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. SACHI VIRESH SHAH

Roll No: 381 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)

NAME: SACHI VIRESH SHAH

Class: S.Y. B.Sc. IT Sem- III **Roll No: 381**

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.	

NAME: SACHI VIRESH SHAH **NEW ROLL NO: 381**

CLASS: SYIT **OLD ROLL NO: 3066**

Practical 1(a)

AIM: Implement the following for Array:

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:

A one-dimensional **array** is a type of linear **array**. Accessing its elements involves a single subscript which can either represent a row or column index.

Searching- Searching is an operation or a technique that helps finds the place of a given element or value in the list. **Binary search-** It is 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.

Sorting- Sorting refers to arranging data in a particular format. Sorting algorithm specifies the way to arrange data in a particular order. Most common orders are in numerical or lexicographical order. In Numpy, we can perform various sorting operations using the various functions that are provided in the library like sort, lexsort, argsort etc.

numpy.sort() : This function returns a sorted copy of an array.

Merging- To merge array elements we have to copy first array's elements into third array first then copy second array's elements into third array after the index of first array elements.

Reversing- Every list in Python has a built-in reverse() method you can call to reverse the contents of the list object in-place. Reversing the list in-place means won't create a new list and copy the existing elements to it in reverse order. Instead, it directly modifies the original list object.

NAME: SACHI VIRESH SHAH **NEW ROLL NO: 381**

CLASS: SYIT **OLD ROLL NO: 3066**

CODE:

practical1a.py - C:\Users\Dhwani\Desktop\sachiit\sem3_bscit\DataStructure\prac\practical1a.py (3.5.3rc

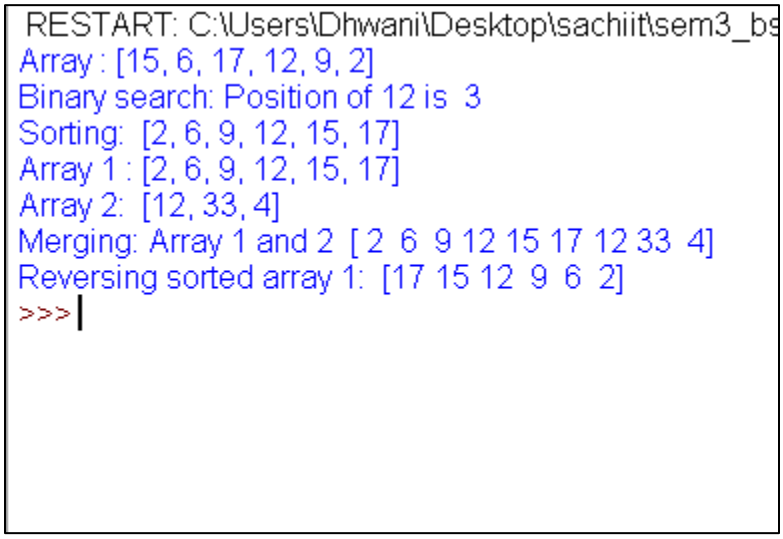
File Edit Format Run Options Window Help

```
# 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.

import numpy as np


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

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

def merge(arr, arr2):
    merged_list = np.concatenate((arr,arr2))
    return merged_list

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

arr = [15, 6, 17, 12, 9, 2]
arr2 = [12, 33, 4]
print("Array :",arr)
print("Binary search: Position of 12 is ", binary_search(arr, 12, 0, len(arr)))
print("Sorting: ",sorting(arr))
print("Array 1 :",arr)
print("Array 2: ",arr2)
print("Merging: Array 1 and 2 ",merge(arr,arr2))
print("Reversing sorted array 1: ",rev())
```

NAME: SACHI VIRESH SHAH **NEW ROLL NO: 381**

CLASS: SYIT **OLD ROLL NO: 3066**

OUTPUT:



NAME: SACHI VIRESH SHAH **NEW ROLL NO: 381**

CLASS: SYIT **OLD ROLL NO: 3066**

Practical 1(b)

AIM: Implement the following for Array:

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

THEORY:

Matrix Addition:

To add the matrices, we simply traverse through both matrices element by element and insert the smaller element (one with smaller row and col value) into the resultant matrix. If we come across an element with the same row and column value, we simply add their values and insert the added data into the resultant matrix.

Matrix Multiplication:

To multiply the matrices, we first calculate transpose of the second matrix to simplify our comparisons and maintain the sorted order. So, the resultant matrix is obtained by traversing through the entire length of both matrices and summing the appropriate multiplied values.

Matrix Transpose:

To transpose a matrix, we can simply change every column value to the row value and vice-versa, however, in this case, the resultant matrix won't be sorted as we require. Hence, we initially determine the number of elements less than the current element's column being inserted in order to get the exact index of the resultant matrix where the current element should be placed. This is done by maintaining an array index [] whose ith value indicates the number of elements in the matrix less than the column i.

NAME: SACHI VIRESH SHAH **NEW ROLL NO: 381**

CLASS: SYIT **OLD ROLL NO: 3066**

CODE:

Practical1b_test.py - C:\Users\Dhwani\Desktop\sachiit\sem3_bscit\D

File Edit Format Run Options Window Help

```
# Write a program to perform the Matrix addition,
# Multiplication and Transpose Operation.

def add(X, Y):

    result = [[0,0,0], [0,0,0], [0,0,0]]

    for i in range(len(X)):
        for j in range(len(X[0])):
            result[i][j] = X[i][j] + Y[i][j]
    for r in result:
        print(r)


def mul(X, Y):

    result = [[0,0,0], [0,0,0], [0,0,0]]

    for i in range(len(X)):              # rows of X
        for j in range(len(Y[0])):       # cols of Y
            for k in range(len(Y)):      # rows of Y
                result[i][j] += X[i][k] * Y[k][j]
    for r in result:
        print(r)


def transpose(matrix):

    result = [[0,0,0], [0,0,0], [0,0,0]]

    for i in range(len(matrix)):             # rows
        for j in range(len(matrix[0])):      # columns
            result[j][i] = matrix[i][j]
    for r in result:
        print(r)
```

```
X = [[1,7,3],
     [4 ,3,6],
     [7 ,8,5]]
Y = [[5,8,1],
     [6,7,3],
     [4,5,9]]

print("X = ",X)
print("Y = ",Y)

print("Addition")
add(X, Y)
print("Multiplication")
mul(X, Y)
print("Transpose of X")
transpose(X)
print("Transpose of Y")
transpose(Y)
```

OUTPUT:

```
 RESTART: C:\Users\Dhwani\Desktop\sachii
X =  [[1, 7, 3], [4, 3, 6], [7, 8, 5]]
Y =  [[5, 8, 1], [6, 7, 3], [4, 5, 9]]
Addition
[6, 15, 4]
[10, 10, 9]
[11, 13, 14]
Multiplication
[59, 72, 49]
[62, 83, 67]
[103, 137, 76]
Transpose of X
[1, 4, 7]
[7, 3, 8]
[3, 6, 5]
Transpose of Y
[5, 6, 4]
[8, 7, 5]
[1, 3, 9]
>>>
```

NAME: SACHI VIRESH SHAH **NEW ROLL NO: 381**

CLASS: SYIT **OLD ROLL NO: 3066**

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 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

- Head points to the first node of the linked list
- Next pointer of the last node is NULL, so if the next current node is NULL, we have reached the end of the linked list.

There are three common types of Linked List.

- Singly Linked List
- Doubly Linked List
- Circular Linked List

To Traverse a Linked List

Displaying the contents of a linked list is very simple. We keep moving the temp node to the next one and display its contents. When temp is NULL, we know that we have reached the end of the linked list so we get out of the while loop.

To insert Elements to a Linked List

You can add elements to either the beginning, middle or end of the linked list.

- **Add to the beginning**
  A. Allocate memory for new node

B. Store data
C. Change next of new node to point to head
D. Change head to point to recently created node

- **Add to the End**
  A. Allocate memory for new node
  B. Store data
  C. Traverse to last node
  D. Change next of last node to recently created node

To Delete from a Linked List

You can delete either from the beginning, end or from a particular position.

- **Delete from beginning**
  A. Point head to the second node
- **Delete from end**
  A. Traverse to second last element
  B. Change its next pointer to null
- **Delete from middle**
  A. Traverse to element before the element to be deleted
  B. Change next pointers to exclude the node from the chain

NAME: SACHI VIRESH SHAH **NEW ROLL NO: 381**

CLASS: SYIT **OLD ROLL NO: 3066**

CODE:

practical2.py - C:\Users\Dhwani\Desktop\sachiit\sem3_bscit\DataStructure\prac\practical2.py

File Edit Format Run Options Window Help

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

class Node1:

    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):
#         return "Displaying list"
        if self.size ==0:
            print("no element")

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

practical2.py - C:\Users\Dhwani\Desktop\sachiit\sem3_bscit\DataStructur

File Edit Format Run Options Window Help

```
    def add_head(self,e):
        temp = self.head
        self.head = Node1(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")
        else:
            print("removing head")
            self.head =  self.head.next
            self.size -=1

    def add_tail(self, e):
        new_value = Node1(e)
        self.get_tail().next = new_value
        self.size += 1

    def find_second_last_el(self):
        #second_last_el = None

        if self.size >= 2:
            first = self.head
            temp_counter = self.size - 1
            while temp_counter > 1:
                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:
            Node1 = self.find_second_last_el()
            if Node1:
                Node1.next = None
                self.size -= 1

    def get_Node1_at(self, index):
        el_Node1 = self.head
        counter = 0
        if index > self.size-1:
            print("index out of bound")
            return None
        while(counter < index):
            el_Node1 = el_Node1.next
            counter +=1
        return el_Node1

    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_Node1 = self.get_Node1_at(position - 1)
        next_Node1 = self.get_Node1_at(position + 1)
        prev_Node1.next = next_Node1
        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_Node1 = self.get_Node1_at(position - 1)
            current_Node1 = self.get_Node1_at(position)
            prev_Node1.next = element
            element.next = current_Node1
            self.size -= 1

    def search(self, search_value):
        index = 0
        while(index < self.size):
            value = self.get_Node1_at(index)
            print("searching at "+ str(index)+ " and val is "+str(value.element))
            if value.element == search_value:
                print("found")
                return True
            index +=1
        print("not found")
        return False

    def merge(self, linkedlist_value):
        if self.size > 0:
            last_Node1 = self.get_Node1_at(self.size - 1)
            last_Node1.next = linkedlist_value.head
```

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

print("Linked List\n")
print("\n1 .  Singly linked list\n")
sy = LinkedList()
print("adding elements\n")
sy.add_head(6)
sy.add_head(5)
sy.add_head(4)
sy.add_tail(3)
sy.add_tail(2)
sy.add_tail(1)
print("final list\n")
sy.display()
sy.remove_head()
print("\nfinal list\n")
sy.display()
sy.remove_tail()
print("\nremoving tail : final list\n")
sy.display()
print("\nget node at 1 \n")
sy.get_Node1_at(1).display()
print("\nremove between list :1 \n")
sy.remove_between_list(0)
print("\nfinal list\n")
sy.display()
print("\nsearching 3 in list\n")
sy.search(3)
print("\nadding elements in list2\n")
fy = LinkedList()
fy.add_head(22)
fy.add_head(55)
```

```
fy.add_tail(11)
print("\nfinal list\n")
fy.display()
sy.merge(fy)
print("merging 2 lists\n")
sy.display()



# doubly linkedlist

class Node:

    def __init__(self, data):
        self.item = data
        self.next = None
        self.prev = None

class DoublyLinkedList:

    def __init__(self):
        self.head = None

    def add_head(self, data):
        if self.head is None:
            new_node = Node(data)
            self.head = new_node
            print("node inserted")
            return
        else:
            new_node = Node(data)
            new_node.next = self.head
            self.head.prev = new_node
            self.head = new_node
```

```
    def add_tail(self, data):
        if self.head is None:
            new_node = Node(data)
            self.head = new_node
            return
        else:
            n = self.head
            while n.next is not None:
                n = n.next
            new_node = Node(data)
            n.next = new_node
            new_node.prev = n
            new_node.next = None

    def traverse_list(self):      # same as display() function
        if self.head is None:
            print("List is empty")
            return
        else:
            n = self.head
            while n is not None:
                print(n.item)
                n = n.next

    def remove_head(self):
        if self.head is None:
            print("List is empty")
            return
        elif self.head.next is None:
            self.head = None
            return
        else:
            self.head = self.head.next
```

```
    def remove_tail(self):
        if self.head is None:
            print("List is empty")
            return
        elif self.head.next is None:
            self.head = None
            return
        else:
            n = self.head
            while n.next is not None:
                n = n.next
            n.prev.next = None

    def reverse_linked_list(self):
        if self.head is None:
            print("List is empty")
            return
        else:
            p = self.head
            q = p.next
            p.next = None
            p.prev = q
            while q is not None:
                q.prev = q.next
                q.next = p
                p = q
                q = q.prev
            self.head = p

print("\n2.  Doubly linked list\n")
print("\nadding elements\n")
l1 = DoublyLinkedList()
l1.add_head('b')
l1.add_head('c')
```

```
l1.add_tail('d')
print("\nfinal list\n")

l1.traverse_list()
print("\nremoving head: final list\n")

l1.remove_head()
l1.traverse_list()

print("\nremoving tail : final list\n")
l1.remove_tail()

l1.traverse_list()

print("\nadding elements\n")
l1.add_head('c')
l1.add_head('e')
l1.add_tail('f')
print("\nfinal list\n")
l1.traverse_list()

print("\nreversed list\n")
l1.reverse_linked_list()

l1.traverse_list()
```

NAME: SACHI VIRESH SHAH **NEW ROLL NO: 381**

CLASS: SYIT **OLD ROLL NO: 3066**

OUTPUT:

```
 RESTART: C:\Users\Dhwani\Desktop\s
Linked List


1 .  Singly linked list

adding elements

final list

4
5
6
3
2
1
removing head

final list

5
6
3
2
1

removing tail : final list

5
6
3
2

get node at 1

6

remove between list :1

removing head
```

```
final list

6
3
2

searching 3 in list

searching at 0 and val is 6
searching at 1 and val is 3
found

adding elements in list2


final list

55
22
11
merging 2 lists

6
3
2
55
22
11

2.  Doubly linked list


adding elements

node inserted

final list

c
b
d
```

```
removing head: final list

b
d

removing tail : final list

b

adding elements


final list

e
c
b
f

reversed list

f
b
c
e
```

NAME: SACHI VIRESH SHAH **NEW ROLL NO: 381**

CLASS: SYIT **OLD ROLL NO: 3066**

Practical 3(a)

AIM: Implement the following for Stack:

Perform Stack operations using Array implementation.

THEORY:

In array implementation, the stack is formed by using the array. All the operations regarding the stack are performed using arrays

Push and Pop operations in arrays

Push

- Check if the stack is full.
- If the stack is full, then display "Stack overflow".
- If the stack is not full, increment top to the next location.
- Assign data to the top element.

Pop

- Check if the stack is empty.
- If the stack is empty, then display "Stack Underflow".
- If the stack is not empty, copy top in a temporary variable.
- Decrement top to the previous location.
- Delete the temporary variable.

NAME: SACHI VIRESH SHAH **NEW ROLL NO: 381**

CLASS: SYIT **OLD ROLL NO: 3066**

CODE:

practical3a.py - C:\Users\Dhwani\Desktop\sachiit\sem3_bscit\DataStructure\

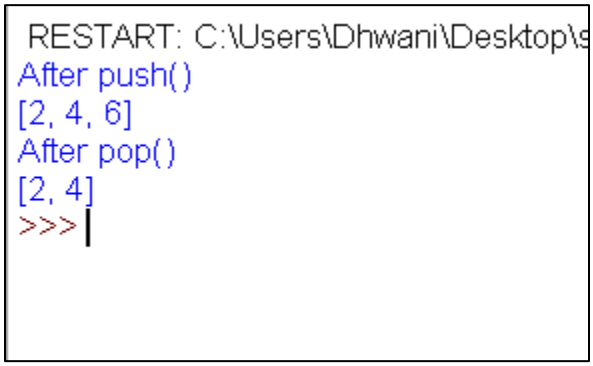
File Edit Format Run Options Window Help

```
# Perform Stack operations using Array implementation.

class ArrayStack:

    def __init__(self):
        self._data = []

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

    def is_empty(self):
        '''checks if stack is empty'''
        return len(self._data) == 0

    def push(self, value):
        self._data.append(value)

    def pop(self):
        '''pops value - last vala 4 in above list - (-1 element)'''
        if not self.is_empty():
            return self._data.pop()
        else:
            raise Exception('Stack is empty')

a1 = ArrayStack()
a1.push(2)
a1.push(4)
a1.push(6)
print("After push() ")
a1.display()

print("After pop() ")
a1.pop()
a1.display()
```

NAME: SACHI VIRESH SHAH **NEW ROLL NO: 381**

CLASS: SYIT **OLD ROLL NO: 3066**

OUTPUT:



NAME: SACHI VIRESH SHAH **NEW ROLL NO: 381**

CLASS: SYIT **OLD ROLL NO: 3066**

Practical 3(b)

AIM: Implement Tower of Hanoi

THEORY:

Tower of Hanoi is a mathematical puzzle which consists of three towers (pegs) and more than one ring.

These rings are of different sizes and stacked upon in an ascending order, i.e. the smaller one sits over the larger one. There are other variations of the puzzle where the number of disks increase, but the tower count remains the same.

Rules

1. The mission is to move all the disks to some another tower without violating the sequence of arrangement. A few rules to be followed for Tower of Hanoi are −
2. Only one disk can be moved among the towers at any given time.
3. Only the "top" disk can be removed.
4. No large disk can sit over a small disk.
5. Tower of Hanoi puzzle with n disks can be solved in minimum $\mathbf{2^n - 1}$ steps

NAME: SACHI VIRESH SHAH **NEW ROLL NO: 381**

CLASS: SYIT **OLD ROLL NO: 3066**

CODE:

practical3b.py - C:\Users\Dhwani\Desktop\sachiit\sem3_bscit\DataStruc

File Edit Format Run Options Window Help

```
# Implement Tower of Hanoi


def TowerOfHanoi(n , from_, to, middle):
    if n == 1:
        print ("Move disk 1 from rod",from_,"to rod",to)
        return

    TowerOfHanoi(n-1, from_, middle, to)
    print("Move disk",n,"from rod",from_,"to rod",to)
    TowerOfHanoi(n-1, middle, to, from_)

n = 3
TowerOfHanoi(n, 'A', 'C', 'B')
```

OUTPUT:

```
 RESTART: C:\Users\Dhwani\Desktop\s
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
>>>
```

NAME: SACHI VIRESH SHAH **NEW ROLL NO: 381**

CLASS: SYIT **OLD ROLL NO: 3066**

Practical 3(c)

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

THEORY:

Linked list is a data structure that stores each element as an object in a node of the list.
Polynomial is a mathematical expression that consists of variables and coefficients.

For example x^2 - 4x + 7

Summing **two polynomials** simply means to sum the coefficients of the same powers, if these situations occur. At this point, you simply need to notice that: The constant term (i.e. 1) appears only in the first **polynomial**, so we have nothing to sum. The same goes with the linear factor (i.e. x)

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

For adding two polynomials that are stored as a linked list. We need to add the coefficients of variables with the same power. In a linked list node contains 3 members, coefficient value link to the next node.

NAME: SACHI VIRESH SHAH **NEW ROLL NO: 381**

CLASS: SYIT **OLD ROLL NO: 3066**

CODE:

practical3c.py - C:\Users\Dhwani\Desktop\sachiit\sem3_bsc

File Edit Format Run Options Window Help

```
# WAP to scan a polynomial using linked list
# and add two polynomial.
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 find_second_last_element(self):
    if self.size >= 2:
        first = self.head
        temp_counter = self.size -2
        while temp_counter > 0:
            first = first.next
            temp_counter -= 1
        return first
    else:
```

```
            print("Size not sufficient")
        return None

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

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

    def 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)
            if value.element == search_value:
                return value.element
            index += 1
        print("Not Found")
        return False

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

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


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 cooeficients 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 + ",
        list1.get_node_at(3).element + list2.get_node_at(3).element)
```

OUTPUT:

```
Polynomial 1:
coefficient for power 3 : 2
coefficient for power 2 : 3
coefficient for power 1 : 2
coefficient for power 0 : 1
Polynomial 2
coefficient for power 3 : 1
coefficient for power 2 : 1
coefficient for power 1 : 1
coefficient for power 0 : 4
Adding cooeficients of polynomial 1 and 2
3 x^3 +  4 x^2 +  3 x +  5
```

NAME: SACHI VIRESH SHAH **NEW ROLL NO: 381**

CLASS: SYIT **OLD ROLL NO: 3066**

Practical 3(d)

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

(i) using recursion

(ii) Using iteration.

THEORY:

Factorial of a given number

If you want to find a factorial of an positive integer, keep multiplying it with all the positive integers less than that number. The final result that you get is the Factorial of that number.

- **Using recursion**

1. User must enter a number and store it in a variable.
2. The number is passed as an argument to a recursive factorial function.
3. The base condition is that the number has to be lesser than or equal to 1 and return 1 if it is
4. Otherwise the function is called recursively with the number minus 1 multiplied by the number itself.
5. The result is returned and the factorial of the number is printed.

- **Using iteration.**

We shall use Python For Loop to find Factorial. For a **range (1, n+1)** of given n, multiply the element over each iteration and return the result after coming out of the loop.

NAME: SACHI VIRESH SHAH **NEW ROLL NO: 381**

CLASS: SYIT **OLD ROLL NO: 3066**

CODE:

practical3d.py - C:\Users\Dhwani\Desktop\sachiit\sem3_bscit\DataStructure\prac\practical3d

File Edit Format Run Options Window Help

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

def recursive_factorial(n):
    if n == 0 or n == 1:
        return 1
    return n * recursive_factorial(n - 1)

def iteration_factorial(n):
    fact=1
    while(n>0):
        fact=fact*n
        n=n-1
    return fact

def iteration_factors(x):
   print("The factors of ",x," are: ")
   for i in range(1, x + 1):
       if x % i == 0:
           print(i)

def recursive_factors(n, i):
    if (i <= n):
        if (n % i == 0):
            print(i, end = " ")
        recursive_factors(n, i + 1)

print('Factorial of 5 using recursive function: ', recursive_factorial(5))
print('Factorial of 6 using iteration function: ', iteration_factorial(6))
print("\nFactors by iteration :")
iteration_factors(6)
print("Factors by recursion")
recursive_factors(8,1)
```

NAME: SACHI VIRESH SHAH **NEW ROLL NO: 381**

CLASS: SYIT **OLD ROLL NO: 3066**

OUTPUT:

```
 RESTART: C:\Users\Dhwani\Desktop\sachiit\sem3_bs
Factorial of 5 using recursive function:  120
Factorial of 6 using iteration function:  720

Factors by iteration :
The factors of  6  are:
1
2
3
6
Factors by recursion
1 2 4 8
```

NAME: SACHI VIRESH SHAH **NEW ROLL NO: 381**

CLASS: SYIT **OLD ROLL NO: 3066**

Practical 4

AIM: Perform Queues operations using Circular Array implementation.

THEORY:

Circular Array 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.

1. Initialize the queue, size of the queue (maxSize), head and tail pointers
2. Enqueue:

- Check if the number of elements (size) is equal to the size of the queue (maxSize):
    a. If yes, throw error message "Queue Full!"
    b. If no, append the new element and increment the tail pointer.

3. Dequeue:

- Check if the number of elements (size) is equal to 0:
    a. If yes, throw error message "Queue Empty!"
    b. If no, increment head pointer.

4. is_empty (): Returns True if queue is empty, False otherwise.
5. Size:

- If tail>=head, size = tail – head
- if head>tail, size = maxSize - (head - tail)

NAME: SACHI VIRESH SHAH **NEW ROLL NO: 381**

CLASS: SYIT **OLD ROLL NO: 3066**

CODE:

practical4.py - C:\Users\Dhwani\Desktop\sachiit\sem3_bscit\DataStructure\prac\prac

File Edit Format Run Options Window Help

```
# Perform Queues operations using Circular Array implementation.

class ArrayQueue:

    default_len = 5

    def __init__ (self):
        self._data = [None] * ArrayQueue.default_len
        self._size = 0
        self._front = 0

    def __len__ (self):
        return self._size

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

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

    def 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))
        avail = (self._front + self._size) % len(self._data)
        self._data[avail] = e
        self._ size += 1
```

```
    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

a = ArrayQueue()
a.enqueue(1)
a.enqueue(12)
print('After enqueue: ', a._data)

print('Deque : ',a.dequeue())
print('After Deque')
print(a._data)

print('Size : ',a._size)

a.enqueue(34)
a.enqueue(2)

print('After enqueue: ', a._data)

print('Size : ',a._size)

print('Deque : ',a.dequeue())
print('After Deque')
print(a._data)

print('Size : ',a._size)
```

OUTPUT:

```
After enqueue:  [1, 12, None, None, None]
Deque :  1
After Deque
[None, 12, None, None, None]
Size :  1
After enqueue:  [None, 12, 34, 2, None]
Size :  3
Deque :  12
After Deque
[None, None, 34, 2, None]
Size :  2
```

NAME: SACHI VIRESH SHAH **NEW ROLL NO: 381**

CLASS: SYIT **OLD ROLL NO: 3066**

Practical 5

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

THEORY:

A. Linear Search

- We start searching a list for a particular value from the first item in the list.
- We move from item to item from the first item to the last item in the list.
- If the desired item/value is found in the list then the position of that item in the list is returned/printed.
- If the desired item/value is not found in the list then we conclude that the desired item was not present in the list.
- The output result will be the position of the searched element/item in the list. In programming, the counting of a list or array starts from 0. So, the first element of a list is said to be in 0th position. The second element of the list is said to be in the 1st position and so on.

B. Binary Search

- Binary search algorithm doesn't go on searching in a sequential manner like in linear search.
- It directly moves to the middle item and compares it with the searched item/value.
- If the compared item is same as the searched item then the algorithm terminates.
- Otherwise, if the compared item is greater than the searched item then we can skip entire second half (right half) of the list of the compared item.
- If the compared item is less than the searched item then we can skip entire first half (left half) of the list from the compared item.
- After that, we repeat the above process again for one of the halves of the list.
- This algorithm runs until the sub-list is reduced to zero, i.e. until there is no possibility of dividing the list into two halves.

NAME: SACHI VIRESH SHAH **NEW ROLL NO: 381**

CLASS: SYIT **OLD ROLL NO: 3066**

CODE:

practical5.py - C:\Users\Dhwani\Desktop\sachiit\sem3_bscit\DataStructure\prac\practical5.py (3.5

File Edit Format Run Options Window Help

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


def linear_search(mylist, el):
    for i in mylist:
        if i == el:
            return mylist.index(i)
    return -1


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


mylist = [1, 2, 5, 7, 13, 15]
print(mylist)
n = input("searching for 5 \nEnter l for linear search and b for binary search: ")
el = 5
if n=='l':
    print(linear_search(mylist, el))
elif n=='b':
    print(binary_search(mylist, el, 0, len(mylist)))
else:
    print("invalid input")
```

OUTPUT:

```
 RESTART: C:\Users\Dhwani\Desktop\sachiit\sem3
[1, 2, 5, 7, 13, 15]
searching for 5
Enter l for linear search and b for binary search: b
2
>>>
```

NAME: SACHI VIRESH SHAH **NEW ROLL NO: 381**

CLASS: SYIT **OLD ROLL NO: 3066**

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:

A. Bubble Sort

- Bubble Sort is a simple sorting algorithm that compares each element in the list with its adjacent element and sorts that pair of elements if they are not in order.
- If there are n items in the list then there will be (n-1) pairs of items that need to be compared in the first pass.
- After the first pass, the largest element of the list will be placed in its proper place.
- Now, in the second pass, there will be (n-1) items left for sorting. This means that there will be (n-2) pairs of items that need to be compared in the second pass.
- In the second pass, the second largest element of the list will be placed in its proper place.
- In this way, other elements of the list are sorted.

B. 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 sub-arrays in a given array.
    i. The sub-array which is already sorted.
    ii. Remaining sub-array which is unsorted.
- In every iteration of selection sort, the minimum element (considering ascending order) from the unsorted sub-array is picked and moved to the sorted sub-array.

C. 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.
- Iterate over the input elements by growing the sorted array at iteration.
- Compare the current element with the largest value available in the sorted array.
- If the current element is greater, then it leaves the element in its place and moves on to the next element else it finds its correct position in the sorted array and moves it to that position in the array.
- This is achieved by shifting all the elements towards the right, which are larger than the current element, in the sorted array to one position ahead.

NAME: SACHI VIRESH SHAH **NEW ROLL NO: 381**

CLASS: SYIT **OLD ROLL NO: 3066**

CODE:

practical6.py - C:\Users\Dhwani\Desktop\sachiit\sem3_bscit\DataStructure\prac\prac

File Edit Format Run Options Window Help

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


# Selection sort

def Selection_sort():
    mylist = [19, 27, 62, 24, 21, 2 ,51]

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


# Insertion sort

def Insertion_sort():
    mylist = [19, 27, 62, 24, 21, 2 ,51]

    print(mylist)
    for i in range(len(mylist)):
        val = mylist[i]
        j = i-1
        while j >= 0 and val < mylist[j]:
            mylist[j+1] = mylist[j]
            j -= 1
        mylist[j+1] = val
    return mylist
```

```
# Bubble sort

def Bubble_sort():
    mylist = [19, 27, 62, 24, 21, 2 ,51]

    print(mylist)
    for i in range(len(mylist)):
        for j in range(len(mylist) - 1):
            if mylist[j] > mylist[j+1]:
                mylist[j], mylist[j+1] = mylist[j+1], mylist[j]
    return mylist


n = input("Enter s : selection sort , i : Insertion sort, b : bubble sort : ")
if n=='s':
    print("After selection sort: ", Selection_sort())
elif n=='i':
    print("After insertion sort: ", Insertion_sort())
elif n=='b':
    print("After bubble sort: ",Bubble_sort())
else:
    print("invalid input")
```

OUTPUT:

```
 RESTART: C:\Users\Dhwani\Desktop\sachiit\sem3_bscit\D
Enter s : selection sort , i : Insertion sort, b : bubble sort : s
[19, 27, 62, 24, 21, 2, 51]
After selection sort:  [2, 19, 21, 24, 27, 51, 62]
>>>
```

NAME: SACHI VIRESH SHAH **NEW ROLL NO: 381**

CLASS: SYIT **OLD ROLL NO: 3066**

Practical 7 (a)

AIM: Implement the following for Hashing:

Write a program to implement the collision technique.

THEORY:

Hashing:

Hashing is the process of converting a given key into another value. A hash function is used to generate the new value according to a mathematical algorithm. The result of a hash function is known as a hash value or simply, a hash

Collision:

Since a hash function gets us a small number for a key which is a big integer or string, there is a possibility that two keys result in the same value. The situation where a newly inserted key maps to an already occupied slot in the hash table is called collision and must be handled using some collision handling technique.

- A collision occurs when two items or values get the same slot or index, i.e. the hashing function generates same slot number for multiple items.
- If proper collision resolution steps are not taken then the previous item in the slot will be replaced by the new item whenever the collision occurs.

There are mainly two methods to handle collision:

1) Separate Chaining:

2) Open Addressing

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

NAME: SACHI VIRESH SHAH **NEW ROLL NO: 381**

CLASS: SYIT **OLD ROLL NO: 3066**

CODE:

practical7a.py - C:\Users\Dhwani\Desktop\sachiit\sem3_bscit\DataStruct

File Edit Format Run Options Window Help

```
# Implement the following for Hashing:
# a. Write a program to implement the collision technique.

size_list = 6

def search_from_hash(key, hash_list):
    searched_index = hash_function(key)
    if hash_list[searched_index] == key:
        print("value found")
    else:
        print("value not in list")

def hash_function(value):
    global size_list
    return value%size_list

def map_hash2index(hash_return_value):
    return hash_return_value

def create_hash_table(list_values, main_list):
    for value in list_values:
        hash_return_value = hash_function(value)
        list_index = map_hash2index(hash_return_value)
        if main_list[list_index]:
            print("collision detected")
        else:
            main_list[list_index] = value

list_values = [1,3,4,5,8,60]
main_list = [None for x in range(size_list)]
print(main_list)
create_hash_table(list_values, main_list)
print(main_list)
search_from_hash(30, main_list)
```

OUTPUT:

```
[None, None, None, None, None, None]
[60, 1, 8, 3, 4, 5]
value not in list
```

NAME: SACHI VIRESH SHAH **NEW ROLL NO: 381**

CLASS: SYIT **OLD ROLL NO: 3066**

Practical 7 (b)

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

THEORY:

- One way to resolve collision is to find another open slot whenever there is a collision and store the item in that open slot.
- The search for open slot starts from the slot where the collision happened.
- It moves sequentially through the slots until an empty slot is encountered.
- The movement is in a circular fashion. It can move to the first slot while searching for an empty slot. Hence, covering the entire hash table.
- Linear probing is a simple open-addressing hashing strategy.
- This kind of sequential search is called Linear Probing.
- **Linear probing** is a scheme in computer programming for resolving collisions in hash tables, data structures for maintaining a collection of key–value pairs and looking up the value associated with a given key.

NAME: SACHI VIRESH SHAH **NEW ROLL NO: 381**

CLASS: SYIT **OLD ROLL NO: 3066**

CODE:

practical7b.py - C:\Users\Dhwani\Desktop\sachiit\sem3_bscit\DataStructure\prac\practic

File Edit Format Run Options Window Help

```
# Implement the following for Hashing:
# b. Write a 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)

linear_probing = True
list_of_keys = [23,43,1,87]
list_of_list_index = [None,None,None,None]
print("Before : " + str(list_of_list_index))

for value in list_of_keys:
    #print(Hash(value,0,len(list_of_keys)).get_key_value())
    list_index = Hash(value,0,len(list_of_keys)).get_key_value()
    print("hash value for " + str(value) + " is :" + str(list_index))

    if list_of_list_index[list_index]:
        print("Collission detected for " + str(value))
        if linear_probing:
            old_list_index = list_index
            if list_index == len(list_of_list_index)-1:
                list_index = 0
            else:
                list_index += 1
            list_full = False
```

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

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

OUTPUT:

```
 RESTART: C:\Users\Dhwani\Desktop\sa
Before : [None, None, None, None]
hash value for 23 is :3
hash value for 43 is :3
Collission detected for 43
hash value for 1 is :1
hash value for 87 is :3
Collission detected for 87
After: [43, 1, 87, 23]
```

NAME: SACHI VIRESH SHAH **NEW ROLL NO: 381**

CLASS: SYIT **OLD ROLL NO: 3066**

Practical 8

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

THEORY:

A. Pre Order Traversal

- In this traversal we first visit root, then left, then right.
- In the below python program, we use the Node class to create place holders for the root node as well as the left and right nodes.
- Then we create an insert function to add data to the tree.
- Finally, the Pre-order traversal logic is implemented by creating an empty list and adding the root node first followed by the left node.
- At last the right node is added to complete the Pre-order traversal. Please note that this process is repeated for each sub-tree until all the nodes are traversed.

B. Inorder traversal.

- In order traversal means visiting first left, then root and then right.
- In the below python program, we use the Node class to create place holders for the root node as well as the left and right nodes.
- Then we create an insert function to add data to the tree.
- Finally, the Inorder traversal logic is implemented by creating an empty list and adding the left node first followed by the root or parent node.
- At last the left node is added to complete the Inorder traversal. Please note that this process is repeated for each sub-tree until all the nodes are traversed.

C. Post Order traversal

- In this traversal we first visit left, then right and then root.
- In the below python program, we use the Node class to create place holders for the root node as well as the left and right nodes.
- Then we create an insert function to add data to the tree.
- Finally, the Post-order traversal logic is implemented by creating an empty list and adding the left node first followed by the right node.
- At last the root or parent node is added to complete the Post-order traversal. Please note that this process is repeated for each sub-tree until all the nodes are traversed.

NAME: SACHI VIRESH SHAH **NEW ROLL NO: 381**

CLASS: SYIT **OLD ROLL NO: 3066**

CODE:

practical8.py - C:\Users\Dhwani\Desktop\sachiit\sem3_

File Edit Format Run Options Window Help

```
# 8. Write a program for inorder, postorder
# and preorder 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()

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

print("After Insert")
root1= Node(None)
root1.insert(20)
root1.insert(4)
root1.insert(13)
root1.insert(130)
root1.insert(130)
root1.insert(123)
root1.printTree()
print("Inorder")
root1.printInOrder()
print("Postorder")
root1.printPostOrder()
print("Preorder")
root1.printPreOrder()
```

NAME: SACHI VIRESH SHAH **NEW ROLL NO: 381**

CLASS: SYIT **OLD ROLL NO: 3066**

OUTPUT:

```
 RESTART: C:\Users\Dhwani\Desk
After Insert
4
13
20
123
130
Inorder
4
13
20
123
130
Postorder
13
4
123
130
20
Preorder
20
4
13
130
123
```