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. __**NIDHEE PANCHAL**________________________________

Roll No: _**378**___ 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)

Nidhee Panchal **Old Roll no.: 3076**

Class: S.Y. B.Sc. IT Sem- III **Roll No: __378____**

Subject: Data Structures

INDEX

Sr. No	**Date**	**Topic**	**Sign**
1	04/09/2020	Implement the following for Array: a) Write a program to store the elements in 1-D array and provide an option to perform the operations like searching, sorting, merging, reversing the elements. b) Write a program to perform the Matrix addition, Multiplication and Transpose Operation.	
2	11/09/2020	Implement Linked List. Include options for insertion, deletion and search of a number, reverse the list and concatenate two linked lists.	
3	18/09/2020	Implement the following for Stack: a) Perform Stack operations using Array implementation. b. b) Implement Tower of Hanoi. c) WAP to scan a polynomial using linked list and add two polynomials. d) WAP to calculate factorial and to compute the factors of a given no. (i) using recursion, (ii) using iteration	
4	25/09/2020	Perform Queues operations using Circular Array implementation.	
5	01/10/2020	Write a program to search an element from a list. Give user the option to perform Linear or Binary search.	
6	09/10/2020	WAP to sort a list of elements. Give user the option to perform sorting using Insertion sort, Bubble sort or Selection sort.	
7	16/10/2020	Implement the following for Hashing: a) Write a program to implement the collision technique. b) Write a program to implement the concept of linear probing.	
8	23/10/2020	Write a program for inorder, postorder and preorder traversal of tree.	

PRACTICAL 1

Implement the following for Array:

PRACTICAL 1 A

Aim:

Write a program to store the elements in 1-D array and provide an option to perform the operations like searching, sorting, merging, reversing the elements.

Theory:

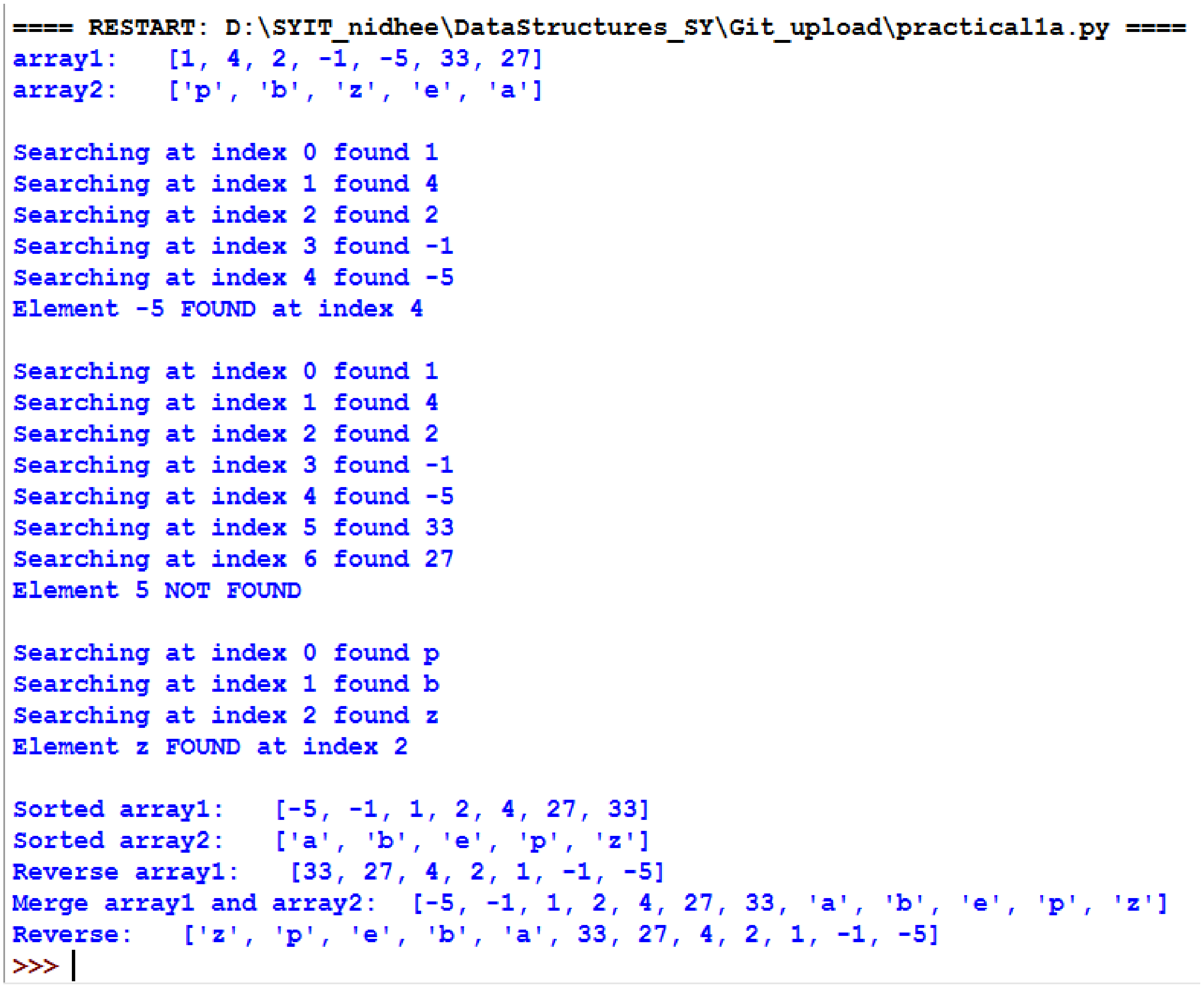
An array is nothing but a variable which can store more than one value at a time. An array can hold many values under a single name, and you can access the values by referring to an index number.

Python provides in-built methods to access, add or remove the elements from the array/list. To access all the list elements we have to iterate each using a loop.

In this program, we have performed operations such as searching any element, sorting all the elements of a list, merge the elements of two different arrays into a single 1-D array and reversing the array.

```
## Practical 1(a)
'''
 Program to store the elements in 1-D array
 and provide operations like
 searching, sorting, merging, reversing the elements
'''

def search_element(my_list,element_to_srch):
    index = 0
    while index < len(my_list):
        print("Searching at index",index,"found",my_list[index])
        if my_list[index] == element_to_srch:
            print("Element",element_to_srch,"FOUND at index",index)
            return False
        index+=1
    print("Element",element_to_srch,"NOT FOUND")

## Bubble Sort
def sort_list(my_list):
    for i in range(len(my_list)-1):
        for j in range((len(my_list)-1)):
            if my_list[j] > my_list[j+1]:
                my_list[j], my_list[j+1] = my_list[j+1], my_list[j]
    return my_list

def merge(list1, list2):
    for i in range(len(list2)):
        list1.append(list2[i])
    return list1
```

```
def reverse_list(my_list):
    my_list = my_list[::-1]
    return my_list

array1 = [1, 4, 2, -1, -5, 33, 27]
array2 = ['p','b','z','e','a']
print("array1:  ",array1)
print("array2:  ",array2,"\n")

search_element(array1,-5)
print()
search_element(array1,5)
print()
search_element(array2,'z')
print()
sort_list(array1)
sort_list(array2)
print("Sorted array1:  ",array1)
print("Sorted array2:  ",array2)
print("Reverse array1:  ",reverse_list(array1))
merge(array1,array2)
print("Merge array1 and array2: ",array1)
print("Reverse:  ",reverse_list(array1))
```



```
==== RESTART: D:\SYIT_nidhee\DataStructures_SY\Git_upload\practical1a.py ====
array1:    [1, 4, 2, -1, -5, 33, 27]
array2:    ['p', 'b', 'z', 'e', 'a']

Searching at index 0 found 1
Searching at index 1 found 4
Searching at index 2 found 2
Searching at index 3 found -1
Searching at index 4 found -5
Element -5 FOUND at index 4

Searching at index 0 found 1
Searching at index 1 found 4
Searching at index 2 found 2
Searching at index 3 found -1
Searching at index 4 found -5
Searching at index 5 found 33
Searching at index 6 found 27
Element 5 NOT FOUND

Searching at index 0 found p
Searching at index 1 found b
Searching at index 2 found z
Element z FOUND at index 2

Sorted array1:    [-5, -1, 1, 2, 4, 27, 33]
Sorted array2:    ['a', 'b', 'e', 'p', 'z']
Reverse array1:    [33, 27, 4, 2, 1, -1, -5]
Merge array1 and array2:   [-5, -1, 1, 2, 4, 27, 33, 'a', 'b', 'e', 'p', 'z']
Reverse:    ['z', 'p', 'e', 'b', 'a', 33, 27, 4, 2, 1, -1, -5]
>>> 
```

PRACTICAL 1 B

Aim:

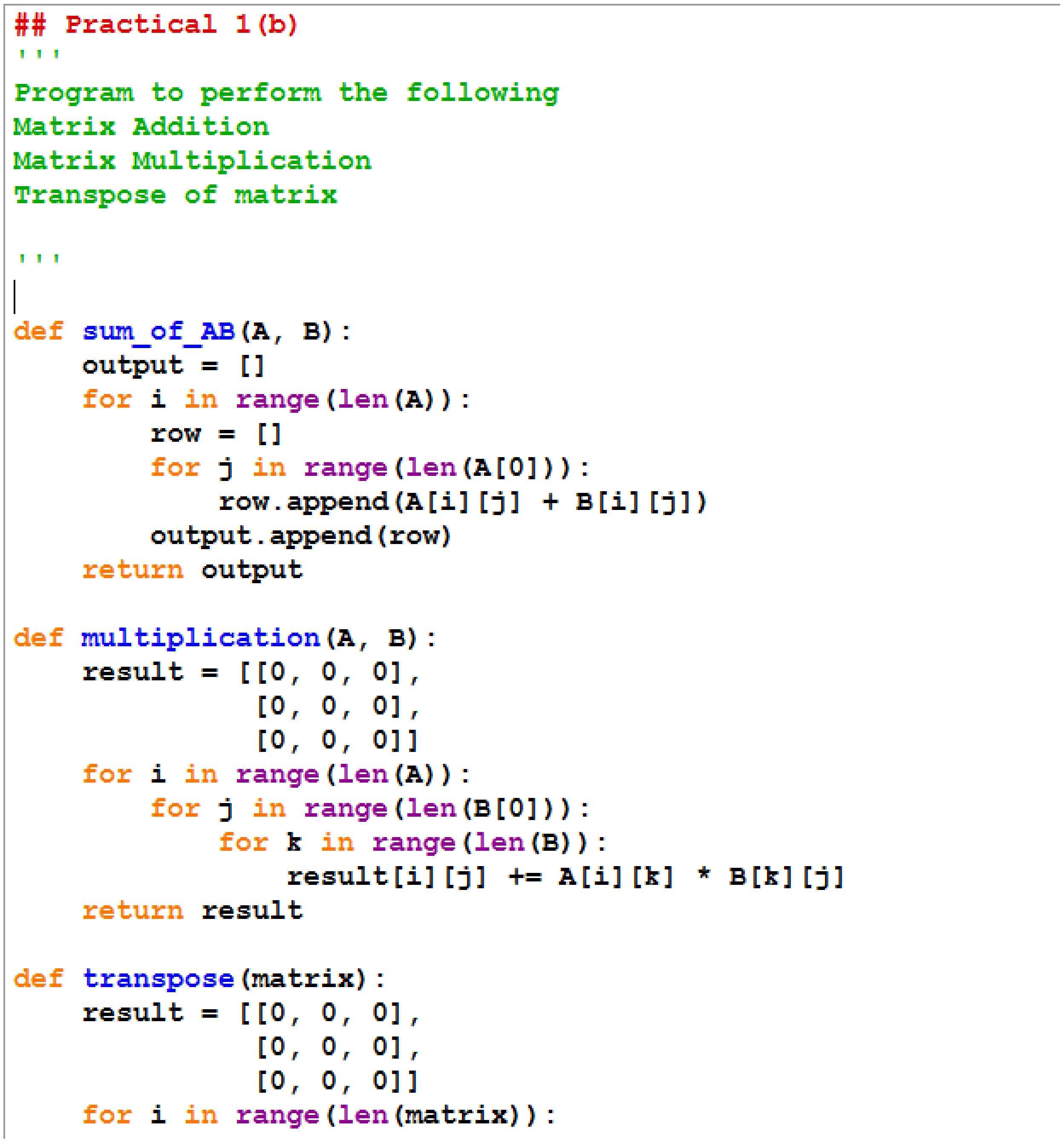
B. Write a program to perform the Matrix addition, Multiplication and Transpose Operation.

Theory:

Matrix is a special case of two dimensional array where each data element is of strictly same size. The horizontal entries in a matrix are called as ‘rows’ while the vertical entries are called as ‘columns’. If a matrix has r number of rows and c number of columns then the order of matrix is given by **r x c**.

In this program we have to use nested for loops to iterate through each row and each column. We accumulate the sum of products in the result

This technique is simple but computationally expensive as we increase the order of the matrix.



```
## Practical 1(b)
'''
Program to perform the following
Matrix Addition
Matrix Multiplication
Transpose of matrix

'''

def sum_of_AB(A, B):
    output = []
    for i in range(len(A)):
        row = []
        for j in range(len(A[0])):
            row.append(A[i][j] + B[i][j])
        output.append(row)
    return output

def multiplication(A, B):
    result = [[0, 0, 0],
              [0, 0, 0],
              [0, 0, 0]]
    for i in range(len(A)):
        for j in range(len(B[0])):
            for k in range(len(B)):
                result[i][j] += A[i][k] * B[k][j]
    return result

def transpose(matrix):
    result = [[0, 0, 0],
              [0, 0, 0],
              [0, 0, 0]]
    for i in range(len(matrix)):
        for j in range(len(matrix[0])):
            result[j][i] = matrix[i][j]
    return result


A = [[1, 5, 1],
     [2, 4, 0],
     [1, 2,-3]]

B = [[0, 1, 2],
     [3, 4, 0],
     [2, 3, 5]]

print("A:  ",A)
print("B:  ",B)

add = sum_of_AB(A,B)
print("\nAddition of matrix A and B:\n",add)
multi = multiplication(A, B)
print("\nMultiplication of A and B:\n",multi)

A_transpose = transpose(A)
print("\nTranspose of matrix A:\n",A_transpose)

B_transpose = transpose(B)
print("\nTranspose of matrix B:\n",B_transpose)
```

Output:

```
==== RESTART: D:\SYIT_nidhee\DataStructures_SY\Git_upload\practical1b.py ====
A:   [[1, 5, 1], [2, 4, 0], [1, 2, -3]]
B:   [[0, 1, 2], [3, 4, 0], [2, 3, 5]]

Addition of matrix A and B:
 [[1, 6, 3], [5, 8, 0], [3, 5, 2]]

Multiplication of A and B:
 [[17, 24, 7], [12, 18, 4], [0, 0, -13]]

Transpose of matrix A:
 [[1, 2, 1], [5, 4, 2], [1, 0, -3]]

Transpose of matrix B:
 [[0, 3, 2], [1, 4, 3], [2, 0, 5]]
>>> 
```

PRACTICAL 2

Aim:

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

Theory:

A **linked list** is a linear collection of data elements whose order is not given by their physical placement in memory.

A linked list is represented by a pointer to the first node of the linked list. The first node is called the head. If the linked list is empty, then the value of the head is NULL.

Each node in a list consists of at least two parts:

1) data

2) Pointer (Or Reference) to the next node

Types of Linked List:

Following are the various types of linked list.

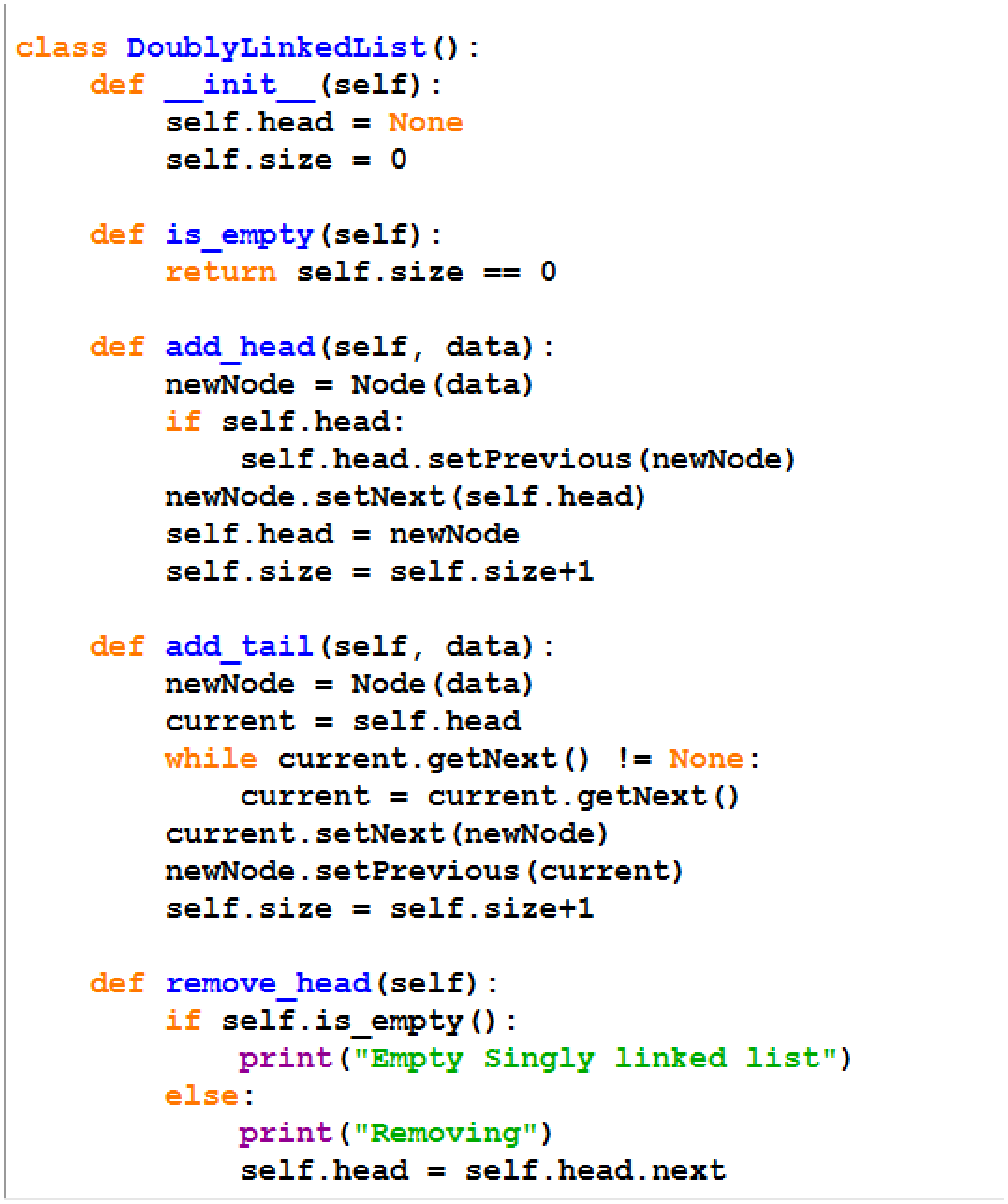
- Simple Linked List − Item navigation is forward only.
- Doubly Linked List − Items can be navigated forward and backward.
- Circular Linked List − Last item contains link of the first element as next and the first element has a link to the last element as previous.

Basic Operations:

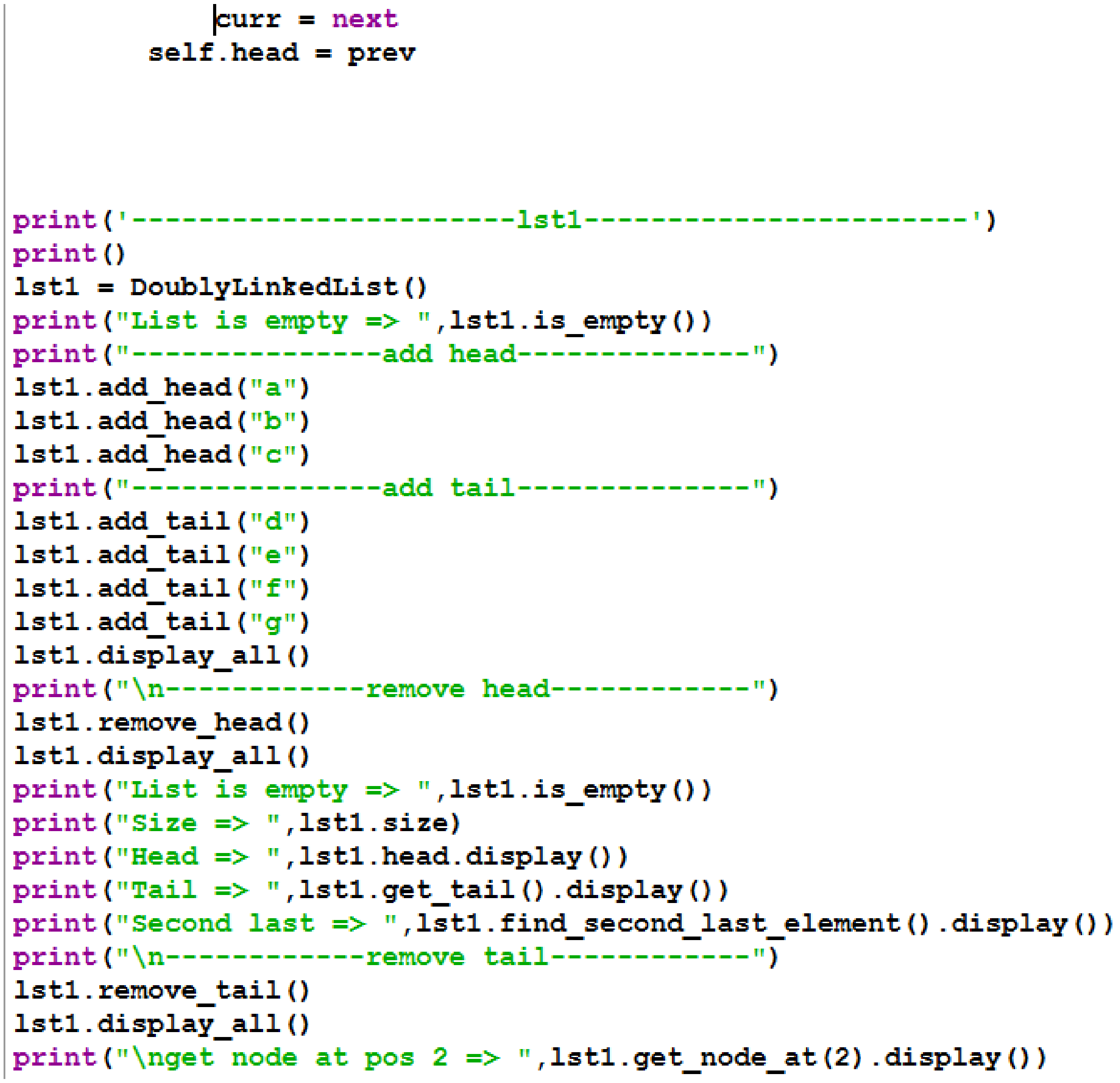
Following are the basic operations supported by a list.

- Insertion − Adds an element at the beginning of the list.
- Deletion − Deletes an element at the beginning of the list.
- Display − Displays the complete list.
- Search − Searches an element using the given key.
- Delete − Deletes an element using the given key.

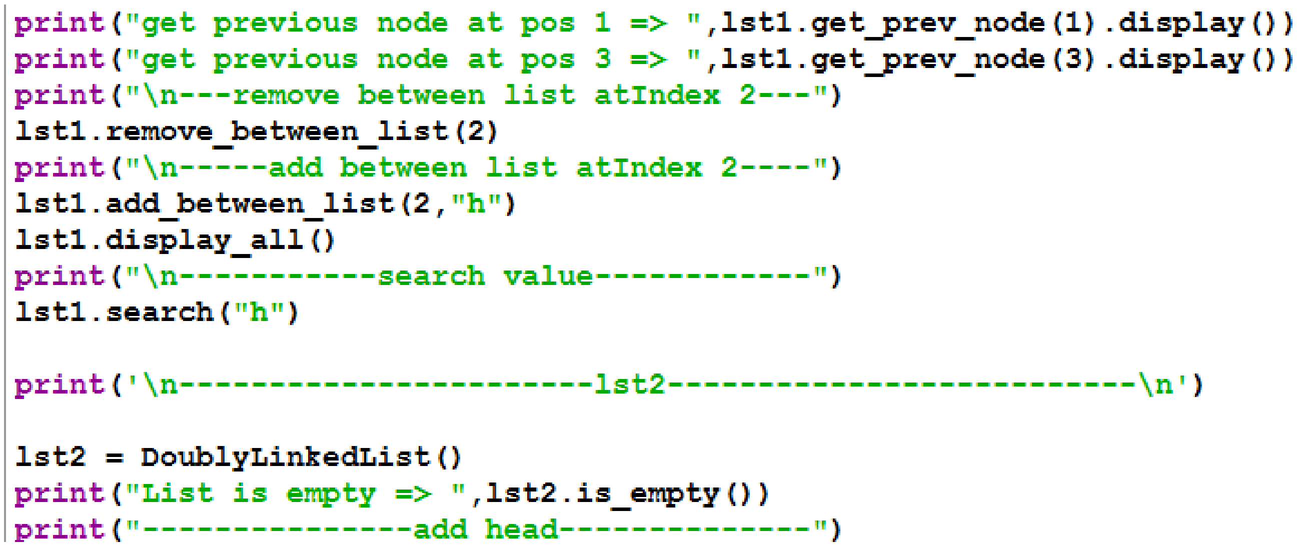
```
## Practical 2
'''
Implementation of doubly linked list
'''

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

    def getNext(self):
        return self.next

    def getPrevious(self):
        return self.previous

    def getData(self):
        return self.data

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

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

    def setPrevious(self, newPrevious):
        self.previous = newPrevious
```



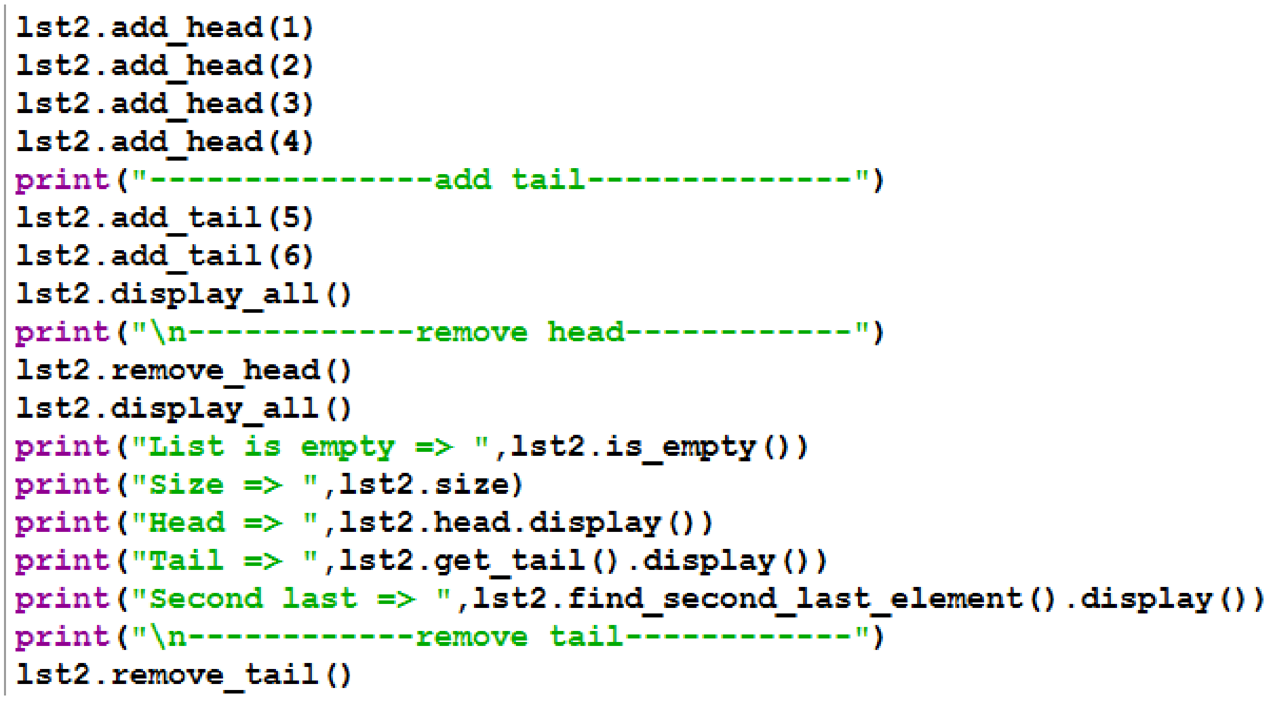
```
class DoublyLinkedList():
    def __init__(self):
        self.head = None
        self.size = 0

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

    def add_head(self, data):
        newNode = Node(data)
        if self.head:
            self.head.setPrevious(newNode)
        newNode.setNext(self.head)
        self.head = newNode
        self.size = self.size+1

    def add_tail(self, data):
        newNode = Node(data)
        current = self.head
        while current.getNext() != None:
            current = current.getNext()
        current.setNext(newNode)
        newNode.setPrevious(current)
        self.size = self.size+1

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

    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(self,pos):
        if pos == 0:
            print('No previous elements')
        return self.get_node_at(pos).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
            self.size -= 1

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

            print(first.data)
            first = first.next



    def find_second_last_element(self):
        #second_last_element = None
        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")

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

    def search(self,search_value):
        index = 0
        while (index < self.size):
            value = self.get_node_at(index)
            print("Searching at " + str(index) + " and value is " + str(value.data))
            if value.data == search_value:
                print("Found value at " + str(index) + " location")
                return True
            index += 1
        print("Not Found")
        return False

    def mergeList(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

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



```
            curr = next
        self.head = prev




print('-----------------------lst1-----------------------')
print()
lst1 = DoublyLinkedList()
print("List is empty => ",lst1.is_empty())
print("---------------add head--------------")
lst1.add_head("a")
lst1.add_head("b")
lst1.add_head("c")
print("---------------add tail--------------")
lst1.add_tail("d")
lst1.add_tail("e")
lst1.add_tail("f")
lst1.add_tail("g")
lst1.display_all()
print("\n------------remove head------------")
lst1.remove_head()
lst1.display_all()
print("List is empty => ",lst1.is_empty())
print("Size => ",lst1.size)
print("Head => ",lst1.head.display())
print("Tail => ",lst1.get_tail().display())
print("Second last => ",lst1.find_second_last_element().display())
print("\n------------remove tail------------")
lst1.remove_tail()
lst1.display_all()
print("\nget node at pos 2 => ",lst1.get_node_at(2).display())
```

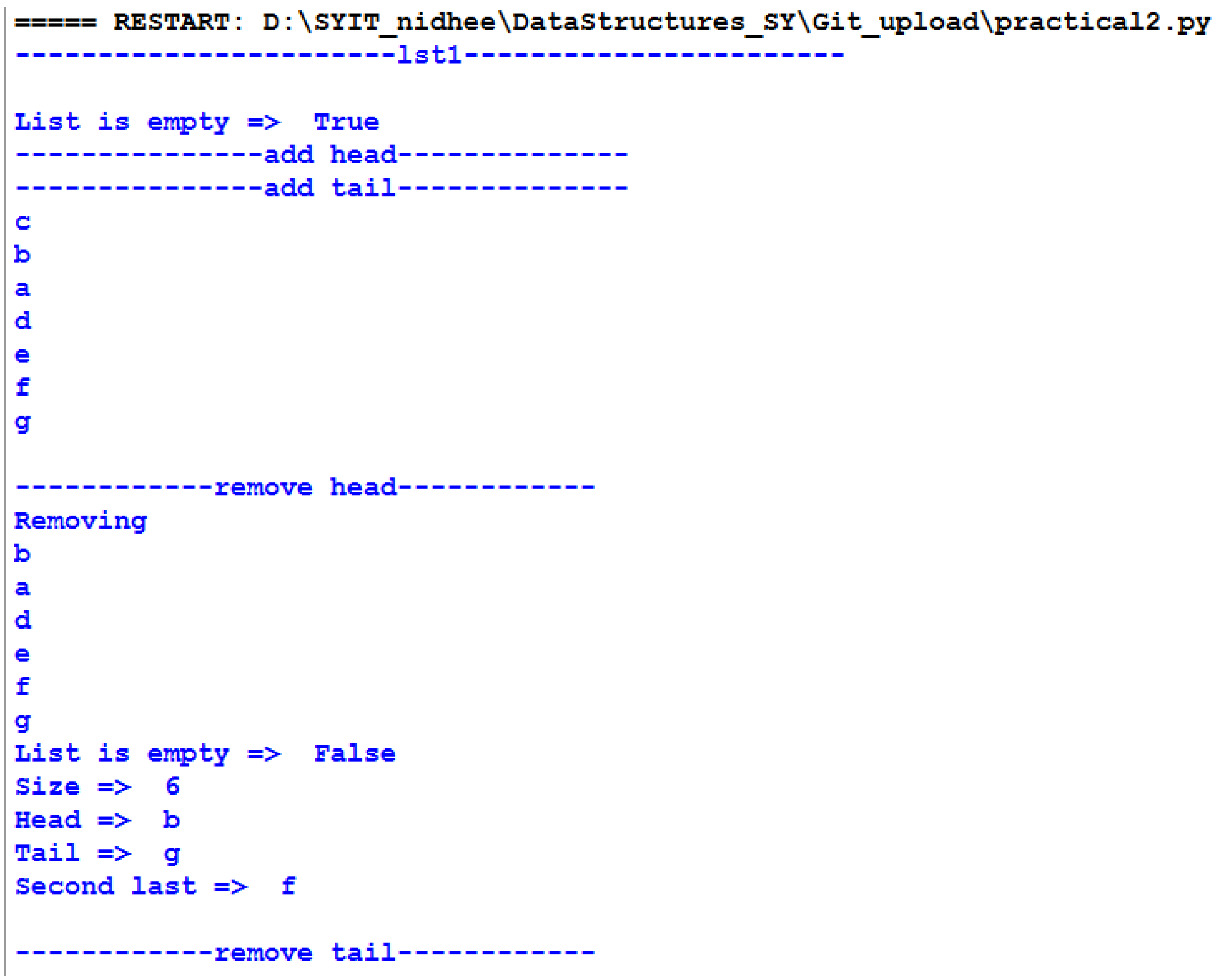


```
print("get previous node at pos 1 => ",lst1.get_prev_node(1).display())
print("get previous node at pos 3 => ",lst1.get_prev_node(3).display())
print("\n---remove between list atIndex 2---")
lst1.remove_between_list(2)
print("\n-----add between list atIndex 2----")
lst1.add_between_list(2,"h")
lst1.display_all()
print("\n-----------search value------------")
lst1.search("h")

print('\n----------------------lst2-------------------------\n')

lst2 = DoublyLinkedList()
print("List is empty => ",lst2.is_empty())
print("---------------add head--------------")
```

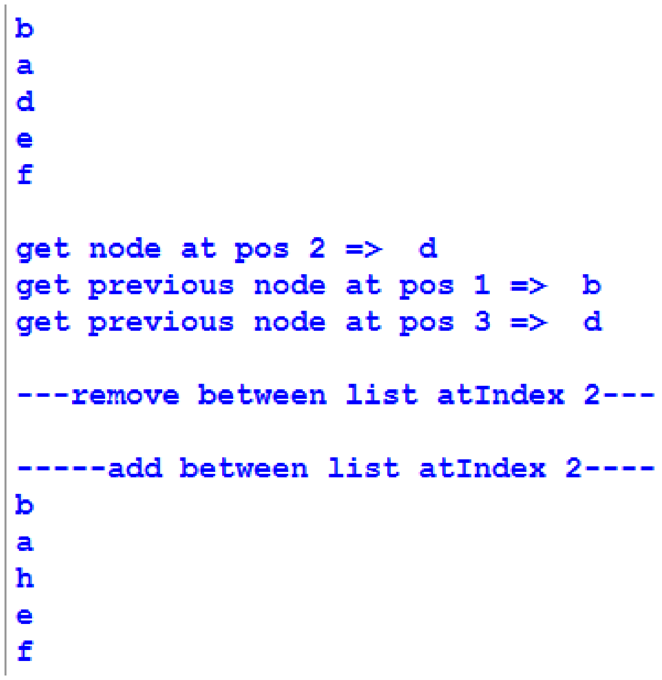


```
lst2.add_head(1)
lst2.add_head(2)
lst2.add_head(3)
lst2.add_head(4)
print("---------------add tail--------------")
lst2.add_tail(5)
lst2.add_tail(6)
lst2.display_all()
print("\n------------remove head------------")
lst2.remove_head()
lst2.display_all()
print("List is empty => ",lst2.is_empty())
print("Size => ",lst2.size)
print("Head => ",lst2.head.display())
print("Tail => ",lst2.get_tail().display())
print("Second last => ",lst2.find_second_last_element().display())
print("\n------------remove tail------------")
lst2.remove_tail()
```

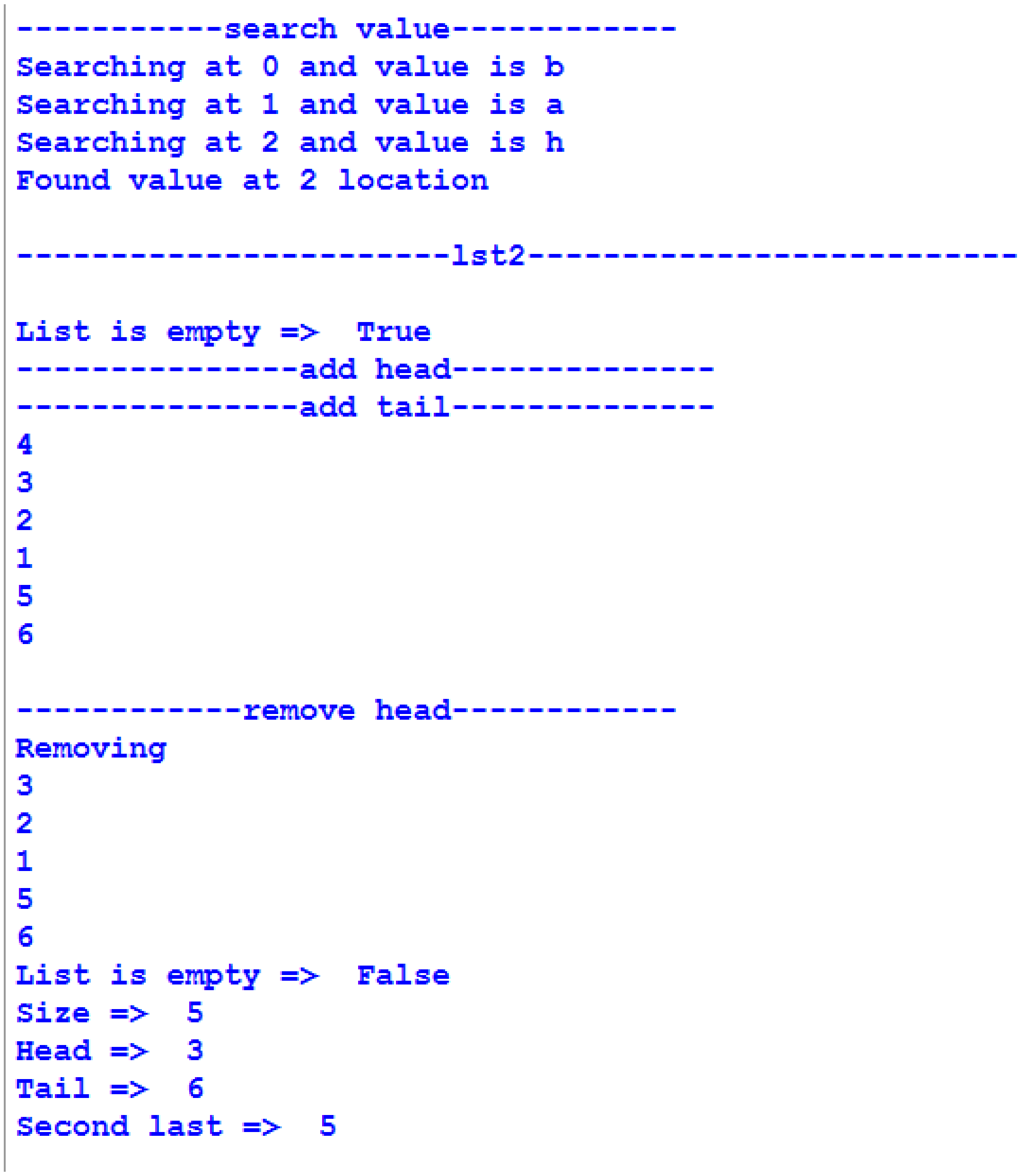
```
lst2.display_all()
print("\nget previous node at pos 1 => ",lst2.get_prev_node(1).display())
print("\nget previous node at pos 3 => ",lst2.get_prev_node(3).display())
print("\nget node at pos 2 => ",lst2.get_node_at(2).display())
print("\n---remove between list atIndex 2---")
lst2.remove_between_list(2)
print("\n-----add between list atIndex 2----")
lst2.add_between_list(2,0)
lst2.display_all()
print("\n-----------search value------------")
lst2.search(1)

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

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



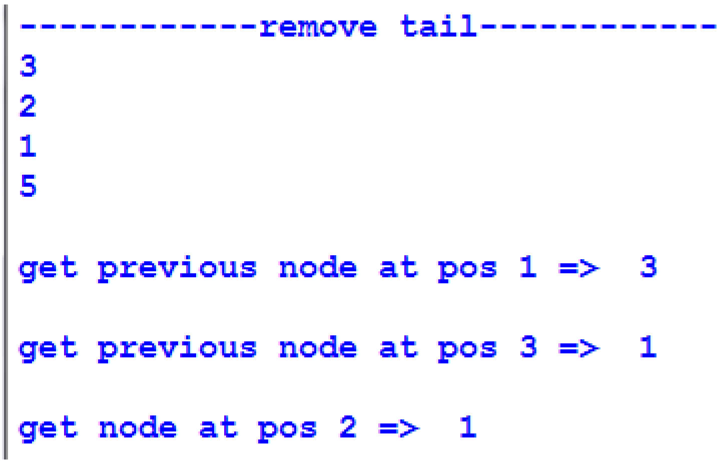
```
===== RESTART: D:\SYIT_nidhee\DataStructures_SY\Git_upload\practical2.py
-----------------------lst1-----------------------

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

------------remove head------------
Removing
b
a
d
e
f
g
List is empty =>  False
Size =>  6
Head =>  b
Tail =>  g
Second last =>  f

------------remove tail------------
```



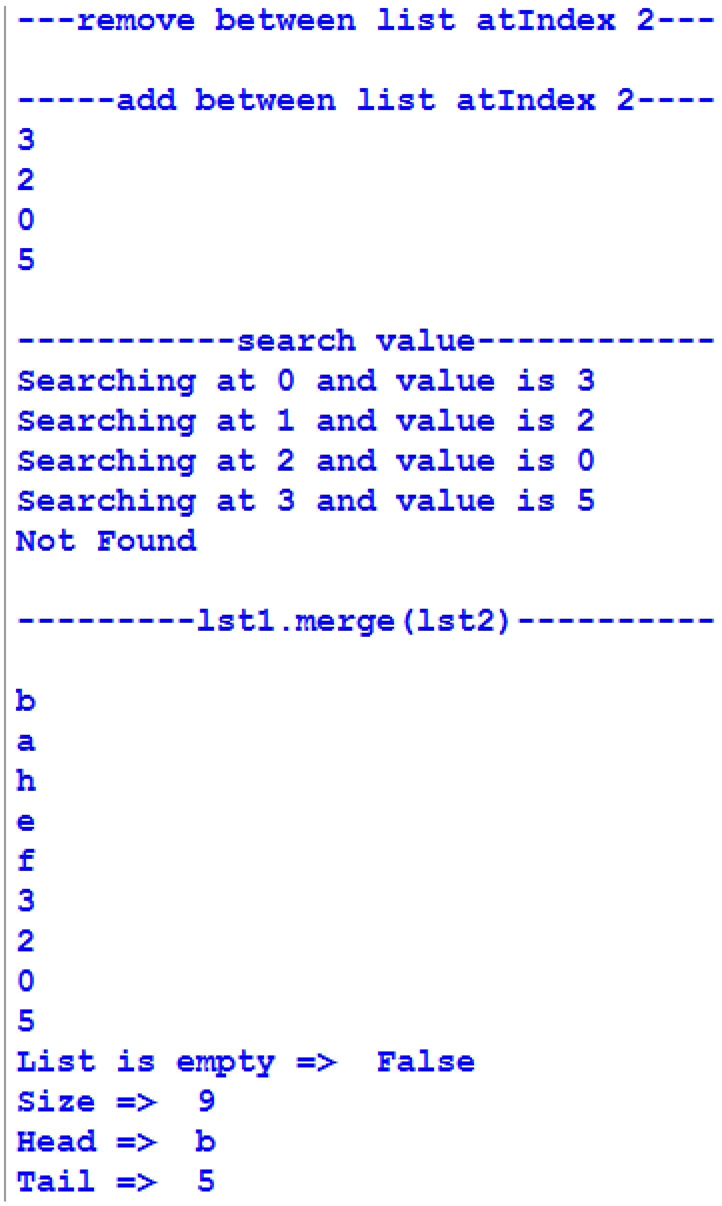
```
b
a
d
e
f

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

---remove between list atIndex 2---

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



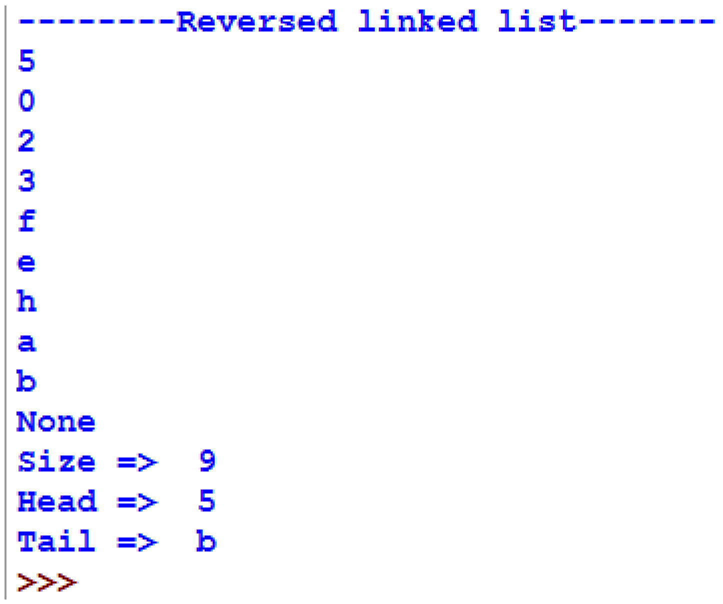
```
-----------search value------------
Searching at 0 and value is b
Searching at 1 and value is a
Searching at 2 and value is h
Found value at 2 location

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

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

------------remove head------------
Removing
3
2
1
5
6
List is empty =>  False
Size =>  5
Head =>  3
Tail =>  6
Second last =>  5
```



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

get previous node at pos 1 =>  3

get previous node at pos 3 =>  1

get node at pos 2 =>  1
```



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

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

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

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

b
a
h
e
f
3
2
0
5
List is empty =>  False
Size =>  9
Head =>  b
Tail =>  5
```



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

PRACTICAL 3

Implement the following for Stack:

Theory:

A stack is an Abstract Data Type (ADT), commonly used in most programming languages. It is named stack as it behaves like a real-world stack, for example – a deck of cards or a pile of plates, etc.

A real-world stack allows operations at one end only. For example, we can place or remove a card or plate from the top of the stack only. Likewise, Stack ADT allows all data operations at one end only. At any given time, we can only access the top element of a stack.

This feature makes it LIFO data structure. LIFO stands for Last-in-first-out. Here, the element which is placed (inserted or added) last, is accessed first.

PRACTICAL 3 A

Aim:

Perform Stack operations using Array implementation.

Theory:

Stack is a linear data structure which follows a particular order in which the operations are performed. The order may be LIFO(Last In First Out) or FILO(First In Last Out).

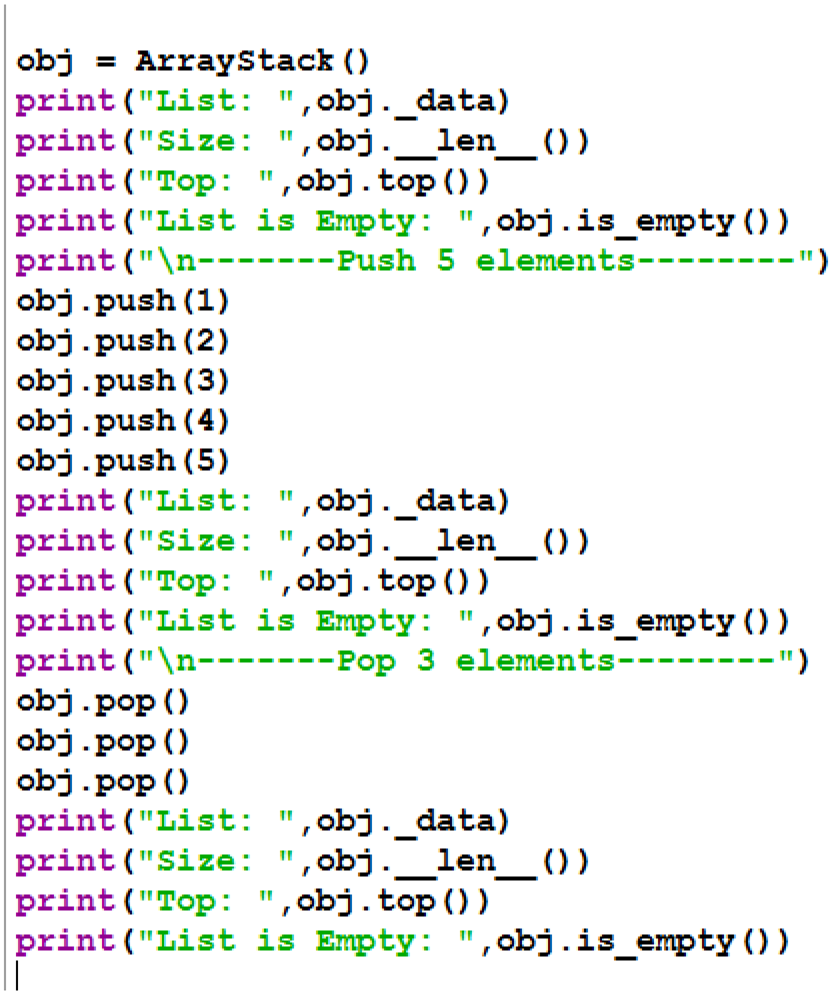
A stack data structure can be implemented using a one-dimensional array. But stack implemented using array stores only a fixed number of data values. This implementation is very simple.

Basic Operations:

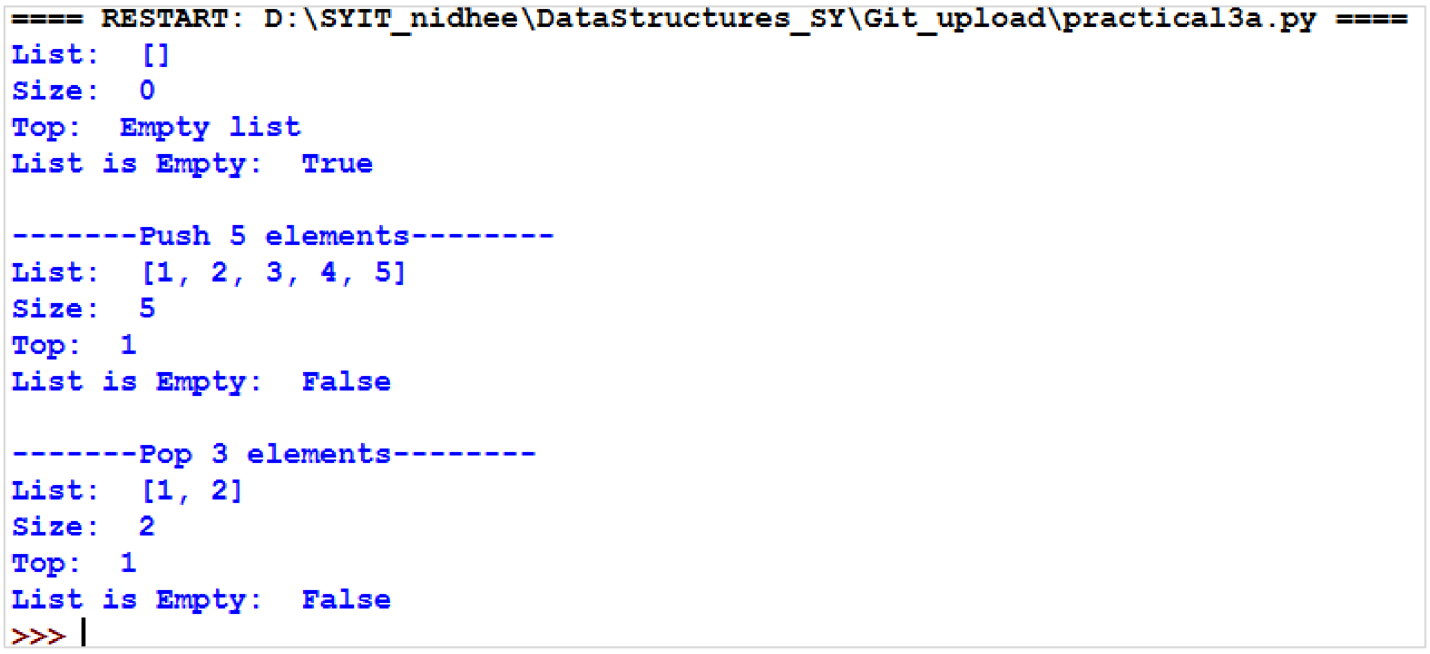
Stack operations may involve initializing the stack, using it and then de-initializing it. Apart from these basic stuffs, a stack is used for the following two primary operations −

- **append()** − Pushing (storing) an element on the stack.
- **pop()** − Removing (accessing) an element from the stack.
- **isEmpty()** – Check if the list is Empty or have some elements in it. If empty it returns a Boolean value True if not then return False.

```
## Practical 3(A)
'''
Perform stack operations using
Array implementation
'''

class ArrayStack:

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

    def __len__(self):
        return len(self._data)

    def is_empty(self):
        if self.__len__() == 0:
            return True
        else:
            return False

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

    def pop(self):
        if self.__len__() == 0:
            return 'Empty list'
        self._data.pop()

    def top(self):
        if self.__len__() == 0:
            return 'Empty list'
        return self._data[0]
```



```
obj = ArrayStack()
print("List: ",obj._data)
print("Size: ",obj.__len__())
print("Top: ",obj.top())
print("List is Empty: ",obj.is_empty())
print("\n-------Push 5 elements--------")
obj.push(1)
obj.push(2)
obj.push(3)
obj.push(4)
obj.push(5)
print("List: ",obj._data)
print("Size: ",obj.__len__())
print("Top: ",obj.top())
print("List is Empty: ",obj.is_empty())
print("\n-------Pop 3 elements--------")
obj.pop()
obj.pop()
obj.pop()
print("List: ",obj._data)
print("Size: ",obj.__len__())
print("Top: ",obj.top())
print("List is Empty: ",obj.is_empty())
```



```
==== RESTART: D:\SYIT_nidhee\DataStructures_SY\Git_upload\practical3a.py ====
List:  []
Size:  0
Top:  Empty list
List is Empty:  True

-------Push 5 elements--------
List:  [1, 2, 3, 4, 5]
Size:  5
Top:  1
List is Empty:  False

-------Pop 3 elements--------
List:  [1, 2]
Size:  2
Top:  1
List is Empty:  False
>>> 
```

PRACTICAL 3 B

Aim:

Implement Tower of Hanoi.

Theory:

Puzzle

-Tower of Hanoi consists of 3 towers and more than one rings

-Rings are stacked in ascending order of their size smaller at the top and larger at the bottom

Rules

-Only one disk can be moved among the tower at any given time

-Only the "top" disk can be removed

-No large disk can sit over a small disk

Minimum Steps to solve the puzzle

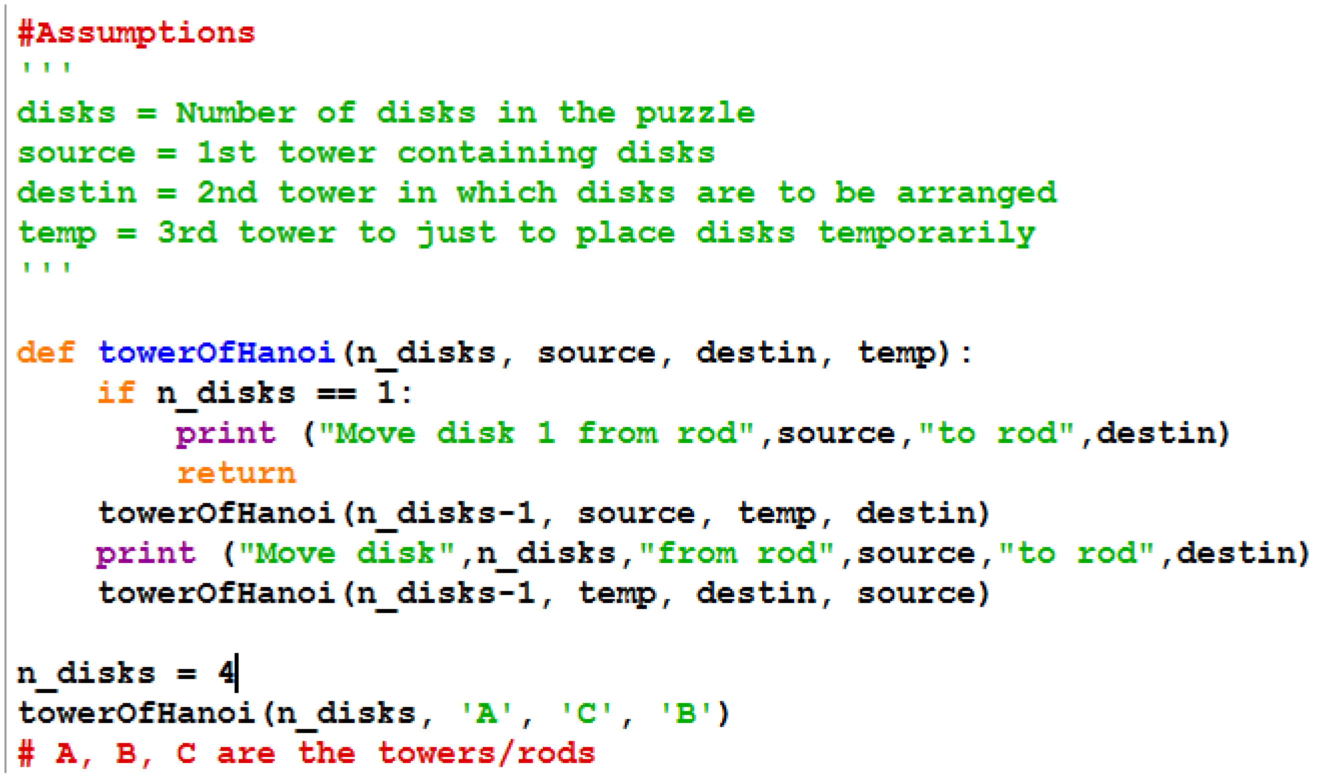
2^n - 1

Example:

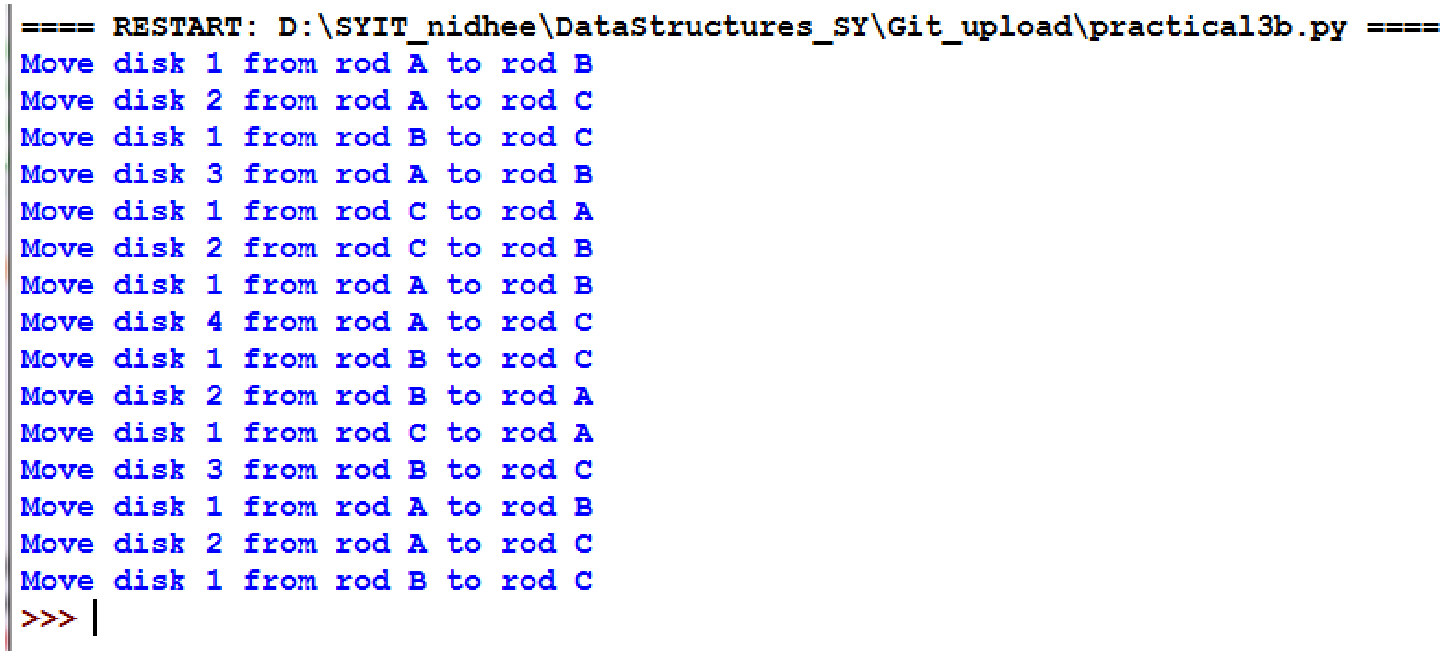
for '2 disks' minimum steps are,

2^2 -1 = 4

Tower of Hanoi puzzle with n disks can be solved in minimum $\mathbf{2^n-1}$ steps. This presentation shows that a puzzle with 3 disks has taken $\mathbf{2^3 - 1 = 7}$ steps.



```
#Assumptions
'''
disks = Number of disks in the puzzle
source = 1st tower containing disks
destin = 2nd tower in which disks are to be arranged
temp = 3rd tower to just to place disks temporarily
'''

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

n_disks = 4
towerOfHanoi(n_disks, 'A', 'C', 'B')
# A, B, C are the towers/rods
```



```
==== RESTART: D:\SYIT_nidhee\DataStructures_SY\Git_upload\practical3b.py ====
Move disk 1 from rod A to rod B
Move disk 2 from rod A to rod C
Move disk 1 from rod B to rod C
Move disk 3 from rod A to rod B
Move disk 1 from rod C to rod A
Move disk 2 from rod C to rod B
Move disk 1 from rod A to rod B
Move disk 4 from rod A to rod C
Move disk 1 from rod B to rod C
Move disk 2 from rod B to rod A
Move disk 1 from rod C to rod A
Move disk 3 from rod B to rod C
Move disk 1 from rod A to rod B
Move disk 2 from rod A to rod C
Move disk 1 from rod B to rod C
>>> 
```

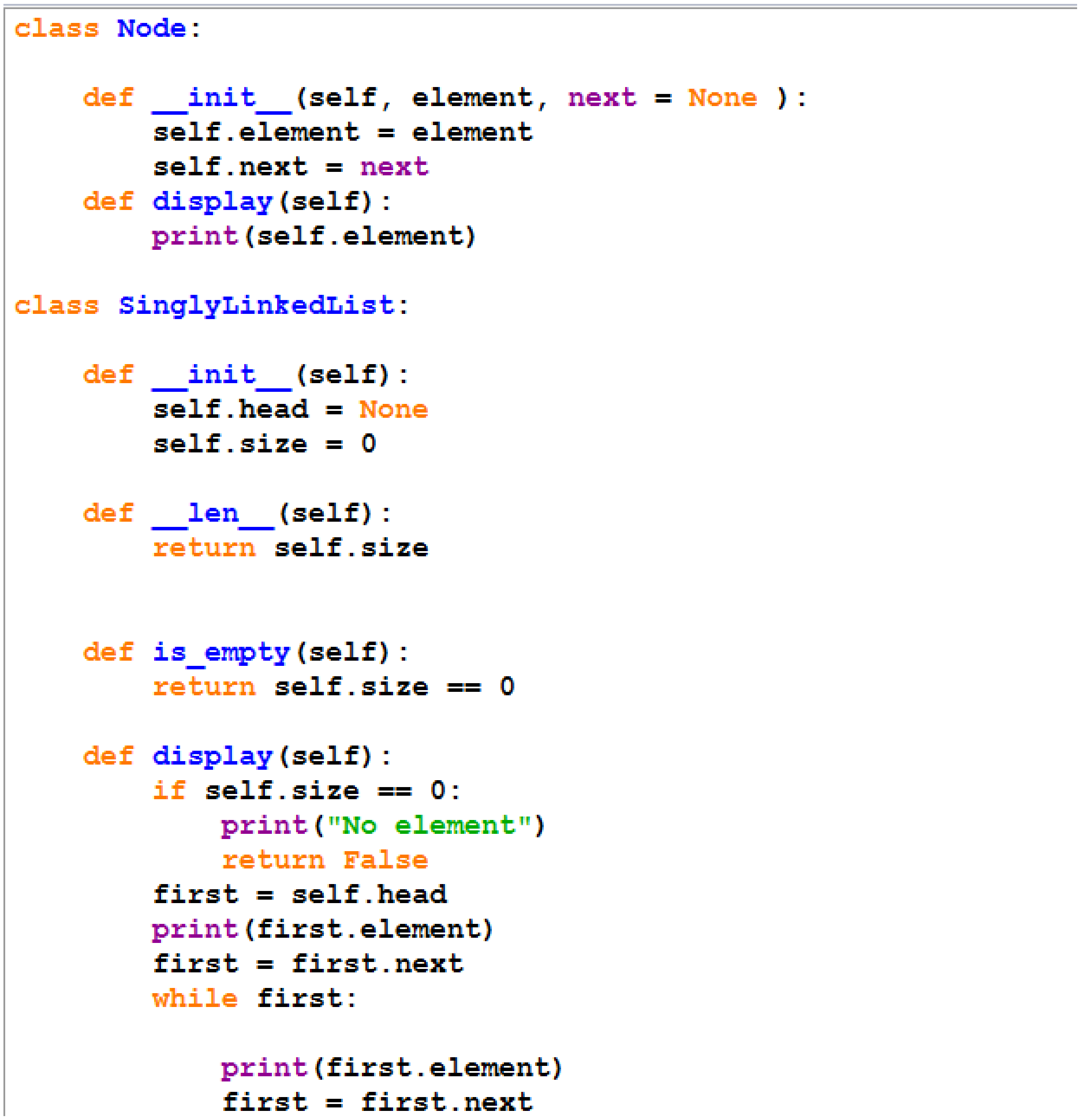
PRACTICAL 3 C

Aim:

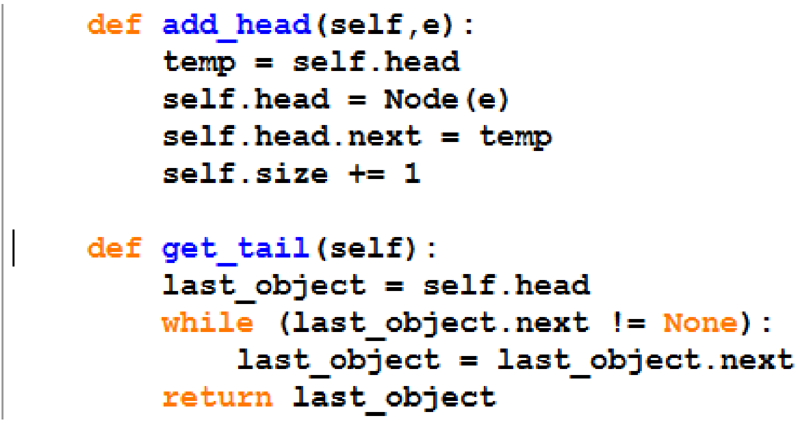
WAP to scan a polynomial using linked list and add two polynomials.

Theory:

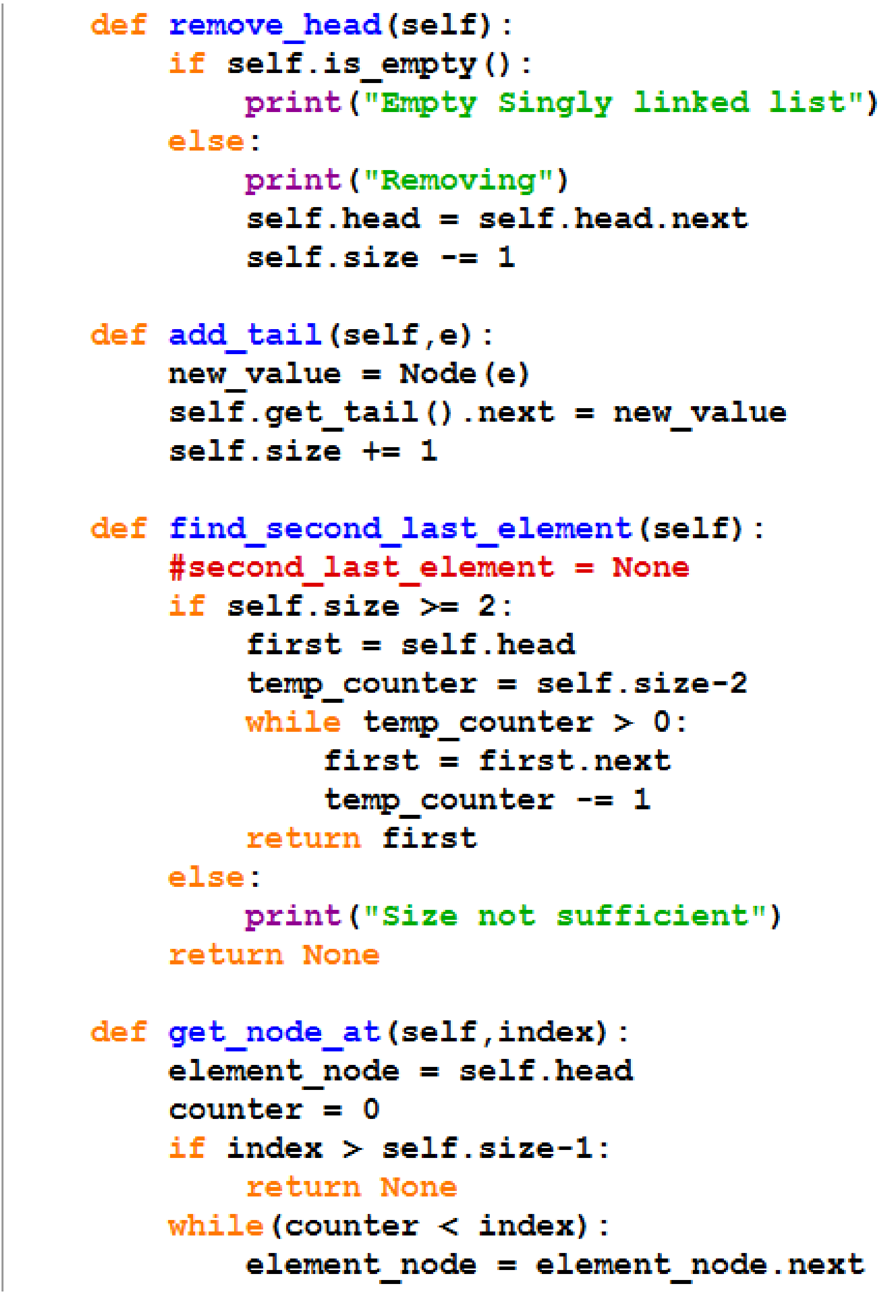
In this program we are going to take two polynomials using Linked List and will add those two polynomials.



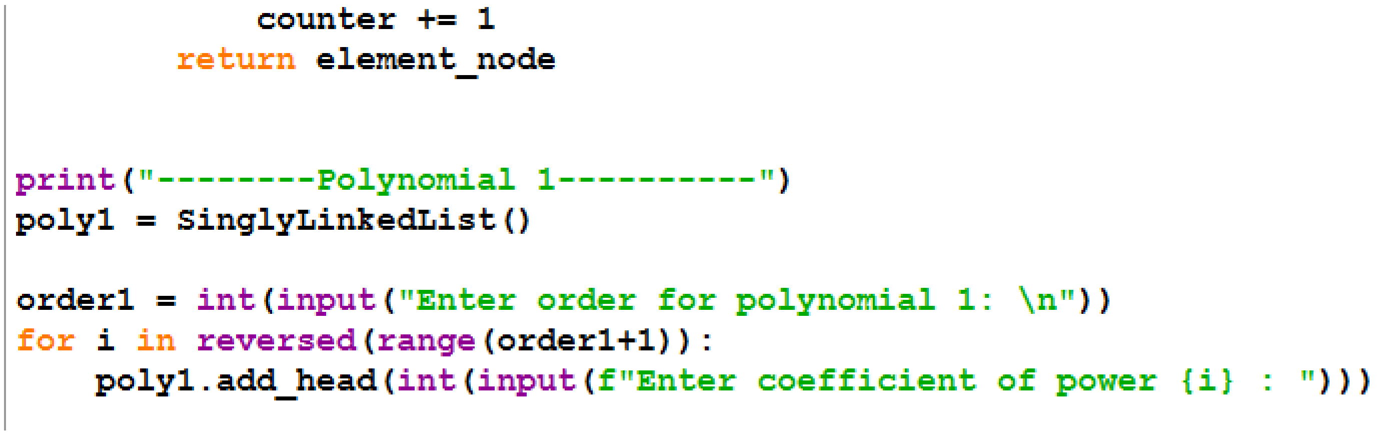
```
class Node:

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

class SinglyLinkedList:

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

            print(first.element)
            first = first.next
```



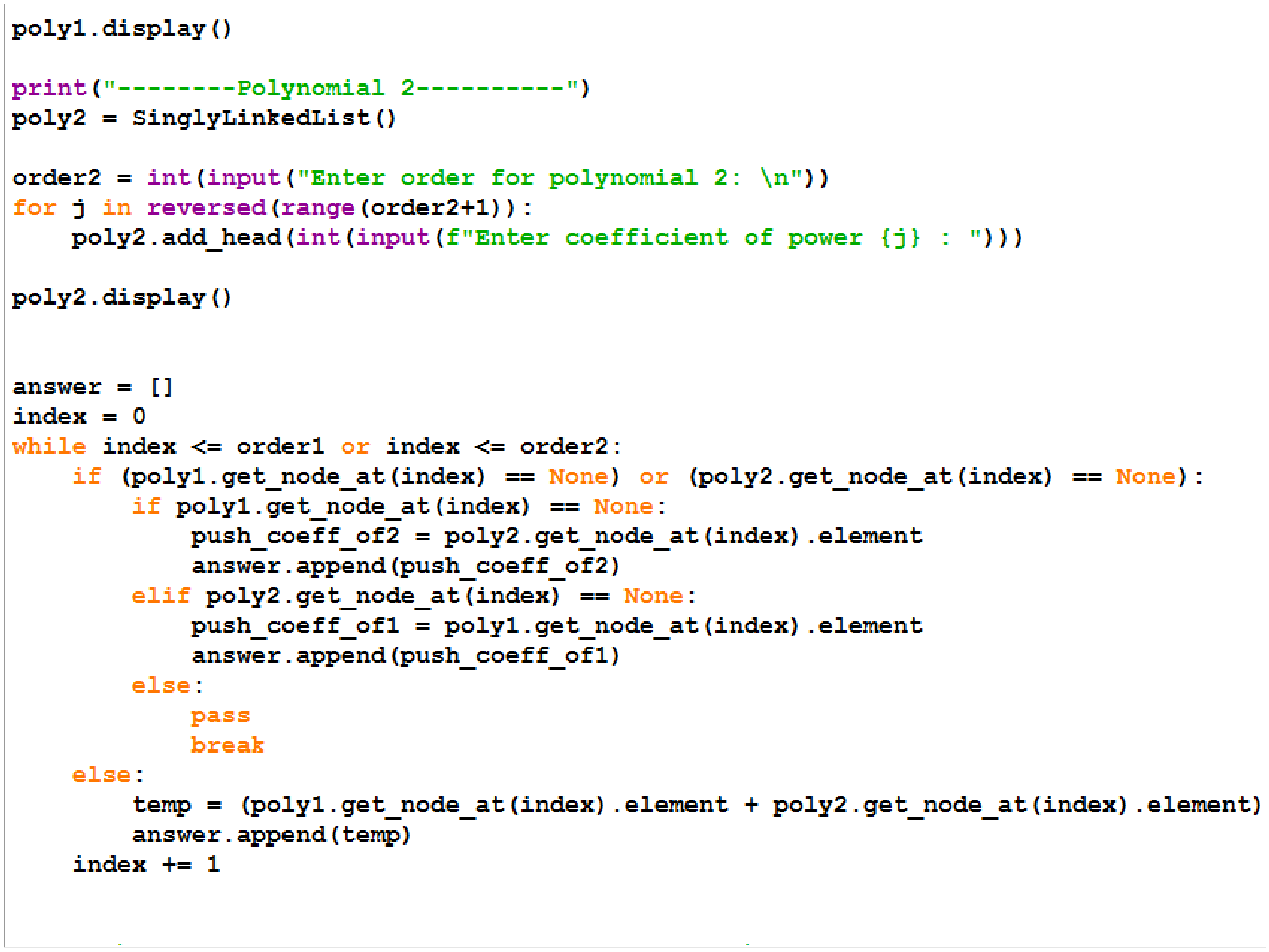
```
    def add_head(self,e):
        temp = self.head
        self.head = Node(e)
        self.head.next = temp
        self.size += 1

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



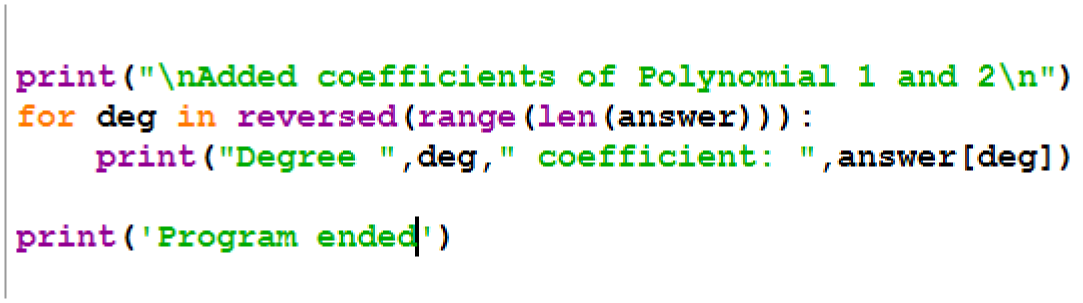
```
    def remove_head(self):
        if self.is_empty():
            print("Empty Singly linked list")
        else:
            print("Removing")
            self.head = self.head.next
            self.size -= 1

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

    def find_second_last_element(self):
        #second_last_element = None
        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:
            return None
        while(counter < index):
            element_node = element_node.next
```



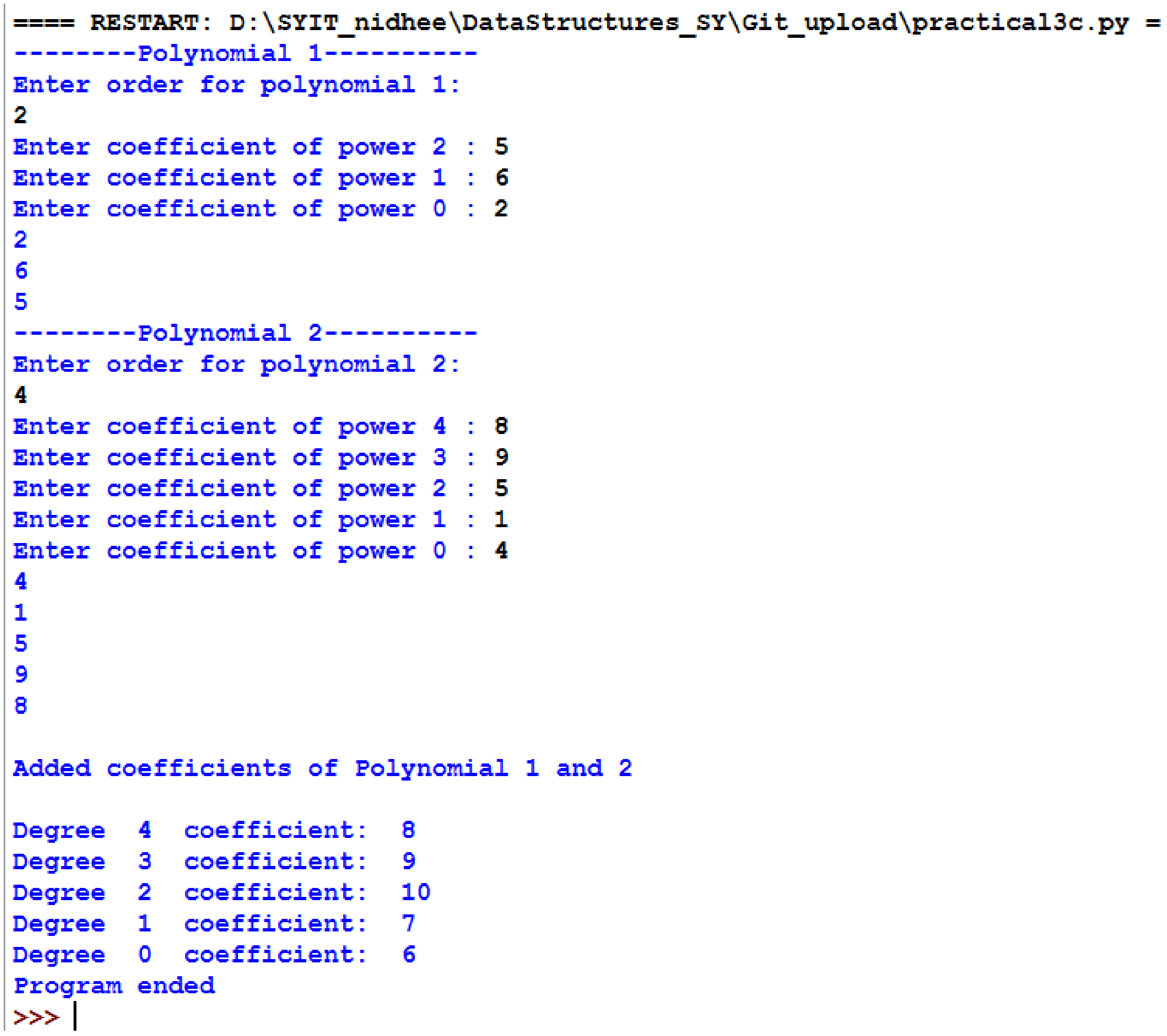
```
            counter += 1
        return element_node


print("--------Polynomial 1----------")
poly1 = SinglyLinkedList()

order1 = int(input("Enter order for polynomial 1: \n"))
for i in reversed(range(order1+1)):
    poly1.add_head(int(input(f"Enter coefficient of power {i} : ")))
```



```
poly1.display()

print("--------Polynomial 2----------")
poly2 = SinglyLinkedList()

order2 = int(input("Enter order for polynomial 2: \n"))
for j in reversed(range(order2+1)):
    poly2.add_head(int(input(f"Enter coefficient of power {j} : ")))

poly2.display()


answer = []
index = 0
while index <= order1 or index <= order2:
    if (poly1.get_node_at(index) == None) or (poly2.get_node_at(index) == None):
        if poly1.get_node_at(index) == None:
            push_coeff_of2 = poly2.get_node_at(index).element
            answer.append(push_coeff_of2)
        elif poly2.get_node_at(index) == None:
            push_coeff_of1 = poly1.get_node_at(index).element
            answer.append(push_coeff_of1)
        else:
            pass
            break
    else:
        temp = (poly1.get_node_at(index).element + poly2.get_node_at(index).element)
        answer.append(temp)
    index += 1
```



```
print("\nAdded coefficients of Polynomial 1 and 2\n")
for deg in reversed(range(len(answer))):
    print("Degree ",deg," coefficient: ",answer[deg])

print('Program ended')
```



```
==== RESTART: D:\SYIT_nidhee\DataStructures_SY\Git_upload\practical3c.py =
--------Polynomial 1----------
Enter order for polynomial 1:
2
Enter coefficient of power 2 : 5
Enter coefficient of power 1 : 6
Enter coefficient of power 0 : 2
2
6
5
--------Polynomial 2----------
Enter order for polynomial 2:
4
Enter coefficient of power 4 : 8
Enter coefficient of power 3 : 9
Enter coefficient of power 2 : 5
Enter coefficient of power 1 : 1
Enter coefficient of power 0 : 4
4
1
5
9
8

Added coefficients of Polynomial 1 and 2

Degree  4  coefficient:  8
Degree  3  coefficient:  9
Degree  2  coefficient:  10
Degree  1  coefficient:  7
Degree  0  coefficient:  6
Program ended
>>> 
```

PRACTICAL 3 D

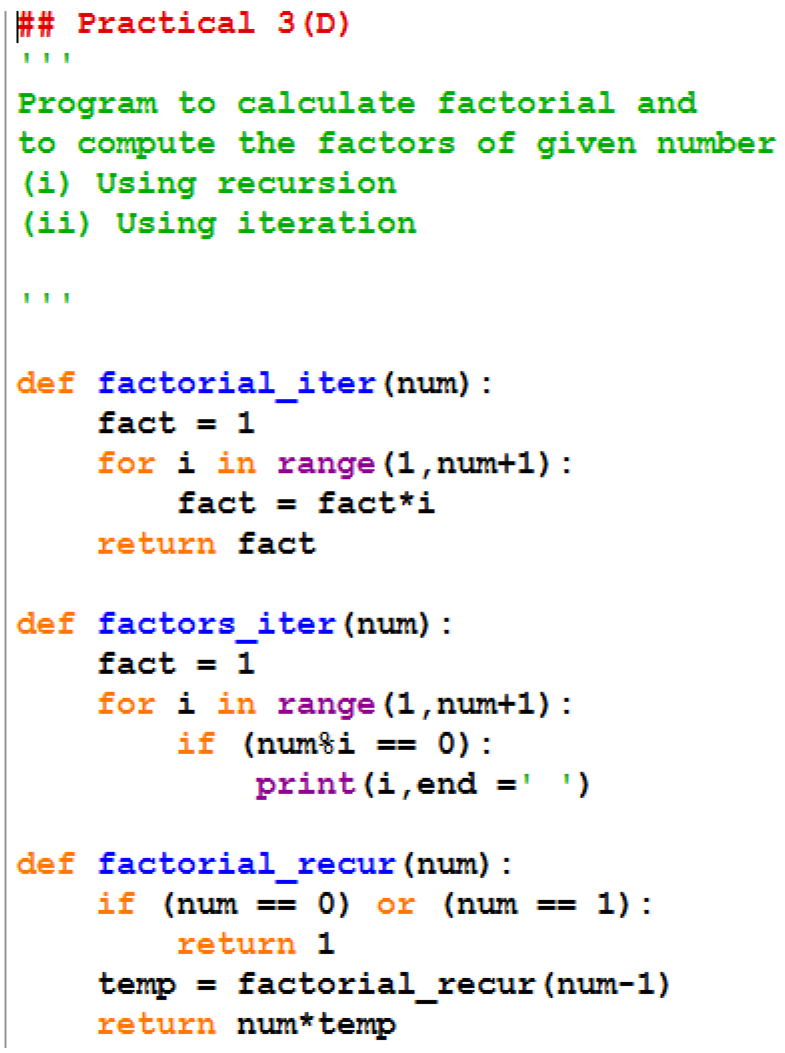
Aim:

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

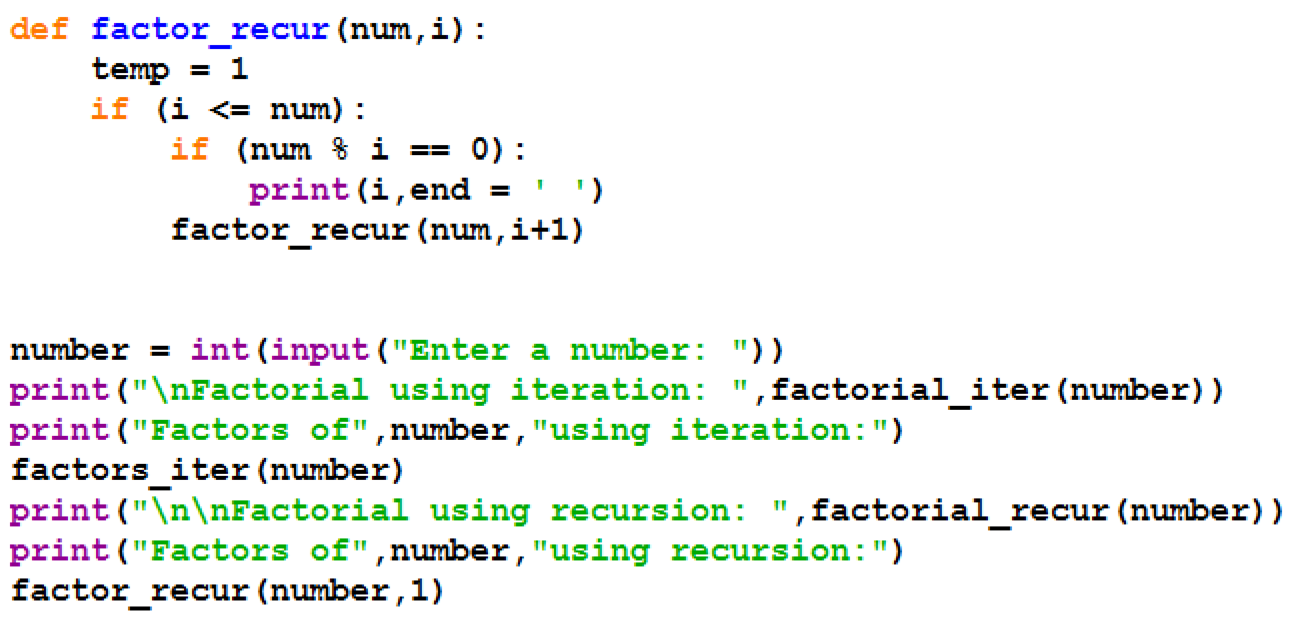
(i) using recursion, (ii) using iteration.

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. 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.



```
## Practical 3(D)
'''
Program to calculate factorial and
to compute the factors of given number
(i) Using recursion
(ii) Using iteration

'''

def factorial_iter(num):
    fact = 1
    for i in range(1,num+1):
        fact = fact*i
    return fact

def factors_iter(num):
    fact = 1
    for i in range(1,num+1):
        if (num%i == 0):
            print(i,end =' ')

def factorial_recur(num):
    if (num == 0) or (num == 1):
        return 1
    temp = factorial_recur(num-1)
    return num*temp
```



```
def factor_recur(num,i):
    temp = 1
    if (i <= num):
        if (num % i == 0):
            print(i,end = ' ')
        factor_recur(num,i+1)


number = int(input("Enter a number: "))
print("\nFactorial using iteration: ",factorial_iter(number))
print("Factors of",number,"using iteration:")
factors_iter(number)
print("\n\nFactorial using recursion: ",factorial_recur(number))
print("Factors of",number,"using recursion:")
factor_recur(number,1)
```

PRACTICAL 4

Aim:

Perform Queues operations using Circular Array implementation.

Theory:

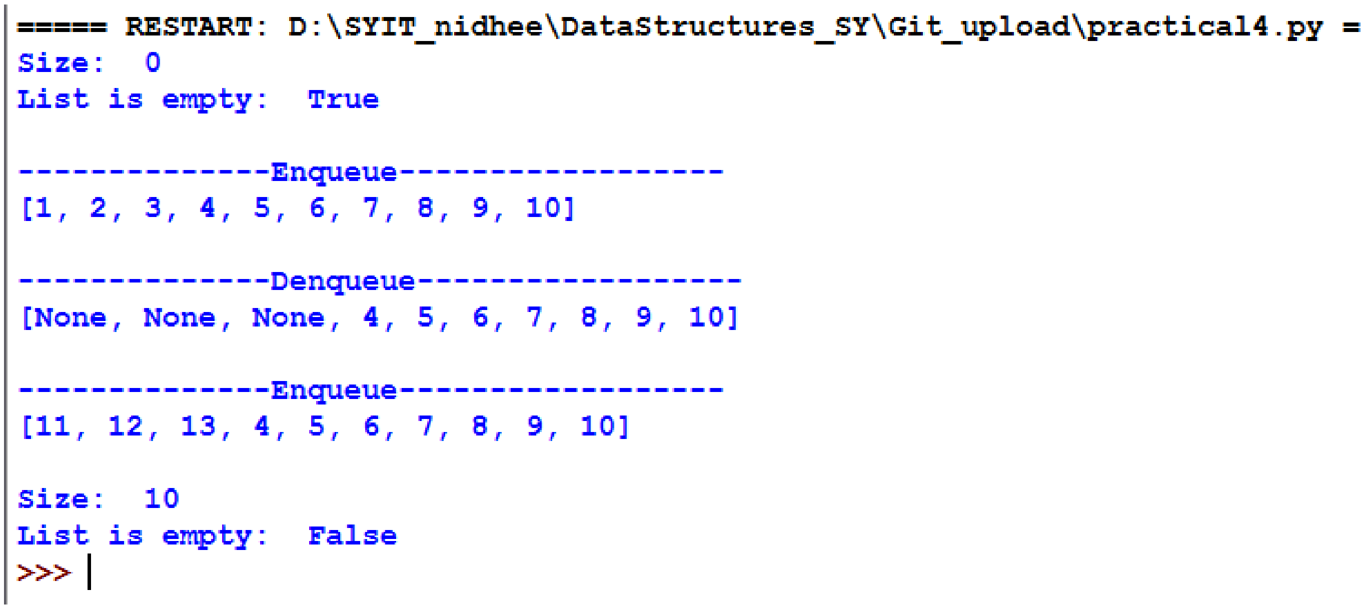
A queue is a linear data structure where an item can be inserted from one end and can be removed from another end. We cannot insert and remove items from the same end. One end of the queue is called a front and the other end is called a rear. Items are inserted in the rear end and are removed from the 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 works by the process of circular increment i.e. when we try to increment the pointer and we reach the end of the queue, we start from the beginning of the queue.

```
class ArrayQueue:
    DEFAULT_CAPACITY = 10

    def __init__(self):
        self._data = [None] * ArrayQueue.DEFAULT_CAPACITY
        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 Empty('Queue is empty')
        return self._data[self._front]

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

    def enqueue(self,value):
        if self._size == len(self._data):
            self._resize(2 * len(self._data))
        avail = (self._front + self._size)%len(self._data)

        self._data[avail] = value
        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
```

```
list_obj = ArrayQueue()
print("Size: ",list_obj.len())
print("List is empty: ",list_obj.is_empty())
print("\n--------------Enqueue------------------")
list_obj.enqueue(1)
list_obj.enqueue(2)
list_obj.enqueue(3)
list_obj.enqueue(4)
list_obj.enqueue(5)
list_obj.enqueue(6)
list_obj.enqueue(7)
list_obj.enqueue(8)
list_obj.enqueue(9)
list_obj.enqueue(10)
print(list_obj._data)
print("\n--------------Denqueue------------------")
list_obj.dequeue()
list_obj.dequeue()
list_obj.dequeue()
print(list_obj._data)
print("\n--------------Enqueue------------------")
list_obj.enqueue(11)
list_obj.enqueue(12)
list_obj.enqueue(13)
print(list_obj._data)
print("\nSize: ",list_obj.len())
print("List is empty: ",list_obj.is_empty())
```

Output:



```
===== RESTART: D:\SYIT_nidhee\DataStructures_SY\Git_upload\practical4.py =
Size:  0
List is empty:  True

--------------Enqueue------------------
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

--------------Denqueue------------------
[None, None, None, 4, 5, 6, 7, 8, 9, 10]

--------------Enqueue------------------
[11, 12, 13, 4, 5, 6, 7, 8, 9, 10]

Size:  10
List is empty:  False
>>> 
```

PRACTICAL 5

Aim:

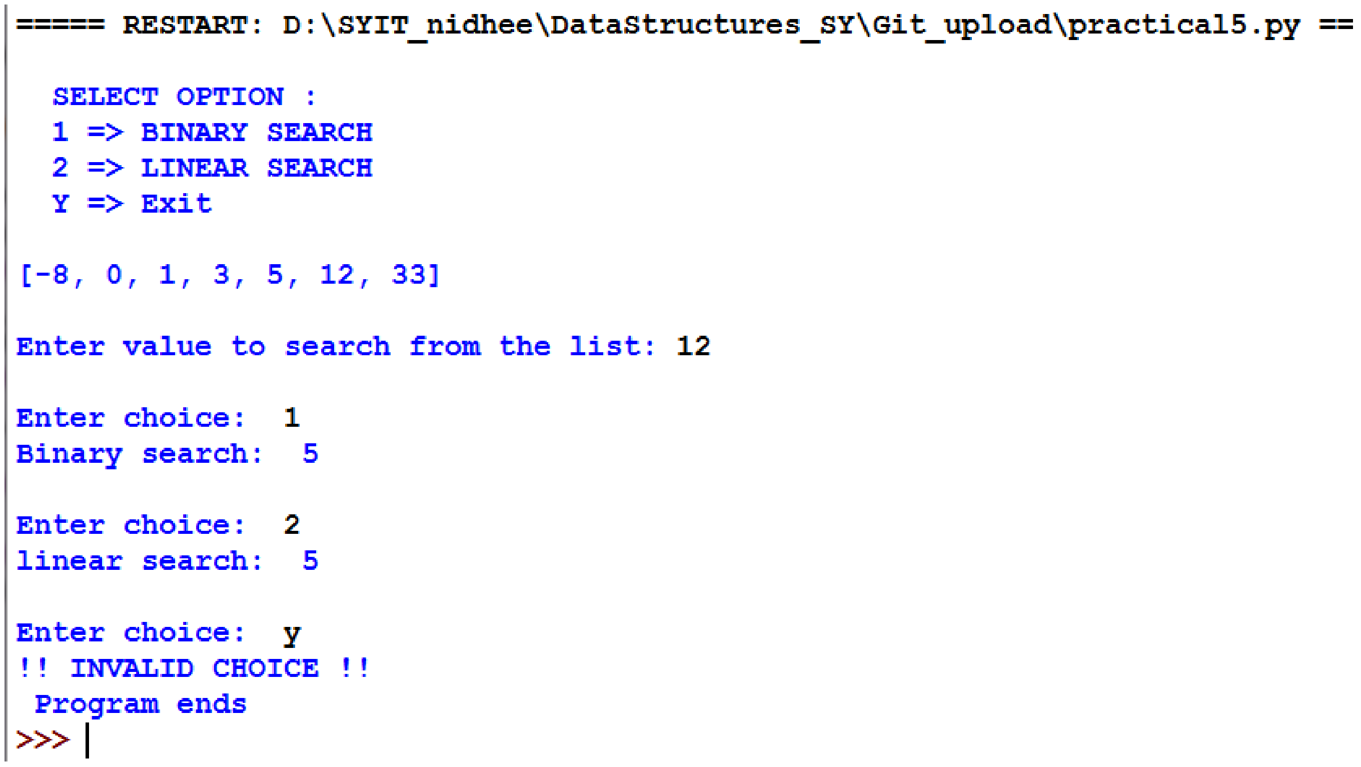
Write a program to search an element from a list. Give user the option to perform Linear or Binary search.

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.

```
## Practical 5
'''
Program to search an element from a list\
Give user the option to perform
    Linear or Binary Search
'''

print("\n  SELECT OPTION : \n  1 => BINARY SEARCH\n  2 => LINEAR SEARCH\n  Y => Exit")
print()
lst = [1,3,-8,5,0,33,12]
lst = sorted(lst)
print(lst)

def run_program(lst):
    def linear_search(lst,n):
        index = 0
        size = len(lst)
        while index < size:
            temp = lst[index]
            if temp == n:
                return index
            index += 1
        return -1

    def binary_search(lst,search,start,end):
        lst = sorted(lst)
        if search > lst[-1]:
            return -1
        if start > end:
            return -1

        mid = (start+end)//2
        if lst[mid] == search:
            return mid
        if search < lst[mid]:
            return binary_search(lst,search,start,mid-1)
        else:
            return binary_search(lst,search,mid+1,end)
    choice = input("\nEnter choice:  ")
    if choice == '1':
        print("Binary search: ",binary_search(lst,search,0,len(lst)))
        run_program(lst)
    elif choice == '2':
        print("linear search: ",linear_search(lst,search))
        run_program(lst)
    else:
        print("!! INVALID CHOICE !!\n Program ends")

search = int(input('\nEnter value to search from the list: '))
run_program(lst)
```



```
===== RESTART: D:\SYIT_nidhee\DataStructures_SY\Git_upload\practical5.py ==

  SELECT OPTION :
  1 => BINARY SEARCH
  2 => LINEAR SEARCH
  Y => Exit

[-8, 0, 1, 3, 5, 12, 33]

Enter value to search from the list: 12

Enter choice:  1
Binary search:  5

Enter choice:  2
linear search:  5

Enter choice:  y
!! INVALID CHOICE !!
 Program ends
>>> 
```

PRACTICAL 6

Aim:

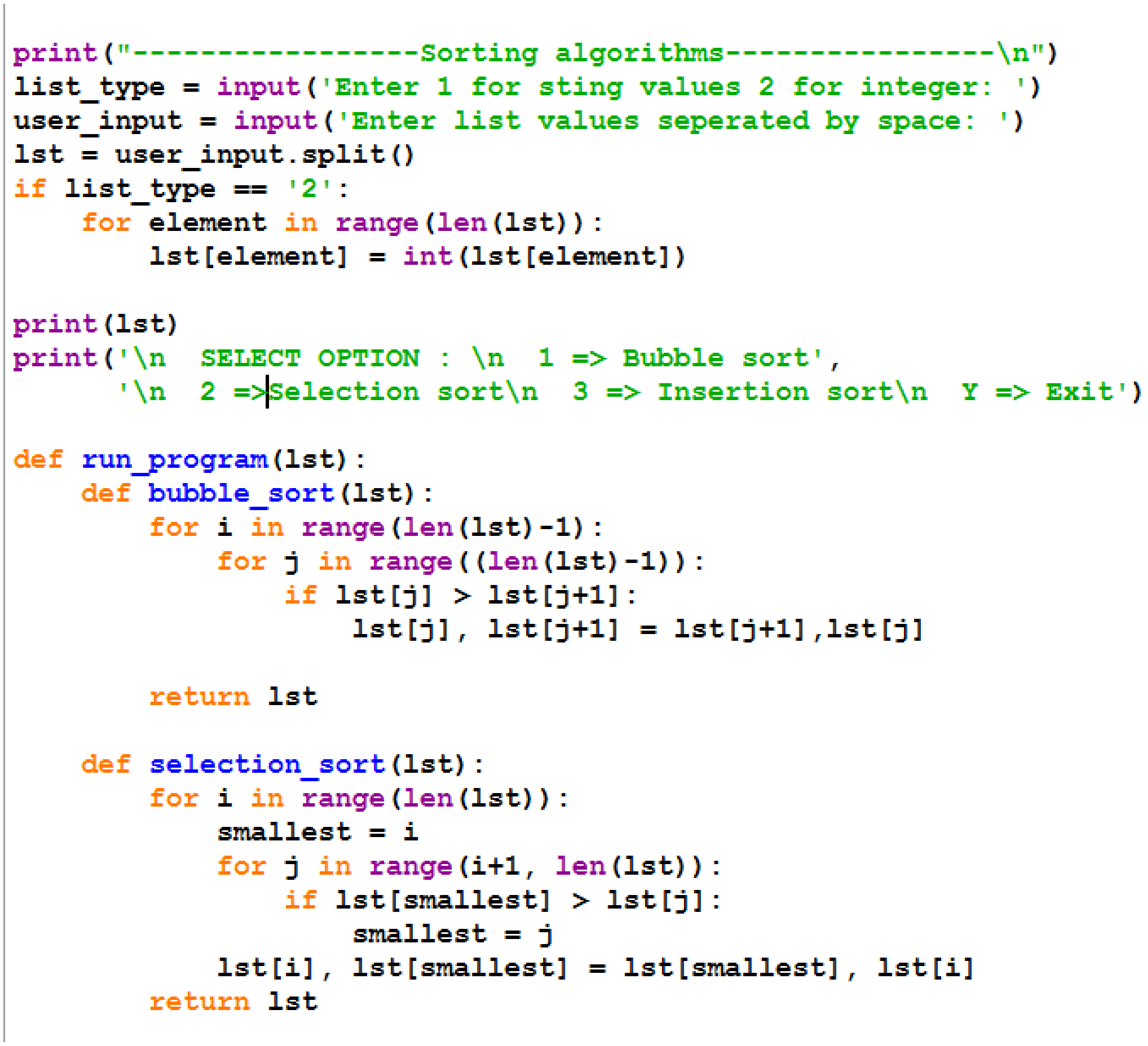
WAP to sort a list of elements. Give user the option to perform sorting using Insertion sort, Bubble sort or Selection sort.

Theory:

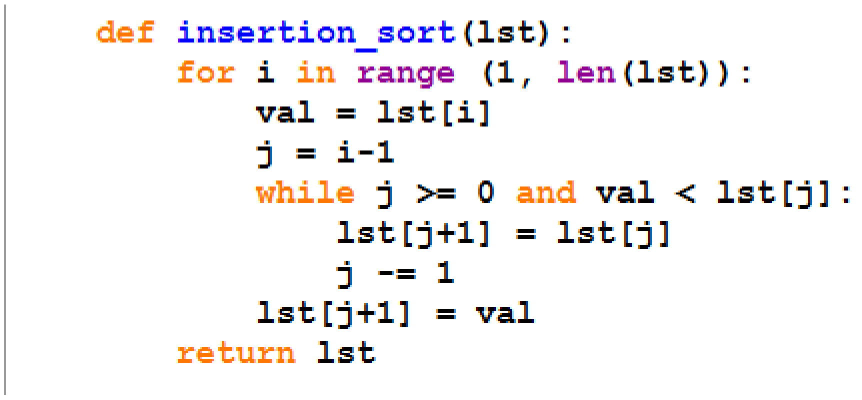
Insertion sort iterates, consuming one input element each repetition, and growing a **sorted** output list. At each iteration, **insertion sort** removes one element from the input data, finds the location it belongs within the **sorted** list, and inserts it there. It repeats until no input elements remain.

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.

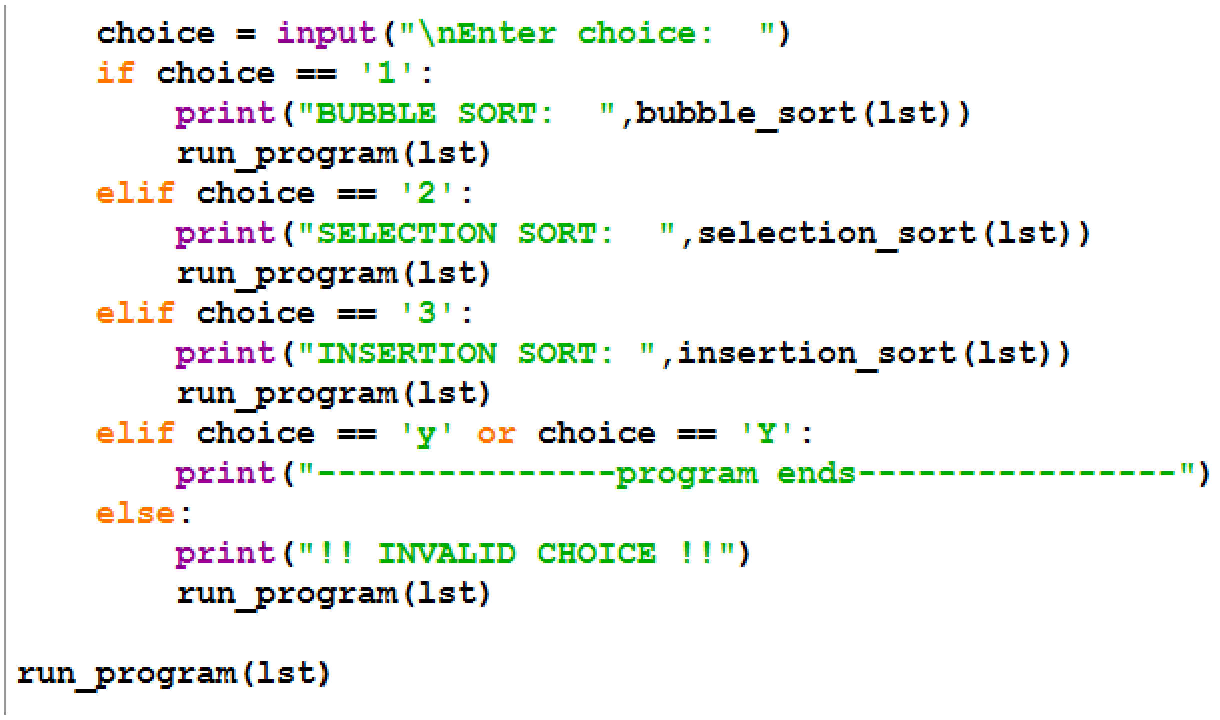
Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order. This algorithm is not suitable for large data sets as its average and worst case complexity are of $O(n^2)$ where **n** is the number of items.



```
print("-----------------Sorting algorithms----------------\n")
list_type = input('Enter 1 for sting values 2 for integer: ')
user_input = input('Enter list values seperated by space: ')
lst = user_input.split()
if list_type == '2':
    for element in range(len(lst)):
        lst[element] = int(lst[element])

print(lst)
print('\n  SELECT OPTION : \n  1 => Bubble sort',
      '\n  2 =>Selection sort\n  3 => Insertion sort\n  Y => Exit')

def run_program(lst):
    def bubble_sort(lst):
        for i in range(len(lst)-1):
            for j in range((len(lst)-1)):
                if lst[j] > lst[j+1]:
                    lst[j], lst[j+1] = lst[j+1],lst[j]

        return lst

    def selection_sort(lst):
        for i in range(len(lst)):
            smallest = i
            for j in range(i+1, len(lst)):
                if lst[smallest] > lst[j]:
                    smallest = j
            lst[i], lst[smallest] = lst[smallest], lst[i]
        return lst
```

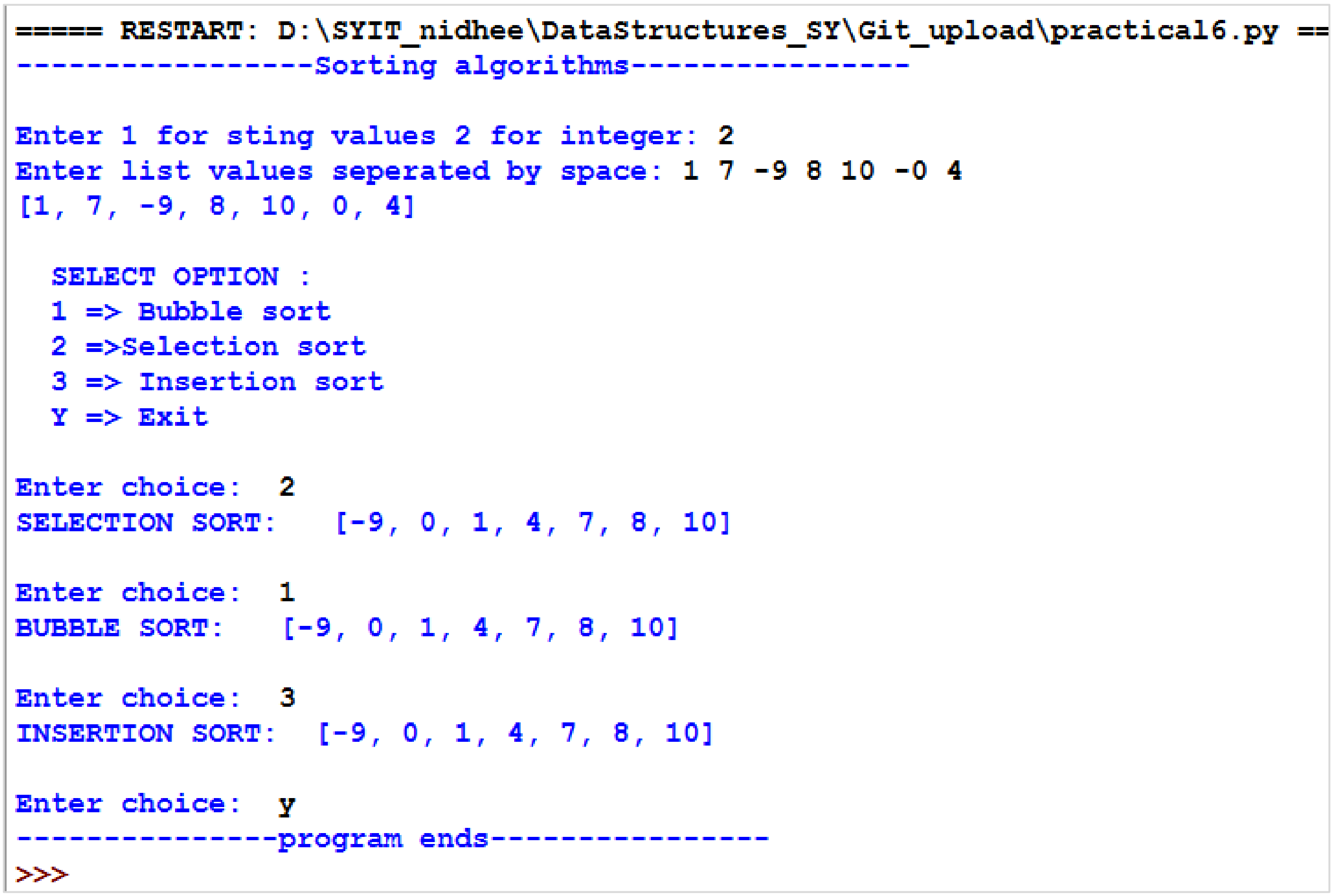


```
    def insertion_sort(lst):
        for i in range (1, len(lst)):
            val = lst[i]
            j = i-1
            while j >= 0 and val < lst[j]:
                lst[j+1] = lst[j]
                j -= 1
            lst[j+1] = val
        return lst
```

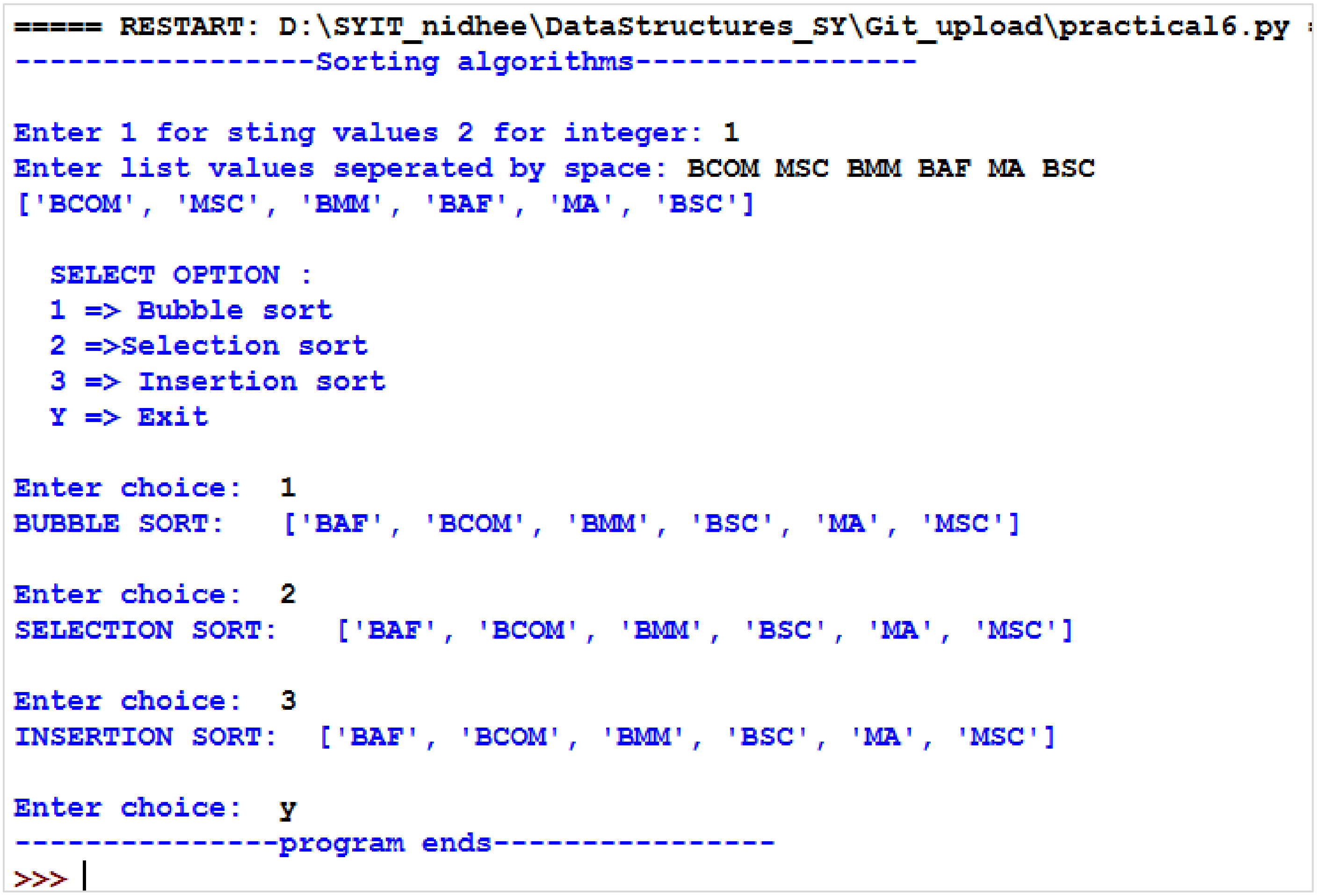


```
    choice = input("\nEnter choice:  ")
    if choice == '1':
        print("BUBBLE SORT:  ",bubble_sort(lst))
        run_program(lst)
    elif choice == '2':
        print("SELECTION SORT:  ",selection_sort(lst))
        run_program(lst)
    elif choice == '3':
        print("INSERTION SORT: ",insertion_sort(lst))
        run_program(lst)
    elif choice == 'y' or choice == 'Y':
        print("---------------program ends----------------")
    else:
        print("!! INVALID CHOICE !!")
        run_program(lst)

run_program(lst)
```

Output:



```
===== RESTART: D:\SYIT_nidhee\DataStructures_SY\Git_upload\practical6.py ==
-----------------Sorting algorithms----------------

Enter 1 for sting values 2 for integer: 2
Enter list values seperated by space: 1 7 -9 8 10 -0 4
[1, 7, -9, 8, 10, 0, 4]

  SELECT OPTION :
  1 => Bubble sort
  2 =>Selection sort
  3 => Insertion sort
  Y => Exit

Enter choice:  2
SELECTION SORT:   [-9, 0, 1, 4, 7, 8, 10]

Enter choice:  1
BUBBLE SORT:   [-9, 0, 1, 4, 7, 8, 10]

Enter choice:  3
INSERTION SORT:  [-9, 0, 1, 4, 7, 8, 10]

Enter choice:  y
---------------program ends----------------
>>>
```



```
===== RESTART: D:\SYIT_nidhee\DataStructures_SY\Git_upload\practical6.py
-----------------Sorting algorithms----------------

Enter 1 for sting values 2 for integer: 1
Enter list values seperated by space: BCOM MSC BMM BAF MA BSC
['BCOM', 'MSC', 'BMM', 'BAF', 'MA', 'BSC']

  SELECT OPTION :
  1 => Bubble sort
  2 =>Selection sort
  3 => Insertion sort
  Y => Exit

Enter choice:  1
BUBBLE SORT:   ['BAF', 'BCOM', 'BMM', 'BSC', 'MA', 'MSC']

Enter choice:  2
SELECTION SORT:   ['BAF', 'BCOM', 'BMM', 'BSC', 'MA', 'MSC']

Enter choice:  3
INSERTION SORT:  ['BAF', 'BCOM', 'BMM', 'BSC', 'MA', 'MSC']

Enter choice:  y
---------------program ends----------------
>>> |
```

PRACTICAL 7

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 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 7 A

Aim:

Write a program to implement the collision technique.

Theory:

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.

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.

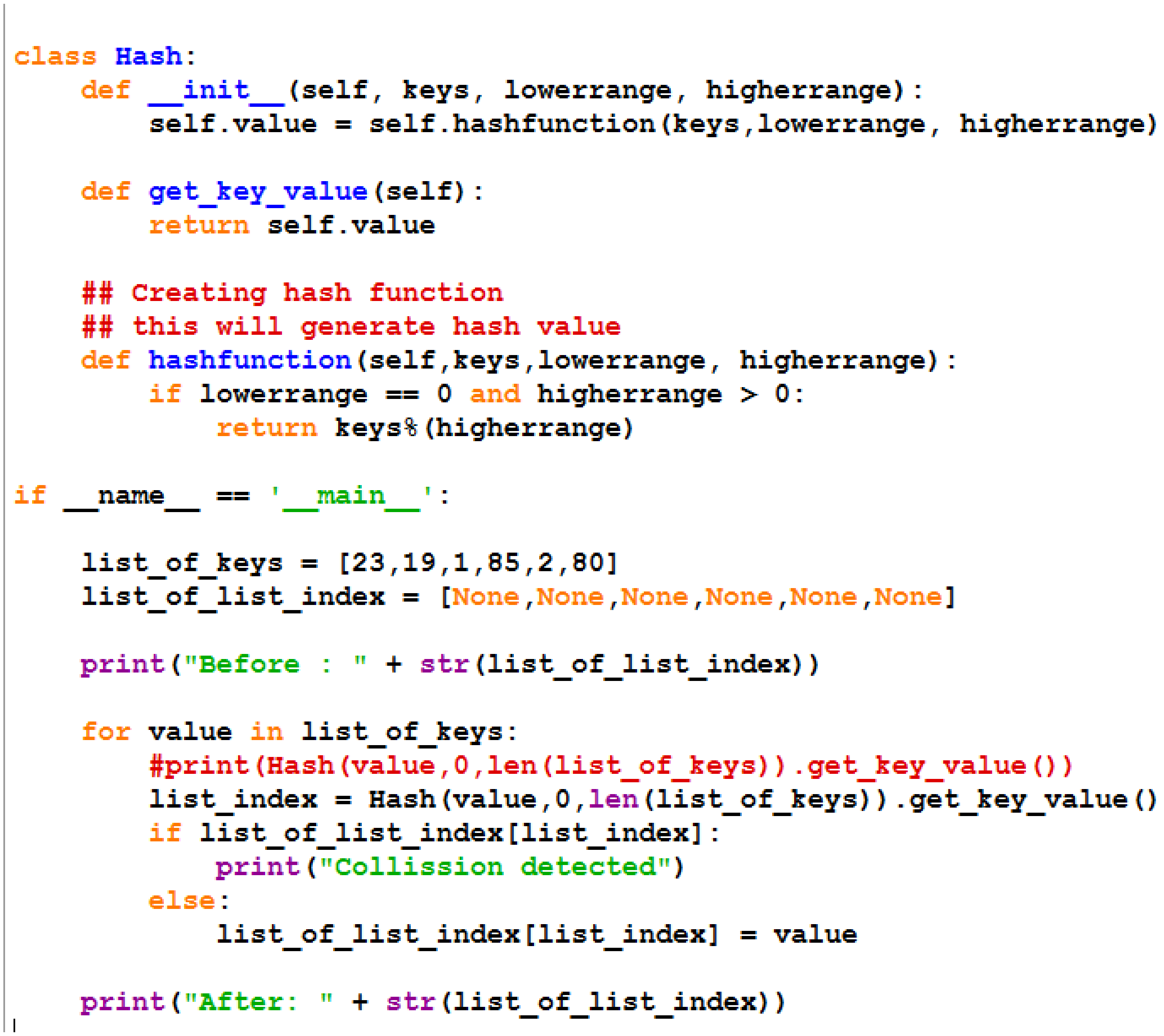
What happens is, more than one keys map to the same location and this is known as a collision. Collision resolution is finding another location to avoid the collision. The most popular resolution techniques are,

- Separate chaining
- Open addressing
- Open addressing can be further divided into,

(i) Linear Probing

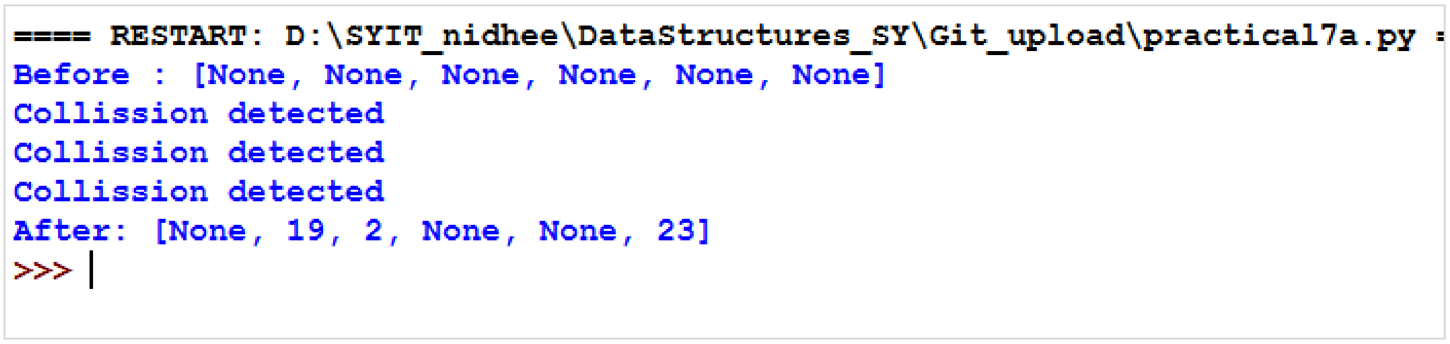
(ii) Quadratic Probing

(iii)Double hashing



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

    def get_key_value(self):
        return self.value

    ## Creating hash function
    ## this will generate hash value
    def hashfunction(self,keys,lowerrange, higherrange):
        if lowerrange == 0 and higherrange > 0:
            return keys%(higherrange)

if __name__ == '__main__':

    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:
        #print(Hash(value,0,len(list_of_keys)).get_key_value())
        list_index = Hash(value,0,len(list_of_keys)).get_key_value()
        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:



```
==== RESTART: D:\SYIT_nidhee\DataStructures_SY\Git_upload\practical7a.py =
Before : [None, None, None, None, None, None]
Collission detected
Collission detected
Collission detected
After: [None, 19, 2, None, None, 23]
>>> |
```

PRACTICAL 7 B

Aim:

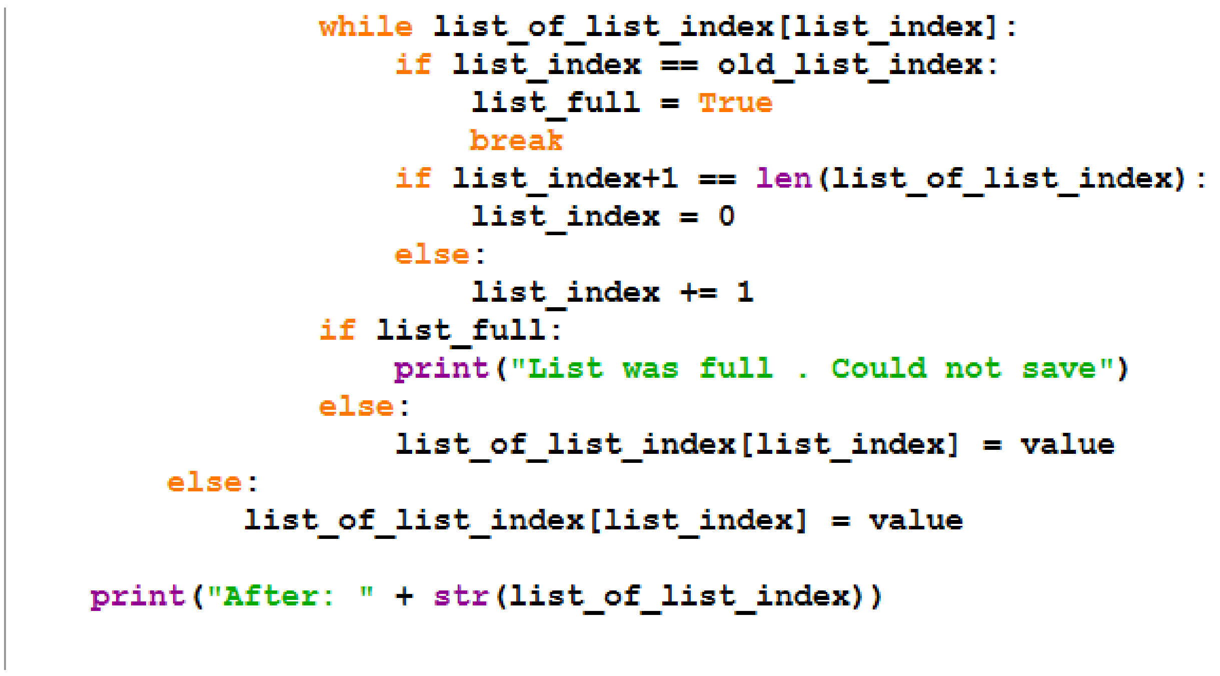
Write a program to implement the concept of linear probing.

Theory:

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

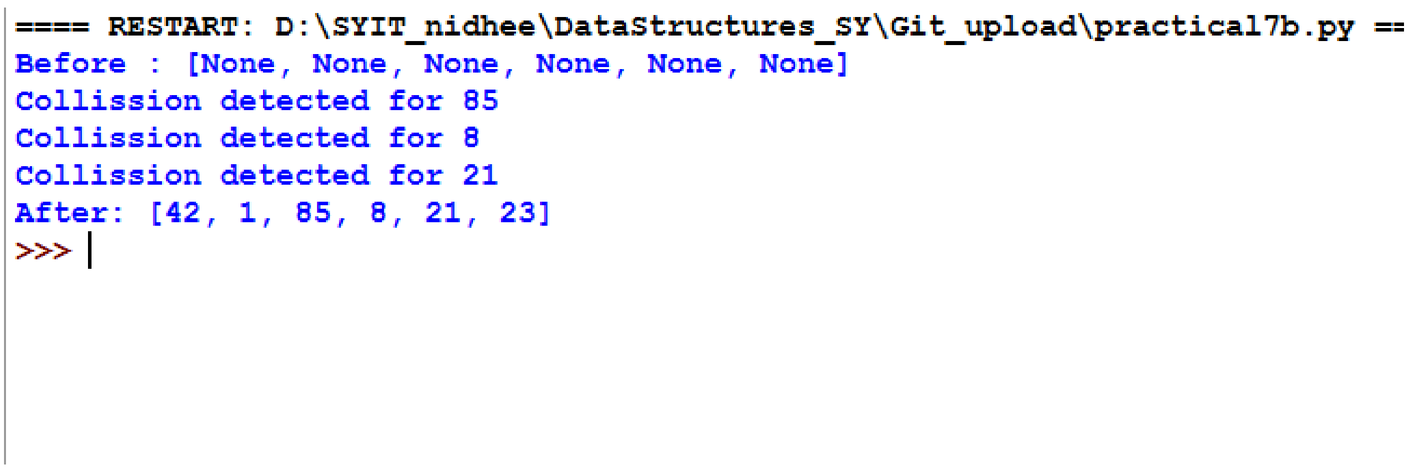
It is a strategy for resolving collisions. In this the new key is placed in the closest following empty cell. Advantage - It is faster due to locality of reference. Disadvantage - It needs five-way independence in the hash function.

```
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__':

    ## Creating a flag as linear_probing
    linear_probing = True
    list_of_keys = [23,42,1,85,8,21]
    list_of_list_index = [None,None,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: D:\SYIT_nidhee\DataStructures_SY\Git_upload\practical7b.py ==
Before : [None, None, None, None, None, None]
Collission detected for 85
Collission detected for 8
Collission detected for 21
After: [42, 1, 85, 8, 21, 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.

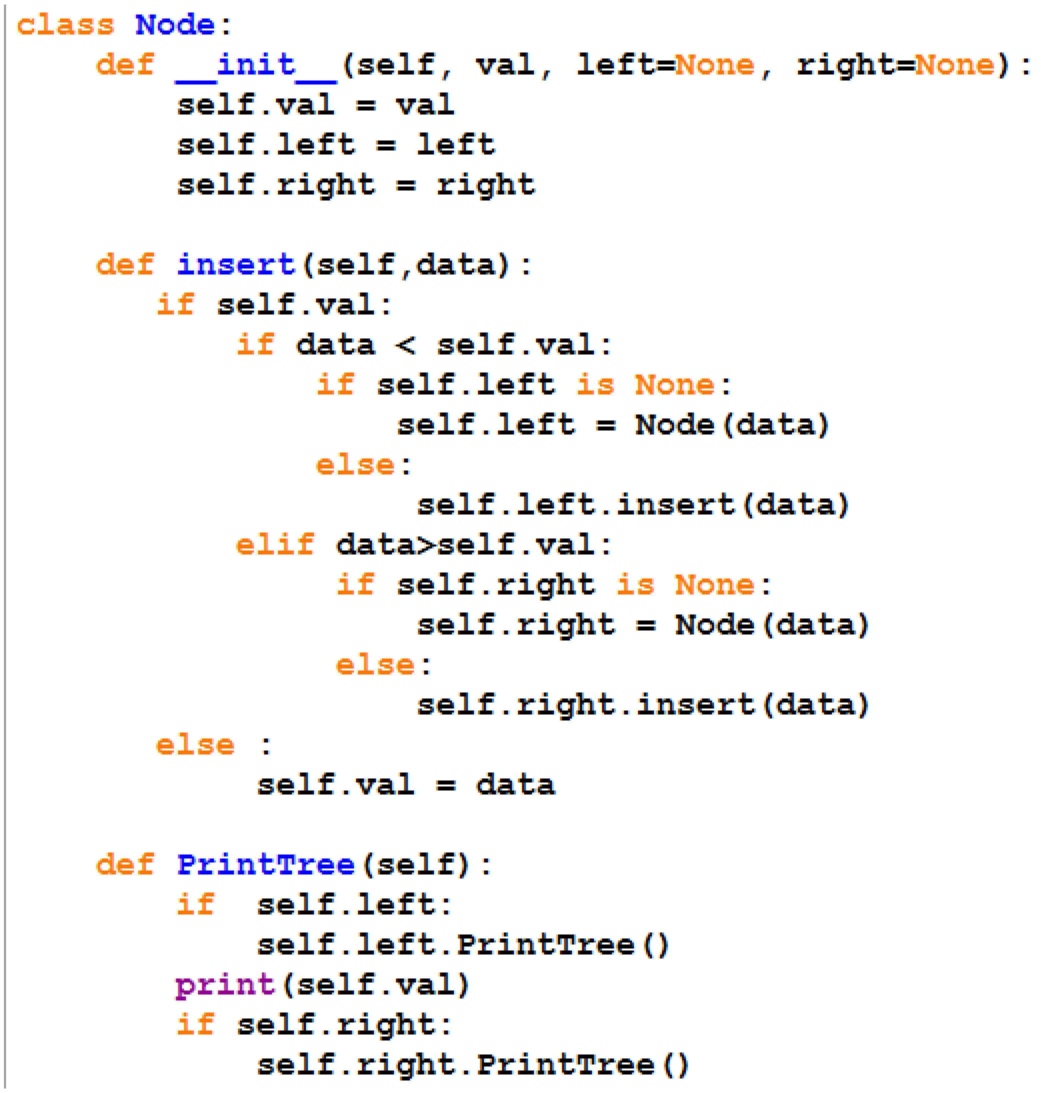
If a binary tree is traversed **in-order**, the output will produce sorted key values in an ascending order.

Pre-order Traversal

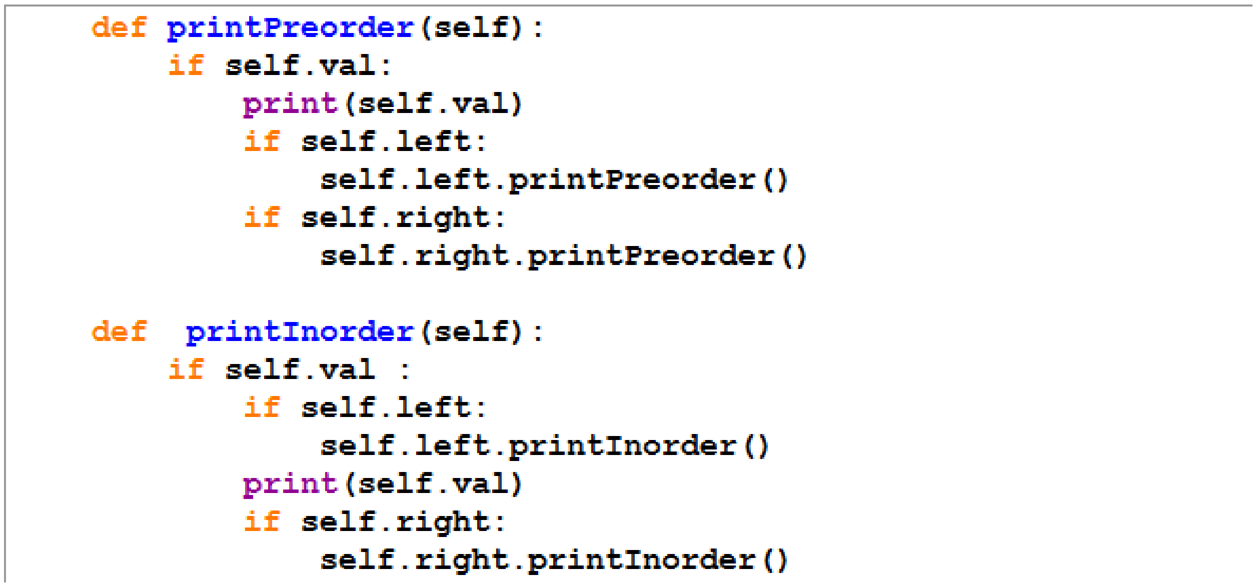
In this traversal method, the root node is visited first, then the left subtree and finally the right subtree.

Post-order Traversal

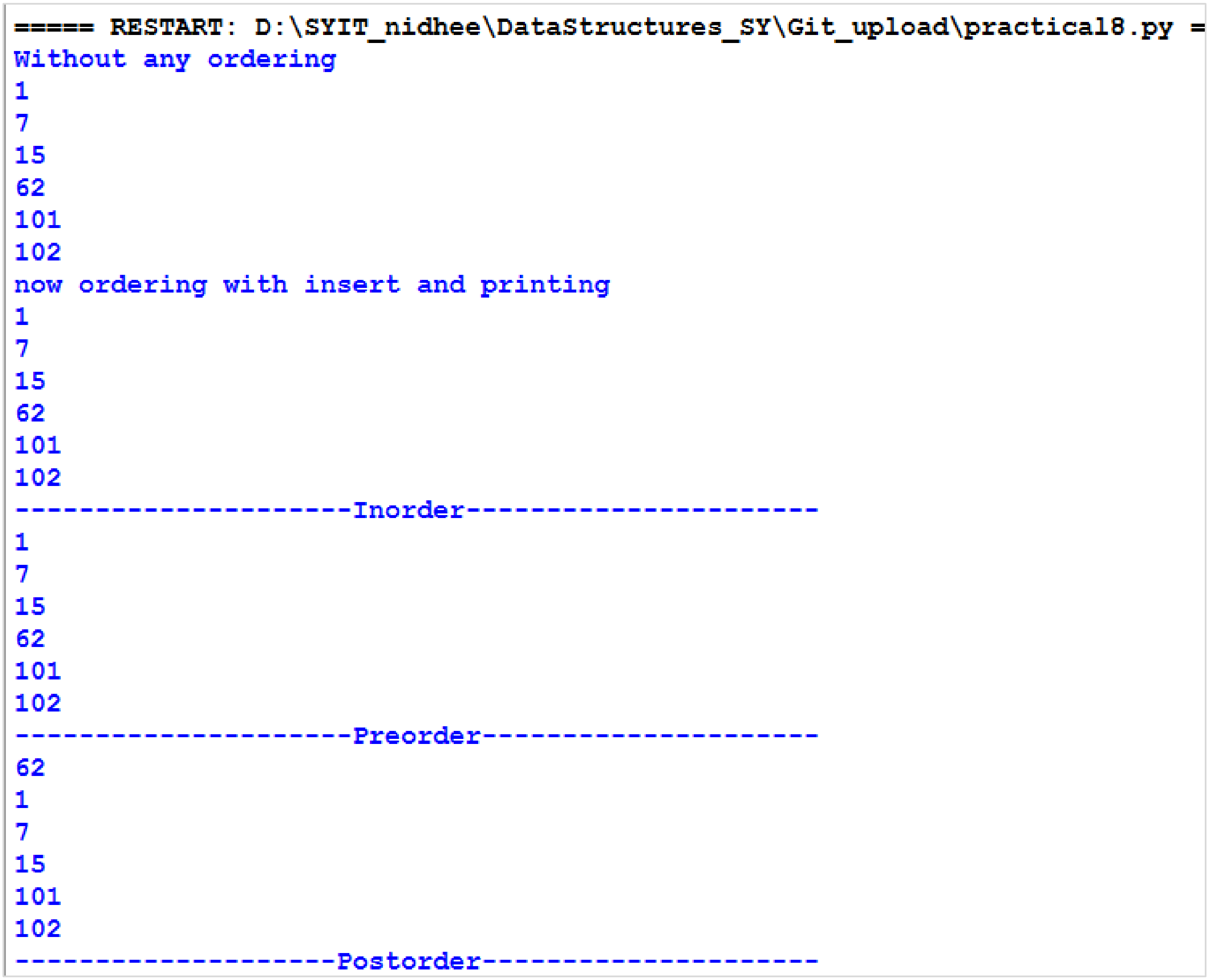
In this traversal method, the root node is visited last, hence the name. First we traverse the left subtree, then the right subtree and finally the root node.



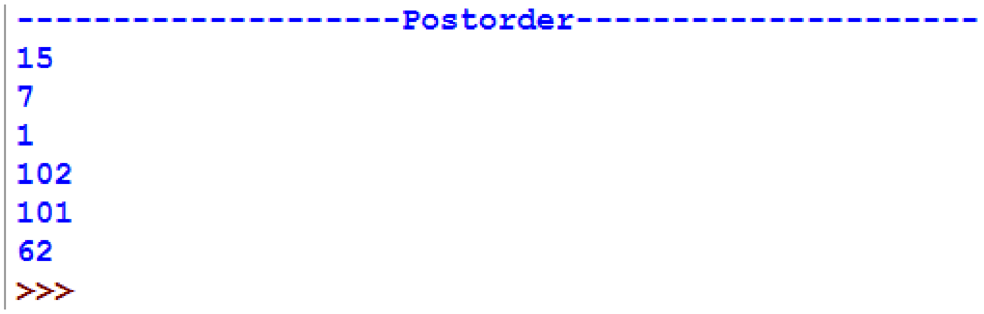
```
class Node:
    def __init__(self, val, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

    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)

root = Node(62)
root.left = Node(1)
root.right = Node(101)
root.left.right = Node(7)
root.left.right.right = Node(15)
root.right.right = Node(102)

print('Without any ordering')
root.PrintTree()

print('now ordering with insert and printing' )
root1 = Node(62)
root1.insert(1)
root1.insert(7)
root1.insert(15)
root1.insert(101)
root1.insert(102)
root1.PrintTree()

print("Inorder".center(50,'-'))
root1.printInorder()

print("Preorder".center(50,'-'))
root1.printPreorder()

print("Postorder".center(50,'-'))
root1.printPostorder()
```



```
===== RESTART: D:\SYIT_nidhee\DataStructures_SY\Git_upload\practical8.py =
Without any ordering
1
7
15
62
101
102
now ordering with insert and printing
1
7
15
62
101
102
---------------------Inorder----------------------
1
7
15
62
101
102
---------------------Preorder---------------------
62
1
7
15
101
102
--------------------Postorder---------------------
```



```
--------------------Postorder---------------------
15
7
1
102
101
62
>>>
```