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 KHUSHI MANSURIA

Roll No: 354 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: 354**

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.	

ROLL NO:354 **NAME:KHUSHI MANSURIA** **CLASS:SYIT**

PRACTICAL-1A

GITHUB LINK: https://github.com/khushi-mansuria/data-structure/tree/master

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

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.

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.

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.

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.

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.

ROLL NO:354 **NAME:KHUSHI MANSURIA** **CLASS:SYIT**

CODE:

```
list1 = [2, 5, 3, 1, 4]
print("List 1: ",list1)

list1.sort()
print("sorted list 1",list1)

def search_list1():
    i = int(input("Enter element to search in the list:"))
    if i in list1:
        print("The element is in the list")
        print("Position of the element is", list1.index(i))
    else:
        print("The element is not in the list")

search_list1()

def reverse_list1():
    print("Reversing the list1: ", list1[::-1])

reverse_list1()

list2 = ["F", "C", "P", "D", "O"]
```

```
print("List 2: ", list2)

list2.sort()
print("sorted list 2", list2)

def merge_list():
    list3 = list1 + list2
    print("Merging two lists: ", list3)
```

ROLL NO:354 **NAME:KHUSHI MANSURIA** **CLASS:SYIT**

OUTPUT:

```
List 1:  [2, 5, 3, 1, 4]
sorted list 1 [1, 2, 3, 4, 5]
Enter element to search in the list:2
The element is in the list
Position of the element is 1
Reversing the list1:  [5, 4, 3, 2, 1]
List 2:  ['F', 'C', 'P', 'D', 'O']
sorted list 2 ['C', 'D', 'F', 'O', 'P']
Merging two lists:  [1, 2, 3, 4, 5, 'C', 'D', 'F', 'O', 'P']
```

ROLL NO:354 **NAME:KHUSHI MANSURIA** **CLASS:SYIT**

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.

ROLL NO:354 **NAME:KHUSHI MANSURIA** **CLASS:SYIT**

CODE:

```
class Matrix:
    def __get_dimensions(self, m_1, m_2=None):
        rows_1 = len(m_1)
        cols_1 = len(m_1[0])
        dim_1 = (rows_1, cols_1)
        if m_2 is not None:
            rows_2 = len(m_2)
            cols_2 = len(m_2[0])
            dim_2 = (rows_2, cols_2)
            same_dim = dim_1 == dim_2
            comp_dim = cols_1 == rows_2
            return dim_1, dim_2, same_dim, comp_dim
        else:
            return dim_1

    def addition(self, m_1, m_2):
        dim_1, dim_2, same_dim, comp_dim = self.__get_dimensions(m_1
            , m_2)
        if not same_dim:
            raise Exception("Matrices do not have the same
                dimensions")
```

```
20          sum_m = [[0 for cols in range(dim_1[1])] for rows in range
                (dim_1[0])]
21          for i in range(0, dim_1[0]):
22              for j in range(0, dim_1[1]):
23                  sum_m[i][j] = m_1[i][j] + m_2[i][j]
24          return sum_m
25
26
27
28      def multiplication(self, m_1, m_2):
29          dim_1, dim_2, same_dim, comp_dim = self.__get_dimensions(m_1
                , m_2)
30          if not comp_dim:
31              raise Exception("Matrices do not have compatible
                    dimensions")
32          product_m = [[0 for cols in range(dim_2[1])] for rows in
                range(dim_1[0])]
33          for i in range(0, dim_1[0]):
34              for j in range(0, dim_2[1]):
35                  for k in range(0, dim_2[0]):
36                      product_m[i][j] += m_1[i][k] * m_2[k][j]
```

```
        return product_m

    def transpose(self, m):
        dim = self.__get_dimensions(m)
        transpose_m = [[0 for cols in range(dim[0])] for rows in
            range(dim[1])]
        for i in range(0, dim[0]):
            for j in range(0, dim[1]):
                transpose_m[j][i] = m[i][j]
        return transpose_m

    def display(self, m):
        for row in m:
            print(row)


if __name__ == '__main__':
    matrix_1 = [[10, 20, 30], [40, 5, 6], [7, 8, 9]]
    matrix_2 = [[10, 20, 30], [40, 5, 6], [7, 8, 9]]
    matrix_3 = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
    matrix_4 = [[1, 2], [4, 5], [7, 8]]

    matrix = Matrix()
    sum = matrix.addition(matrix_1, matrix_2)
    transpose = matrix.transpose(matrix_3)
    product = matrix.multiplication(matrix_3, matrix_4)
    print('Matrix 1:')
    matrix.display(matrix_1)
    print('Matrix 2:')
    matrix.display(matrix_2)
    print('Matrix 3:')
    matrix.display(matrix_3)
    print('Matrix 4:')
    matrix.display(matrix_4)
    print('Transpose of Matrix 3:')
    matrix.display(transpose)
    print('Addition of Matrix 1 & 2:')
    matrix.display(sum)
    print('Multiplication of Matrix 3 & 4:')
    matrix.display(product)
```

ROLL NO:354 **NAME:KHUSHI MANSURIA** **CLASS:SYIT**

OUTPUT:

```
Matrix 1:
[10, 20, 30]
[40, 5, 6]
[7, 8, 9]
Matrix 2:
[10, 20, 30]
[40, 5, 6]
[7, 8, 9]
Matrix 3:
[1, 2, 3]
[4, 5, 6]
[7, 8, 9]
Matrix 4:
[1, 2]
[4, 5]
[7, 8]
Transpose of Matrix 3:
[1, 4, 7]
[2, 5, 8]
[3, 6, 9]
Addition of Matrix 1 & 2:
[20, 40, 60]
[80, 10, 12]
```

```
[14, 16, 18]
Multiplication of Matrix 3 & 4:
[30, 36]
[66, 81]
[102, 126]
```

ROLL NO:354 **NAME:KHUSHI MANSURIA** **CLASS:SYIT**

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

Singly Linked List:

A singly linked list, in its simplest form, is a collection of nodes that collectively form a linear sequence. Each node stores a reference to an object that is an element of the sequence, as well as a reference to the next node of the list

Doubly Linked List:

In a singly linked list, each node maintains a reference to the node that is immediately after it. However, there are limitations that stem from the asymmetry of a singly linked list. To provide greater symmetry, we define a linked list in which each node keeps an explicit reference to the node before it and a reference to the node after it.

Such a structure is known as a doubly linked list. These lists allow a greater variety of

O (1)-time update operations, including insertions and deletions at arbitrary positions

within the list. We continue to use the term “next” for the reference to the

node that follows another, and we introduce the term “prey” for the reference to the

node that precedes it. With array-based sequences, an integer index was a

convenient means for describing a position within a sequence. However, an index

is not convenient for linked lists as there is no efficient way to find the jth element;

it would seem to require a traversal of a portion of the list.

When working with a linked list, the most direct way to describe the location

of an operation is by identifying a relevant node of the list. However, we prefer

to encapsulate the inner workings of our data structure to avoid having users directly

access nodes of a list.

ROLL NO:354 **NAME:KHUSHI MANSURIA** **CLASS:SYIT**

CODE:

```
class Node():
    def __init__(self, data, Next = None, Previous = None):
        self.data = data
        self.next = Next
        self.previous = Previous
    def display(self):
        return self.data

    def getNext(self):
        return self.next

    def getPrevious(self):
        return self.previous

    def getData(self):
        return self.data

    def setData(self, newData):
        self.data = newData

    def setNext(self, newNext):
        self.next = newNext
```

```
24      def setPrevious(self, newPrevious):
25          self.previous = newPrevious
26
27
28  class DoublyLinkedList():
29      def __init__(self):
30          self.head = None
31          self.size = 0
32
33      def is_empty(self):
34          return self.size == 0
35
36      def add_head(self, data):
37          newNode = Node(data)
38          if self.head:
39              self.head.setPrevious(newNode)
40          newNode.setNext(self.head)
41          self.head = newNode
42          self.size = self.size+1
43
44      def add_tail(self, data):
45          newNode = Node(data)
```

```
46          current = self.head
47          while current.getNext() != None:
48              current = current.getNext()
49          current.setNext(newNode)
50          newNode.setPrevious(current)
51          self.size = self.size+1
52
53      def remove_head(self):
54          if self.is_empty():
55              print("Empty Singly linked list")
56          else:
57              print("Removing")
58              self.head = self.head.next
59              self.head.previous = None
60              self.size -= 1
61
62      def get_tail(self):
63          last_object = self.head
64          while (last_object.next != None):
65              last_object = last_object.next
66          return last_object
67
```

```
68      def get_node_at(self,index):
69          element_node = self.head
70          counter = 0
71          if index > self.size-1:
72              print("Index out of bound")
73              return None
74          while(counter < index):
75              element_node = element_node.next
76              counter += 1
77          return element_node
78
79      def get_prev_node(self,pos):
80          if pos == 0:
81              print('No previous elements')
82          return self.get_node_at(pos).previous
83
84
85      def remove_between_list(self,position):
86          if position > self.size-1:
87              print("Index out of bound")
88          elif position == self.size-1:
89              self.remove_tail()
```

```
90          elif position == 0:
91              self.remove_head()
92          else:
93              prev_node = self.get_node_at(position-1)
94              next_node = self.get_node_at(position+1)
95              prev_node.next = next_node
96              self.size -= 1
97
98      def display_all(self):
99          if self.size == 0:
100             print("No element")
101             return False
102         first = self.head
103         print(first.data)
104         first = first.next
105         while first:
106
107             print(first.data)
108             first = first.next
109
110
111
```

```
112     def find_second_last_element(self):
113         #second_last_element = None
114         if self.size >= 2:
115             first = self.head
116             temp_counter = self.size-2
117             while temp_counter > 0:
118                 first = first.next
119                 temp_counter -= 1
120             return first
121 
122 
123         else:
124             print("Size not sufficient")
125 
126     def remove_tail(self):
127         if self.is_empty():
128             print("Empty Singly linked list")
129         elif self.size == 1:
130             self.head == None
131             self.size -= 1
132         else:
133             Node = self.find_second_last_element()
```

```
            if Node:
                Node.next = None
                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
            self.size += 1

    def search(self,search_value):
        index = 0
```

```
156         while (index < self.size):
157             value = self.get_node_at(index)
158             print("Searching at " + str(index) + " and value is " +
                    str(value.data))
159             if value.data == search_value:
160                 print("Found value at " + str(index) + " location")
161                 return True
162             index += 1
163         print("Not Found")
164         return False
165
166     def mergeList(self,linkedlist_value):
167         if self.size > 0:
168             last_node = self.get_node_at(self.size-1)
169             last_node.next = linkedlist_value.head
170             self.size = self.size + linkedlist_value.size
171
172         else:
173             self.head = linkedlist_value.head
174             self.size = linkedlist_value.size
175
176     def reverse_list(self):
```

```
        prev = None
        curr = self.head
        while (curr != None):
            next = curr.next
            curr.next = prev
            prev = curr
            curr = next
        self.head = prev
        
        


print('-----------------------lst1-----------------------')
print()
lst1 = DoublyLinkedList()
print("List is empty => ",lst1.is_empty())
print("---------------add head--------------")
lst1.add_head("a")
lst1.add_head("b")
lst1.add_head("c")
print("---------------add tail--------------")
lst1.add_tail("d")
```

```
199 lst1.add_tail("e")
200 lst1.add_tail("f")
201 lst1.add_tail("g")
202 lst1.display_all()
203 print("\n------------remove head------------")
204 lst1.remove_head()
205 lst1.display_all()
206 print("List is empty => ",lst1.is_empty())
207 print("Size => ",lst1.size)
208 print("Head => ",lst1.head.display())
209 print("Tail => ",lst1.get_tail().display())
210 print("Second last => ",lst1.find_second_last_element().display())
211 print("\n------------remove tail------------")
212 lst1.remove_tail()
213 lst1.display_all()
214 print("\nget node at pos 2 => ",lst1.get_node_at(2).display())
215 print("get previous node at pos 1 => ",lst1.get_prev_node(1
        ).display())
216 print("get previous node at pos 3 => ",lst1.get_prev_node(3
        ).display())
217 print("\n---remove between list atIndex 2---")
218 lst1.remove_between_list(2)
```

```
print("\n-----add between list atIndex 2----")
lst1.add_between_list(2,"h")
lst1.display_all()
print("\n-----------search value------------")
lst1.search("h")


print()
print('-----------------------lst2--------------------------')
print()

lst2 = DoublyLinkedList()
print("List is empty => ",lst2.is_empty())
print("---------------add head--------------")
lst2.add_head(1)
lst2.add_head(2)
lst2.add_head(3)
lst2.add_head(4)
print("---------------add tail--------------")
lst2.add_tail(5)
lst2.add_tail(6)
```

```
lst2.display_all()
print("\n------------remove head------------")
lst2.remove_head()
lst2.display_all()
print("List is empty => ",lst2.is_empty())
print("Size => ",lst2.size)
print("Head => ",lst2.head.display())
print("Tail => ",lst2.get_tail().display())
print("Second last => ",lst2.find_second_last_element().display())
print("\n------------remove tail------------")
lst2.remove_tail()
lst2.display_all()
print("\nget previous node at pos 1 => ",lst2.get_prev_node(1
    ).display())
print("\nget previous node at pos 3 => ",lst2.get_prev_node(3
    ).display())
print("\nget node at pos 2 => ",lst2.get_node_at(2).display())
print("\n---remove between list atIndex 2---")
lst2.remove_between_list(2)
print("\n-----add between list atIndex 2----")
lst2.add_between_list(2,0)
lst2.display_all()
```

```
print("\n-----------search value------------")
lst2.search(1)

print("\n---------lst1.merge(lst2)----------\n")
lst1.mergeList(lst2)
lst1.display_all()
print("List is empty => ",lst1.is_empty())
print("Size => ",lst1.size)
print("Head => ",lst1.head.display())
print("Tail => ",lst1.get_tail().display())
lst1.reverse_list()
print("\n--------Reversed linked list-------")
print(lst1.display_all())

print("Size => ",lst1.size)
print("Head => ",lst1.head.display())
print("Tail => ",lst1.get_tail().display())
```

ROLL NO:354 **NAME:KHUSHI MANSURIA** **CLASS:SYIT**

OUTPUT:

```
------------------------lst1-----------------------

List is empty =>  True
---------------add head--------------
---------------add tail--------------
c
b
a
d
e
f
g

------------remove head------------
Removing
b
a
d
e
f
g
List is empty =>  False
Size =>  6
```

```
Head =>  b
Tail =>  g
Second last =>  f

------------remove tail------------
b
a
d
e
f

get node at pos 2 =>  d
get previous node at pos 1 =>  b
get previous node at pos 3 =>  d

---remove between list atIndex 2---

-----add between list atIndex 2----
b
a
h
e
f
```

```
----------search value-----------
Searching at 0 and value is b
Searching at 1 and value is a
Searching at 2 and value is h
Found value at 2 location

-----------------------lst2--------------------------

List is empty =>  True
---------------add head--------------
---------------add tail--------------
4
3
2
1
5
6

------------remove head------------
Removing
3
2
```

```
1
5
6
List is empty =>  False
Size =>  5
Head =>  3
Tail =>  6
Second last =>  5

------------remove tail------------
3
2
1
5

get previous node at pos 1 =>  3

get previous node at pos 3 =>  1

get node at pos 2 =>  1

---remove between list atIndex 2---
```

```
-----add between list atIndex 2----
3
2
0
5

-----------search value------------
Searching at 0 and value is 3
Searching at 1 and value is 2
Searching at 2 and value is 0
Searching at 3 and value is 5
Not Found

---------lst1.merge(lst2)----------

b
a
h
e
f
3
2
0
```

```
5
List is empty =>  False
Size =>  9
Head =>  b
Tail =>  5

--------Reversed linked list-------
5
0
2
3
f
e
h
a
b
None
Size =>  9
Head =>  5
Tail =>  b
```

ROLL NO:354 **NAME:KHUSHI MANSURIA** **CLASS:SYIT**

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:

S.push(e): Add element e to the top of stack S.

S.pop(): Remove and return the top element from the stack S;

an error occurs if the stack is empty.

CODE:

```
1  class Stack:
2
3      def __init__(self, array_size=1000):
4          self.array = [None] * array_size
5          self.top = -1
6          self.size = 0
7
8      def push(self, element):
9          assert self.top != len(self.array) - 1, "the stack is full
               of elements"
10         self.top = self.top + 1
11         self.array[self.top] = element
12         self.size += 1
13
14     def is_empty(self):
15         if self.size == 0:
16             return True
17         else:
18             return False
19
20     def pop(self):
21         # return None if stack is empty
```

```
        while not self.is_empty():
            element = self.array[self.top]
            self.array[self.top] = None
            self.top = self.top - 1
            self.size -= 1
            return element
        return None

    def create_copy_of_data(self):
        element_list = []
        for i in self.array:
            if i is not None:
                element_list.append(i)
        return element_list


def test_stack():

    st = Stack()
    for element in [5, 3, 'hi', 'dina', 8]:
        st.push(element)
    for element in [8, 'dina', 'hi', 3, 5]:
        st_element = st.pop()
        assert st_element == element
    assert st.pop() is None
    for element in [5, 3, 'hi', 'dina', 8]:
        st.push(element)
    print(st.create_copy_of_data())

test_stack()
```

OUTPUT:

```
[5, 3, 'hi', 'dina', 8]
>
```

ROLL NO:354 **NAME:KHUSHI MANSURIA** **CLASS:SYIT**

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.

ROLL NO:354 **NAME:KHUSHI MANSURIA** **CLASS:SYIT**

CODE:

```
def towerOfHanoi(n_disks, source, destin, temp):
    if n_disks == 1:
        print ("Move disk 1 from rod",source,"to rod",destin)
        return
    towerOfHanoi(n_disks-1, source, temp, destin)
    print ("Move disk",n_disks,"from rod",source,"to rod",destin)
    towerOfHanoi(n_disks-1, temp, destin, source)

n_disks = 3
towerOfHanoi(n_disks, 'A', 'C', 'B')
```

OUTPUT:

```
Move disk 1 from rod A to rod C
Move disk 2 from rod A to rod B
Move disk 1 from rod C to rod B
Move disk 3 from rod A to rod C
Move disk 1 from rod B to rod A
Move disk 2 from rod B to rod C
Move disk 1 from rod A to rod C
>
```

ROLL NO:354 **NAME:KHUSHI MANSURIA** **CLASS:SYIT**

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.

Every Polynomial in the program is a Doubly Linked List object. The corresponding terms are added and displayed in the form of an expression.

ROLL NO:354 **NAME:KHUSHI MANSURIA** **CLASS:SYIT**

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)
```

ROLL NO:354 **NAME:KHUSHI MANSURIA** **CLASS:SYIT**

OUTPUT:

```
Enter the order for polynomial : 3
Enter coefficient for power 3 : 2
Enter coefficient for power 2 : 1
Enter coefficient for power 1 : 0
Enter coefficient for power 0 : 0
Enter coefficient for power 3 : 4
Enter coefficient for power 2 : 8
Enter coefficient for power 1 : 56
Enter coefficient for power 0 : 987
6
9
56
987
>
```

ROLL NO:354 **NAME:KHUSHI MANSURIA** **CLASS:SYIT**

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.

Factorial:

Factors are the numbers you multiply to get another number. For instance, factors of 15 are 3 and 5, because 3×5 = 15. Some numbers have more than one factorization (more than one way of being factored). For instance, 12 can be factored as 1×12, 2×6, or 3×4. A number that can only be factored as 1 time itself is called "prime".

You can find it using recursion as well as iteration to calculate the factors of a number.

ROLL NO:354 **NAME:KHUSHI MANSURIA** **CLASS:SYIT**

CODE:

```
factorial = 1
n = int(input('Enter Number: '))
for i in range(1,n+1):
    factorial = factorial * i

print(f'Factorial is : {factorial}')

fact = []
for i in range(1,n+1):
    if (n/i).is_integer():
        fact.append(i)

print(f'Factors of the given numbers is : {fact}')

factorial = 1
index = 1
n = int(input("Enter number : "))
def calculate_factorial(n,factorial,index):
    if index == n:
        print(f'Factorial is : {factorial}')
        return True
    else:
        index = index + 1
```

ROLL NO:354 **NAME:KHUSHI MANSURIA** **CLASS:SYIT**

```
        calculate_factorial(n,factorial * index,index)
calculate_factorial(n,factorial,index)

fact = []
def calculate_factors(n,factors,index):
    if index == n+1:
        print(f'Factors of the given numbers is : {factors}')
        return True
    elif (n/index).is_integer():
        factors.append(index)
        index += 1
        calculate_factors(n,factors,index)
    else:
        index += 1
        calculate_factors(n,factors,index)

index = 1
factors = []
calculate_factors(n,factors,index)
```

OUTPUT:

```
Enter Number: 2
Factorial is : 2
Factors of the given numbers is : [1, 2]
Enter number : 5
Factorial is : 120
Factors of the given numbers is : [1, 5]
>
```

ROLL NO:354 **NAME:KHUSHI MANSURIA** **CLASS:SYIT**

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:

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.

ROLL NO:354 **NAME:KHUSHI MANSURIA** **CLASS:SYIT**

CODE:

```
class Array_Queue:
    DEFAULT_CAPACITY = 10

    def __init__(self):
        self._data = [None] * Array_Queue.DEFAULT_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 Start_dequeue(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 End_dequeue(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 End_enqueue(self, e):
```

```
44          if self._size == len(self._data):
45              self._resize(2 * len(self.data))
46          avail = (self._front + self._size) % len(self._data)
47          self._data[avail] = e
48          self._size += 1
49          self._back = (self._front + self._size - 1) % len(self._data
                )
50
51      def Start_enqueue(self, e):
52          if self._size == len(self._data):
53              self._resize(2 * len(self._data))     # double the array
                    size
54          self._front = (self._front - 1) % len(self._data)
55          avail = (self._front + self._size) % len(self._data)
56          self._data[self._front] = e
57          self._size += 1
58          self._back = (self._front + self._size - 1) % len(self._data
                )
59
60      def resize(self, cap):
61          old = self._data
62          self._data = [None] * cap
63          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 = Array_Queue()
queue.End_enqueue(1)
print(f"First Element: {queue._data[queue._front]}, Last Element:
    {queue._data[queue._back]}")
queue._data
queue.End_enqueue(2)
print(f"First Element: {queue._data[queue._front]}, Last Element:
    {queue._data[queue._back]}")
queue._data
queue.Start_dequeue()
print(f"First Element: {queue._data[queue._front]}, Last Element:
    {queue._data[queue._back]}")
queue.End_enqueue(3)
print(f"First Element: {queue._data[queue._front]}, Last Element:
    {queue._data[queue._back]}")
```

```
queue.End_enqueue(4)
print(f"First Element: {queue._data[queue._front]}, Last Element:
    {queue._data[queue._back]}")
queue.Start_dequeue()
print(f"First Element: {queue._data[queue._front]}, Last Element:
    {queue._data[queue._back]}")
queue.Start_enqueue(5)
print(f"First Element: {queue._data[queue._front]}, Last Element:
    {queue._data[queue._back]}")
queue.End_dequeue()
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
>
```

PRACTICAL-5

ROLL NO:354 **NAME:KHUSHI MANSURIA** **CLASS:SYIT**

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.

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.

CODE:

ROLL NO:354 **NAME:KHUSHI MANSURIA** **CLASS:SYIT**

```
class Search:
    def __init__(self, lst, ele):
        self.lst = lst
        self.ele = ele

    def binary_search(self):
        sorted_lst = sorted(self.lst)
        start = 0
        end = len(sorted_lst) - 1
        while start <= end:
            mid = (end + start) // 2
            if sorted_lst[mid] < self.ele:
                start = mid + 1
            elif sorted_lst[mid] > self.ele:
                end = mid - 1
            else:
                return mid
        return False

    def linear_search(self):
        for i in range(len(self.lst)):
            if self.lst[i] == self.ele:
                return i
        return False


if __name__ == '__main__':
    test_list = [35, 10, 43, 82, 66, 51, 97]
    test_value_1 = 43
    test_value_2 = 44
    test_search = Search(test_list, test_value_1)
    print(test_search.binary_search())
    print(test_search.linear_search())
    test_search_2 = Search(test_list, test_value_2)
    print(test_search_2.binary_search())
    print(test_search_2.linear_search())
```

OUTPUT:

```
2
2
False
False
> 
```

PRACTICAL-6

ROLL NO:354 **NAME:KHUSHI MANSURIA** **CLASS:SYIT**

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.

This 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 'insert'ed 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:

```
# Selection sort


def selection_sort(num):
    for i in range(len(num)):
        lowest_value_index = i
        for j in range(i + 1, len(num)):
            if num[j] < num[lowest_value_index]:
                lowest_value_index = j
                num[i], num[lowest_value_index] =
                    num[lowest_value_index], num[i]
    return num


list2 = [23, 14, 66, 8, 2]
print(selection_sort(list2))


# Insertion sort

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

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


# main
arr = ['t', 'u', 't', 'o', 'r', 'i', 'a', 'l']
print("The sorted array is:")
print(insertionSort(arr))


# Bubble sort

def bubble_sort(num):
    swap = True
    while swap:
        swap = False
        for i in range(len(num) - 1):
            if num[i] > num[i + 1]:
                num[i], num[i + 1] = num[i + 1], num[i]
                swap = True
     return num


list = [23, 14, 66, 8, 2]
print(bubble_sort(list))
```

ROLL NO:354 **NAME:KHUSHI MANSURIA** **CLASS:SYIT**

OUTPUT:

```
[2, 8, 14, 23, 66]
The sorted array is:
['a', 'i', 'l', 'o', 'r', 't', 't', 'u']
[2, 8, 14, 23, 66]
>
```

ROLL NO:354 **NAME:KHUSHI MANSURIA** **CLASS:SYIT**

PRACTICAL-7

Aim: 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.

THEORY

A map M supports the abstraction of using keys as indices with a

syntax such as M[k]. As a mental warm-up, consider a restricted setting in which

a map with n items uses keys that are known to be integers in a range from 0 to

N −1 for some N ≥ n. In this case, we can represent the map using a lookup table

of length N.

we store the value associated with key k at index k of the

table (presuming that we have a distinct way to represent an empty slot). Basic map

operations of getitem , setitem , and delitem can be implemented in

O(1) worst-case time.

The goal of a hash function, h, is to map each key k to an integer in the range

[0,N −1], where N is the capacity of the bucket array for a hash table. Equipped

with such a hash function, h, the main idea of this approach is to use the hash

function value, h(k), as an index into our bucket array, A, instead of the key k

(which may not be appropriate for direct use as an index). That is, we store the

item (k,v) in the bucket A[h(k)].

If there are two or more keys with the same hash value, then two different items

will be mapped to the same bucket in A. In this case, we say that a collision has

occurred. To be sure, there are ways of dealing with collisions, which we will

discuss later, but the best strategy is to try to avoid them in the first place. We say that a hash function is "good" if it maps the keys in our map so as to sufficiently minimize collisions. For practical reasons, we also would like a hash function to be fast and easy to compute.

It is common to view the evaluation of a hash function, h(k), as consisting of two portions—a hash code that maps a key k to an integer, and a compression function that maps the hash code to an integer within a range of indices, [0,N −1],

for a bucket array.

The advantage of separating the hash function into two such components is that the hash code portion of that computation is independent of a specific hash table size. This allows the development of a general hash code for each object that can be used for a hash table of any size; only the compression function depends upon the table size. This is particularly convenient, because the underlying bucket array for a hash table may be dynamically resized, depending on the number of items currently stored in the map.

ROLL NO:354 **NAME:KHUSHI MANSURIA** **CLASS:SYIT**

CODE:

```
class Hash:
    def __init__(self, keys: int, lower_range: int, higher_range:
        int) -> None:
        self.value = self.hash_function(keys, lower_range,
            higher_range)

    def get_key_value(self) -> int:
        return self.value

    @staticmethod
    def hash_function(keys: int, lower_range: int, higher_range: int
        ) -> int:
        if lower_range == 0 and higher_range > 0:
            return keys % higher_range


if __name__ == '__main__':
    linear_probing = True
    list_of_keys = [23, 43, 1, 87]
    list_of_list_index = [None]*4
    print("Before : " + str(list_of_list_index))
    for value in list_of_keys:
        list_index = Hash(value, 0, len(list_of_keys)).get_key_value
```

```
            ()
21      print("Hash value for " + str(value) + " is :" + str
            (list_index))
22      if list_of_list_index[list_index]:
23          print("Collision detected for " + str(value))
24          if linear_probing:
25              old_list_index = list_index
26              if list_index == len(list_of_list_index) - 1:
27                  list_index = 0
28              else:
29                  list_index += 1
30              list_full = False
31              while list_of_list_index[list_index]:
32                  if list_index == old_list_index:
33                      list_full = True
34                      break
35                  if list_index + 1 == len(list_of_list_index):
36                      list_index = 0
37                  else:
38                      list_index += 1
39              if list_full:
40                  print("List was full . Could not save")
41              else:
42                  list_of_list_index[list_index] = value
43      else:
44          list_of_list_index[list_index] = value
45  print("After: " + str(list_of_list_index))
```

ROLL NO:354 **NAME:KHUSHI MANSURIA** **CLASS:SYIT**

OUTPUT:

```
Before : [None, None, None, None]
Hash value for 23 is :3
Hash value for 43 is :3
Collision detected for 43
Hash value for 1 is :1
Hash value for 87 is :3
Collision detected for 87
After: [43, 1, 87, 23]
>
```

ROLL NO:354 **NAME:KHUSHI MANSURIA** **CLASS:SYIT**

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.

ROLL NO:354 **NAME:KHUSHI MANSURIA** **CLASS:SYIT**

CODE:

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

    def PrintTree(self):
        if self.left:
            self.left.PrintTree()
        print(self.value)
        if self.right:
            self.right.PrintTree()

    def Printpreorder(self):
        if self.value:
            print(self.value)
            if self.left:
                self.left.Printpreorder()
            if self.right:
                self.right.Printpreorder()

    def Printinorder(self):
        if self.value:
```

```
24            if self.left:
25                self.left.Printinorder()
26            print(self.value)
27            if self.right:
28                self.right.Printinorder()
29
30    def Printpostorder(self):
31        if self.value:
32            if self.left:
33                self.left.Printpostorder()
34            if self.right:
35                self.right.Printpostorder()
36            print(self.value)
37
38    def insert(self, data):
39        if self.value:
40            if data < self.value:
41                if self.left is None:
42                    self.left = Node(data)
43                else:
44                    self.left.insert(data)
45            elif data > self.value:
46                if self.right is None:
```

```
                    self.right = Node(data)
                else:
                    self.right.insert(data)
        else:
            self.value = data


if __name__ == '__main__':
    root = Node(10)
    root.left = Node(12)
    root.right = Node(5)
    print("Without any order")
    root.PrintTree()
    root_1 = Node(None)
    root_1.insert(28)
    root_1.insert(4)
    root_1.insert(13)
    root_1.insert(130)
    root_1.insert(123)
    print("Now ordering with insert")
    root_1.PrintTree()
    print("Pre order")
    root_1.Printpreorder()
```

```
    print("In Order")
    root_1.Printinorder()
    print("Post Order")
    root_1.Printpostorder()
```

ROLL NO:354 **NAME:KHUSHI MANSURIA** **CLASS:SYIT**

OUTPUT:

```
Without any order
12
10
5
Now ordering with insert
4
13
28
123
130
Pre order
28
4
13
130
123
In Order
4
13
28
123
130
```

```
Post Order
13
4
123
130
28
```