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.~~ Satyam Thaker

Roll No: 386 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: 386**

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.	

PRACTICAL-1A

GITHUB LINK: https://github.com/satyamthaker/Data-Structure

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

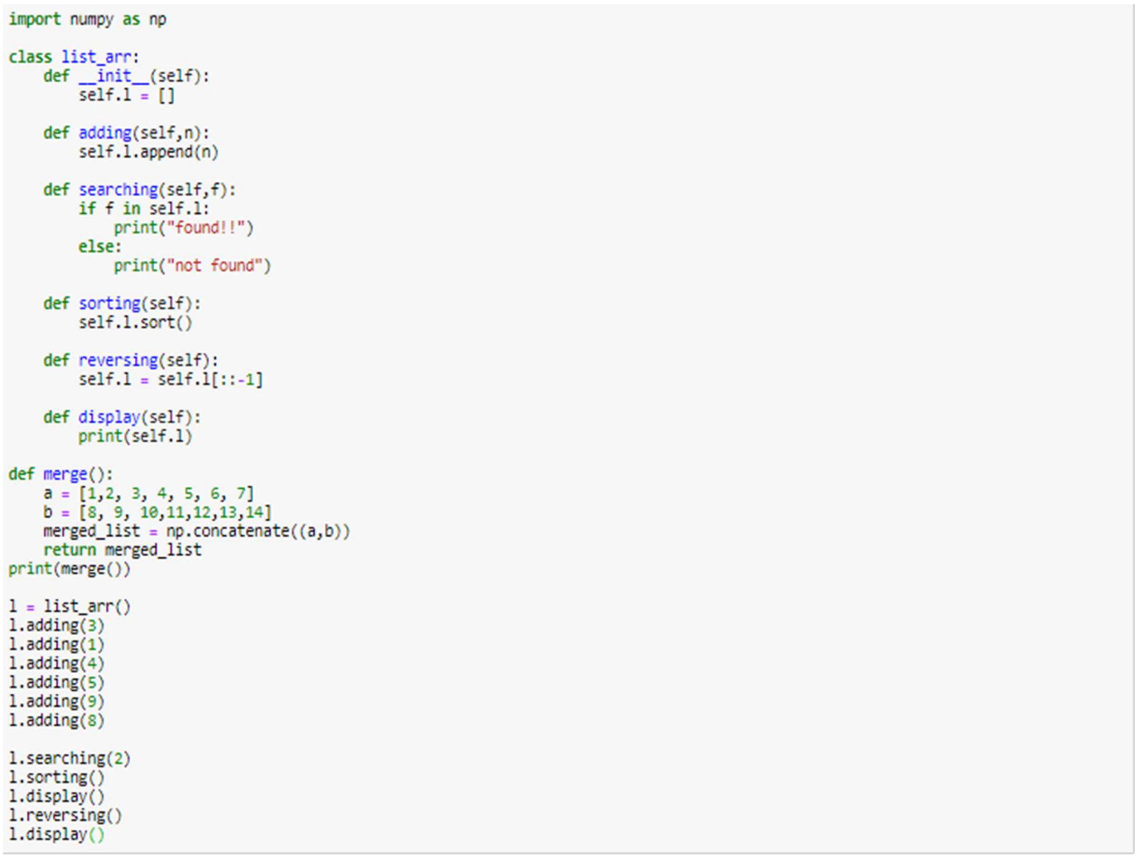
- Storing Data in Arrays.
- Assigning values to an element in an array is similar to assigning values to scalar variables.
- Simply reference an individual element of an array using the array name and the index inside parentheses, then use the assignment operator (=) followed by a value.
- Following are the basic operations supported by an array.
    1. Traverse − print all the array elements one by one.
    2. Insertion − Adds an element at the given index.
    3. Deletion − Deletes an element at the given index.
    4. Search − Searches an element using the given index or by the value.

Searching is a very basic necessity when you store data in different data structures. The simplest approach is to go across every element in the data structure and match it with the value you are searching for.

The sort () function returns a sorted list of the specified iterable object.
You can specify ascending or descending order. Strings are sorted alphabetically, and numbers are sorted numerically.

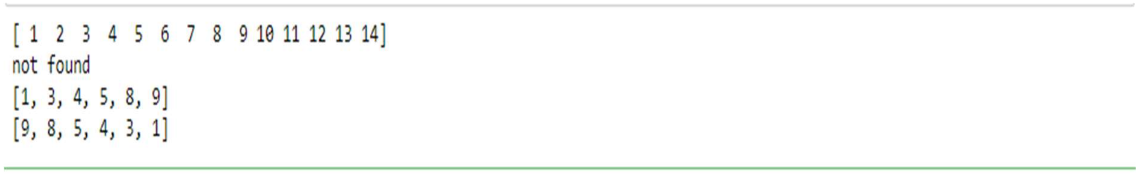
Merge: First we have to copy all the elements of the first list into a new list. Using a for loop we can append every element of the second list in the new list.
Reverse: First we have to create a new list. Using a reversed for loop we can append all the elements of the original list into the new list in reverse order.

Code:



```
import numpy as np

class list_arr:
    def __init__(self):
        self.l = []

    def adding(self,n):
        self.l.append(n)

    def searching(self,f):
        if f in self.l:
            print("found!!")
        else:
            print("not found")

    def sorting(self):
        self.l.sort()

    def reversing(self):
        self.l = self.l[::-1]

    def display(self):
        print(self.l)

def merge():
    a = [1,2, 3, 4, 5, 6, 7]
    b = [8, 9, 10,11,12,13,14]
    merged_list = np.concatenate((a,b))
    return merged_list
print(merge())

l = list_arr()
l.adding(3)
l.adding(1)
l.adding(4)
l.adding(5)
l.adding(9)
l.adding(8)

l.searching(2)
l.sorting()
l.display()
l.reversing()
l.display()
```

Output:



```
[ 1  2  3  4  5  6  7  8  9 10 11 12 13 14]
not found
[1, 3, 4, 5, 8, 9]
[9, 8, 5, 4, 3, 1]
```

PRACTICAL-1B

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

THEORY

- **Matrix:** In mathematics, a matrix (plural matrices) is a rectangular array or table of numbers, symbols, or expressions, arranged in rows and columns. For example, the dimension of the matrix below is 2 × 3 (read "two by three"), because there are two rows and three columns. This is represented in the form of nested lists in Python.

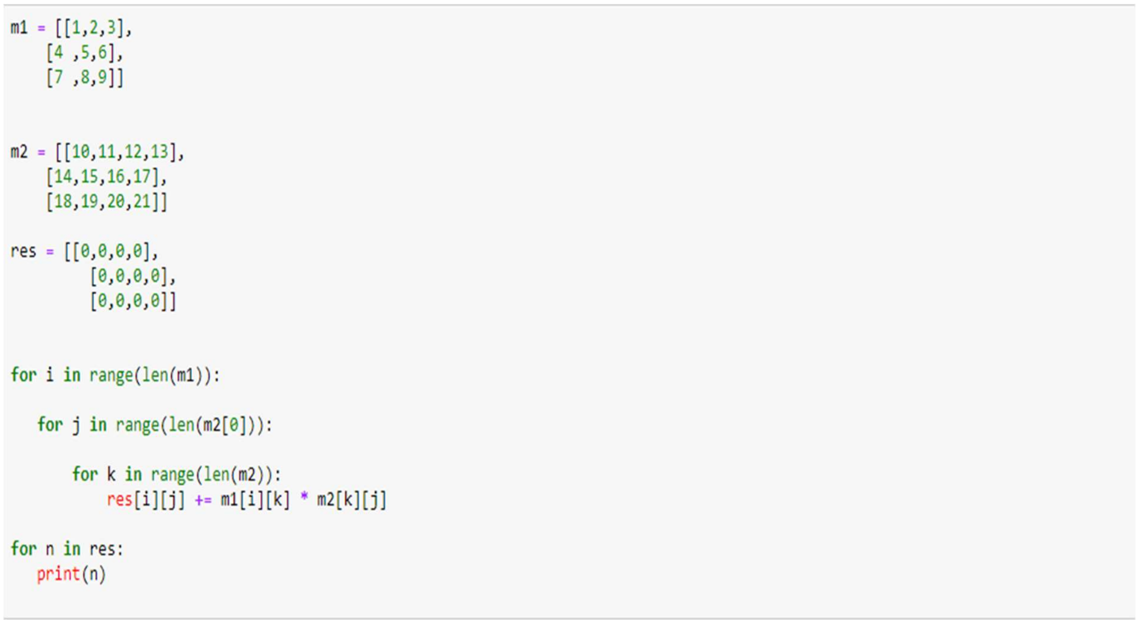
- **Addition:** Matrix addition is the operation of adding two matrices by adding the corresponding entries together. Two matrices need to have the same dimensions in order to be eligible for addition. We have to create a new empty matrix (nested list) and use 2 nested for loops to add the elements with the corresponding index positions and write it to the new matrix.

- **Subtraction:** Matrix addition is the operation of subtracting two matrices by subtracting the corresponding entries together .Two matrices need to have the same dimensions in order to be eligible for subtraction .We have to create a new empty matrix(nested list) and use 2 nested for loops to subtract the elements with the corresponding index positions and write it to the new matrix.

- **Multiplication:** Matrix multiplication is a binary operation that produces a matrix from two matrices. For matrix multiplication, the number of columns in the first matrix must be equal to the number of rows in the second matrix. The resulting matrix, known as the matrix product, has the number of rows of the first and the number of columns of the second matrix. We have to create a new empty matrix (nested list) and use 3 nested for loops and multiply elements in the appropriate index positions and write it to the new matrix.

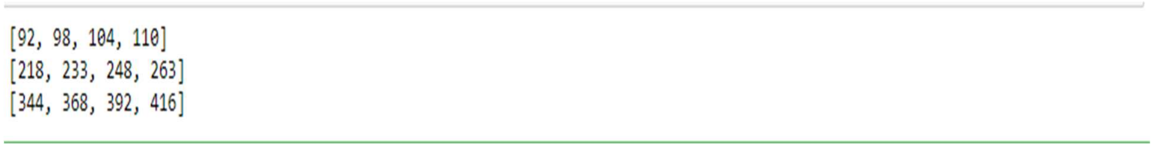
- **Transpose**: The transpose of a matrix is an operator which flips a matrix over its diagonal; that is, it switches the row and column indices of the matrix A by producing another matrix, often denoted by AT (among other notations). We have to create a new empty matrix (nested list) and use 2 nested for loops to swap the row and column of the elements in the appropriate index positions and write it to the new matrix.

Code:



```
m1 = [[1,2,3],
    [4 ,5,6],
    [7 ,8,9]]


m2 = [[10,11,12,13],
    [14,15,16,17],
    [18,19,20,21]]

res = [[0,0,0,0],
        [0,0,0,0],
        [0,0,0,0]]


for i in range(len(m1)):

   for j in range(len(m2[0])):

       for k in range(len(m2)):
           res[i][j] += m1[i][k] * m2[k][j]

for n in res:
   print(n)
```

Output:



```
[92, 98, 104, 110]
[218, 233, 248, 263]
[344, 368, 392, 416]
```

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 implement the concept of linked lists using the concept of nodes as discussed in the previous chapter. We have already seen how we create a node class and how to traverse the elements of a node. In this chapter 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.

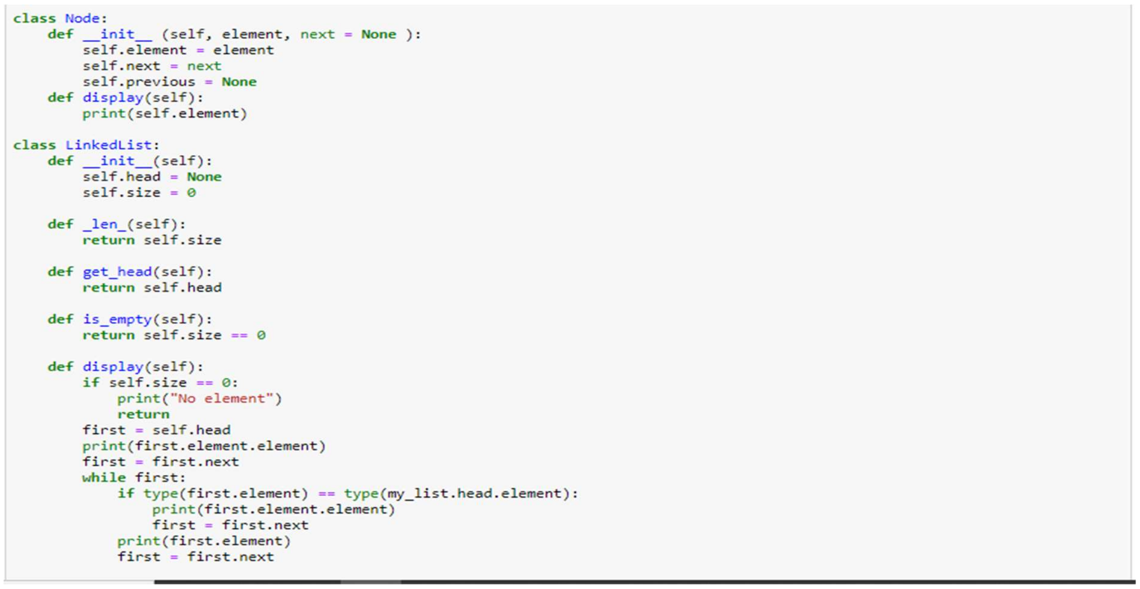
- Insertion in a Linked List:
- Inserting element in the linked list involves reassigning the pointers from the existing nodes to the newly inserted node. Depending on whether the new data element is getting inserted at the beginning or at the middle or at the end of the linked list.

- Deleting an Item form a Linked List:
- We can remove an existing node using the key for that node. In the below program we locate the previous node of the node which is to be deleted. Then point the next pointer of this node to the next node of the node to be deleted.

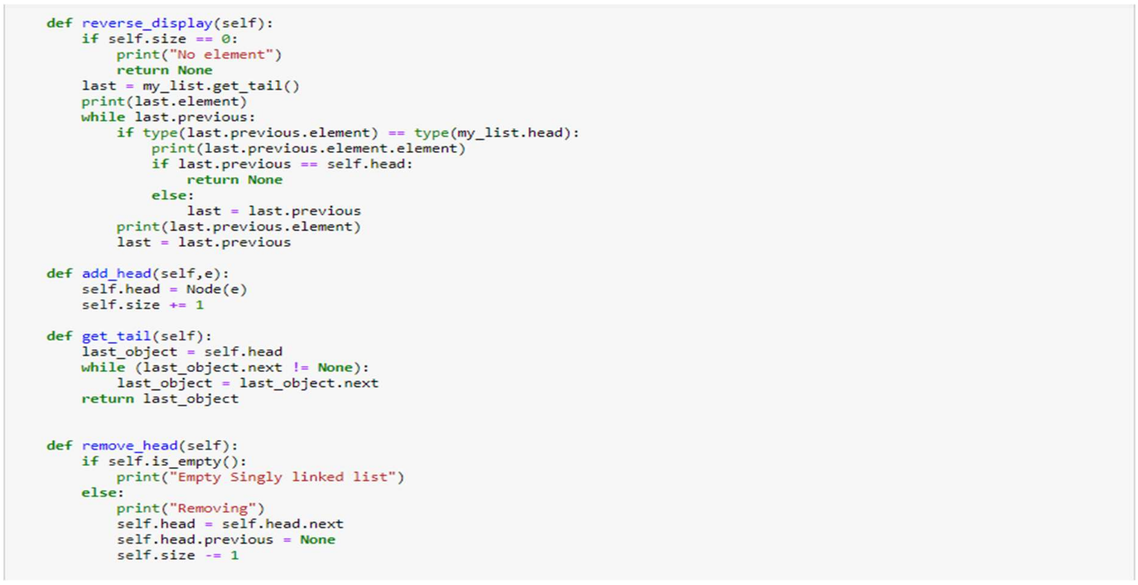
- Searching in linked list:
- Searching is performed in order to find the location of a particular element in the list. Searching any element in the list needs traversing through the list and make the comparison of every element of the list with the specified element. If the element is matched with any of the list element then the location of the element is returned from the function.

- Reversing a Linked List:
- To reverse a LinkedList recursively we need to divide the LinkedList into two parts: head and remaining. Head points to the first element initially. Remaining points to the next element from the head. We traverse the LinkedList recursively until the second last element.

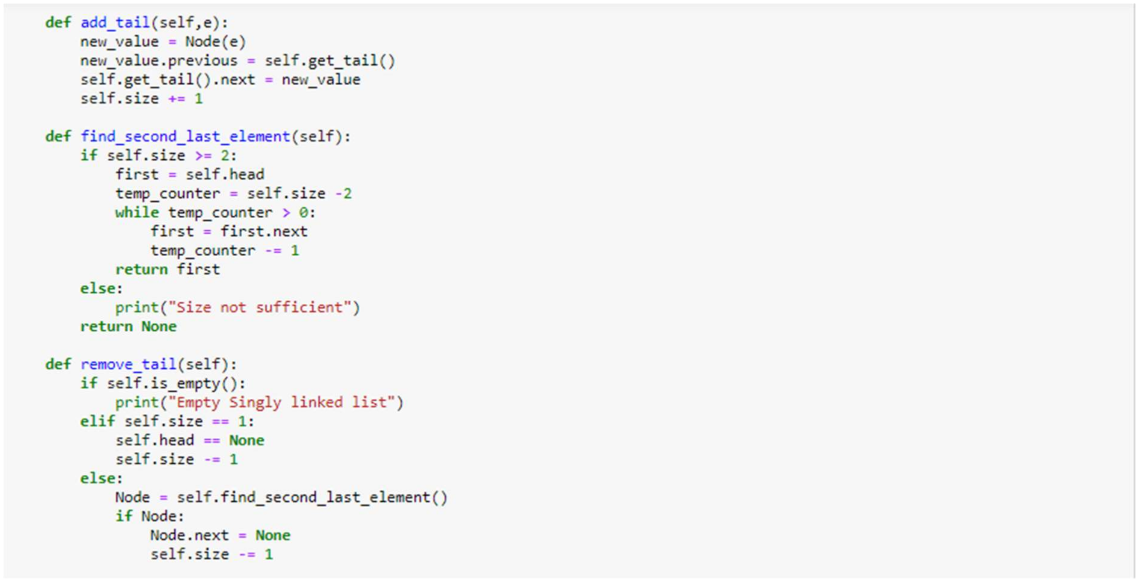
- Concatenating Linked Lists:
- Concatenate the two lists by traversing the first list until we reach it’s a tail node and then point the next of the tail node to the head node of the second list. Store this concatenated list in the first list



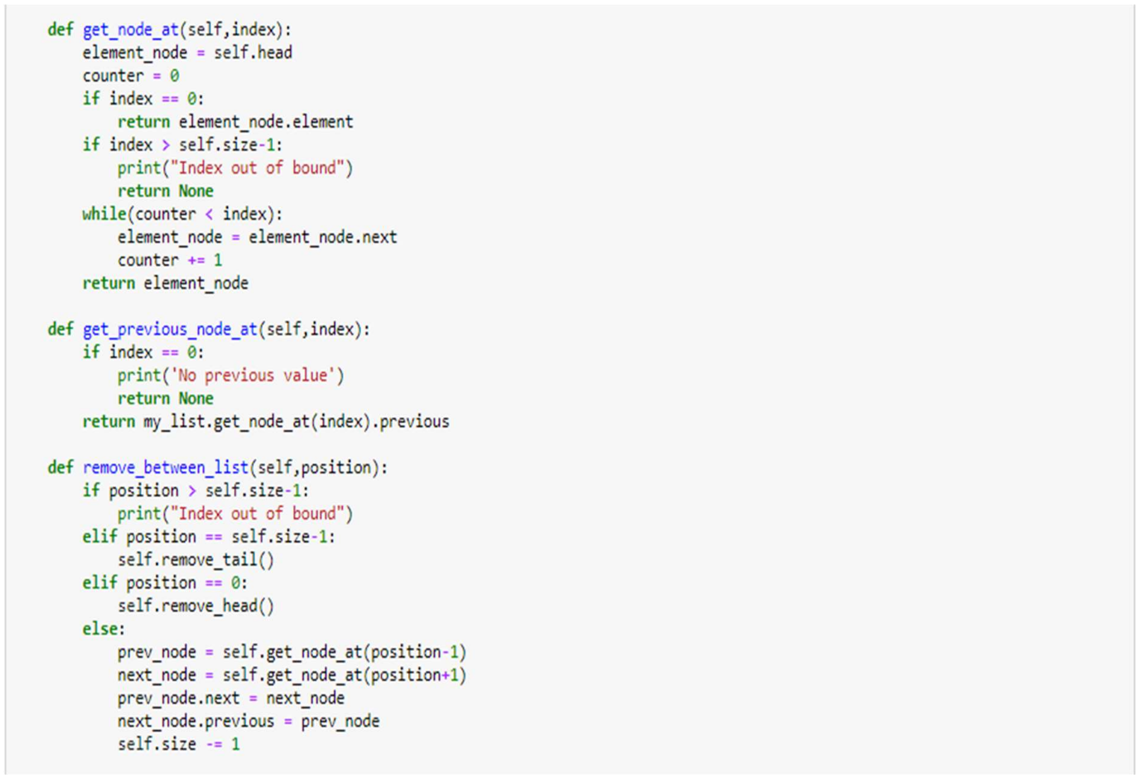
```
class Node:
    def __init__ (self, element, next = None ):
        self.element = element
        self.next = next
        self.previous = None
    def display(self):
        print(self.element)

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
```



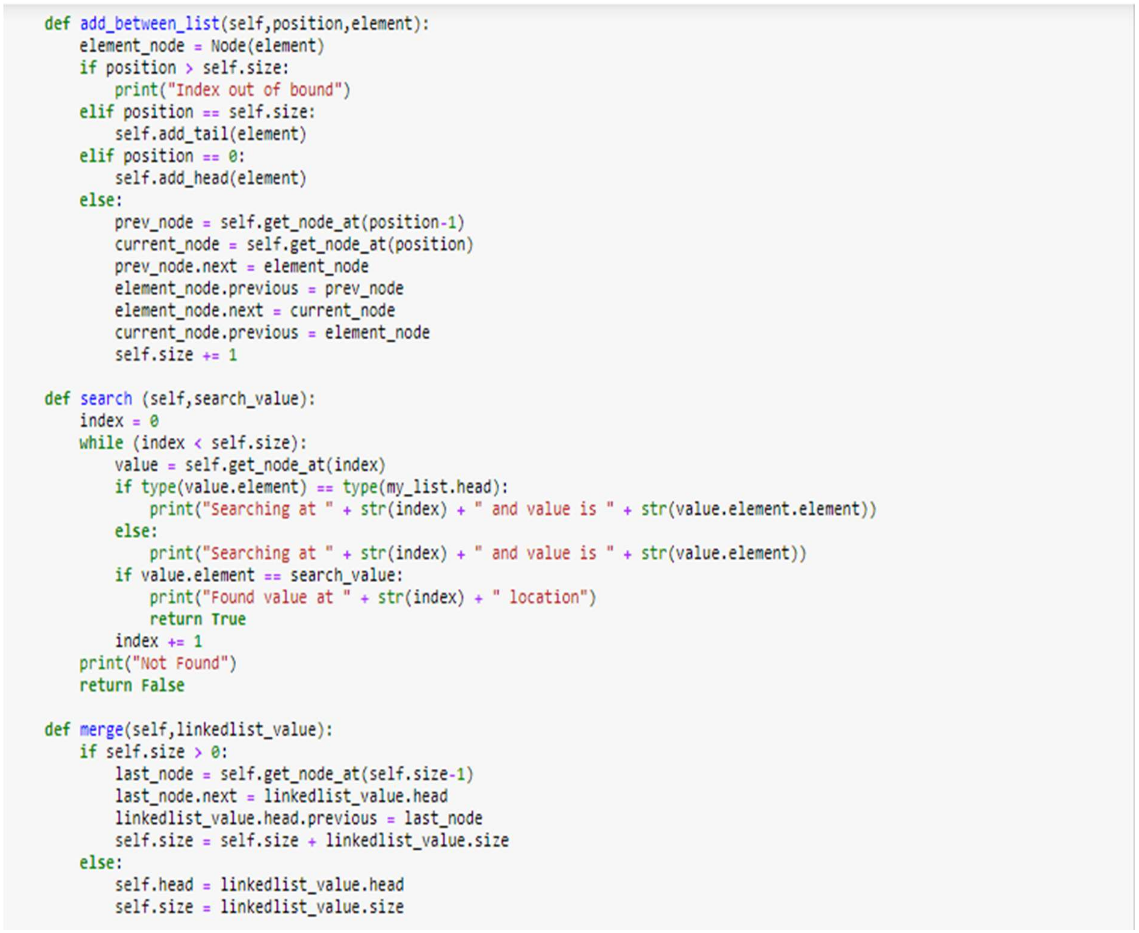
```
    def reverse_display(self):
        if self.size == 0:
            print("No element")
            return None
        last = my_list.get_tail()
        print(last.element)
        while last.previous:
            if type(last.previous.element) == type(my_list.head):
                print(last.previous.element.element)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



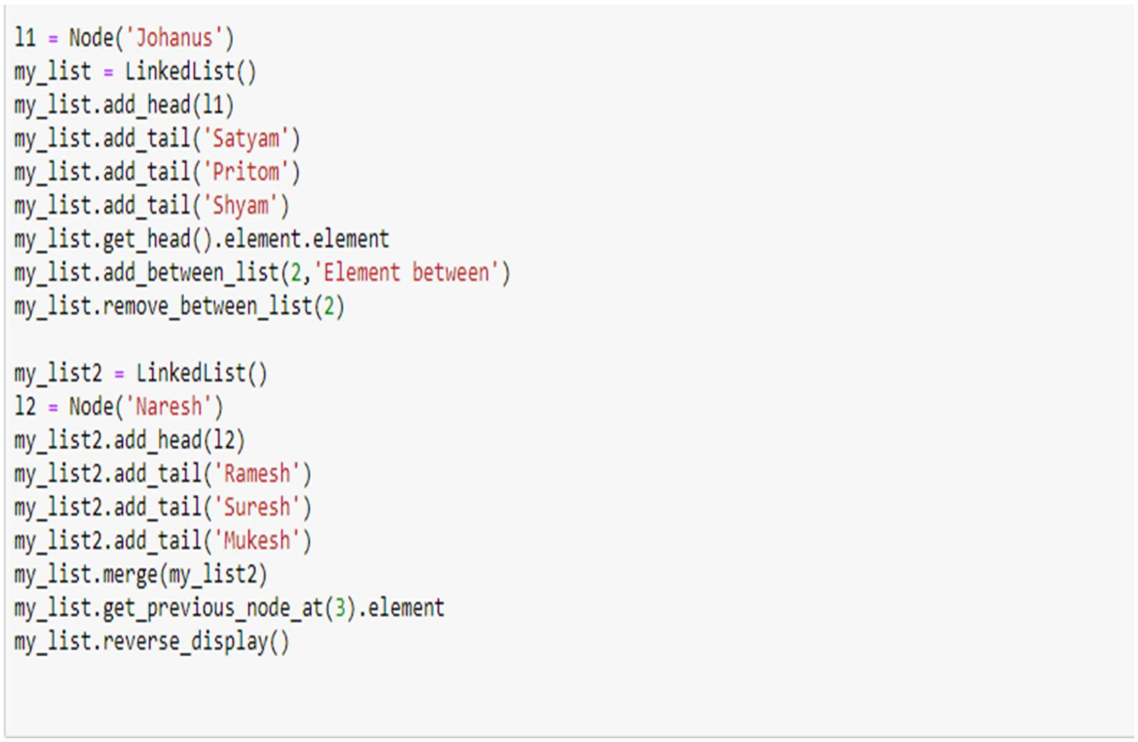
```
    def add_tail(self,e):
        new_value = Node(e)
        new_value.previous = self.get_tail()
        self.get_tail().next = new_value
        self.size += 1

    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
```



```python
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
```

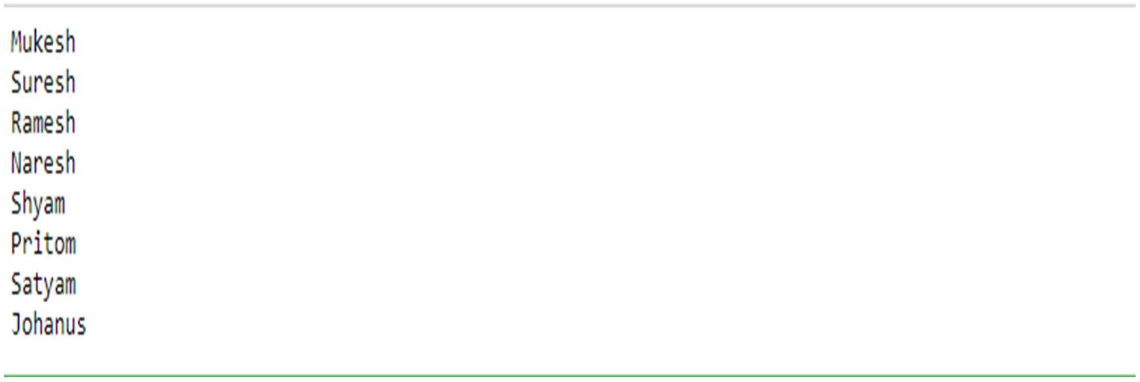


```python
def add_between_list(self,position,element):
    element_node = Node(element)
    if position > self.size:
        print("Index out of bound")
    elif position == self.size:
        self.add_tail(element)
    elif position == 0:
        self.add_head(element)
    else:
        prev_node = 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('Johanus')
my_list = LinkedList()
my_list.add_head(l1)
my_list.add_tail('Satyam')
my_list.add_tail('Pritom')
my_list.add_tail('Shyam')
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('Naresh')
my_list2.add_head(l2)
my_list2.add_tail('Ramesh')
my_list2.add_tail('Suresh')
my_list2.add_tail('Mukesh')
my_list.merge(my_list2)
my_list.get_previous_node_at(3).element
my_list.reverse_display()
```

Output:



```
Mukesh
Suresh
Ramesh
Naresh
Shyam
Pritom
Satyam
Johanus
```

PRACTICAL-3A

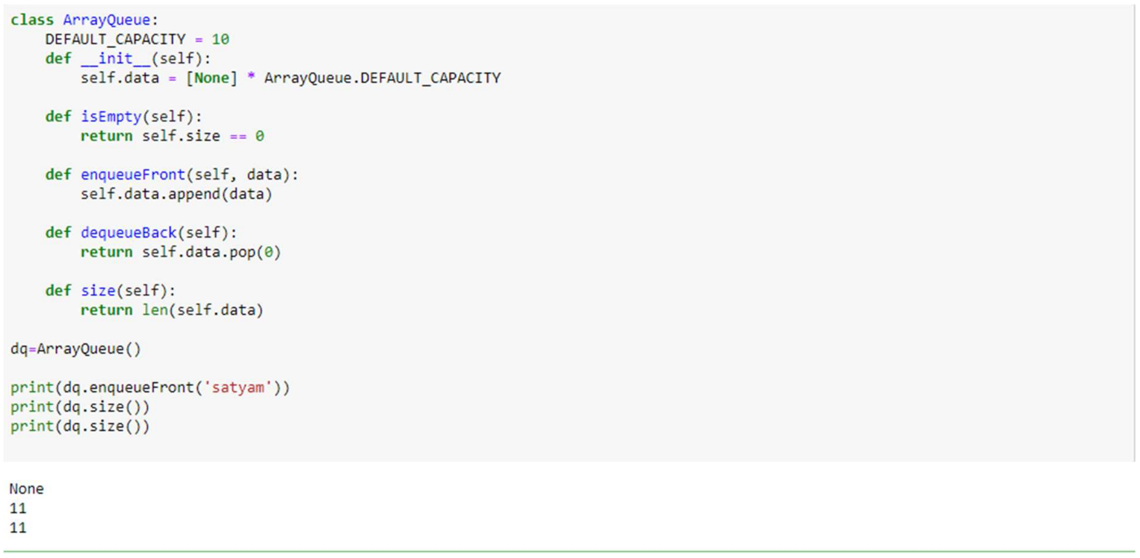
AIM: Perform Stack operations using Array implementation. b.

THEORY

Stack:

- A stack is a collection of objects that are inserted and removed according to the last-in, first-out (LIFO) principle.
- A user may insert objects into a stack at any time, but may only access or remove the most recently inserted object that remains (at the so-called "top" of the stack).
- We can implement a stack quite easily by storing its elements in a Python list.
- The list class already supports adding an element to the end with the append method, and removing the last element with the pop method, so it is natural to align the top of the stack at the end of the list.
- Stack is an abstract data type (ADT) such that an instance S supports the following two methods:
    1. S.push(e): Add element e to the top of stack S.
    2. S.pop(): Remove and return the top element from the stack S;
    3. an error occurs if the stack is empty.

Code:



```python
class ArrayQueue:
    DEFAULT_CAPACITY = 10
    def __init__(self):
        self.data = [None] * ArrayQueue.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=ArrayQueue()

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

```
None
11
11
```

PRACTICAL-3B

AIM: Implement Tower of Hanoi.

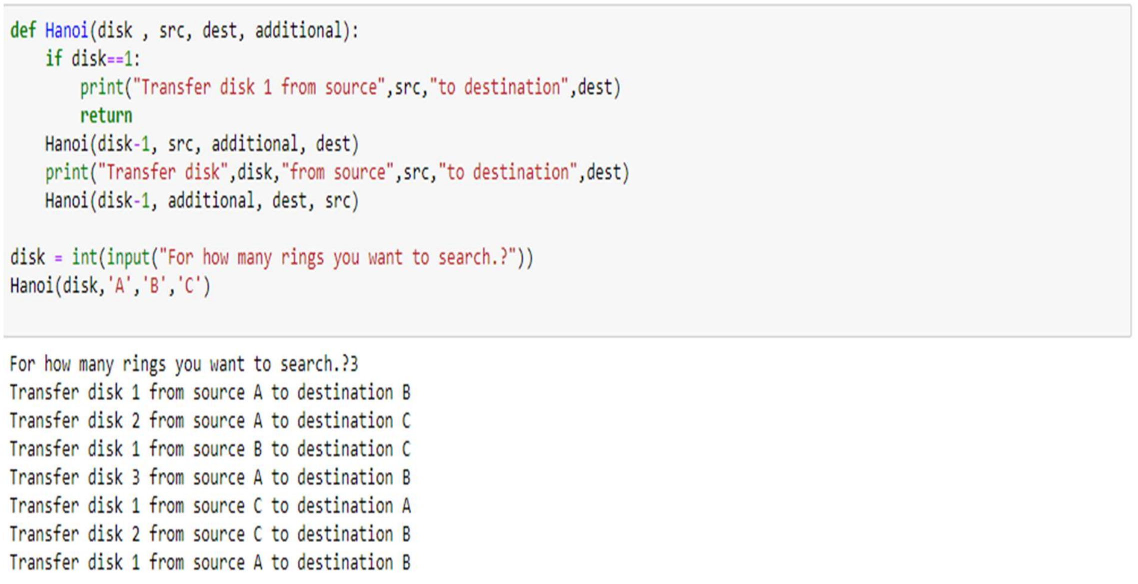
THEORY

Tower of Hanoi is a mathematical puzzle where we have three rods and n disks. The objective of the puzzle is to move the entire stack to another rod, obeying the following simple rules:

1) Only one disk can be moved at a time.
2) Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack i.e. a disk can only be moved if it is the uppermost disk on a stack.
3) No disk may be placed on top of a smaller disk.

To write an algorithm for Tower of Hanoi, first we need to learn how to solve this problem with lesser number of disks, say → 1 or 2. We mark three towers with name, source, destination and aux (only to help moving the disks). If we have only one disk, then it can easily be moved from source to destination peg.
If we have 2 disks −
First, we move the smaller (top) disk to aux peg.
Then, we move the larger (bottom) disk to destination peg.
And finally, we move the smaller disk from aux to destination peg.
So now, we are in a position to design an algorithm for Tower of Hanoi with more than two disks. We divide the stack of disks in two parts. The largest disk (nth disk) is in one part and all other (n-1) disks are in the second part.
Our ultimate aim is to move disk n from source to destination and then put all other (n1) disks onto it. We can imagine to apply the same in a recursive way for all given set of disks.
Each peg is a Stack object.

Code:



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

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

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

PRACTICAL-3C

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

THEORY

- Different operations can be performed on the polynomials like addition, subtraction, multiplication, and division.

- A polynomial is an expression within which a finite number of constants and variables are combined using addition, subtraction, multiplication, and exponents.

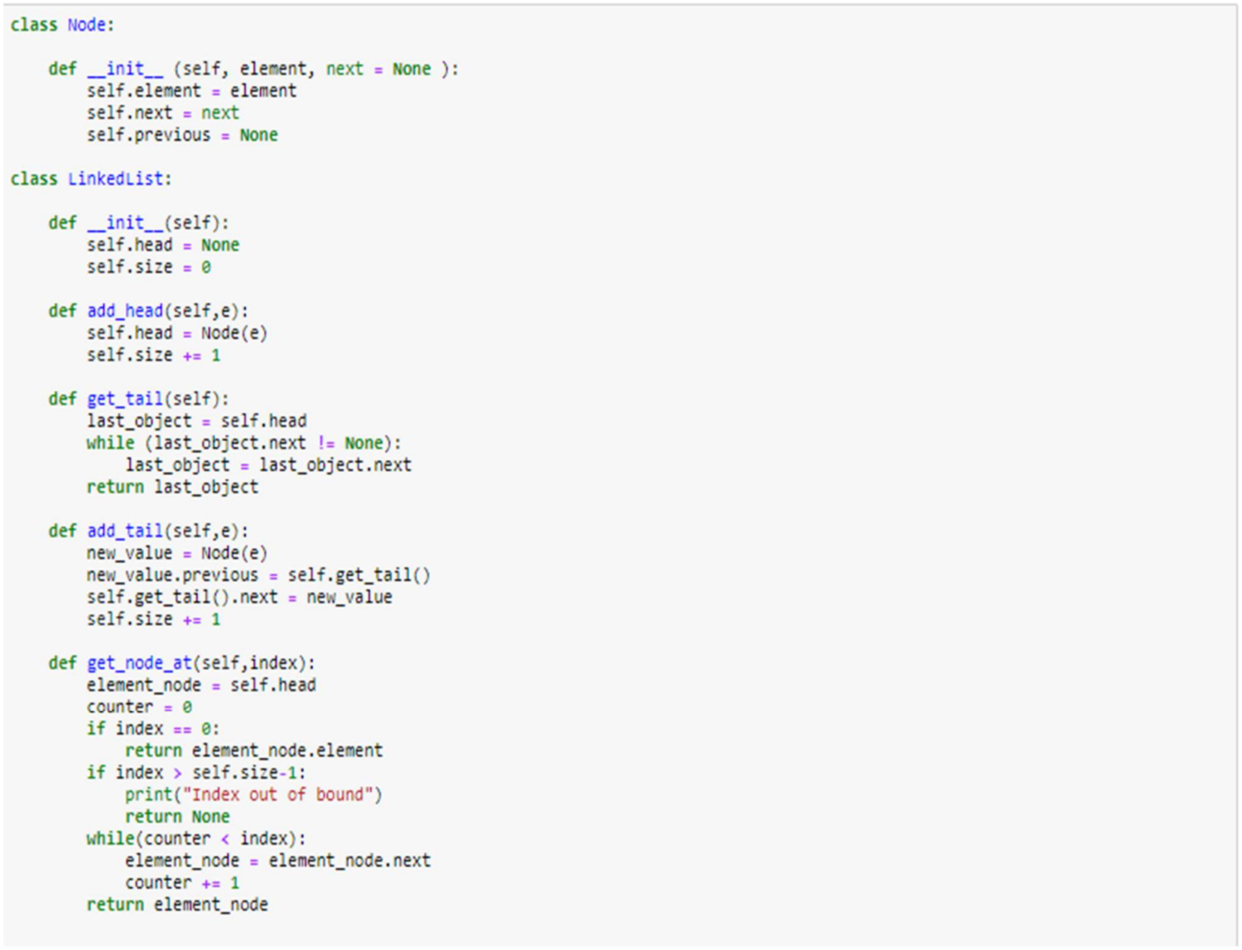
- Adding and subtracting polynomials is just adding and subtracting their like terms. The sum of two monomials is called a binomial and the sum of three monomials is called a trinomial.

- The sum of a finite number of monomials in x is called a polynomial in x. The coefficients of the monomials in a polynomial are called the coefficients of the polynomial.

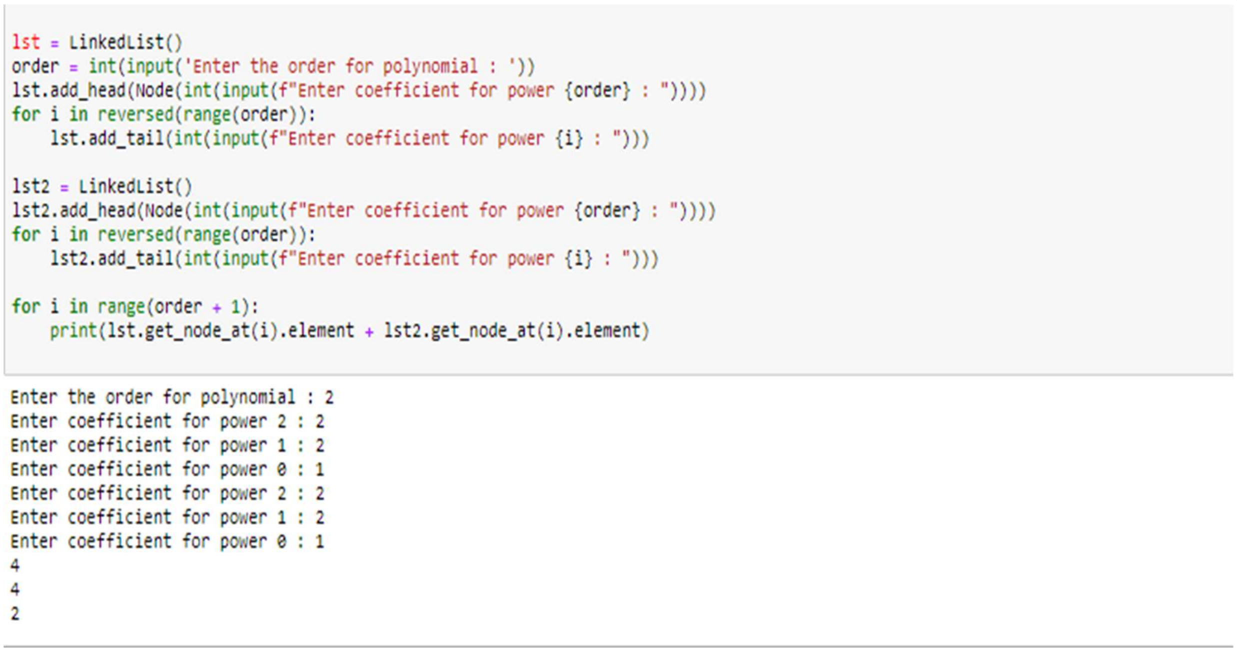
- If all the coefficients of a polynomial are zero, then the polynomial is called the zero polynomial.

- Two polynomials can be added by using arithmetic operator plus (+). Adding polynomials is simply “combining like terms” and then add the like terms.

Code:



```
class Node:

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

class LinkedList:

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

    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 add_tail(self,e):
        new_value = Node(e)
        new_value.previous = self.get_tail()
        self.get_tail().next = new_value
        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
```



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

lst2 = LinkedList()
lst2.add_head(Node(int(input(f"Enter coefficient for power {order} : "))))
for i in reversed(range(order)):
    lst2.add_tail(int(input(f"Enter coefficient for power {i} : ")))

for i in range(order + 1):
    print(lst.get_node_at(i).element + lst2.get_node_at(i).element)
```

```
Enter the order for polynomial : 2
Enter coefficient for power 2 : 2
Enter coefficient for power 1 : 2
Enter coefficient for power 0 : 1
Enter coefficient for power 2 : 2
Enter coefficient for power 1 : 2
Enter coefficient for power 0 : 1
4
4
2
```

PRACTICAL-3D

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

THEORY

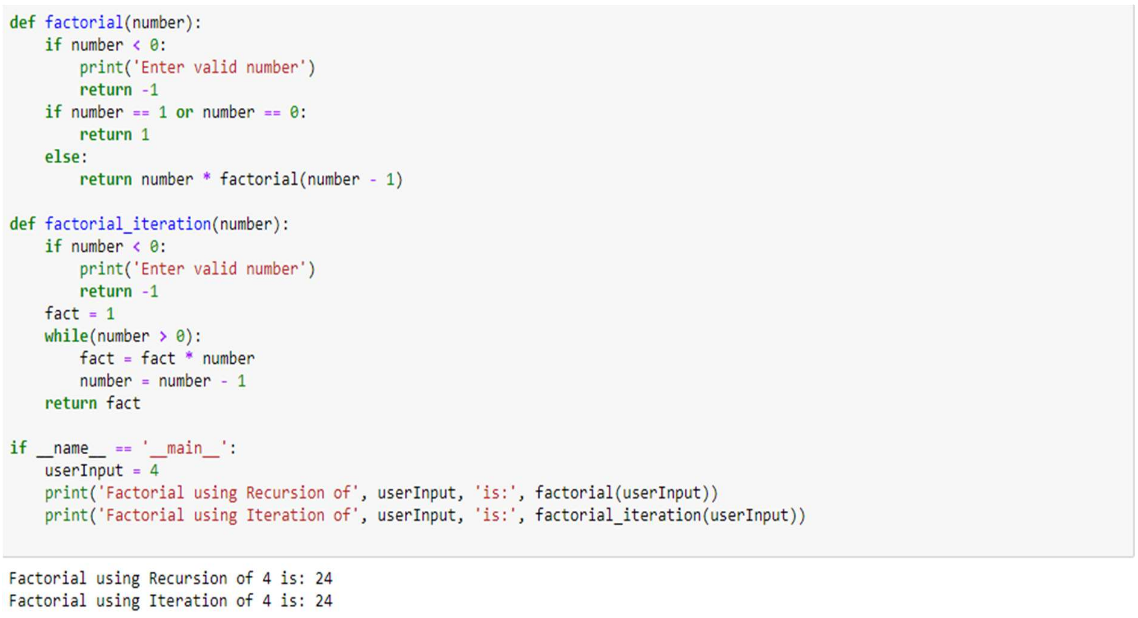
Factorial:

- The factorial of a number is the product of all the integers from 1 to that number.
- For example, the factorial of 6 (denoted as 6!) is 1*2*3*4*5*6 = 720.
- Factorial is not defined for negative numbers and the factorial of zero is one, 0! = 1.
- You can find it using recursion as well as iteration to calculate the factorial of a number.
- Recursion:

  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.
- Iteration:

  Repeating identical or similar tasks without making errors is something that computers do well and people do poorly. Repeated execution of a set of statements is called iteration. Because iteration is so common, Python provides several language features to make it easier.

Code:



```python
def factorial(number):
    if number < 0:
        print('Enter valid 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('Enter valid number')
        return -1
    fact = 1
    while(number > 0):
        fact = fact * number
        number = number - 1
    return fact

if __name__ == '__main__':
    userInput = 4
    print('Factorial using Recursion of', userInput, 'is:', factorial(userInput))
    print('Factorial using Iteration of', userInput, 'is:', factorial_iteration(userInput))
```

```
Factorial using Recursion of 4 is: 24
Factorial using Iteration of 4 is: 24
```

PRACTICAL-4

Aim: Perform Queues operations using Circular Array implementation.

THEORY

Queue:

- The queue abstract data type defines a collection that keeps objects in a sequence, where element access and deletion are restricted to the first element in the queue, and element insertion is restricted to the back of the sequence.

- This restriction enforces the rule that items are inserted and deleted in a queue according to the first-in, first-out (FIFO) principle.

- The queue abstract data type (ADT) supports the following two fundamental methods for a queue

- **Q.enqueue(e): Add element e to the back of queue Q.**
- **Q.dequeue( ): Remove and return the first element from queue Q;**

- an error occurs if the queue is empty. For the stack ADT, we created a very simple adapter class that used a Python list as the underlying storage.

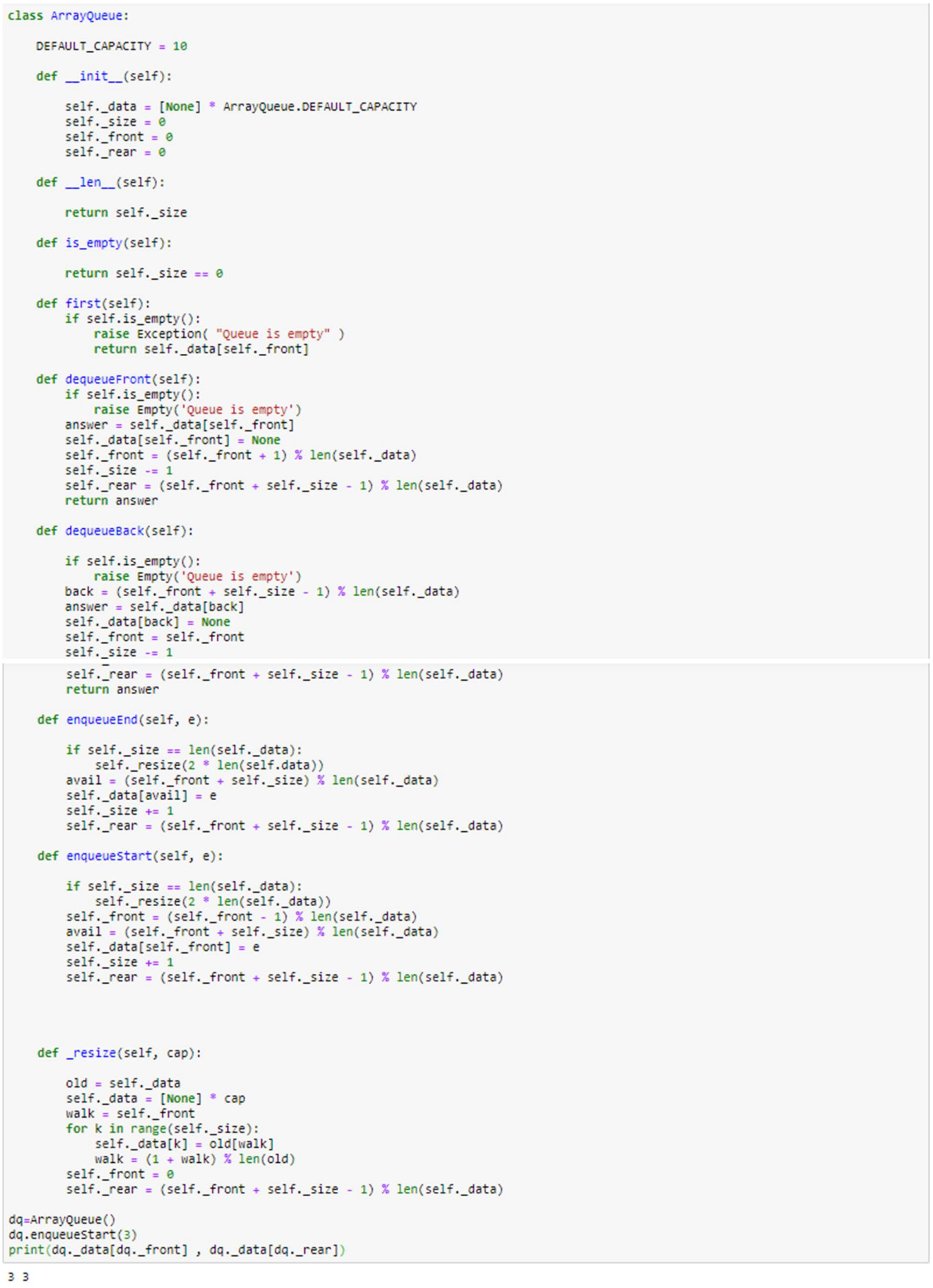
- Double Ended Queue

- We next consider a queue-like data structure that supports insertion and deletion at both the front and the back of the queue.

- Such a structure is called a double ended queue, or deque, which is usually pronounced “deck” to avoid confusion with the dequeue method of the regular queue ADT, which is

- pronounced like the abbreviation “D.Q.”
- The deque abstract data type is more general than both the stack and the queue ADTs.

Code:



```
class ArrayQueue:

    DEFAULT_CAPACITY = 10

    def __init__(self):

        self._data = [None] * ArrayQueue.DEFAULT_CAPACITY
        self._size = 0
        self._front = 0
        self._rear = 0

    def __len__(self):

        return self._size

    def is_empty(self):

        return self._size == 0

    def first(self):
        if self.is_empty():
            raise Exception( "Queue is empty" )
            return self._data[self._front]

    def dequeueFront(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._rear = (self._front + self._size - 1) % len(self._data)
        return answer

    def dequeueBack(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._rear = (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._rear = (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._rear = (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._rear = (self._front + self._size - 1) % len(self._data)

dq=ArrayQueue()
dq.enqueueStart(3)
print(dq._data[dq._front] , dq._data[dq._rear])
```

```
3 3
```

PRACTICAL-5

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

THEORY

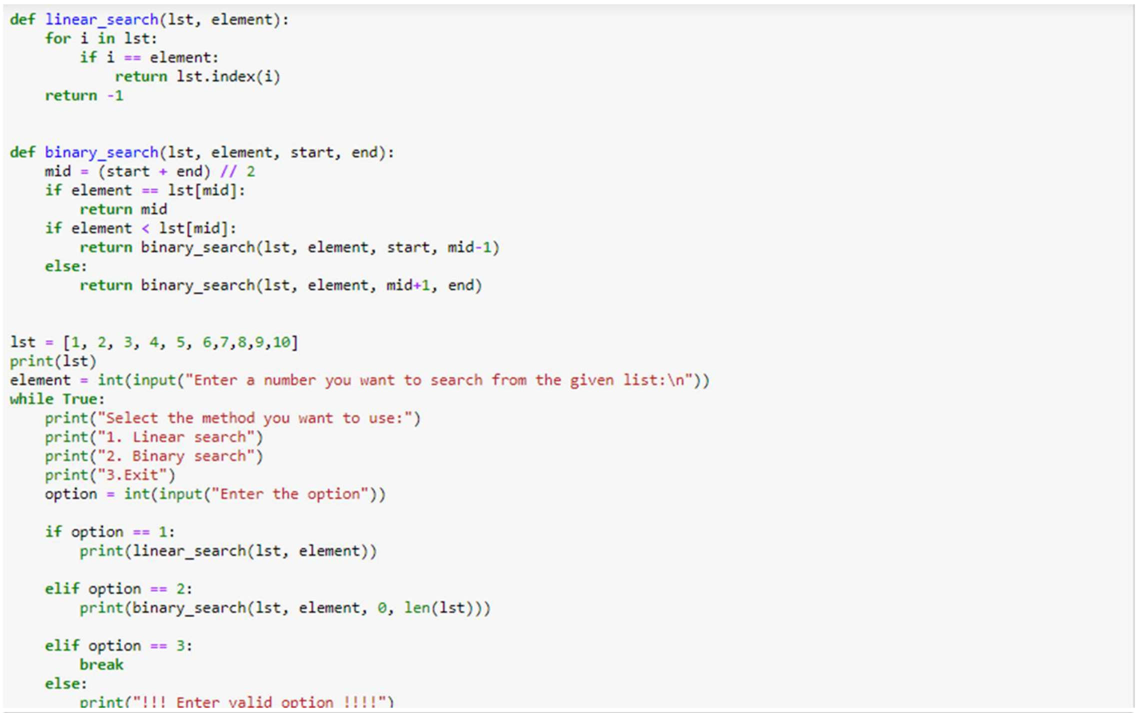
Binary Search:

- Search a sorted array by repeatedly dividing the search interval in half. Begin with an interval covering the whole array.
- If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise narrow it to the upper half. Repeatedly check until the value is found or the interval is empty.
- A binary search is an algorithm to find a particular element in the list. Suppose we have a list of thousand elements, and we need to get an index position of a particular element.
- We can find the element's index position very fast using the binary search algorithm. There are many searching algorithms but the binary search is most popular among them.
- The elements in the list must be sorted to apply the binary search algorithm. If elements are not sorted then sort them first.

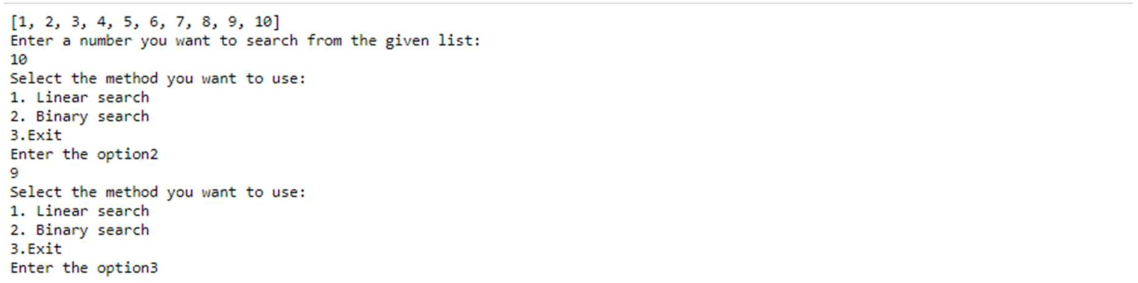
Linear Search:

- A Linear Search is the most basic type of searching algorithm.
- A Linear Search sequentially moves through your collection (or data structure) looking for a matching value.
- In other words, it looks down a list, one item at a time, without jumping.
- Linear search is a method of finding elements within a list. It is also called a sequential search.
- It is the simplest searching algorithm because it searches the desired element in a sequential manner.
- It compares each and every element with the value that we are searching for. If both are matched, the element is found, and the algorithm returns the key's index position.

Code:



```python
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, 6,7,8,9,10]
print(lst)
element = int(input("Enter a number you want to search from the given list:\n"))
while True:
    print("Select the method you want to use:")
    print("1. Linear search")
    print("2. Binary search")
    print("3.Exit")
    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)))

    elif option == 3:
        break
    else:
        print("!!! Enter valid option !!!!")
```



```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Enter a number you want to search from the given list:
10
Select the method you want to use:
1. Linear search
2. Binary search
3.Exit
Enter the option2
9
Select the method you want to use:
1. Linear search
2. Binary search
3.Exit
Enter the option3
```

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.

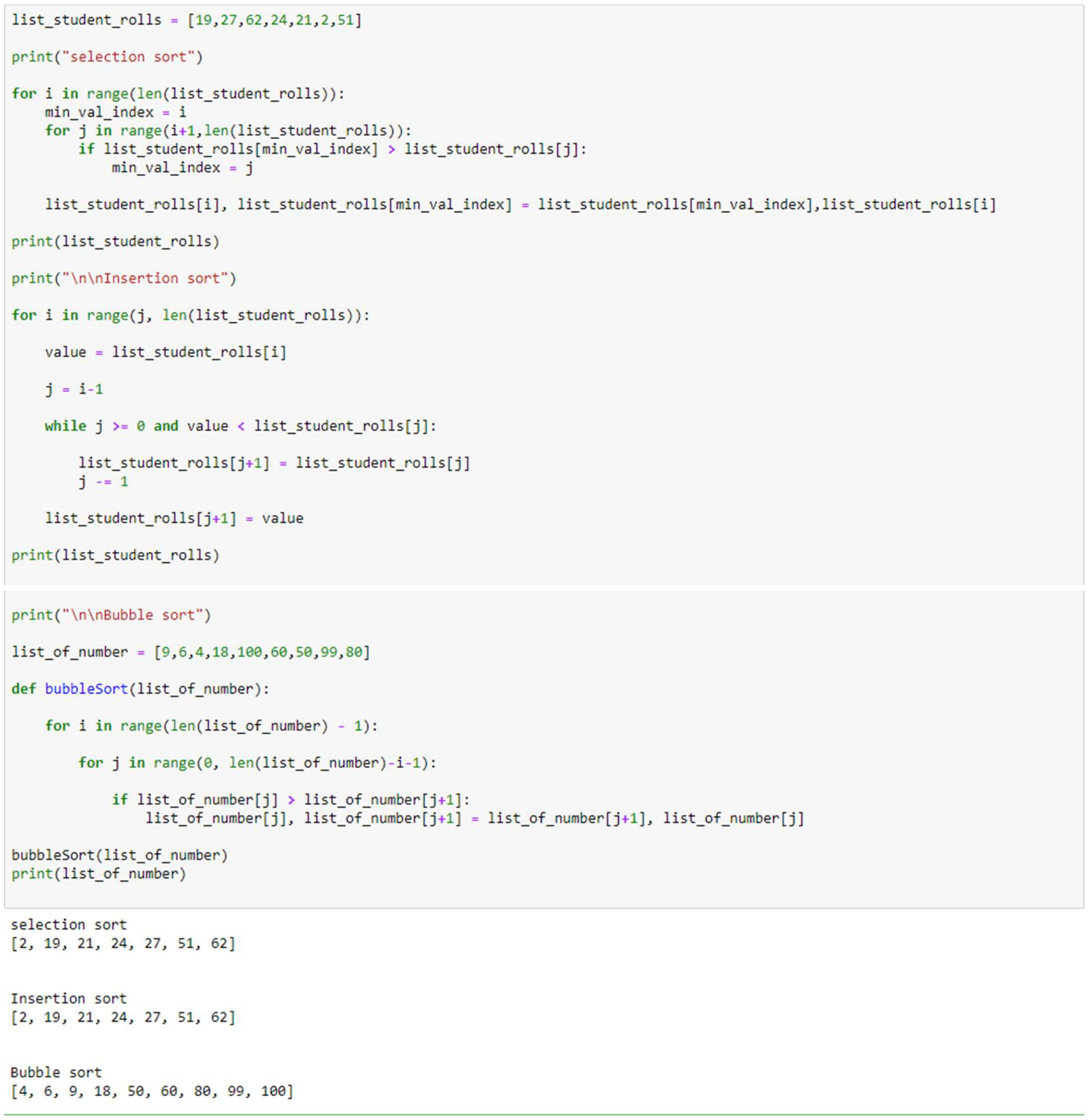
THEORY

- 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:** 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.
- **Selection Sort:** The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning. The algorithm maintains two subarrays in a given array.
    1. The subarray which is already sorted.
    2. Remaining subarray which is unsorted.
    3. In every iteration of selection sort, the minimum element (considering ascending order) from the unsorted subarray is picked and moved to the sorted subarray.
- **Insertion Sort:** Insertion sort is a simple sorting algorithm that works similar to the way you sort playing cards in your hands. The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part.

Code:



```
list_student_rolls = [19,27,62,24,21,2,51]

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 = [9,6,4,18,100,60,50,99,80]

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)
```

```
selection sort
[2, 19, 21, 24, 27, 51, 62]


Insertion sort
[2, 19, 21, 24, 27, 51, 62]


Bubble sort
[4, 6, 9, 18, 50, 60, 80, 99, 100]
```

PRACTICAL-7A

AIM: Write a program to implement the collision technique

THEORY

Hashing:
Hashing is an important Data Structure which is designed to use a special function called the Hash function which is used to map a given value with a particular key for faster access of elements. The efficiency of mapping depends of the efficiency of the hash function used.

Collisions:
A Hash Collision Attack is an attempt to find two input strings of a hash function that produce the same hash result. If two separate inputs produce the same hash output, it is called a collision.

Collision Techniques:

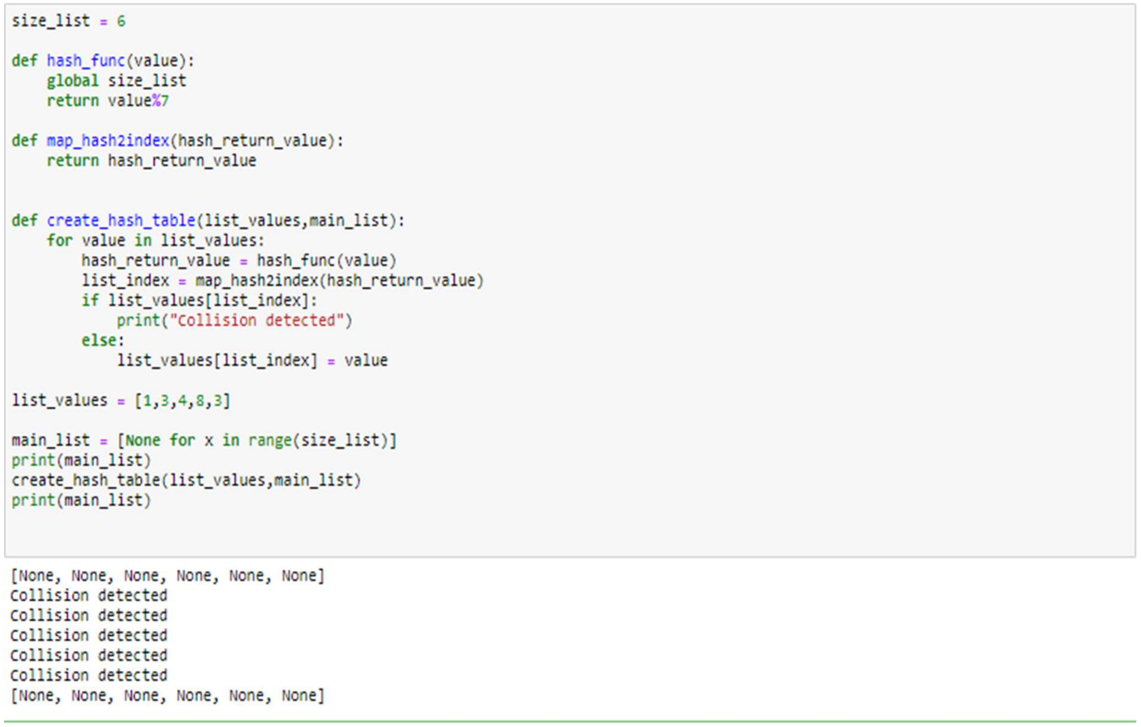
- Separate Chaining:

The idea is to make each cell of hash table point to a linked list of records that have same hash function value.

- Open Addressing:

Like separate chaining, open addressing is a method for handling collisions. In Open Addressing, all elements are stored in the hash table itself. So at any point, the size of the table must be greater than or equal to the total number of keys (Note that we can increase table size by copying old data if needed)

Code:



```
size_list = 6

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

def map_hash2index(hash_return_value):
    return hash_return_value


def create_hash_table(list_values,main_list):
    for value in list_values:
        hash_return_value = hash_func(value)
        list_index = map_hash2index(hash_return_value)
        if list_values[list_index]:
            print("Collision 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)
```

```
[None, None, None, None, None, None]
Collision detected
Collision detected
Collision detected
Collision detected
Collision detected
[None, None, None, None, None, None]
```

PRACTICAL-7B

AIM: Write a program to implement the concept of linear probing

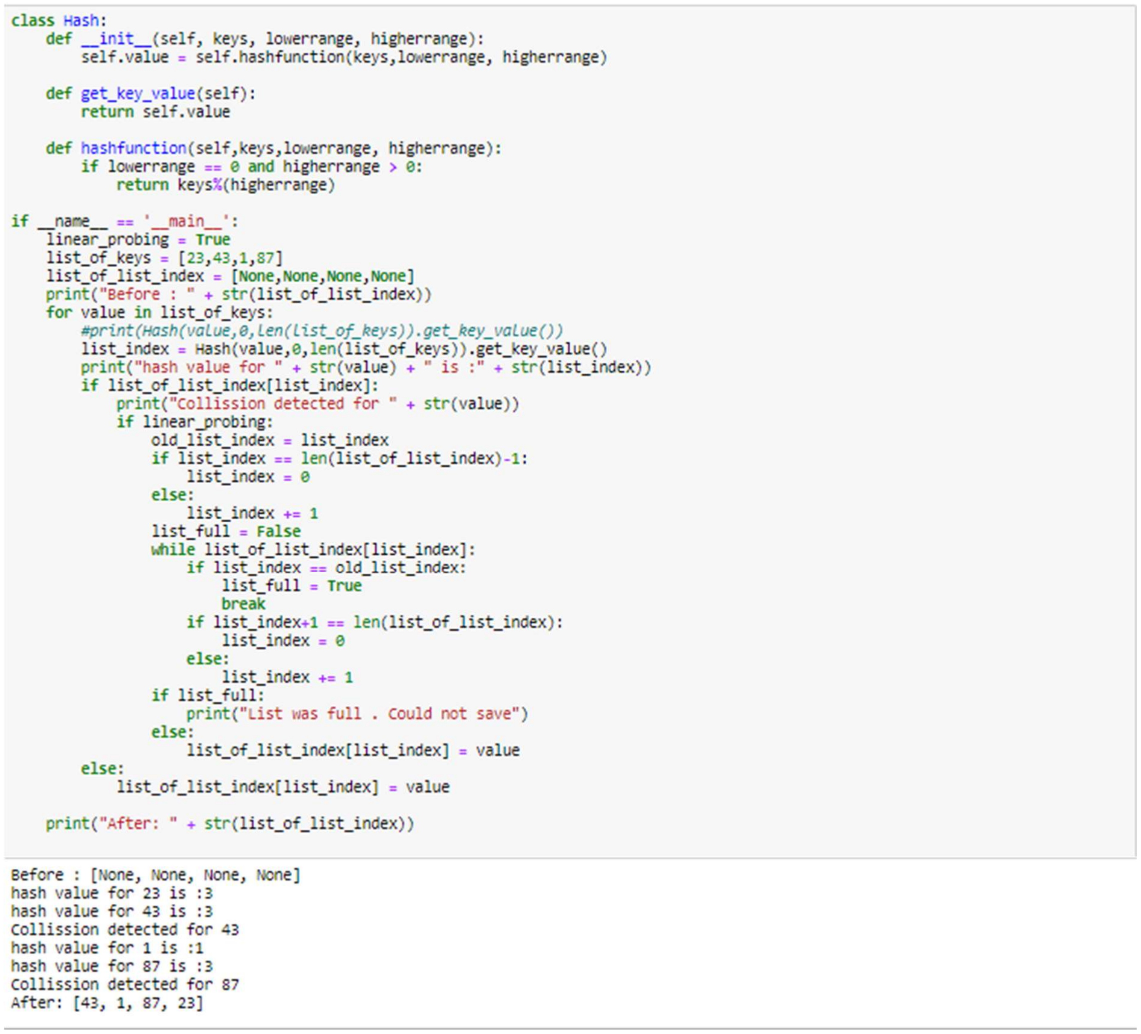
THEORY

Hashing: Hashing is an important Data Structure which is designed to use a special function called the Hash function which is used to map a given value with a particular key for faster access of elements. The efficiency of mapping depends of the efficiency of the hash function used.

Linear Probing

As we can see, it may happen that the hashing technique is used to create an already used index of the array. In such a case, we can search the next empty location in the array by looking into the next cell until we find an empty cell. This technique is called linear probing.

Code:



```python
class Hash:
    def __init__(self, keys, lowerrange, higherrange):
        self.value = self.hashfunction(keys,lowerrange, higherrange)

    def get_key_value(self):
        return self.value

    def hashfunction(self,keys,lowerrange, higherrange):
        if lowerrange == 0 and higherrange > 0:
            return keys%(higherrange)

if __name__ == '__main__':
    linear_probing = True
    list_of_keys = [23,43,1,87]
    list_of_list_index = [None,None,None,None]
    print("Before : " + str(list_of_list_index))
    for value in list_of_keys:
        #print(Hash(value,0,len(list_of_keys)).get_key_value())
        list_index = Hash(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))
```

```
Before : [None, None, None, None]
hash value for 23 is :3
hash value for 43 is :3
Collission detected for 43
hash value for 1 is :1
hash value for 87 is :3
Collission detected for 87
After: [43, 1, 87, 23]
```

PRACTICAL-8

AIM: Write a program for Inorder, Postorder and Preorder traversal of tree.

THEORY

- **Inorder Traversal**: For binary search trees (BST), Inorder Traversal specifies the nodes in non-descending order. In order to obtain nodes from BST in non-increasing order, a variation of inorder traversal may be used where inorder traversal is reversed.
- **Preorder Traversal:** Preorder traversal will create a copy of the tree. Preorder Traversal is also used to get the prefix expression of an expression.
- **Postorder Traversal:** Postorder traversal is used to get the postfix expression of an expression given

Inorder(root)

- Traverse the left sub-tree, (recursively call inorder(root -> left).
- Visit and print the root node.
- Traverse the right sub-tree, (recursively call inorder(root -> right).

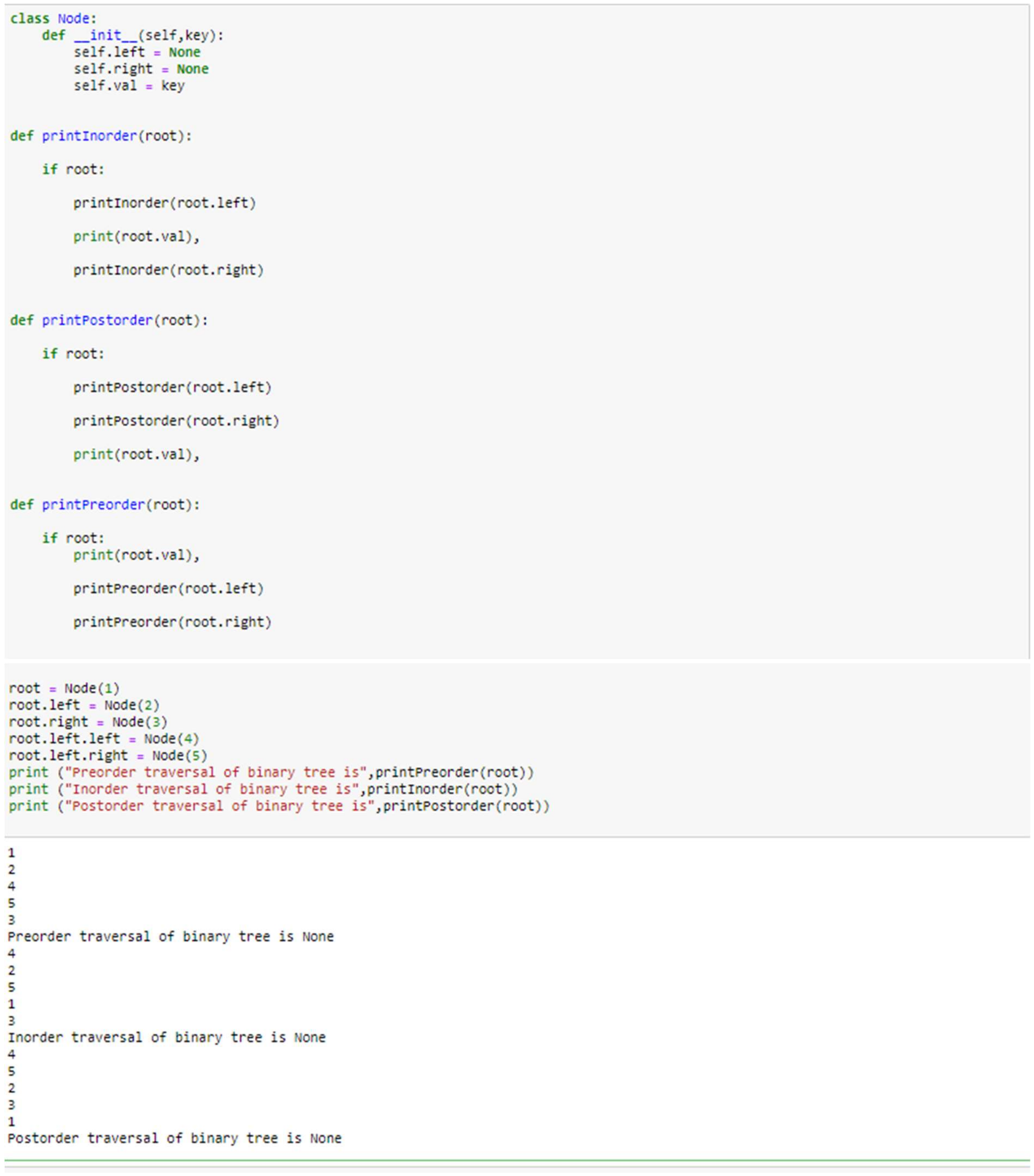
Preorder(root)

- Visit and print the root node.
- Traverse the left sub-tree, (recursively call inorder(root -> left).
- Traverse the right sub-tree, (recursively call inorder(root -> right).

Postorder(root)

- Traverse the left sub-tree, (recursively call inorder(root -> left).
- Traverse the right sub-tree, (recursively call inorder(root -> right).Visit and print the root node.
- Visit and print the root node.

Code:



```
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 ("Preorder traversal of binary tree is",printPreorder(root))
print ("Inorder traversal of binary tree is",printInorder(root))
print ("Postorder traversal of binary tree is",printPostorder(root))
```

```
1
2
4
5
3
Preorder traversal of binary tree is None
4
2
5
1
3
Inorder traversal of binary tree is None
4
5
2
3
1
Postorder traversal of binary tree is None
```