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: Ms. SHRADDHA PANDEY

Roll No: 329 Programme: BSc IT 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. IT Sem- III **Roll No:** 329

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.	

Link for repository:

https://github.com/shraddha1261/DS

Practical 1

Implement the following for Array

Practical 1 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.

Theory:

- NumPy is a library for the Python programming language, adding support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays.
- One dimensional array contains elements only in one dimension. In other words, the shape of the numpy array should contain only one value in the tuple.
  Numpy array() functions takes a list of elements as argument and returns a one-dimensional array.

Code:

```
import numpy as np
arr = [87, 9, 13, 120, 14, 34]
def binary_search(arr, el, start, end):
    mid = (start + end) // 2
    if el == arr[mid]:
        return mid
    if el < arr[mid]:
        return binary_search(arr, el, start, mid-1)
    else:
        return binary_search(arr, el, mid+1, end)
print(binary_search(arr, 120, 0, len(arr)))

def sorting(arr):
    arr.sort()
    return arr
print(sorting(arr))
```

```
def merge():
    a = [17, 3, 23, 45, 64, 12]
    b = [1, 33, 47]
    merged_list = np.concatenate((a,b))
    return merged_list
print(merge())

def rev():
    rev_list = np.flipud(arr)
    return rev_list
print(rev())
```

Output:

```
3
[9, 13, 14, 34, 87, 120]
[17  3 23 45 64 12  1 33 47]
[120  87  34  14  13   9]
>>>
```

Practical 1 B

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

Theory:

- A Python matrix is a specialized two-dimensional rectangular array of data stored in rows and columns. The data in a matrix can be numbers, strings, expressions, symbols, etc. Matrix is one of the important data structures that can be used in mathematical and scientific calculations.
  The python matrix makes use of arrays, and the same can be implemented.

- Addition:

  Addition of matrix can be done in two ways:

  1) Using FOR loop
  2) Using nested list comprehension

- Multiplication:
  To multiply the matrices, the number of columns in first matrix should be equal to number of rows in second matrix.

- Transpose:
  Transpose of a matrix is the interchanging of rows and columns. It is denoted as X'. The element at ith row and jth column in X will be placed at jth row and ith column in X'. So, if X is a 3x2 matrix, X' will be a 2x3 matrix.

Code:

```
import numpy as np

x = np.array([[12,3,21],[2,43,5],[5,32,53]])
y = np.array([[11,4,6],[21,7,9],[4,8,43]])
a = np.multiply(x,y)
b = np.add(x,y)
c = np.transpose(x)
d = np.transpose(y)
print("Multiplication of two matrices")
print(a)
print("Addition of two matrices")
print(b)
print("Transpose of two matrices")
print(c)
print(d)
```

Output:

```
Multiplication of two matrices
[[ 132   12  126]
 [  42  301   45]
 [  20  256 2279]]
Addition of two matrices
[[23  7 27]
 [23 50 14]
 [ 9 40 96]]
Transpose of two matrices
[[12  2  5]
 [ 3 43 32]
 [21  5 53]]
[[11 21  4]
 [ 4  7  8]
 [ 6  9 43]]
>>>
```

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 linear collection of data elements whose order is not given by their physical placement in memory. Instead, each element points to the next. It is a data structure consisting of a collection of nodes which together represent a sequence.
  The principal benefit of a linked list over a conventional array is that the list elements can be easily inserted or removed without reallocation or reorganization of the entire structure.

- Types of Linked List:
  Simple Linked List − Item navigation is forward only.
  Doubly Linked List − Items can be navigated forward and backward.
  Circular Linked List − Last item contains link of the first element as next and the first element has a link to the last element as previous.

- Basic Operations:
  Insertion Operation:
  Adding a new node in linked list is a more than one step activity. We shall learn this with diagrams here. First, create a node using the same structure and find the location where it has to be inserted.

  Deletion Operation:
  Deletion is also a more than one step process. We shall learn

with pictorial representation. First, locate the target node to be removed, by using searching algorithms.

Reverse Operation:
This operation is a thorough one. We need to make the last node to be pointed by the head node and reverse the whole linked list.
Searching in singly linked list
Searching is performed in order to find the location of a particular element in the list. Searching any element in the list needs traversing through the list and make the comparison of every element of the list with the specified element. If the element is matched with any of the list element then the location of the element is returned from the function.

Concatenate two linked lists:
We just need to follow some very simple steps and the steps to join two lists (say ‘a’ and ‘b’) are as follows:
Traverse over the linked list ‘a’ until the element next to the node is not NULL.
If the element next to the current element is NULL (a->next == NULL) then change the element next to it to ‘b’ (a->next = b).

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):
    self.head = Node(e)
    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):
    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 type(value.element) == type(my_list.head):
                print("Searching at " + str(index) + " and value is " + str(value.element.element))
            else:
                print("Searching at " + str(index) + " and value is " + str(value.element))
            if value.element == search_value:
                print("Found value at " + str(index) + " location")
                return True
```

```
            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
```

```
l1 = Node('A')
my_list = LinkedList()
my_list.add_head(l1)
my_list.add_tail('B')
my_list.add_tail('C')
my_list.add_tail('D')
my_list.get_head().element.element
my_list.add_between_list(2,'Element between')
my_list.remove_between_list(2)
my_list2 = LinkedList()
l2 = Node('E')
my_list2.add_head(l2)
my_list2.add_tail('F')
my_list2.add_tail('G')
my_list2.add_tail('H')
my_list.merge(my_list2)
my_list.get_previous_node_at(3).element
my_list.reverse_display()
```

Output:

Practical 3

Implement the following for Stack

Practical 3 A

Aim: Perform Stack operations using Array implementation.

Theory:

- A stack data structure can be implemented using a one-dimensional array. But stack implemented using array stores only a fixed number of data values. This implementation is very simple. Just define a one-dimensional array of specific size and insert or delete the values into that array by using LIFO principle with the help of a variable called 'top'. Initially, the top is set to -1.
- Whenever we want to insert a value into the stack, increment the top value by one and then insert. Whenever we want to delete a value from the stack, then delete the top value and decrement the top value by one.

- Stack Operations using Array:
  A stack can be implemented using array as follows...

  Before implementing actual operations, first follow the below steps to create an empty stack.

  Step 1 - Include all the header files which are used in the program and define a constant 'SIZE' with specific value.
  Step 2 - Declare all the functions used in stack implementation.
  Step 3 - Create a one dimensional array with fixed size (int

stack[SIZE])

Step 4 - Define a integer variable 'top' and initialize with '-1'. (int top = -1)

Step 5 - In main method, display menu with list of operations and make suitable function calls to perform operation selected by the user on the stack.

Code:

```
class ArrayQ:
    default_capacity = 10
    def __init__(self):
        self.data = [None] * ArrayQ.default_capacity

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

    def enqueueFront(self, data):
        self.data.append(data)

    def dequeueBack(self):
        return self.data.pop(0)

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

dq=ArrayQ()

print(dq.enqueueFront('Shraddha'))
print(dq.size())
print(dq.size())
```

Output:

```
None
11
11
>>>
```

Practical 3 B

Aim: Implement Tower of Hanoi.

Theory:

- Tower of Hanoi is a mathematical puzzle. It consists of three poles and a number of disks of different sizes which can slide onto any poles. The puzzle starts with the disk in a neat stack in ascending order of size in one pole, the smallest at the top thus making a conical shape.
- The objective of the puzzle is to move all the disks from one pole (say ‘source pole’) to another pole (say ‘destination pole’) with the help of the third pole (say auxiliary pole).

- The puzzle has the following two rules:
    1. You can’t place a larger disk onto smaller disk
    2. Only one disk can be moved at a time

- For n disks, total (2^n)– 1 moves are required.

Code:

```
def Tower_of_Hanoi(disk , src, dest, auxiliary):
    if disk==1:
        print("Transfer disk 1 from source",src,"to destination",dest)
        return
    Tower_of_Hanoi(disk-1, src, auxiliary, dest)
    print("Transfer disk",disk,"from source",src,"to destination",dest)
    Tower_of_Hanoi(disk-1, auxiliary, dest, src)

disk = int(input("For how many rings you want to search.?"))
Tower_of_Hanoi(disk,'A','B','C')
```

Output:

```
For how many rings you want to search.?4
Transfer disk 1 from source A to destination C
Transfer disk 2 from source A to destination B
Transfer disk 1 from source C to destination B
Transfer disk 3 from source A to destination C
Transfer disk 1 from source B to destination A
Transfer disk 2 from source B to destination C
Transfer disk 1 from source A to destination C
Transfer disk 4 from source A to destination B
Transfer disk 1 from source C to destination B
Transfer disk 2 from source C to destination A
Transfer disk 1 from source B to destination A
Transfer disk 3 from source C to destination B
Transfer disk 1 from source A to destination C
Transfer disk 2 from source A to destination B
Transfer disk 1 from source C to destination B
>>>
```

Practical 3 C

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

Theory:

- A polynomial p(x) is the expression in variable x which is in the form (axn + bxn-1-1-1 + …. + jx+ k), where a, b, c …., k fall in the category of real numbers and 'n' is non negative integer, which is called the degree of polynomial.

An essential characteristic of the polynomial is that each term in the polynomial expression consists of two parts:

one is the coefficient
other is the exponent

Polynomial can be represented in the various ways. These are:
By the use of arrays
By the use of Linked List

In the Polynomial linked list, the coefficients and exponents of the polynomial are defined as the data node of the list.

For adding two polynomials that are stored as a linked list. We need to add the coefficients of variables with the same power. In a linked list node contains 3 members, coefficient value link to the next node.
Adding two polynomials that are represented by a linked list. We check values at the exponent value of the node. For the same values of exponent, we will add the coefficients.

Explanation − For all power, we will check for the coefficients of the exponents that have the same value of exponents and add them. The return the final polynomial.
Algorithm

Input − polynomial p1 and p2 represented as a linked list.

Step 1: loop around all values of linked list and follow step 2& 3.

Step 2: if the value of a node's exponent. is greater copy this node to result node and head towards the next node.

Step 3: if the values of both node's exponent is same adding the coefficients and then copy the added value with node to the result.

Step 4: Print the resultant 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):
    self.head = Node(e)
    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):
    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
```

```
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)
```

Output:

```
Enter the order for polynomial : 4
Enter coefficient for power 4 : 3
Enter coefficient for power 3 : 5
Enter coefficient for power 2 : 3
Enter coefficient for power 1 : 1
Enter coefficient for power 0 : 3
Enter coefficient for power 4 : 4
Enter coefficient for power 3 : 3
Enter coefficient for power 2 : 2
Enter coefficient for power 1 : 1
Enter coefficient for power 0 : 2
7
8
5
2
5
>>>
```

Practical 3 D

Aim: WAP to calculate factorial and to compute the factors of a given no.

(i) using recursion, (ii) using iteration

Theory:

- Recursion is the process of defining something in terms of itself.

- A physical world example would be to place two parallel mirrors facing each other. Any object in between them would be reflected recursively.

- Python Recursive Function:

  In Python, we know that a function can call other functions. It is even possible for the function to call itself. These types of construct are termed as recursive functions.

  The following image shows the working of a recursive function called recurse.
  When we call this function with a positive integer, it will recursively call itself by decreasing the number.

  Each function multiplies the number with the factorial of the number below it until it is equal to one.

- Iteration, in the context of computer programming, is a process wherein a set of instructions or structures are repeated in a sequence a specified number of times or until a condition is met. When the first set of instructions is executed again, it is called an iteration.

- Factorial of a number is the product of all integers between 1 and itself. To find factorial of a given number, let us form a for loop over a range from 1 to itself. Remember that range() function excludes the stop value. Hence stop value should be one more than the input number.

  Each number in range is cumulatively multiplied in a variable f which is initialised to 1

Code:

```
def factorial(number):
    if number < 0:
        print('Invalid entry! Cannot find factorial of a negative number')
        return -1
    if number == 1 or number == 0:
        return 1
    else:
        return number * factorial(number - 1)

def factorial_iteration(number):
    if number < 0:
        print('Invalid entry! Cannot find factorial of a negative number')
        return -1
    fact = 1
    while(number > 0):
        fact = fact * number
        number = number - 1
    return fact

if __name__ == '__main__':
    userInput = 5
    print('Factorial using Recursion of', userInput, 'is:', factorial(userInput))
    print('Factorial using Iteration of', userInput, 'is:', factorial_iteration(userInput))
```

Output:

```
Factorial using Recursion of 5 is: 120
Factorial using Iteration of 5 is: 120
>>>
```

Practical 4

Aim: Perform Queues operations using Circular Array implementation.

Theory:

- Queue is an abstract data structure, somewhat similar to Stacks. Unlike stacks, a queue is open at both its ends. One end is always used to insert data (enqueue) and the other is used to remove data (dequeue). Queue follows First-In-First-Out methodology, i.e., the data item stored first will be accessed first.

- Basic Operations:
  Queue operations may involve initializing or defining the queue, utilizing it, and then completely erasing it from the memory. Here we shall try to understand the basic operations associated with queues −
  enqueue() − add (store) an item to the queue.
  dequeue() − remove (access) an item from the queue
  Few more functions are required to make the above-mentioned queue operation efficient. These are −
  peek() − Gets the element at the front of the queue without removing it.
  isfull() − Checks if the queue is full.
  isempty() − Checks if the queue is empty.

- Operations On A Circular Queue:
  Enqueue- adding an element in the queue if there is space in the queue.
  Dequeue- Removing elements from a queue if there are any elements in the queue
  Front- get the first item from the queue.

Rear- get the last item from the queue.
isEmpty/isFull- checks if the queue is empty or full.

Code:

```
class ArrayQ:
    def_capacity = 10

    def __init__(self):
        self._data = [None] * ArrayQ.def_capacity
        self._size = 0
        self._front = 0
        self._back = 0

    def __len__(self):
        return self._size

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

    def first(self):
        if self.is_empty():
            raise Empty('Queue is empty')
        return self._data[self._front]
```

```
    def dequeueStart(self):
        if self.is_empty():
            raise Empty('Queue is empty')
        answer = self._data[self._front]
        self._data[self._front] = None
        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):
        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
        self._front = self._front
        self._size -= 1
        self._back = (self._front + self._size - 1) % len(self._data)
        return answer
```

```
    def enqueueEnd(self, e):
        if self._size == len(self._data):
            self._resize(2 * len(self.data))
        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):
        if self._size == len(self._data):
            self._resize(2 * len(self._data))
        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):
        old = self._data
        self._data = [None] * cap
        walk = self._front
        for k in range(self._size):
            self._data[k] = old[walk]
            walk = (1 + walk) % len(old)
        self._front = 0
        self._back = (self._front + self._size - 1) % len(self._data)

queue = ArrayQ()
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]}")
```

Output:

```
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:

- Searching is the process of finding a given value position in a list of values. It decides whether a search key is present in the data or not. It is the algorithmic process of finding a particular item in a collection of items. It can be done on internal data structure or on external data structure.

- Searching Techniques
  To search an element in a given array, it can be done in following ways:
  1. Sequential Search
  2. Binary Search
  Linear search is a very simple search algorithm. In this type of search, a sequential search is made over all items one by one. Every item is checked and if a match is found then that particular item is returned, otherwise the search continues till the end of the data collection
  Algorithm
  Linear Search (Array A, Value x)

  Binary search is a fast search algorithm with run-time complexity of O(log n). This search algorithm works on the principle of divide and conquer. For this algorithm to work properly, the data collection should be in the sorted form.

  Binary search looks for a particular item by comparing the

middle most item of the collection. If a match occurs, then the index of item is returned. If the middle item is greater than the item, then the item is searched in the sub-array to the left of the middle item. Otherwise, the item is searched for in the sub-array to the right of the middle item. This process continues on the sub-array as well until the size of the subarray reduces to zero.

Code:

```
def linear_search(lst, element):
    for i in lst:
        if i == element:
            return lst.index(i)
    return -1

def binary_search(lst, element, start, end):
    mid = (start + end) // 2
    if element == lst[mid]:
        return mid
    if element < lst[mid]:
        return binary_search(lst, element, start, mid-1)
    else:
        return binary_search(lst, element, mid+1, end)

lst = [1,2,3,4,5,6,7,8,9]
element = 8
print("Select the method you want to use:")
print("1. Linear search")
print("2. Binary search")
option = int(input("Enter the option"))

if option == 1:
    print(linear_search(lst, element))
elif option == 2:
    print(binary_search(lst, element, 0, len(lst)))
```

Output:

```
Select the method you want to use:
1. Linear search
2. Binary search
Enter the option1
7
>>>
```

```
Select the method you want to use:
1. Linear search
2. Binary search
Enter the option2
7
>>>
```

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:

- Sorting refers to arranging data in a particular format. Sorting algorithm specifies the way to arrange data in a particular order. The importance of sorting lies in the fact that data searching can be optimized to a very high level, if data is stored in a sorted manner.

- Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order. This algorithm is not suitable for large data sets as its average and worst case complexity are of O(n2) where n is the number of items.

- Insertion sort is an in-place comparison-based sorting algorithm. Here, a sub-list is maintained which is always sorted. For example, the lower part of an array is maintained to be sorted. An element which is to be inserted in this sorted sub-list, has to find its appropriate place and then it has to be inserted there. Hence the name, insertion sort.

  The array is searched sequentially and unsorted items are moved and inserted into the sorted sub-list (in the same array). This algorithm is not suitable for large data sets as its average and worst case complexity are of O(n2), where n is

the number of items.

- Selection sort is a simple sorting algorithm. This sorting algorithm is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is the entire list.

  The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array. This process continues moving unsorted array boundary by one element to the right.This algorithm is not suitable for large data sets as its average and worst case complexities are of O(n2), where n is the number of items.

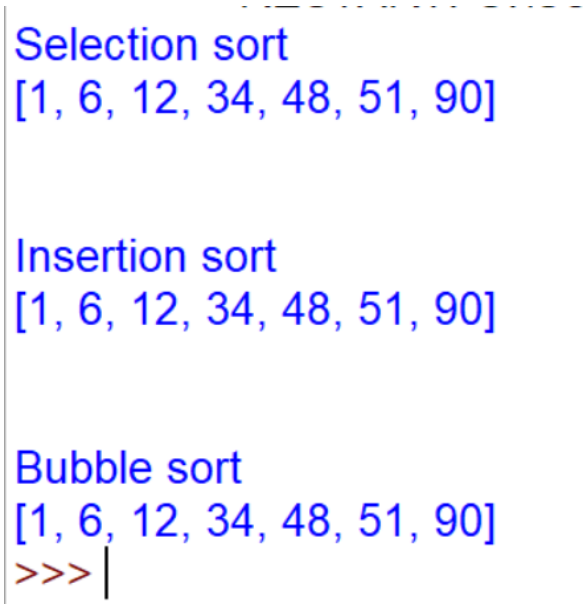
Code:

```
list_student_rolls = [1,90,51,12,48,6,34]

print("Selection sort")

for i in range(len(list_student_rolls)):
    min_val_index = i
    for j in range(i+1,len(list_student_rolls)):
        if list_student_rolls[min_val_index] > list_student_rolls[j]:
            min_val_index = j

    list_student_rolls[i], list_student_rolls[min_val_index] = list_student_rolls[min_val_index],list_student_rolls[i]

print(list_student_rolls)
```

```
print("\n\nInsertion sort")

for i in range(j, len(list_student_rolls)):

    value = list_student_rolls[i]

    j = i-1

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

        list_student_rolls[j+1] = list_student_rolls[j]
        j -= 1

    list_student_rolls[j+1] = value

print(list_student_rolls)
```

```
print("\n\nBubble sort")

list_of_number = [1,90,51,12,48,6,34]

def bubbleSort(list_of_number):

    for i in range(len(list_of_number) - 1):

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

            if list_of_number[j] > list_of_number[j+1]:
                list_of_number[j], list_of_number[j+1] = list_of_number[j+1], list_of_number[j]

bubbleSort(list_of_number)
print(list_of_number)
```

Output:


Selection sort
[1, 6, 12, 34, 48, 51, 90]
Insertion sort
[1, 6, 12, 34, 48, 51, 90]
Bubble sort
[1, 6, 12, 34, 48, 51, 90]
>>>

Practical 7

Implement the following for Hashing

Practical 7 A

Aim: Write a program to implement the collision technique.

Theory:

- When one or more hash values compete with a single hash table slot, collisions occur. To resolve this, the next available empty slot is assigned to the current hash value. The most common methods are open addressing, chaining, probabilistic hashing, perfect hashing and coalesced hashing technique.

a) Chaining:

This technique implements a linked list and is the most popular collision resolution techniques

b) Open Addressing:

This technique depends on space usage and can be done with linear or quadratic probing techniques. As the name says, this technique tries to find an available slot to store the record. It can be done in one of the 3 ways –

Linear probing – Here, the next probe interval is fixed to 1. It supports best caching but miserably fails at clustering.

Quadratic probing – the probe distance is calculated based on the quadratic equation. This is considerably a better option as it balances clustering and caching.

Double hashing – Here, the probing interval is fixed for each record by a second hashing function. This technique has poor cache performance although it does not have any clustering

issues.
Below are some of the hashing techniques that can help in resolving collision.

c) Probabilistic hashing:
This is memory based hashing that implements caching. When collision occurs, either the old record is replaced by the new or the new record may be dropped. Although this scenario has a risk of losing data, it is still preferred due to its ease of implementation and high performance.

d) Perfect hashing:
When the slots are uniquely mapped, there is very less chances of collision. However, it can be done where there is a lot of spare memory.

e) Coalesced hashing:
This technique is a combo of open address and chaining methods. A chain of items is stored in the table when there is a collision. The next available table space is used to store the items to prevent collision.

Code:

```
size_list = 6

def hash_f(value):
    global size_list
    return value%7

def map_hash(hash_return_value):
    return hash_return_value


def create_hash_table(list_values,main_list):
    for value in list_values:
        hash_return_value = hash_f(value)
        list_index = map_hash(hash_return_value)
        if list_values[list_index]:
            print("Collision is detected")
        else:
            list_values[list_index] = value

list_values = [1,3,4,8,3]

main_list = [None for x in range(size_list)]
print(main_list)
create_hash_table(list_values,main_list)
print(main_list)
```

Output:

```
[None, None, None, None, None, None]
Collision is detected
Collision is detected
Collision is detected
Collision is detected
Collision is detected
[None, None, None, None, None, None]
>>>
```

Practical 7 B

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

Theory:

- Linear probing is a scheme in computer programming for resolving collisions in hash tables, data structures for maintaining a collection of key–value pairs and looking up the value associated with a given key.

- Challenges in Linear Probing :
  Primary Clustering: One of the problems with linear probing is Primary clustering, many consecutive elements form groups and it starts taking time to find a free slot or to search an element.

  Secondary Clustering: Secondary clustering is less severe, two records do only have the same collision chain(Probe Sequence) if their initial position is the same.

  b) Quadratic Probing We look for i2‘th slot in i’th iteration.

  c) Double Hashing We use another hash function hash2(x) and look for i*hash2(x) slot in i’th rotation.

  Comparison of above three:
  Linear probing has the best cache performance but suffers from clustering. One more advantage of Linear probing is easy to compute.
  Quadratic probing lies between the two in terms of cache

performance and clustering.
Double hashing has poor cache performance but no clustering.
Double hashing requires more computation time as two hash functions need to be computed.

Code:

```
class Hash_f:
    def __init__(self, keys, low_range, high_range):
        self.value = self.hashfun(keys,low_range, high_range)

    def get_key_value(self):
        return self.value

    def hashfun(self,keys,low_range, high_range):
        if low_range == 0 and high_range > 0:
            return keys%(high_range)

if __name__ == '__main__':
    linear_probing = True
    list_of_keys = [33,4,51,14]
    list_of_list_index = [None,None,None,None]
    print("Before : " + str(list_of_list_index))

    for value in list_of_keys:
        list_index = Hash_f(value,0,len(list_of_keys)).get_key_value()
        print("hash value for " + str(value) + " is :" + str(list_index))
        if list_of_list_index[list_index]:
            print("Collission detected for " + str(value))
            if linear_probing:
                old_list_index = list_index
                if list_index == len(list_of_list_index)-1:
                    list_index = 0
                else:
                    list_index += 1
                list_full = False
```

```
            while list_of_list_index[list_index]:
                if list_index == old_list_index:
                    list_full = True
                    break
                if list_index+1 == len(list_of_list_index):
                    list_index = 0
                else:
                    list_index += 1
            if list_full:
                print("List was full . Could not save")
            else:
                list_of_list_index[list_index] = value
        else:
            list_of_list_index[list_index] = value

    print("After: " + str(list_of_list_index))
```

Output:

```
Before : [None, None, None, None]
hash value for 33 is :1
hash value for 4 is :0
hash value for 51 is :3
hash value for 14 is :2
After: [4, 33, 14, 51]
>>>
```

Practical 8

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

Theory:

- Unlike linear data structures (Array, Linked List, Queues, Stacks, etc) which have only one logical way to traverse them, trees can be traversed in different ways. Following are the generally used ways for traversing trees.
  Depth First Traversals:
  (a) Inorder (Left, Root, Right) : 4 2 5 1 3
  (b) Preorder (Root, Left, Right) : 1 2 4 5 3
  (c) Postorder (Left, Right, Root) : 4 5 2 3 1

  Breadth First or Level Order Traversal : 1 2 3 4 5

- Uses of Inorder:
  In case of binary search trees (BST), Inorder traversal gives nodes in non-decreasing order. To get nodes of BST in non-increasing order, a variation of Inorder traversal where Inorder traversal s reversed can be used.

- Uses of Preorder:
  Preorder traversal is used to create a copy of the tree. Preorder traversal is also used to get prefix expression on of an expression tree.

- Uses of Postorder:
  Postorder traversal is used to delete the tree. Please see the question for deletion of tree for details. Postorder traversal is also useful to get the postfix expression of an expression tree.

Code:

```
class Node_order:
    def __init__(self,key):
        self.left = None
        self.right = None
        self.val = key

def Inorder(root):
    if root:
        Inorder(root.left)
        print(root.val),
        Inorder(root.right)

def Postorder(root):
    if root:
        Postorder(root.left)
        Postorder(root.right)
        print(root.val),

def Preorder(root):
    if root:
        print(root.val),
        Preorder(root.left)
        Preorder(root.right)


root = Node_order(1)
root.left = Node_order(2)
root.right = Node_order(3)
root.left.left = Node_order(4)
root.left.right = Node_order(5)
print ("Preorder traversal is",Preorder(root))
print ("Inorder traversal is",Inorder(root))
print ("Postorder traversal is",Postorder(root))
```

Output:

```
1
2
4
5
3
Preorder traversal is None
4
2
5
1
3
Inorder traversal is None
4
5
2
3
1
Postorder traversal is None
>>>
```