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. Dikshita Shetty**

Roll No: **385** 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: 385**

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.	

GITHUB LINK OF DS PRACTICALS REPOSITORY:
https://github.com/dikshita18/DS

PRACTICAL 1

AIM: Implement the following for Array:

THEORY:

Array is a container which can hold a fix number of items and these items should be of the same type. Most of the data structures make use of arrays to implement their algorithms. Following are the important terms to understand the concept of Array.

- **Element** − Each item stored in an array is called an element.
- **Index** − Each location of an element in an array has a numerical index, which is used to identify the element. Index starts with 0.

PRACTICAL 1A

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:

In arrays we can perform various operations, some of them are:

Search − Searches an element using the given index or by the value.

Sort – Sorting the elements of an array in ascending order.

Merge – Merging two or more arrays .

Reverse – Reverses the elements in the array.

These array operations are implemented in the following program using 1-D array.

CODE:

```python
#Program to perform the operations like searching, sorting, merging, reversing the elements.
list1 = [4, 5, 3, 1, 2]
print("List 1: ", list1)

#sorting the 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 = ["E", "C", "B", "D", "A"]
print("List 2: ", list2)

#sorting the list2
list2.sort()
print("sorted list 2", list2)

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

merge_list()
```

OUTPUT:

```
File  Edit  Shell  Debug  Options  Window  Help
Python 3.9.0 (tags/v3.9.0:9cf6752, Oct  5 2020, 15:34:40) [MSC v.1927 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
============== RESTART: C:\DIKSHITA\SUBJECTS\DS\DS PRACS\prac1a.py =============
List 1:  [4, 5, 3, 1, 2]
sorted list 1 [1, 2, 3, 4, 5]
Enter element to search in the list:4
The element is in the list.
Position of the element is 3
Reversing the list1:  [5, 4, 3, 2, 1]
List 2:  ['E', 'C', 'B', 'D', 'A']
sorted list 2 ['A', 'B', 'C', 'D', 'E']
Merging two lists:  [1, 2, 3, 4, 5, 'A', 'B', 'C', 'D', 'E']
>>>
```

PRACTICAL 1B

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

THEORY: In this program we have performed matrix operations like addition, multiplication and transpose on an array using loops.

CODE:

```
#Performing Matrix Operations
M1 = [ [1, 2, 3],
       [4, 5, 6],
       [7, 8, 9] ]

M2 = [ [9, 8, 7],
       [6, 5, 4],
       [3, 2, 1] ]

M3 = [ [0, 0, 0],
        [0, 0, 0],
        [0, 0, 0] ]

print("\nMatrix 1: \n")
for i in M1:
    print(i)

print("\nMatrix 2: \n")
for i in M2:
    print(i)

#Matrix multiplication
for i in range(len(M1)):
    for j in range(len(M2[0])):
        for k in range(len(M2)):
            M3[i][j] += M1[i][k] * M2[k][j]

print("\nMatrix Multiplication: \n")
for i in M3:
    print(i)
```

```python
#Matrix addition
for i in range(len(M3)):
    for j in range(len(M3[0])):
        M3[i][j] = M1[i][j] + M2[i][j]

print("\nMatrix Addition: \n")
for i in M3:
    print(i)

#Transpose of matrix
#Matrix 1
for i in range(len(M3)):
    for j in range(len(M3[0])):
        M3[i][j] = M1[j][i]

print("\nTranspose of matrix 1: \n")
for i in M3:
    print(i)

#Matrix 2
for i in range(len(M3)):
    for j in range(len(M3[0])):
        M3[i][j] = M2[j][i]

print("\nTranspose of matrix 2: \n")
for i in M3:
    print(i)
```

OUTPUT:

```
Python 3.9.0 (tags/v3.9.0:9cf6752, Oct  5 2020, 15:34:40) [MSC v.1927 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
=========== RESTART: C:\DIKSHITA\SUBJECTS\DS\DS PRACS\Practical_1B.py ==========

Matrix 1:

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

Matrix 2:

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

Matrix Multiplication:

[30, 24, 18]
[84, 69, 54]
[138, 114, 90]

Matrix Addition:

[10, 10, 10]
[10, 10, 10]
[10, 10, 10]

Transpose of matrix 1:

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

Transpose of matrix 2:

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

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.

Following are the important terms to understand the concept of Linked List.

- **Link** − Each link of a linked list can store a data called an element.
- **Next** − Each link of a linked list contains a link to the next link called Next.
- **LinkedList** − A Linked List contains the connection link to the first link called First.

Following are the various types of linked list.

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

In this program we are going to perform Singly Linked List and Doubly Linked List and performing operations like insertion, deletion, searching, merging, reversing, etc. on these Linked lists.

CODE1:

Singly Linked List:

```
#Program to implement Singly Linked List
class Node:

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

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

class LinkedList:

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

    def _len_(self):
        return self.size

    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)
        first = first.next
        while first:
            print(first.element)
            first = first.next

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

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

```
    def 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 Elements")
            self.head = self.head.next
            self.size -= 1

    def add_tail(self, e):
        new_value = Node(e)
        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 > self.size-1:
        print("Index out of bound")
        return None
    while(counter < index):
        element_node = element_node.next
        counter += 1
    return element_node

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
        self.size -= 1

def add_between_list(self, position, 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
        element.next = current_node
        self.size -= 1
```

```
    def search (self, search_value):
        index = 0
        while (index < self.size):
            value = self.get_node_at(index)
            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
            self.size = self.size + linkedlist_value.size

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


list1=LinkedList()
list1.add_head(3)
list1.add_head(2)
list1.add_head(1)
list1.add_tail(4)
list1.add_tail(5)
print("List 1:")
list1.display()
list1.remove_head()
list1.remove_tail()
list1.add_between_list(3,9)
list1.remove_between_list(2)
print("Now the List is:")
list1.display()

list2=LinkedList()
list2.add_head(8)
list2.add_head(7)
list2.add_head(6)
print("List 2:")
list2.display()
list1.merge(list2)
print('Merging List1 and List2')
list1.display()
```

OUTPUT1:

```
Python 3.9.0 (tags/v3.9.0:9cf6752, Oct  5 2020, 15:34:40) [MSC v.1927 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
========= RESTART: C:\DIKSHITA\SUBJECTS\DS\codes\single_linked_list.py =========
List 1:
1
2
3
4
5
Removing Elements
Now the List is:
2
3
9
List 2:
6
7
8
Merging List1 and List2
2
3
9
6
7
8
>>> 
```

CODE2:

Doubly Linked List:

```
#Program to implement Doubly Linked 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 DlinkedList:

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

    def _len_(self):
        return self.size

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

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

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

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

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

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

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

    def 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 get_node_at(self,index):
        element_node = self.head
        counter = 0
        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_prev_node_at(self,position):
        if position == 0:
            print('No previous element')
            return None
        return self.get_node_at(position).previous

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

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

```
    def search(self,search_value):
        index = 0
        while (index < self.size):
            value = self.get_node_at(index)
            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


#List 1
print('List 1')
list1 = DlinkedList()
list1.add_head(2)
list1.add_head(1)
list1.add_tail(3)
list1.add_tail(4)
list1.add_tail(5)
list1.add_tail(7)
list1.display()
list1.remove_head()
list1.remove_tail()
print('Head and Tail elements removed')
list1.display()

print('Added element between the list')
list1.add_between_list(2,6)
list1.display()

print('Removed element from between the list')
list1.remove_between_list(2)
list1.display()
```

```
#List 2
list2 = DlinkedList()
list2.add_head(7)
list2.add_head(6)
list2.add_tail(8)
list2.add_tail(9)
print('List 2')
list2.display()

#merging lists
list1.merge(list2)
print('List after merging')
list1.display()
```

OUTPUT2:

```
Python 3.9.0 (tags/v3.9.0:9cf6752, Oct  5 2020, 15:34:40) [MSC v.1927 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
============== RESTART: C:\DIKSHITA\SUBJECTS\DS\DS PRACS\prac2.py ==============
List 1
1
2
3
4
5
7
Removing
Head and Tail elements removed
2
3
4
5
Added element between the list
2
3
6
4
5
Removed element from between the list
2
3
4
5
List 2
6
7
8
9
List after merging
2
3
4
5
6
7
8
9
>>>
```

<u>**PRACTICAL3**</u>

AIM: Implement the following for Stack:

THEORY:

Data structures are the key to organize storage in computers so that we can efficiently access and edit data. Stacks is one of the earliest data structures defined in computer science. In simple words, Stack is a linear collection of items. It is a collection of objects that supports fast last-in, first-out (LIFO) semantics for insertion and deletion. It is an array or list structure of function calls and parameters used in modern computer programming and CPU architecture. Similar to a stack of plates at a restaurant, elements in a stack are added or removed from the top of the stack, in a “last in, first out” order. Unlike lists or arrays, random access is not allowed for the objects contained in the stack.

There are two types of operations in Stack-

- **Push** – To add data into the stack.
- **Pop** – To remove data from the stack.

Stacks are simple to learn and easy to implement, they are extensively incorporated in many software for carrying out various tasks. They can be implemented with an Array or Linked List.

<u>**PRACTICAL 3A**</u>

AIM: Perform Stack operations using Array implementation.

THEORY: In this program we are going to perform stack operations like push, pop and also top which returns the topmost element in the stack using array.

CODE:

```
#Performing Stack operations

class Arraystack:

    def __init__(self):
        self._data = [2, 4, 6, 8, 10]

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

    def is_empty(self):
        if len(self._data) == 0:
            return 0

        else:
            return 1

    def push(self, value):
        self._data.append(value)
        print(self._data,' element added')

    def pop(self):
        if self.is_empty() == 1:
            print('element removed which is ', self._data.pop())
            print(self._data)

        else:
            print('The stack is empty')

    def top(self):
        if self.is_empty() == 1:
            print('The topmost element in the stack is: ', self._data[-1])

        else:
            print('The stack is empty')

c = Arraystack()
c.push(12)
c.push(14)
c.push(16)
c.display()
c.pop()
c.top()
```

OUTPUT:

```
Python 3.9.0 (tags/v3.9.0:9cf6752, Oct  5 2020, 15:34:40) [MSC v.1927 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
=========== RESTART: C:\DIKSHITA\SUBJECTS\DS\DS PRACS\Practical_3A.py ==========
[2, 4, 6, 8, 10, 12]  element added
[2, 4, 6, 8, 10, 12, 14]  element added
[2, 4, 6, 8, 10, 12, 14, 16]  element added
[2, 4, 6, 8, 10, 12, 14, 16]
element removed which is  16
[2, 4, 6, 8, 10, 12, 14]
The topmost element in the stack is:  14
>>>
```

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:

- Only one disk can be moved at a time.
- 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.
- No disk may be placed on top of a smaller disk.

CODE:

```
#Tower of Hanoi program
from_rod = 'A'
to_rod = 'C'
using_rod = 'B'

def hanoi(n, from_rod, to_rod, using_rod):
    if n > 0:
        hanoi(n-1, from_rod, using_rod, to_rod)
        print('Move disk from ', from_rod, ' to ', to_rod)
        hanoi(n-1, using_rod, to_rod, from_rod)

disks = int(input("Enter number of disks: "))
hanoi(disks, from_rod, to_rod, using_rod)
```

OUTPUT:

```
Python 3.9.0 (tags/v3.9.0:9cf6752, Oct  5 2020, 15:34:40) [MSC v.1927 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
=========== RESTART: C:\DIKSHITA\SUBJECTS\DS\DS PRACS\Practical_3B.py ==========
Enter number of disks: 2
Move disk from  A  to  B
Move disk from  A  to  C
Move disk from  B  to  C
>>> 
```

PRACTICAL 3C

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

THEORY: In this program we are going to take two polynomials using Linked List and will add those two polynomials.

CODE:

```
# Program to scan a polynomial using linked list and add two polynomials.
class Node:

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

    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")
        first = self.head
        print(first.element.element)
        first = first.next
        while first:
            print(first.element)
            first = first.next

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

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

    def remove_head(self):
        if self.is_empty():
            print("Empty Singly linked list")
        else:
            print("Removing")
            self.head = self.head.next
            self.size -= 1

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

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

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

```
order = 3

list1 = LinkedList()
print("Polynomial 1:")

list1.add_head(Node(int(input(f"coefficient for power {order} : "))))
for i in reversed(range(order)):
    list1.add_tail(int(input(f"coefficient for power {i} : ")))

list2 = LinkedList()
print("Polynomial 2")

list2.add_head(Node(int(input(f"coefficient for power {order} : "))))
for i in reversed(range(order)):
    list2.add_tail(int(input(f"coefficient for power {i} : ")))

print("Adding coeficients of polynomial 1 and 2: ")
print(list1.get_node_at(0).element + list2.get_node_at(0).element,'x^3 + ',
         list1.get_node_at(1).element + list2.get_node_at(1).element,'x^2 + ',
         list1.get_node_at(2).element + list2.get_node_at(2).element,'x^1 + ',
         list1.get_node_at(3).element + list2.get_node_at(3).element)
```

OUTPUT:

```
Python 3.9.0 (tags/v3.9.0:9cf6752, Oct  5 2020, 15:34:40) [MSC v.1927 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
=========== RESTART: C:\DIKSHITA\SUBJECTS\DS\DS PRACS\Practical_3C.py ==========
Polynomial 1:
coefficient for power 3 : 2
coefficient for power 2 : 3
coefficient for power 1 : 1
coefficient for power 0 : 6
Polynomial 2
coefficient for power 3 : 1
coefficient for power 2 : 3
coefficient for power 1 : 2
coefficient for power 0 : 6
Adding coeficients of polynomial 1 and 2:
3 x^3 +  6 x^2 +  3 x^1 +  12
>>>
```

PRACTICAL 3D

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

THEORY: The factorial of a number is the product of all the integers from 1 to that number. Factorial is not defined for negative numbers, and the factorial of zero is one.
In this program we are going to find factorial of a number by using recursion and iteration.
Recursion is when a statement in a function calls itself repeatedly.
The iteration is when a loop repeatedly executes until the controlling condition becomes false. The primary difference between recursion and iteration is that recursion is a process, always applied to a function and iteration is applied to the set of instructions which we want to get repeatedly executed.

CODE:

```
##Program to calculate factorial and to compute the factors of the given number using recursion and iteration
#factorial program using recursion
def factorial_r(n):
    if n < 0:
        return 0

    if (n == 1 or n == 0):
        return 1

    else:
        return n * factorial_r(n - 1)

num = int(input("Enter a number for finding it's factorial using recursion: "))
print("Factorial of ", num, "using recursion is ", factorial_r(num))

#Factorial Program using iteration
def factorial_i(n):
    if n < 0:
        return 0

    elif n == 0 or n == 1:
        return 1

    else:
        fact = 1
        while(n > 1):
            fact = fact * n
            n -= 1
        return fact

num = int(input("Enter a number for finding it's factorial using iteration: "))
print("Factorial of", num, "using iteration is", factorial_i(num))

#Finding factor of number using recursion
def factors_r(n, i):
    if (i <= n):
        if (n % i == 0):
            print(i, end = " ")
        factors_r(n, i + 1)

num = int(input("\nEnter a number for finding it's factors using recursion: "))
print("The factors of", num, "are:")
factors_r(num, 1)

#Finding factor of number using iteration
def factors_i(n):
   print("The factors of",n,"are:")
   for i in range(1, n + 1):
       if n % i == 0:
           print(i, end = " ")

num = int(input("Enter a number for finding it's factors using iteration: "))
factors_i(num)
```

OUTPUT:

```
Python 3.9.0 (tags/v3.9.0:9cf6752, Oct  5 2020, 15:34:40) [MSC v.1927 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
=========== RESTART: C:\DIKSHITA\SUBJECTS\DS\DS PRACS\Practical_3D.py ==========
Enter a number for finding it's factorial using recursion: 5
Factorial of  5 using recursion is  120
Enter a number for finding it's factorial using iteration: 5
Factorial of 5 using iteration is 120

Enter a number for finding it's factors using recursion: 60
The factors of 60 are:
1 2 3 4 5 6 10 12 15 20 30 60
Enter a number for finding it's factors using iteration: 60
The factors of 60 are:
1 2 3 4 5 6 10 12 15 20 30 60
>>> |
```

PRACTICAL 4

AIM: Perform Queues operations using Circular Array implementation.

THEORY: A queue is a linear data structure where an item can be inserted from one end and can be removed from another end. We cannot insert and remove items from the same end.

One end of the queue is called a front and the other end is called a rear. Items are inserted in the rear end and are removed from the front end. A queue is called a First In First Out data structure because the item that goes in first comes out first.

Circular Queue is also a linear data structure, which follows the principle of FIFO(First In First Out), but instead of ending the queue at the last position, it again starts from the first position after the last, hence making the queue behave like a circular data structure.

In this program we are going to implement Circular Queue.

CODE:

```
#Performing Queue operations using Circular Array implementation
class ArrayQueue:

    '''FIFO queue implementation using a Python list as underlying storage.'''
    DEFAULT_CAPACITY = 6

    # moderate capacity for all new queues
    def __init__ (self):
        '''Create an empty queue.'''
        self._data = [None] * ArrayQueue.DEFAULT_CAPACITY
        self._size = 0
        self._front = 0

    def display(self):
        return self._data

    def __len__(self):
        return self._size

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

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

    def dequeue(self):
        if self.is_empty():
            raise Exception( "Queue is empty" )
        answer = self._data[self._front]
        self._data[self._front] = None
        self._front = (self._front + 1) % len(self._data)
        self._size -= 1
        return answer

    def enqueue(self, e):
        if self._size == len(self._data):
            self._resize(2  * len(self._data)) # double the array size
        avail = (self._front + self._size) % len(self._data)
        self._data[avail] = e
        self._size += 1
```

```python
    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


c = ArrayQueue()
c.enqueue(1)
c.enqueue(2)
c.enqueue(3)
c.enqueue(4)
c.enqueue(5)
c.enqueue(6)
print(c.display())
c.dequeue()
c.dequeue()
c.dequeue()
print(c.display())
c.enqueue(1)
print(c.display())
c.enqueue(2)
print(c.display())
c.enqueue(3)
print(c.display())
```

OUTPUT:

```
Python 3.9.0 (tags/v3.9.0:9cf6752, Oct  5 2020, 15:34:40) [MSC v.1927 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
============== RESTART: C:\DIKSHITA\SUBJECTS\DS\DS PRACS\Prac4.py ==============
[1, 2, 3, 4, 5, 6]
[None, None, None, 4, 5, 6]
[1, None, None, 4, 5, 6]
[1, 2, None, 4, 5, 6]
[1, 2, 3, 4, 5, 6]
>>> |
```

PRACTICAL 5

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

THEORY: Linear search checks for a value within an array one by one until it finds in. In an unordered array, linear search would have to check the entire array until it found the desired value. But ordered arrays, it is different. The reason is once linear search finds a value larger than its desired value, then it can stop and say it found the value or not.

Binary search basically takes the value you are looking for and goes to the middle of the ordered array. It now thinks if the desired value is greater or lesser than the middle value. If higher, binary search goes to that middle and asks higher or lower again, which goes on until it finds the desired value. The same applies to a lower value. The important thing to remember is binary search can only happen with an ordered array. If it was unordered, binary search could not ask the higher or lower value to speed up the search.
In this program we are going to find an element in the list using Linear and Binary search.

CODE:

```
#Program to search an element in the list using Linear and Binary search
list_values = [2, 4, 6, 8, 10]
def binary_search(list_values, search, start, end):
    if start > end:
        return -1
    mid = (start + end) // 2

    if list_values[mid] == search:
        return mid

    elif search < list_values[mid]:
        return binary_search(list_values, search, start, mid - 1)

    else:
        return binary_search(list_values, search, mid - 1, end)

def linear_search(list_values, search):
    index_counter = 0
    list_size = len(list_values)
    while index_counter < list_size:
        temp_value = list_values[index_counter]
        if temp_value == search:
            return index_counter
        index_counter += 1
    return -1

#Taking input from the user
ip = int(input("Enter 1 for binary search OR \nEnter 2 for linear search on list:"))
if ip == 1:
    print(binary_search(list_values, 6, 0, len(list_values)))
else:
    print(linear_search(list_values, 6))
```

OUTPUT:

```
============== RESTART: C:\DIKSHITA\SUBJECTS\DS\DS PRACS\prac5.py ==============
Enter 1 for binary search OR
Enter 2 for linear search on list:1
2
>>> |
```

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: Insertion sort works similarly as we sort cards in our hand in a card game. We assume that the first card is already sorted then, we select an unsorted card. If the unsorted card is greater than the card in hand, it is placed on the right otherwise, to the left. In the same way, other unsorted cards are taken and put at their right place. A similar approach is used by insertion sort. Insertion sort is a sorting algorithm that places an unsorted element at its suitable place in each iteration.
Selection sort is an algorithm that selects the smallest element from an unsorted list in each iteration and places that element at the beginning of the unsorted list.
Bubble sort is an algorithm that compares the adjacent elements and swaps their positions if they are not in the intended order. The order can be ascending or descending.

CODE:

```
#Program to implement Selection, Insertion and Bubble sort on the given list.
list_students_rolls = [ 19, 27, 62, 24, 21, 2, 51]

#Selection sort
def select_sort(list_students_rolls):
    for i in range(len(list_students_rolls)):
        min_val_index = i
        for j in range(i + 1, len(list_students_rolls)):
            if list_students_rolls[min_val_index] > list_students_rolls[j]:
                min_val_index = j
        list_students_rolls[i], list_students_rolls[min_val_index] = list_students_rolls[min_val_index], list_students_rolls[i]
    print("Selection Sort: ", list_students_rolls)

#Insertion sort
def insert_sort(list_students_rolls):
    for i in range(len(list_students_rolls)):
        value = list_students_rolls[i]
        j = i - 1

        while j >= 0 and value == list_students_rolls[j]:
            list_students_rolls[j + 1] = list_students_rolls[j]
            j -= 1

        list_students_rolls[j + 1] = value
    print("Insertion Sort: ", list_students_rolls)

#Bubble sort
def bubble_sort(list_students_rolls):

    for i in range(len(list_students_rolls)):

        for j in range(len(list_students_rolls) - 1):
            if list_students_rolls[j] > list_students_rolls[j+1]:
                list_students_rolls[j], list_students_rolls[j+1] = list_students_rolls[j+1], list_students_rolls[j]
    print("Bubble Sort: ", list_students_rolls)

#Taking input from the user
ip = int(input("Enter 1 to perform selection sort,\nEnter 2 to perform Insertion sort, \nEnter 3 to perform Bubble sort: "))
if ip == 1:
    select_sort(list_students_rolls)
elif ip == 2:
    insert_sort(list_students_rolls)
else:
    bubble_sort(list_students_rolls)
```

OUTPUT:

```
Python 3.9.0 (tags/v3.9.0:9cf6752, Oct  5 2020, 15:34:40) [MSC v.1927 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
============== RESTART: C:\DIKSHITA\SUBJECTS\DS\DS PRACS\prac6.py ==============
Enter 1 to perform selection sort,
Enter 2 to perform Insertion sort,
Enter 3 to perform Bubble sort: 2
Insertion Sort:  [19, 27, 62, 24, 21, 2, 51]
>>>
```

PRACTICAL 7

AIM: Implement the following for Hashing:

THEORY: Hashing is the process of mapping large amount of data item to smaller table with the help of hashing function.

Hashing is also known as Hashing Algorithm or Message Digest Function.

It is a technique to convert a range of key values into a range of indexes of an array.

It is used to facilitate the next level searching method when compared with the linear or binary search.

Hashing allows to update and retrieve any data entry in a constant time O(1).

Constant time O(1) means the operation does not depend on the size of the data.

Hashing is used with a database to enable items to be retrieved more quickly.

It is used in the encryption and decryption of digital signatures.

PRACTICAL 7A

AIM: Write a program to implement the collision technique.

THEORY: Hash functions are there to map different keys to unique locations (index in the hash table), and any hash function which is able to do so is known as the perfect hash function. Since the size of the hash table is very less comparatively to the range of keys, the perfect hash function is practically impossible. What happens is, more than one keys map to the same location and this is known as a collision. A good hash function should have less number of collisions.

Collision resolution is finding another location to avoid the collision. The most popular resolution techniques are,

- Separate chaining
- Open addressing

Open addressing can be further divided into,

- Linear Probing
- Quadratic Probing
- Double hashing

CODE:

```
#Program to implement the concept of linear probing
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, 19, 1, 85, 2, 80]
    list_of_list_index = [None, None, None, None, None, None]
    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()
        print("hash value for " + str(value) + " is :" + str(list_index))
        if list_of_list_index[list_index]:
            print("Collission detected!")
        else:
            list_of_list_index[list_index] = value

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

OUTPUT:

```
Python 3.9.0 (tags/v3.9.0:9cf6752, Oct  5 2020, 15:34:40) [MSC v.1927 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
=========== RESTART: C:\DIKSHITA\SUBJECTS\DS\DS PRACS\Practical_7A.py ==========
Before: [None, None, None, None, None, None]
hash value for 23 is :5
hash value for 19 is :1
hash value for 1 is :1
Collission detected!
hash value for 85 is :1
Collission detected!
hash value for 2 is :2
hash value for 80 is :2
Collission detected!
After: [None, 19, 2, None, None, 23]
>>>
```

PRACTICAL 7B

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

THEORY: Linear probing is a collision resolving technique in Open Addressed Hash tables. In this method, each cell of a hash table stores a single key–value pair. If a collision is occurred by mapping a new key to a cell of the hash table that is already occupied by another key, this method searches the table for the following closest free location and inserts the new key there.

In this program we are going to use this Linear Probing technique to resolve collision.

CODE:

```
#Program to implement the concept of linear probing
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:
        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 = True

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

        else:
            list_of_list_index[list_index] = value

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

OUTPUT:

```
Python 3.9.0 (tags/v3.9.0:9cf6752, Oct  5 2020, 15:34:40) [MSC v.1927 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
=========== RESTART: C:\DIKSHITA\SUBJECTS\DS\DS PRACS\Practical_7B.py ==========
Before : [None, None, None, None]
hash value for 23 is :3
hash value for 43 is :3
Collission detected for 43
List was full . Could not save
hash value for 1 is :1
hash value for 87 is :3
Collission detected for 87
List was full . Could not save
After: [None, 1, None, 23]
>>>
```

PRACTICAL 8

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

THEORY: Traversal is a process to visit all the nodes of a tree and may print their values too. Because, all nodes are connected via edges (links) we always start from the root (head) node. That is, we cannot randomly access a node in a tree. There are three ways which we use to traverse a tree −

- In-order Traversal
- Pre-order Traversal
- Post-order Traversal

In-order Traversal

In this traversal method, the left subtree is visited first, then the root and later the right sub-tree. We should always remember that every node may represent a subtree itself.

Pre-order Traversal

In this traversal method, the root node is visited first, then the left subtree and finally the right subtree.

Post-order Traversal

In this traversal method, the root node is visited last. First, we traverse the left subtree, then the right subtree and finally the root node.

In this program we are going to implement Binary Tree traversal using In-order, Pre-order and Post-order traversal methods.

A Binary Tree is a data structure where every node has at most two children. We call the topmost node as the Root node.

CODE:

```
#Program for preorder, inorder and postorder traversal of tree
class Node :
    def __init__(self,key):
        self.left = None
        self.right = None
        self.val = key

    def insert(self,data):
       if self.val:
           if data < self.val:
               if self.left is None:
                   self.left = Node(data)
               else:
                    self.left.insert(data)
           elif data>self.val:
                if self.right is None:
                    self.right = Node(data)
                else:
                    self.right.insert(data)
       else :
            self.val = data

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

    def printPreorder(self):
        if self.val:
            print(self.val)
            if self.left:
                self.left.printPreorder()
            if self.right:
                self.right.printPreorder()

    def  printInorder(self):
        if self.val :
            if self.left:
                self.left.printInorder()
            print(self.val)
            if self.right:
                self.right.printInorder()
```

```python
    def printPostorder(self):
        if self.val:
            if self.left:
                self.left.printPostorder()
            if self.right:
                self.right.printPostorder()
            print(self.val)


root1 = Node(None)
root1.insert(60)
root1.insert(1)
root1.insert(4)
root1.insert(12)
root1.insert(100)
root1.insert(90)

print("Node:  ")
root1.PrintTree()

print("Preorder Traversal:  ")
root1.printPreorder()

print("Inorder Traversal:  ")
root1.printInorder()

print("Postorder Traversal:  ")
root1.printPostorder()
```

OUTPUT:

```
============ RESTART: C:\DIKSHITA\SUBJECTS\DS\DS PRACS\Practical_8.py ============
Node:
1
4
12
60
90
100
Preorder Traversal:
60
1
4
12
100
90
Inorder Traversal:
1
4
12
60
90
100
Postorder Traversal:
12
4
1
90
100
60
>>> |
```