynomial is:
50x^ 1 + 10x^ 2 + 6x^ 3
Second polynomial is:
12x^ 1 + 4x^ 2
Sum of polynomial is:
60x^ 1 + 0x^ 2 + 0x^ 3
```

D) Aim: WAP to calculate factorial and to compute the factors of a given no. (i) using recursion, (ii) using iteration.

Code and Output:

```
def fact_r(num):
    if num < 0:
        print('Factorial does not exist')
    elif num == 0:
        print('The Factorial of 0 is 1')
    else:
        if num == 1:
            return num
        else:
            return num * fact_r(num-1)

print(fact_r(5))
```

```
120
```

```python
def fact_i(num):
    factorial = 1
    if num < 0:
        print("Sorry, factorial does not exist for negative numbers")
    elif num == 0:
        print("The factorial of 0 is 1")
    else:
        for i in range(1,num + 1):
           factorial = factorial*i
        print("The factorial is: ",factorial)

fact_i(5)
```

```
The factorial is:  120
```

Practical 4

Aim: Perform Queues operations using Circular Array implementation.

Theory:

An array is called circular if we consider first element as next of last element. Circular arrays are used to implement queue. The approach takes of O(n) time but takes extra space of order O(n). An efficient solution is to deal with circular arrays using the same array. If a careful observation is run through the array, then after nth index, the next index always starts from 0 so using mod operator, we can easily access the elements of the circular list, if we use (i)%n and run the loop from ith index to n+i th index. and apply mod we can do the traversal in a circular array within the given array without using any extra space.

Code and Output:

```
class ArrayQueue:
    """FIFO queue implementation using a Python list as underlying storage."""
    DEFAULT_CAPACITY = 10          # moderate capacity for all new queues

    def __init__(self):
        """Create an empty queue."""
        self._data = [None] * ArrayQueue.DEFAULT_CAPACITY
        self._size = 0
        self._front = 0
        self._back = 0

    def __len__(self):
        """Return the number of elements in the queue."""
        return self._size

    def is_empty(self):
        """Return True if the queue is empty."""
        return self._size == 0

    def first(self):
        """Return (but do not remove) the element at the front of the queue.
        Raise Empty exception if the queue is empty.
        """
        if self.is_empty():
            raise Empty('Queue is empty')
        return self._data[self._front]


    def dequeueStart(self):
        """Remove and return the first element of the queue (i.e., FIFO).
        Raise Empty exception if the queue is empty.
        """
        if self.is_empty():
            raise Empty('Queue is empty')
        answer = self._data[self._front]
        self._data[self._front] = None         # help garbage collection
        self._front = (self._front + 1) % len(self._data)
        self._size -= 1
        self._back = (self._front + self._size - 1) % len(self._data)
        return answer
```

```
    def dequeueEnd(self):
        """Remove and return the Last element of the queue.
        Raise Empty exception if the queue is empty.
        """
        if self.is_empty():
            raise Empty('Queue is empty')
        back = (self._front + self._size - 1) % len(self._data)
        answer = self._data[back]
        self._data[back] = None         # help garbage collection
        self._front = self._front
        self._size -= 1
        self._back = (self._front + self._size - 1) % len(self._data)
        return answer

    def enqueueEnd(self, e):
        """Add an element to the back of queue."""
        if self._size == len(self._data):
            self._resize(2 * len(self.data))     # double the array size
        avail = (self._front + self._size) % len(self._data)
        self._data[avail] = e
        self._size += 1
        self._back = (self._front + self._size - 1) % len(self._data)

    def enqueueStart(self, e):
        """Add an element to the start of queue."""
        if self._size == len(self._data):
            self._resize(2 * len(self._data))     # double the array size
        self._front = (self._front - 1) % len(self._data)
        avail = (self._front + self._size) % len(self._data)
        self._data[self._front] = e
        self._size += 1
        self._back = (self._front + self._size - 1) % len(self._data)

    def _resize(self, cap):                 # we assume cap >= len(self)
        """Resize to a new list of capacity >= len(self)."""
        old = self._data                       # keep track of existing list
        self._data = [None] * cap              # allocate list with new capacity
        walk = self._front
        for k in range(self._size):            # only consider existing elements
            self._data[k] = old[walk]            # intentionally shift indices
            walk = (1 + walk) % len(old)         # use old size as modulus
        self._front = 0                        # front has been realigned
        self._back = (self._front + self._size - 1) % len(self._data)


queue = ArrayQueue()
queue.enqueueEnd(1)
print(f"First Element: {queue._data[queue._front]}, Last Element: {queue._data[queue._back]}")
queue._data
queue.enqueueEnd(2)
print(f"First Element: {queue._data[queue._front]}, Last Element: {queue._data[queue._back]}")
queue._data
queue.dequeueStart()
print(f"First Element: {queue._data[queue._front]}, Last Element: {queue._data[queue._back]}")
queue.enqueueEnd(3)
print(f"First Element: {queue._data[queue._front]}, Last Element: {queue._data[queue._back]}")
queue.enqueueEnd(4)
print(f"First Element: {queue._data[queue._front]}, Last Element: {queue._data[queue._back]}")
queue.dequeueStart()
print(f"First Element: {queue._data[queue._front]}, Last Element: {queue._data[queue._back]}")
queue.enqueueStart(5)
print(f"First Element: {queue._data[queue._front]}, Last Element: {queue._data[queue._back]}")
queue.dequeueEnd()
print(f"First Element: {queue._data[queue._front]}, Last Element: {queue._data[queue._back]}")
queue.enqueueEnd(6)
print(f"First Element: {queue._data[queue._front]}, Last Element: {queue._data[queue._back]}")
```

```
First Element: 1, Last Element: 1
First Element: 1, Last Element: 2
First Element: 2, Last Element: 2
First Element: 2, Last Element: 3
First Element: 2, Last Element: 4
First Element: 3, Last Element: 4
First Element: 5, Last Element: 4
First Element: 5, Last Element: 3
First Element: 5, Last Element: 6
```

Practical 5

Aim: Write a program to search an element from a list. Give user the option to perform Linear or Binary search.

Theory: An array is called circular if we consider first element as next of last element. Circular arrays are used to implement queue. The approach takes of O(n) time but takes extra space of order O(n). An efficient solution is to deal with circular arrays using the same array. If a careful observation is run through the array, then after nth index, the next index always starts from 0 so using mod operator, we can easily access the elements of the circular list, if we use (i)%n and run the loop from ith index to (n+i) th index . and apply mod we can do the traversal in a circular array within the given array without using any extra space.

Code and Output:

```
class search:
    def __init__(self, l, e, type):
        self.l = l
        self.e = e
        self.type = type
        if type == 'l' or 'L':
            if self.linear():
                print('Element Present at ', self.linear())
            else:
                print("Element Not there!")
        elif type == 'B' or 'b':
            if self.binary() != -1:
                print("Element is present at index", str(self.binary()))
            else:
                print("Element is not present in array")
        else:
            print('Enter a valide type of Search')

    def linear(self):
        for i in range(len(self.l)):
            if self.l[i] == self.e:
                return i
        return False

    def binary(self):
        low = 0
        high = len(self.l) - 1
        mid = 0

        while low <= high:
            mid = (high + low) // 2
            if self.l[mid] < self.e:
                low = mid + 1
            elif self.l[mid] > self.e:
                high = mid - 1
            else:
                return mid
        return -1


a = [1,8,3,4,5,9,2,7]
search(a, 8,'b')
```

```
Element Present at  1
```

Practical 6

Aim: WAP to sort a list of elements. Give user the option to perform sorting using Insertion sort, Bubble sort or Selection sort.

Theory:

Selection sort is to repetitively pick up the smallest element and put it into the right position. A loop through the array finds the smallest element easily. After the smallest element is put in the first position, it is fixed and then we can deal with the rest of the array. The following implementation uses a nested loop to repetitively pick up the smallest element and swap it to its final position. The swap() method exchanges two elements in an array Output:. Insertion sort maintains a sorted subarray, and repetitively inserts new elements into it. Suppose the array length is n. The outer loop runs roughly n times, and the inner loop on average runs n/2 times. The total time is about t(n) = n * (n/2) = O(n^2). In terms of the efficiency, this is the same as selection sort. Bubble sort repetitively compares adjacent pairs of elements and swaps if necessary. Suppose the array length is n. The outer loop runs roughly n times, and the inner loop on average runs n/2 times. The total time is about t(n) = n * (n/2) = O(n^2). In terms of the efficiency, this is the same as selection sort and insertion sort.

Code and Output:

```
class sort:
    def __init__(self,arr,type):
        self.arr = arr
        self.type = type
        if self.type == 'i' or 'I':
            self.insertionSort()
        elif self.type == 'b' or 'B':
            self.bubbleSort()
        elif self.type == 's' or 'S':
            self.selection()
        else:
            print('Invalid Sort type')

    def insertionSort(self):
        print('Using Insertion Sort')
        for i in range(1, len(self.arr)):
            key = self.arr[i]
            j = i-1
            while j >=0 and key < self.arr[j] :
                    self.arr[j+1] = self.arr[j]
                    j -= 1
            self.arr[j+1] = key

    def bubbleSort(self):
        print('Using Bubble Sort')
        n = len(self.arr)
        for i in range(n-1):
            for j in range(0, n-i-1):
                if self.arr[j] > self.arr[j+1] :
                    self.arr[j], self.arr[j+1] = self.arr[j+1], self.arr[j]

    def selection(self):
        print('Using Selection Sort')
        for i in range(len(self.arr)):
            min_ = i
            for j in range(i+1, len(self.arr)):
                if self.arr[min_] > self.arr[j]:
                    min_ = j
            self.arr[i], self.arr[min_] = self.arr[min_], self.arr[i]

a = [7,8,6,2,3,5,7,2,0,1,3]
sort(a, 'I')
print(a)
```

```
Using Insertion Sort
[0, 1, 2, 2, 3, 3, 5, 6, 7, 7, 8]
```

Practical 7

Aim: Implement the following for Hashing:

Theory:

Hash Table: An array that stores pointers to records corresponding to a given phone number. An entry in hash table is NIL if no existing phone number has hash function value equal to the index for the entry. Collision Handling: Since a hash function gets us a small number for a big key, there is possibility that two keys result in same value. The situation where a newly inserted key maps to an already occupied slot in hash table is called collision and must be handled using some collision handling technique. Following are the ways to handle collisions: Chaining: The idea is to make each cell of hash table point to a linked list of records that have same hash function value. Chaining is simple, but requires additional memory outside the table. Open Addressing: In open addressing, all elements are stored in the hash table itself. Each table entry contains either a record or NIL. When searching for an element, we one by one examine table slots until the desired element is found or it is clear that the element is not in the table.

A) Aim: Write a program to implement the collision technique.

Code and Output:

```
def display_hash(hashTable):
    for i in range(len(hashTable)):
        print(i, end = " ")

        for j in hashTable[i]:
            print("-->", end = " ")
            print(j, end = " ")
        print()
HashTable = [[] for _ in range(10)]
def Hashing(keyvalue):
    return keyvalue % len(HashTable)
def insert(Hashtable, keyvalue, value):

    hash_key = Hashing(keyvalue)
    Hashtable[hash_key].append(value)

insert(HashTable, 10, 'Allahabad')
insert(HashTable, 25, 'Mumbai')
insert(HashTable, 20, 'Mathura')
insert(HashTable, 9, 'Delhi')
insert(HashTable, 21, 'Punjab')
insert(HashTable, 21, 'Noida')

display_hash (HashTable)
```

```
0 --> Allahabad --> Mathura
1 --> Punjab --> Noida
2
3
4
5 --> Mumbai
6
7
8
9 --> Delhi
```

B) Aim: Write a program to implement the concept of linear probing.

Code and Output:

```
list_ = [113 , 117 , 97 , 100 , 114 , 108 , 116 , 105 , 99]


hash_values = []
def hash_func(list_):
    list_2 = [None for i in range(11)]
    for i in list_:
        #print(i % len(list_2))
        hash_values.append(i % len(list_2))
        list_2[i % len(list_2)] = i
    print(list_2)
    print(list_)
    print(hash_values)
    print(116 % 11)
    print(97 % 11)



print(hash_func(list_))
```

```
[99, 100, None, 113, 114, None, 105, 117, None, 108, None]
[113, 117, 97, 100, 114, 108, 116, 105, 99]
[3, 7, 9, 1, 4, 9, 6, 6, 0]
6
9
None
```

Practical 8

Aim: Write a program for inorder, postorder and preorder traversal of tree.

Theory:

Tree represents the nodes connected by edges. It is a non-linear data structure. It has the following properties. 1. One node is marked as Root node. 2. Every node other than the root is associated with one parent node. 3. Each node can have an arbitrary number of child node. We create a tree data structure in python by using the concept of nodes. We designate one node as root node and then add more nodes as child nodes. To insert into a tree we use the same node class created above and add an insert method to it The insert method compares the value of the node to the parent node and decides to add it as a left node or a right node. Finally, the PrintTree method is used to print the tree. A tree can be traversed by deciding on a sequence to visit each node. As we can clearly see we can start at a node then visit the left sub-tree first and right sub-tree next. Or we can also visit the right sub-tree first and left sub-tree next. Accordingly, there are different names for these tree traversal methods.

Code and Output:

```
class Queue(object):
    def __init__(self):
        self.items = []

    def enqueue(self, item):
        self.items.insert(0, item)

    def dequeue(self):
        if not self.is_empty():
            return self.items.pop()

    def is_empty(self):
        return len(self.items) == 0

    def peek(self):
        if not self.is_empty():
            return self.items[-1].value

    def __len__(self):
        return self.size()

    def size(self):
        return len(self.items)


class Node(object):
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None


class BinaryTree(object):
    def __init__(self, root):
        self.root = Node(root)

    def print_tree(self, traversal_type):
        if traversal_type == "preorder":
            return self.preorder_print(tree.root, "")
        elif traversal_type == "inorder":
            return self.inorder_print(tree.root, "")
        elif traversal_type == "postorder":
            return self.postorder_print(tree.root, "")
        else:
            print("Traversal type " + str(traversal_type) + " is not supported.")
            return False
```

```
    def preorder_print(self, start, traversal):
        """Root->Left->Right"""
        if start:
            traversal += (str(start.value) + "-")
            traversal = self.preorder_print(start.left, traversal)
            traversal = self.preorder_print(start.right, traversal)
        return traversal

    def inorder_print(self, start, traversal):
        """Left->Root->Right"""
        if start:
            traversal = self.inorder_print(start.left, traversal)
            traversal += (str(start.value) + "-")
            traversal = self.inorder_print(start.right, traversal)
        return traversal

    def postorder_print(self, start, traversal):
        """Left->Right->Root"""
        if start:
            traversal = self.inorder_print(start.left, traversal)
            traversal = self.inorder_print(start.right, traversal)
            traversal += (str(start.value) + "-")
        return traversal

tree = BinaryTree(1)
tree.root.left = Node(2)
tree.root.right = Node(3)
tree.root.left.left = Node(4)
tree.root.left.right = Node(5)

print(tree.print_tree("preorder"))
print(tree.print_tree("inorder"))
print(tree.print_tree("postorder"))
```

```
1-2-4-5-3-
4-2-5-1-3-
4-2-5-3-1-
```

Practical 9

Aim: Write a program to generate the adjacency matrix.

Theory:

In graph theory and computer science, an adjacency matrix is a square matrix used to represent a finite graph. The elements of the matrix indicate whether pairs of vertices are adjacent or not in the graph. In the special case of a finite simple graph, the adjacency matrix is a (0,1)-matrix with zeros on its diagonal. If the graph is undirected (i.e. all of its edges are bidirectional), the adjacency matrix is symmetric. The relationship between a graph and the eigenvalues and eigenvectors of its adjacency matrix is studied in spectral graph theory. The adjacency matrix of a graph should be distinguished from its incidence matrix, a different matrix representation whose elements indicate whether vertex– edge pairs are incident or not, and its degree matrix, which contains information about the degree of each vertex.

Code and Output:

```
class Graph(object):
    def __init__(self, size):
        self.adjMatrix = []
        for i in range(size):
            self.adjMatrix.append([0 for i in range(size)])
        self.size = size
    def add_edge(self, v1, v2):
        if v1 == v2:
            print("Same vertex %d and %d" % (v1, v2))
        self.adjMatrix[v1][v2] = 1
        self.adjMatrix[v2][v1] = 1
    def remove_edge(self, v1, v2):
        if self.adjMatrix[v1][v2] == 0:
            print("No edge between %d and %d" % (v1, v2))
            return
        self.adjMatrix[v1][v2] = 0
        self.adjMatrix[v2][v1] = 0
    def __len__(self):
        return self.size
    def print_matrix(self):
        for row in self.adjMatrix:
            for val in row:
                print('{:4}'.format(val)),
            print
def main():
    g = Graph(5)
    g.add_edge(0, 1)
    g.add_edge(0, 2)
    g.add_edge(1, 2)
    g.add_edge(2, 0)
    g.add_edge(2, 3)
    g.print_matrix()
if __name__ == '__main__':
    main()
```

0
1
1
0
0
1
0
1
0
0
1
1
0
1
0
0
0
1
0
0
0
0
0
0
0

Practical 10

Aim: Write a program for shortest path diagram.

Theory: In graphs, to reach from one point to another, we use shortest path algorithms. There are many algorithms to do it. Few of them are Breath First Search, Depth First Search, Djiktras Algorithm, etx. Djikstra's algorithm is a path-finding algorithm, like those used in routing and navigation. We will be using it to find the shortest path between two nodes in a graph. It fans away from the starting node by visiting the next node of the lowest weight and continues to do so until the next node of the lowest weight is the end node.

Code and Output:

```
def BFS_SP(graph, start, goal):
    explored = []
    queue = [[start]]
    if start == goal:
        print("Same Node")
        return
    while queue:
        path = queue.pop(0)
        node = path[-1]
        if node not in explored:
            neighbours = graph[node]
            for neighbour in neighbours:
                new_path = list(path)
                new_path.append(neighbour)
                queue.append(new_path)
                if neighbour == goal:
                    print("Shortest path = ", *new_path)
                    return
            explored.append(node)
    print("So sorry, but a connecting path doesnt exist :")
    return

if __name__ == "__main__":
    graph = {'A': ['B', 'E', 'C'],
            'B': ['A', 'D', 'E'],
            'C': ['A', 'F', 'G'],
            'D': ['B', 'E'],
            'E': ['A', 'B', 'D'],
            'F': ['C'],
            'G': ['C']}
    BFS_SP(graph, 'A', 'D')
```

```
Shortest path =  A B D
```