Nitin Singh SYCS 4059

MALAD KANDIVALI EDUCATION SOCIETY'S

NAGINDAS KHANDWALA COLLEGE OF COMMERCE, ARTS & MANAGEMENT STUDIES & SHANTABEN NAGINDAS KHANDWALA COLLEGE OF SCIENCE
MALAD [W], MUMBAI – 64
AUTONOMOUS INSTITUTION
(Affiliated To University Of Mumbai)
Reaccredited 'A' Grade by NAAC | ISO 9001:2015 Certified

CERTIFICATE

Name: Mr./Ms. Nitin Singh

Roll No: 373 Programme: BSc CS 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)

Nitin Singh SYCS 4059

Class: S.Y. B.Sc. CS Sem- III **Roll No:** 373

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 No. 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.

Github Repository Link :
https://github.com/NitinSingh1071/DS/blob/main/practical1a.py

Theory :

Searching: You can search an array for a certain value, and return the indexes that get a match. To search an array, use the search() method.

Sorting: Sorting means putting elements in an ordered sequence. Ordered sequence is any sequence that has an order corresponding to elements, like numeric or alphabetical, ascending or descending. The NumPy ndarray object has a function called sort(), that will sort a specified array.

Merging: There are several ways to join, or concatenate, two or more lists in Python. One of the easiest ways are by using the + operator. Another way to join two lists are by appending all the items from list2 into list1, one by one or you can use the extend() method, which purpose is to add elements from one list to another list.

Reversing: Using the reversed() built-in function.

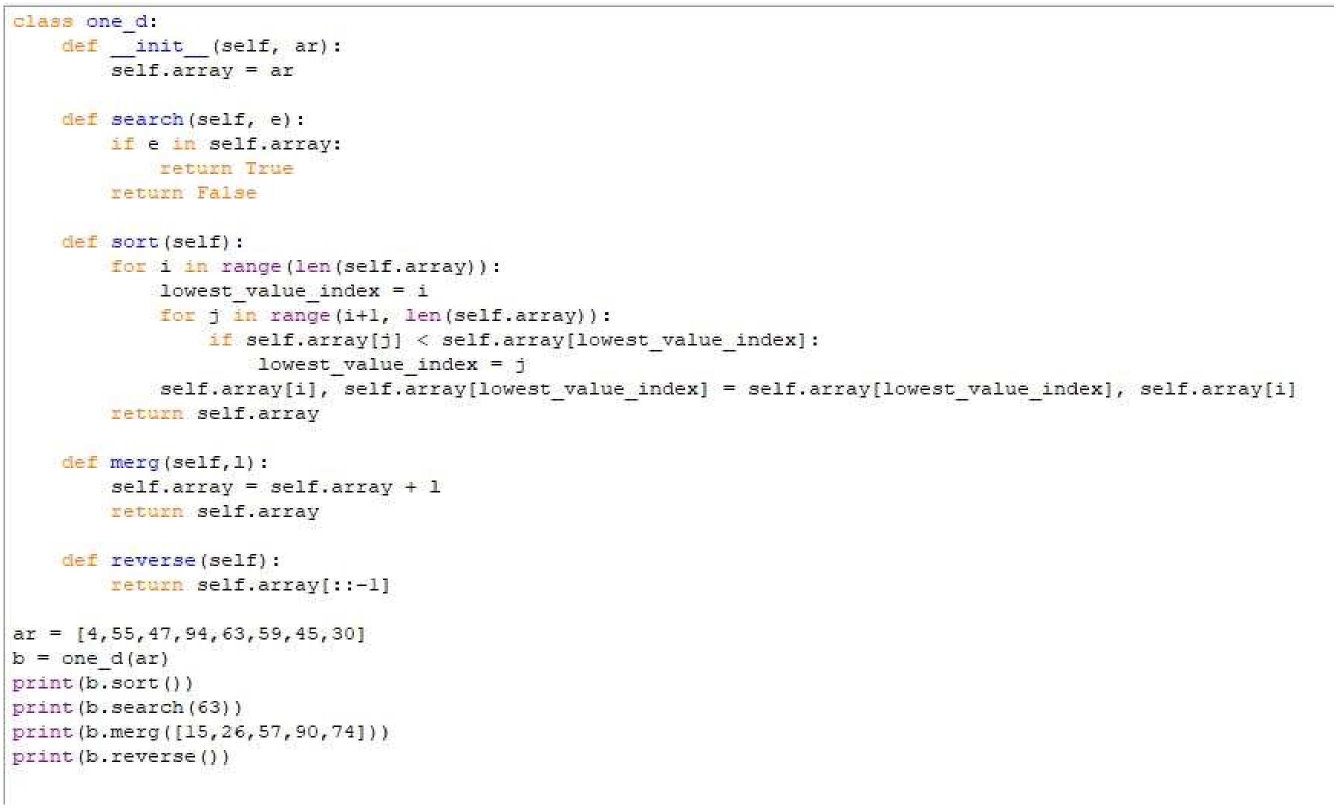
In this method, we neither reverse a list in-place(modify the original list), nor we create any copy of the list. Instead, we get a reverse iterator which we use to cycle through the list.

Using the reverse() built-in function.

Using the reverse() method we can reverse the contents of the list object in-place i.e., we don't need to create a new list instead we just copy the existing elements to the original list in reverse order. This method directly modifies the original list.

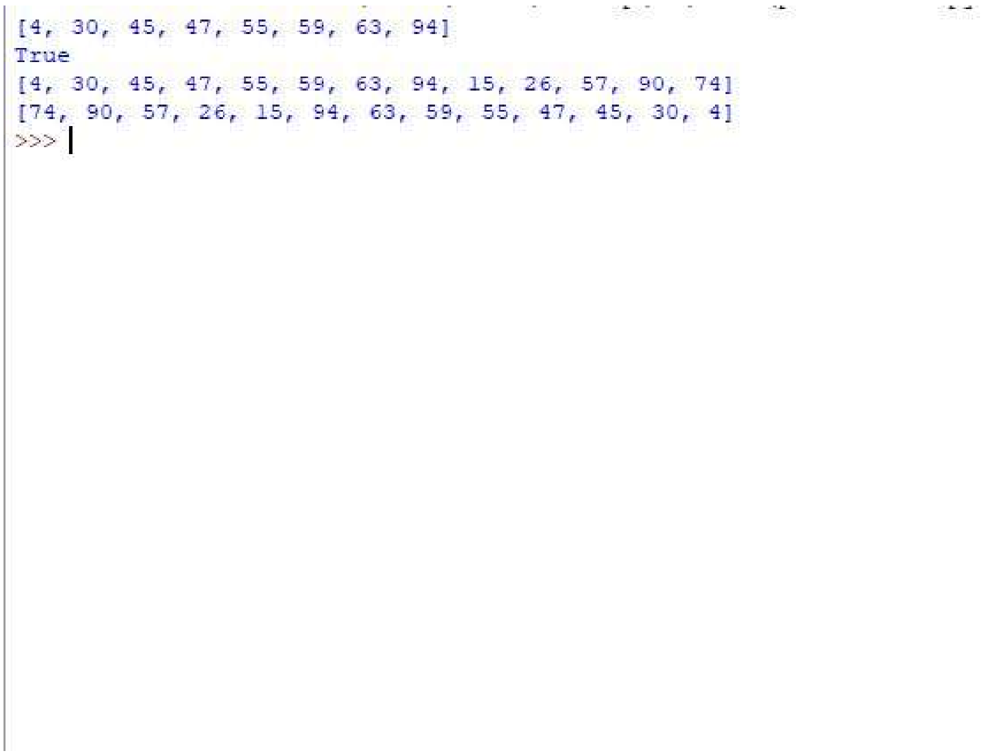
Nitin Singh SYCS 4059

Code :



```
class one_d:
    def __init__(self, ar):
        self.array = ar

    def search(self, e):
        if e in self.array:
            return True
        return False

    def sort(self):
        for i in range(len(self.array)):
            lowest_value_index = i
            for j in range(i+1, len(self.array)):
                if self.array[j] < self.array[lowest_value_index]:
                    lowest_value_index = j
            self.array[i], self.array[lowest_value_index] = self.array[lowest_value_index], self.array[i]
        return self.array

    def merg(self,l):
        self.array = self.array + l
        return self.array

    def reverse(self):
        return self.array[::-1]

ar = [4,55,47,94,63,59,45,30]
b = one_d(ar)
print(b.sort())
print(b.search(63))
print(b.merg([15,26,57,90,74]))
print(b.reverse())
```

Output :



```
[4, 30, 45, 47, 55, 59, 63, 94]
True
[4, 30, 45, 47, 55, 59, 63, 94, 15, 26, 57, 90, 74]
[74, 90, 57, 26, 15, 94, 63, 59, 55, 47, 45, 30, 4]
>>>
```

Practical No. 1.B

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

Github Repository Link :
https://github.com/NitinSingh1071/DS/blob/main/practical1b.py

Theory :

Matrix Addition: In Python, we can implement a matrix as a nested list (list inside a list). We can treat each element as a row of the matrix.

For example X = [[1, 2], [4, 5], [3, 6]] would represent a 3x2 matrix. First row can be selected as X[0] and the element in first row, first column can be selected as X[0][0].

We can perform matrix addition in various ways in Python:

Matrix Addition using Nested Loop.

Matrix Addition using Nested List Comprehension.

Matrix Multiplication: In Python, we can implement a matrix as nested list (list inside a list). We can treat each element as a row of the matrix.

For example X = [[1, 2], [4, 5], [3, 6]] would represent a 3x2 matrix.

The first row can be selected as X[0]. And, the element in first row, first column can be selected as X[0][0].

Multiplication of two matrices X and Y is defined only if the number of columns in X is equal to the number of rows Y.

If X is a n x m matrix and Y is a m x l matrix then, XY is defined and has the dimension n x l (but YX is not defined). Here are a couple of ways to implement matrix multiplication in Python.

We can perform matrix multiplication in various ways in Python:

Matrix Multiplication using Nested Loop.

Matrix Multiplication using Nested List Comprehension.

Matrix Transpose: In Python, we can implement a matrix as a nested list (list inside a list). We can treat each element as a row of the matrix. For example X = [[1, 2], [4, 5], [3, 6]] would represent a 3x2 matrix. The first row can be selected as X[0]. And, the element in the first-row first column can be selected as X[0][0].

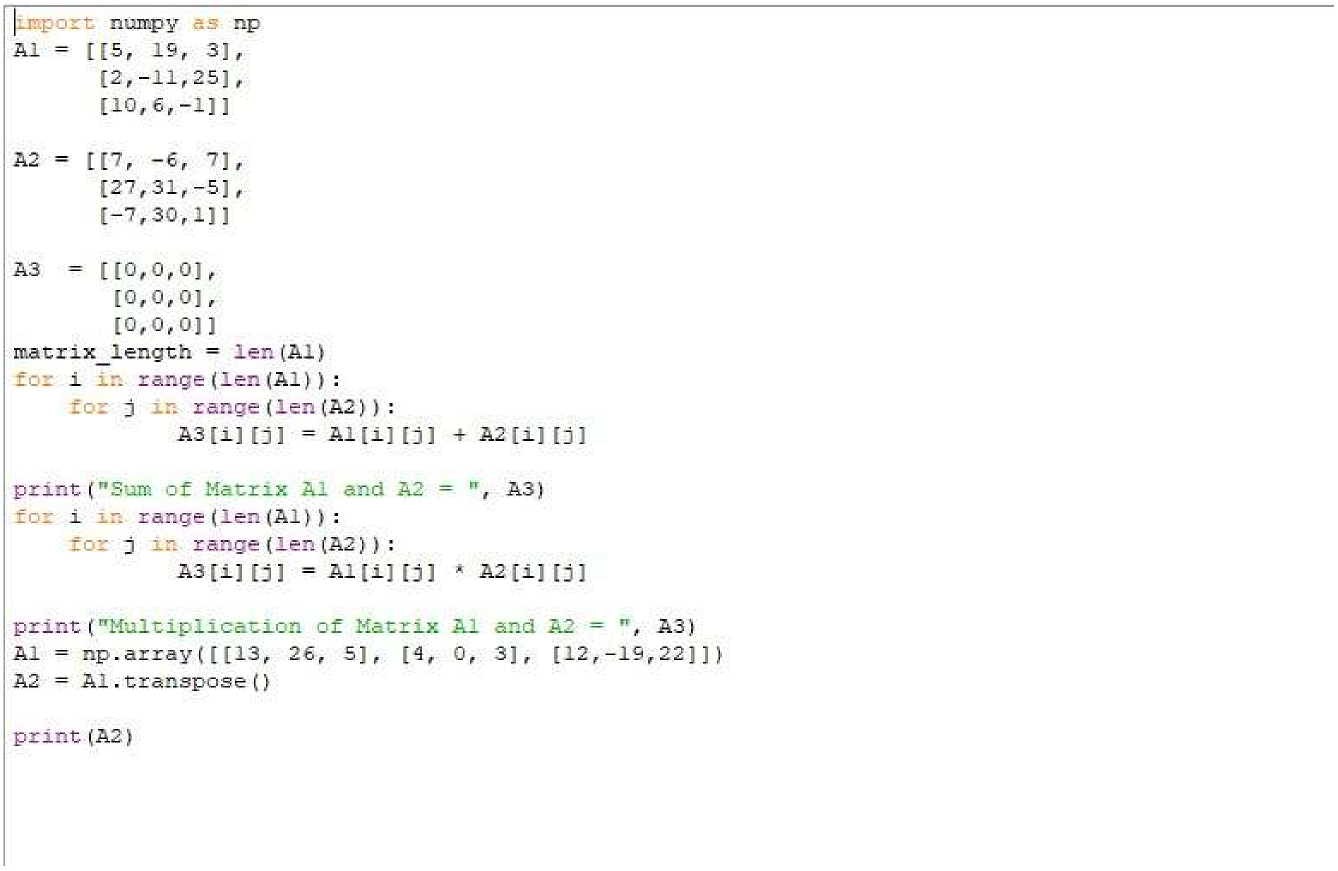
Transpose of a matrix is the interchanging of rows and columns. It is denoted as X'. The element at ith row and jth column in X will be placed at jth row and ith column in X'. So if X is a 3x2 matrix, X' will be a 2x3 matrix.

We can perform matrix transpose various ways in Python:

Matrix Transpose using Nested Loop.

Matrix Transpose using Nested List Comprehension.

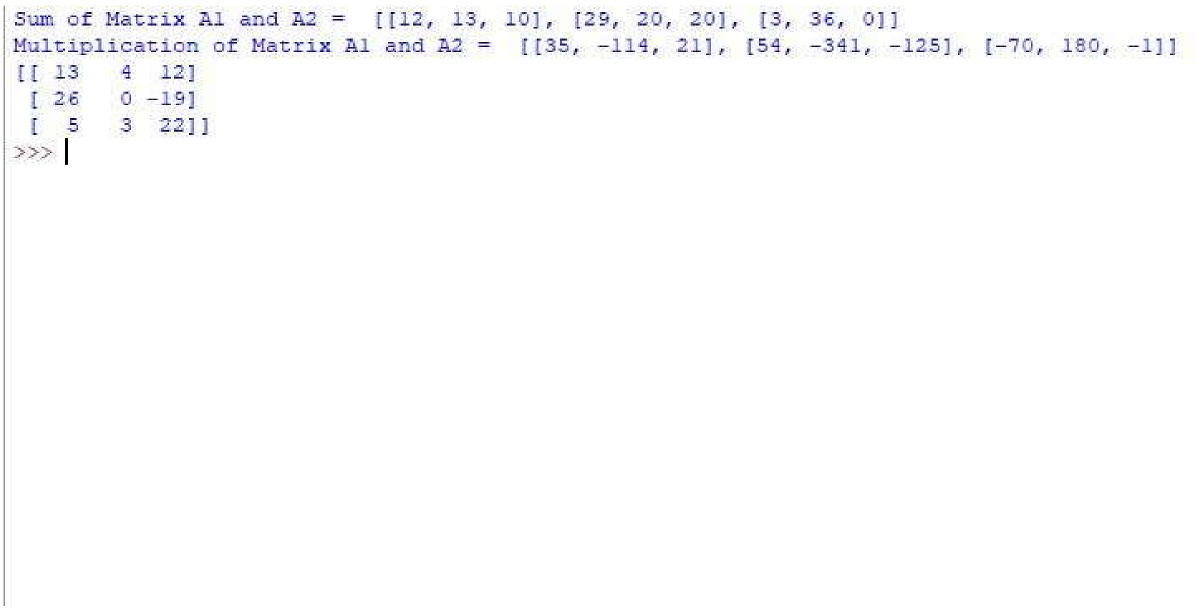
Code :



```
import numpy as np
A1 = [[5, 19, 3],
      [2,-11,25],
      [10,6,-1]]

A2 = [[7, -6, 7],
      [27,31,-5],
      [-7,30,1]]

A3  = [[0,0,0],
       [0,0,0],
       [0,0,0]]
matrix_length = len(A1)
for i in range(len(A1)):
    for j in range(len(A2)):
            A3[i][j] = A1[i][j] + A2[i][j]

print("Sum of Matrix A1 and A2 = ", A3)
for i in range(len(A1)):
    for j in range(len(A2)):
            A3[i][j] = A1[i][j] * A2[i][j]

print("Multiplication of Matrix A1 and A2 = ", A3)
A1 = np.array([[13, 26, 5], [4, 0, 3], [12,-19,22]])
A2 = A1.transpose()

print(A2)
```

Nitin Singh SYCS 4059

Output :



```
Sum of Matrix A1 and A2 =  [[12, 13, 10], [29, 20, 20], [3, 36, 0]]
Multiplication of Matrix A1 and A2 =  [[35, -114, 21], [54, -341, -125], [-70, 180, -1]]
[[ 13   4  12]
 [ 26   0 -19]
 [  5   3  22]]
>>> 
```

Nitin Singh SYCS 4059

Practical No. 2

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

Github Repository Link : https://github.com/NitinSingh1071/DS/blob/main/practical2.py

Theory :

Inserting element in the linked list involves reassigning the pointers from the existing nodes to the newly inserted node. Depending on whether the new data element is getting inserted at the beginning or at the middle or at the end of the linked list, we have the below scenarios.

Inserting at the Beginning of the Linked List

This involves pointing the next pointer of the new data node to the current head of the linked list. So the current head of the linked list becomes the second data element and the new node becomes the head of the linked list.

Inserting at the End of the Linked List

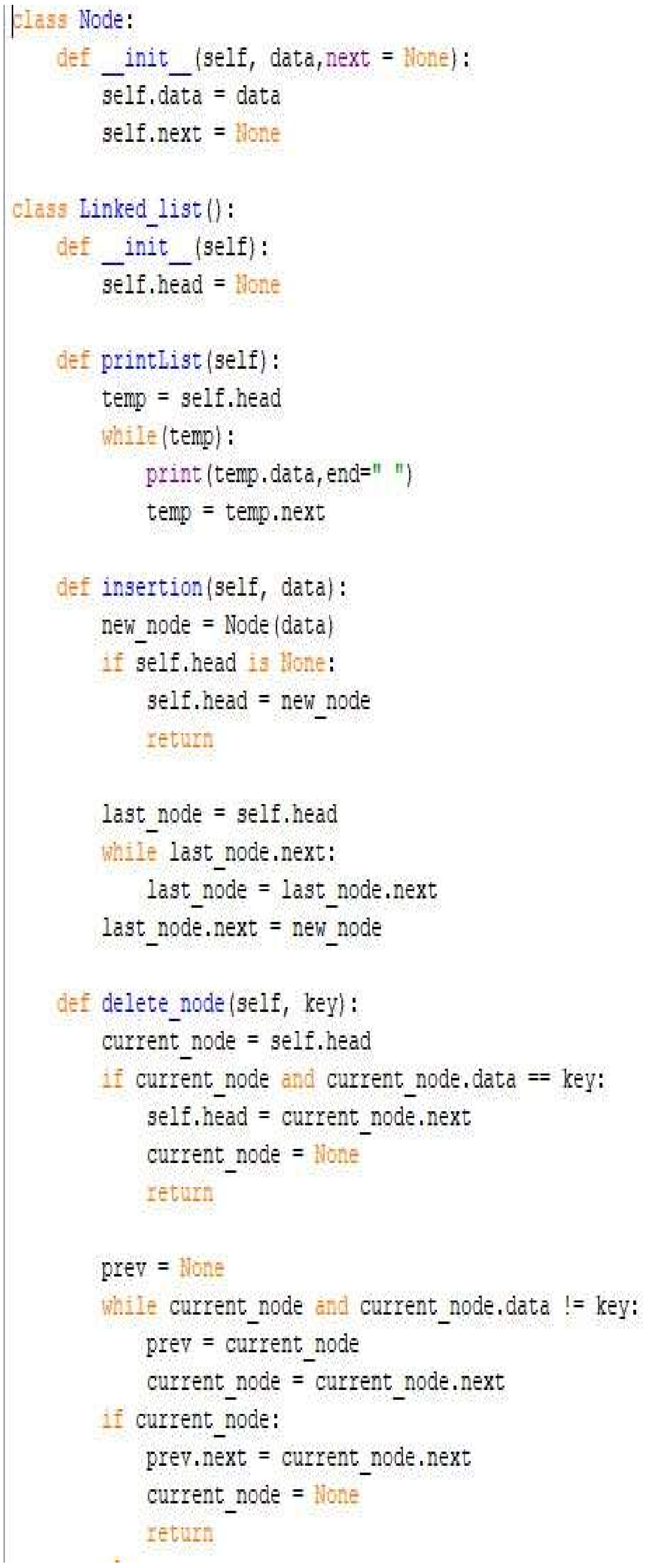
This involves pointing the next pointer of the the current last node of the linked list to the new data node. So the current last node of the linked list becomes the second last data node and the new node becomes the last node of the linked list.

We can remove an existing node using the key for that node. In the below program we locate the previous node of the node which is to be deleted. Then point the next pointer of this node to the next node of the node to be deleted.

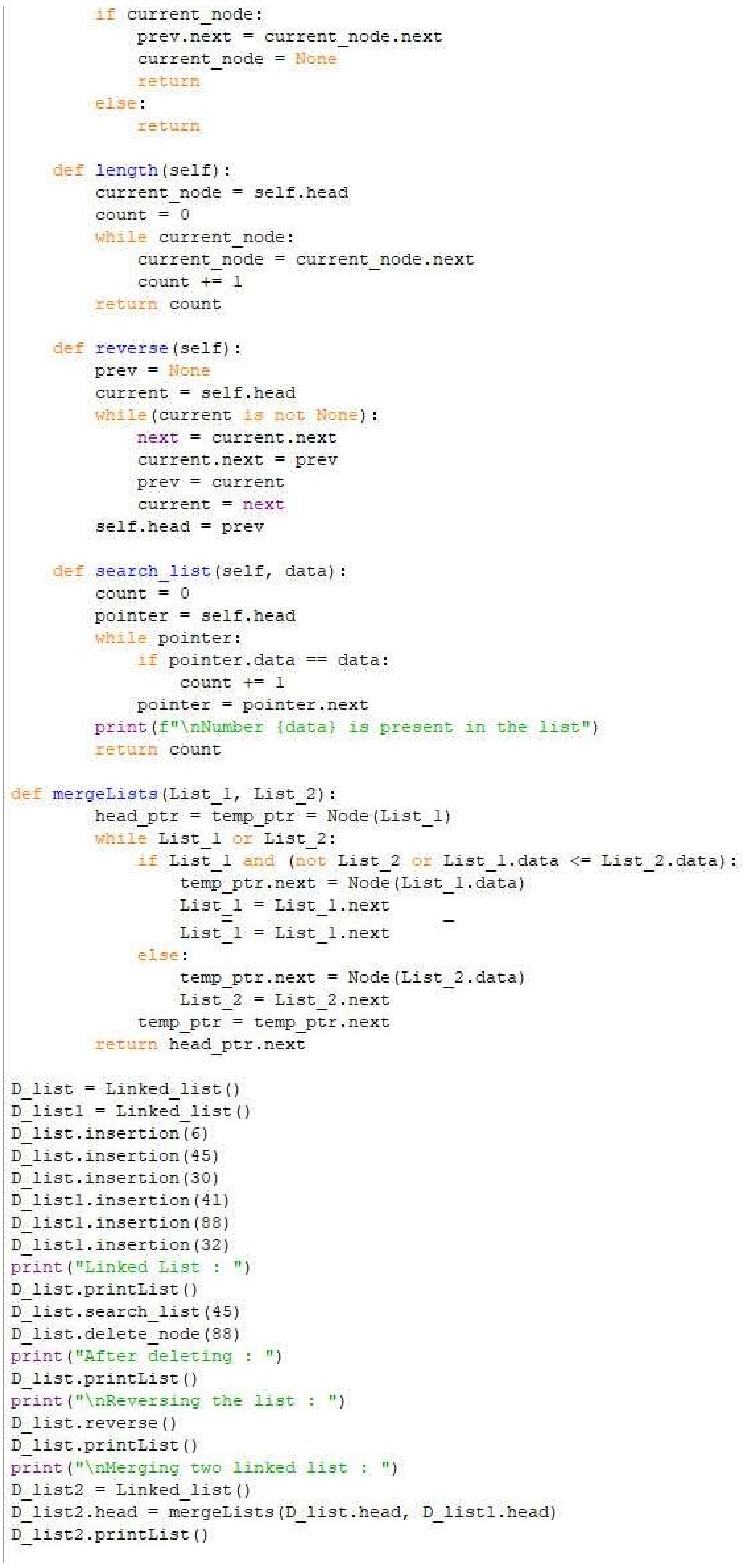
Singly linked lists can be traversed in only forward direction starting form the first data element. We simply print the value of the next data element by assigning the pointer of the next node to the current data element.

Nitin Singh SYCS 4059

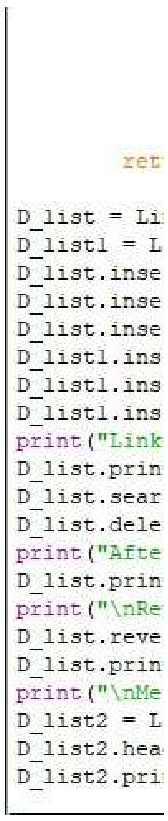
Code :



```
class Node:
    def __init__(self, data,next = None):
        self.data = data
        self.next = None

class Linked_list():
    def __init__(self):
        self.head = None

    def printList(self):
        temp = self.head
        while(temp):
            print(temp.data,end=" ")
            temp = temp.next

    def insertion(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
            return

        last_node = self.head
        while last_node.next:
            last_node = last_node.next
        last_node.next = new_node

    def delete_node(self, key):
        current_node = self.head
        if current_node and current_node.data == key:
            self.head = current_node.next
            current_node = None
            return

        prev = None
        while current_node and current_node.data != key:
            prev = current_node
            current_node = current_node.next
        if current_node:
            prev.next = current_node.next
            current_node = None
            return
```

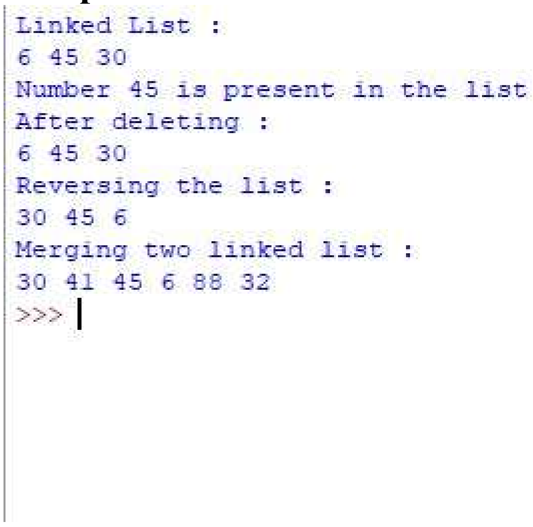


```
        if current_node:
            prev.next = current_node.next
            current_node = None
            return
        else:
            return

    def length(self):
        current_node = self.head
        count = 0
        while current_node:
            current_node = current_node.next
            count += 1
        return count

    def reverse(self):
        prev = None
        current = self.head
        while(current is not None):
            next = current.next
            current.next = prev
            prev = current
            current = next
        self.head = prev

    def search_list(self, data):
        count = 0
        pointer = self.head
        while pointer:
            if pointer.data == data:
                count += 1
            pointer = pointer.next
        print(f"\nNumber {data} is present in the list")
        return count

def mergeLists(List_1, List_2):
        head_ptr = temp_ptr = Node(List_1)
        while List_1 or List_2:
            if List_1 and (not List_2 or List_1.data <= List_2.data):
                temp_ptr.next = Node(List_1.data)
                List_1 = List_1.next
                List_1 = List_1.next
            else:
                temp_ptr.next = Node(List_2.data)
                List_2 = List_2.next
            temp_ptr = temp_ptr.next
        return head_ptr.next

D_list = Linked_list()
D_list1 = Linked_list()
D_list.insertion(6)
D_list.insertion(45)
D_list.insertion(30)
D_list1.insertion(41)
D_list1.insertion(88)
D_list1.insertion(32)
print("Linked List : ")
D_list.printList()
D_list.search_list(45)
D_list.delete_node(88)
print("After deleting : ")
D_list.printList()
print("\nReversing the list : ")
D_list.reverse()
D_list.printList()
print("\nMerging two linked list : ")
D_list2 = Linked_list()
D_list2.head = mergeLists(D_list.head, D_list1.head)
D_list2.printList()
```



```
                ret

D_list = Li
D_list1 = L
D_list.inse
D_list.inse
D_list.inse
D_list1.ins
D_list1.ins
D_list1.ins
print("Link
D_list.prin
D_list.sear
D_list.dele
print("Afte
D_list.prin
print("\nRe
D_list.reve
D_list.prin
print("\nMe
D_list2 = L
D_list2.hea
D_list2.pri
```

Output :


```
Linked List :
6 45 30
Number 45 is present in the list
After deleting :
6 45 30
Reversing the list :
30 45 6
Merging two linked list :
30 41 45 6 88 32
>>>
```

Practical No. 3.A

Aim : Perform Stack operations using Array implementation.

Github Repository Link :
https://github.com/NitinSingh1071/DS/blob/main/Practical3a.py

Theory :

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. Just define a one dimensional array of specific size and insert or delete the values into that array by using **LIFO principle** with the help of a variable called **'top'**. Initially, the top is set to -1. Whenever we want to insert a value into the stack, increment the top value by one and then insert. Whenever we want to delete a value from the stack, then delete the top value and decrement the top value by one.

push(value) - Inserting value into the stack

In a stack, push() is a function used to insert an element into the stack. In a stack, the new element is always inserted at top position. Push function takes one integer value as parameter and inserts that value into the stack. We can use the following steps to push an element on to the stack...

Step 1 - Check whether stack is FULL. (top == SIZE-1)

Step 2 - If it is FULL, then display "Stack is FULL!!! Insertion is not possible!!!" and terminate the function.

Step 3 - If it is NOT FULL, then increment top value by one (top++) and set stack[top] to value (stack[top] = value)

pop() - Delete a value from the Stack

In a stack, pop() is a function used to delete an element from the stack. In a stack, the element is always deleted from top position. Pop function does not take any value as parameter. We can use the following steps to pop an element from the stack...

Step 1 - Check whether stack is EMPTY. (top == -1)

Step 2 - If it is EMPTY, then display "Stack is EMPTY!!! Deletion is not possible!!!" and terminate the function

Step 3 - If it is NOT EMPTY, then delete stack[top] and decrement top value by one (top).

display() - Displays the elements of a Stack

We can use the following steps to display the elements of a stack...

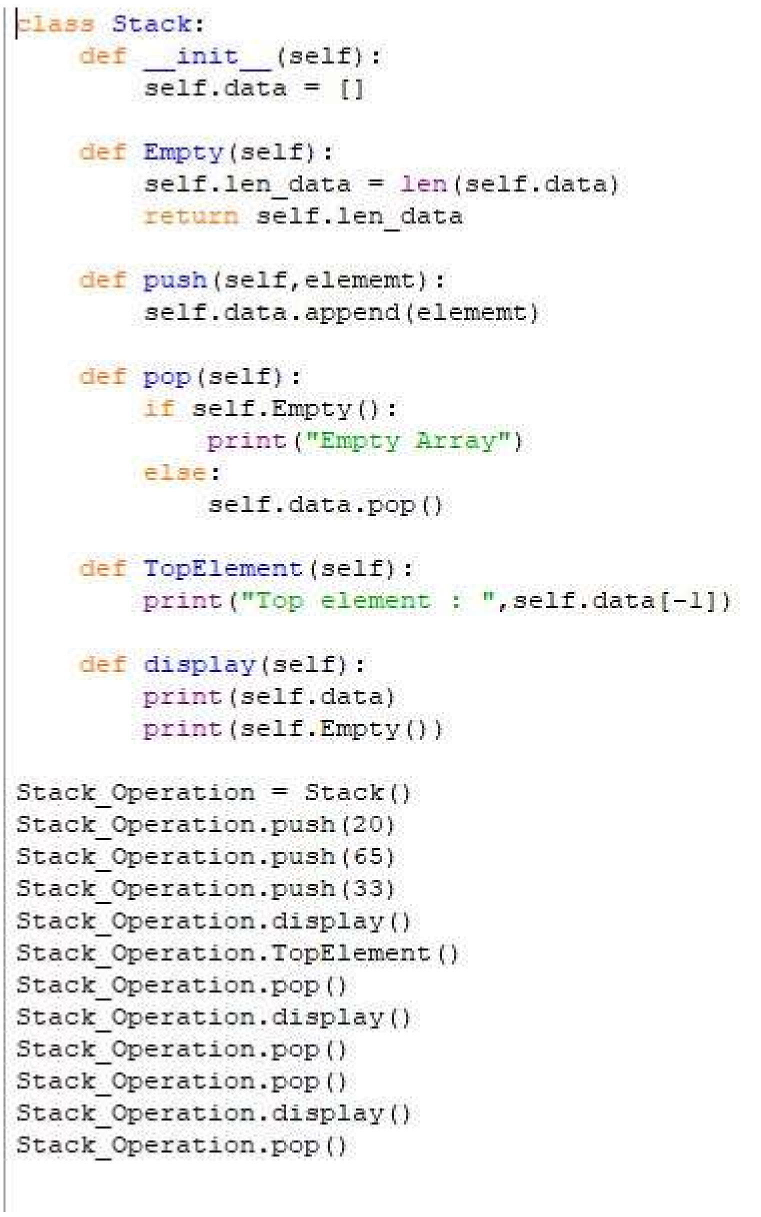
Step 1 - Check whether stack is EMPTY. (top == -1)

Step 2 - If it is EMPTY, then display "Stack is EMPTY!!!" and terminate the function.

Step 3 - If it is NOT EMPTY, then define a variable 'i' and initialize with top. Display stack[i] value and decrement i value by one (i--).

Step 4 - Repeat above step until i value becomes '0'.

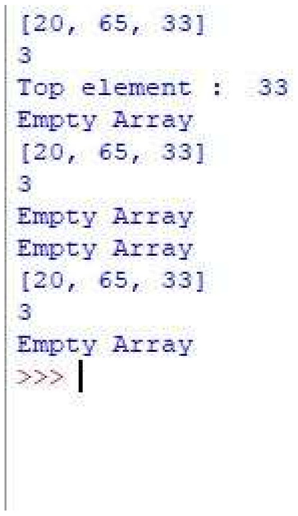
Code :



```
class Stack:
    def __init__(self):
        self.data = []

    def Empty(self):
        self.len_data = len(self.data)
        return self.len_data

    def push(self,elememt):
        self.data.append(elememt)

    def pop(self):
        if self.Empty():
            print("Empty Array")
        else:
            self.data.pop()

    def TopElement(self):
        print("Top element : ",self.data[-1])

    def display(self):
        print(self.data)
        print(self.Empty())

Stack_Operation = Stack()
Stack_Operation.push(20)
Stack_Operation.push(65)
Stack_Operation.push(33)
Stack_Operation.display()
Stack_Operation.TopElement()
Stack_Operation.pop()
Stack_Operation.display()
Stack_Operation.pop()
Stack_Operation.pop()
Stack_Operation.display()
Stack_Operation.pop()
```

Nitin Singh SYCS 4059

Output :



```
[20, 65, 33]
3
Top element :  33
Empty Array
[20, 65, 33]
3
Empty Array
Empty Array
[20, 65, 33]
3
Empty Array
>>> 
```

Practical No. 3.B

Aim : Implement Tower of Hanoi.

Github Repository Link :

https://github.com/NitinSingh1071/DS/blob/main/practical3b.py

Theory :

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

Simple rules:

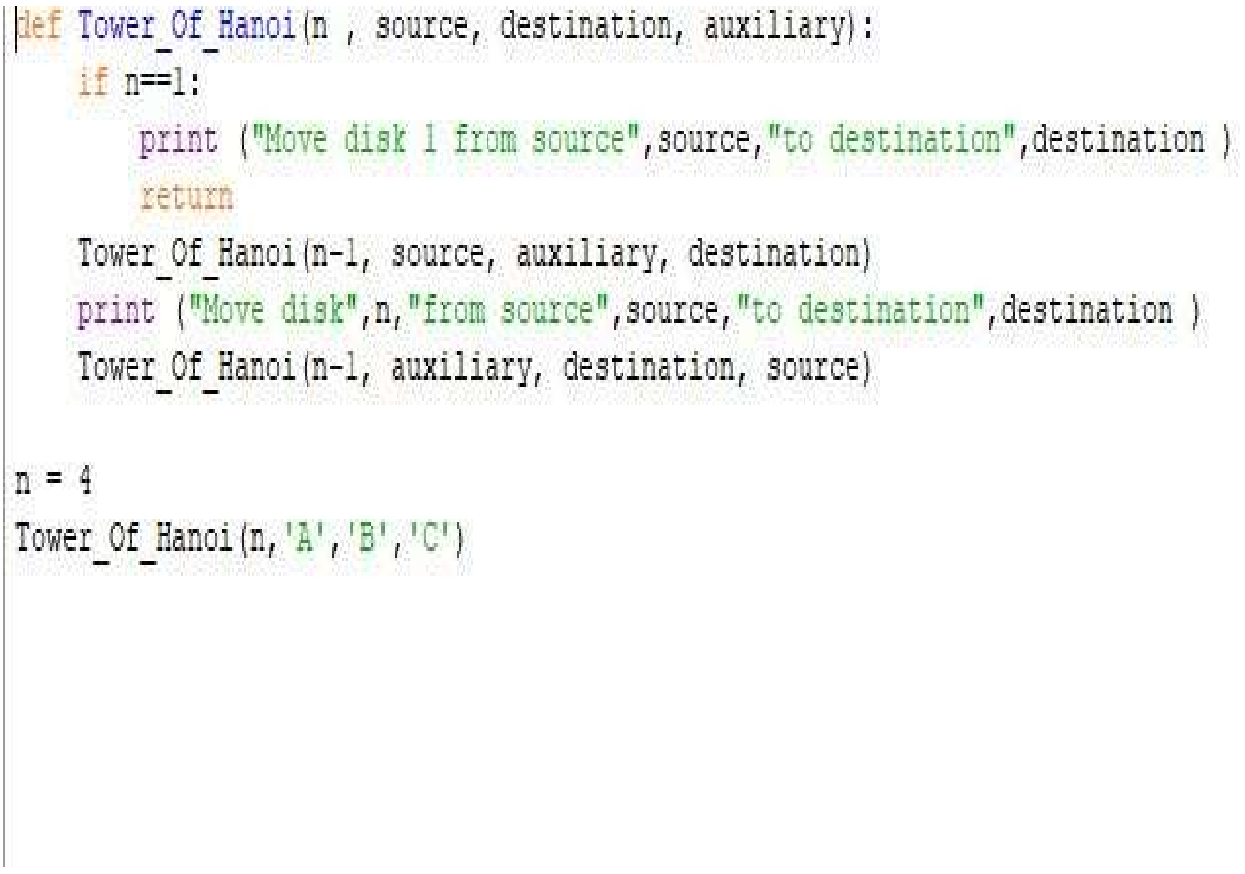
Only one disk can be moved at a time.

Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack i.e. a disk can only be moved if it is the uppermost disk on a stack.

No disk may be placed on top of a smaller disk.

Note: Transferring the top n-1 disks from source rod to Auxiliary rod can again be thought of as a fresh problem and can be solved in the same manner.

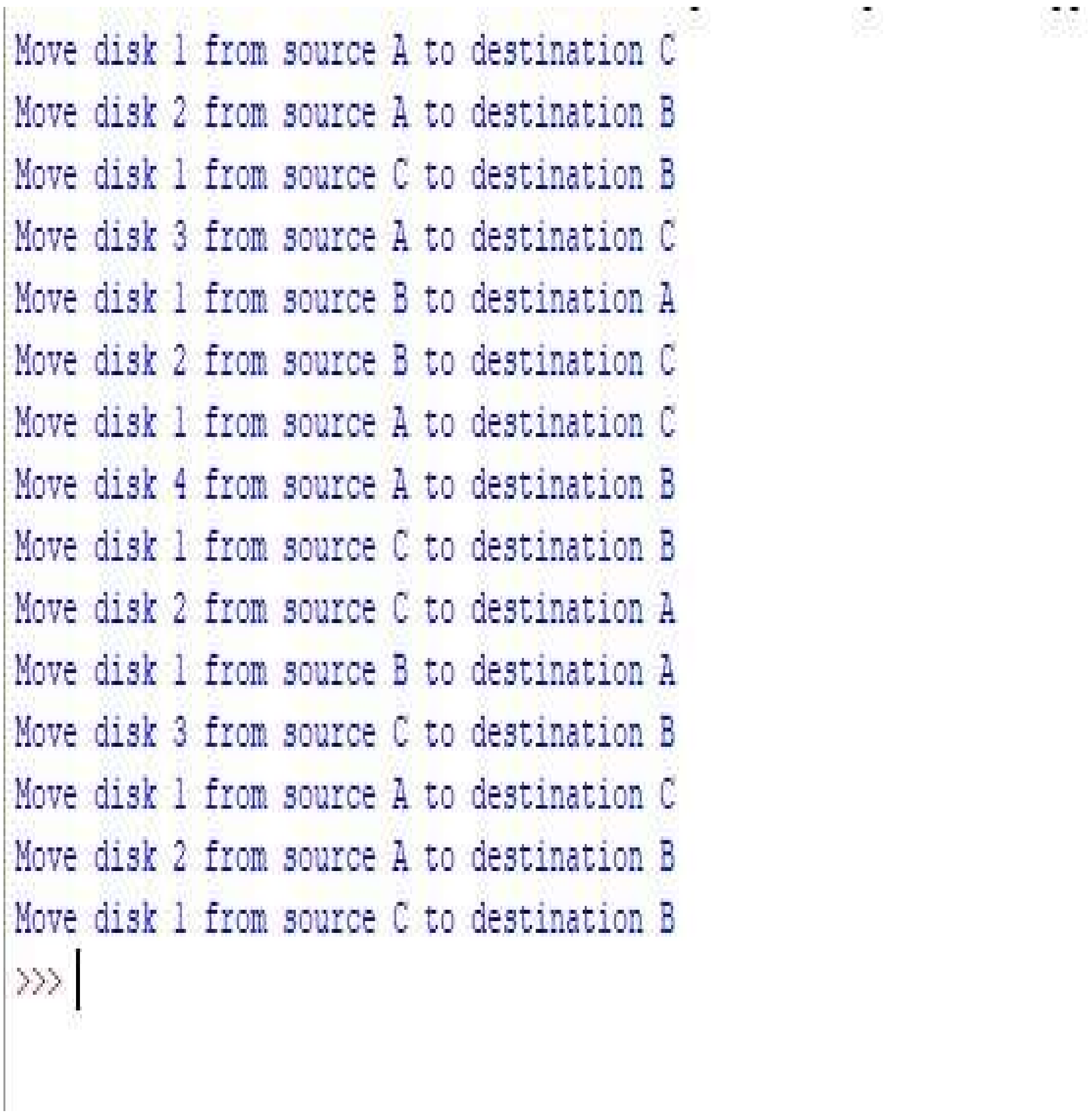
Code :



```
def Tower_Of_Hanoi(n , source, destination, auxiliary):
    if n==1:
        print ("Move disk 1 from source",source,"to destination",destination )
        return
    Tower_Of_Hanoi(n-1, source, auxiliary, destination)
    print ("Move disk",n,"from source",source,"to destination",destination )
    Tower_Of_Hanoi(n-1, auxiliary, destination, source)

n = 4
Tower_Of_Hanoi(n,'A','B','C')
```

Nitin Singh SYCS 4059

Output :



```
Move disk 1 from source A to destination C
Move disk 2 from source A to destination B
Move disk 1 from source C to destination B
Move disk 3 from source A to destination C
Move disk 1 from source B to destination A
Move disk 2 from source B to destination C
Move disk 1 from source A to destination C
Move disk 4 from source A to destination B
Move disk 1 from source C to destination B
Move disk 2 from source C to destination A
Move disk 1 from source B to destination A
Move disk 3 from source C to destination B
Move disk 1 from source A to destination C
Move disk 2 from source A to destination B
Move disk 1 from source C to destination B
>>>
```

Practical No. 3.C

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

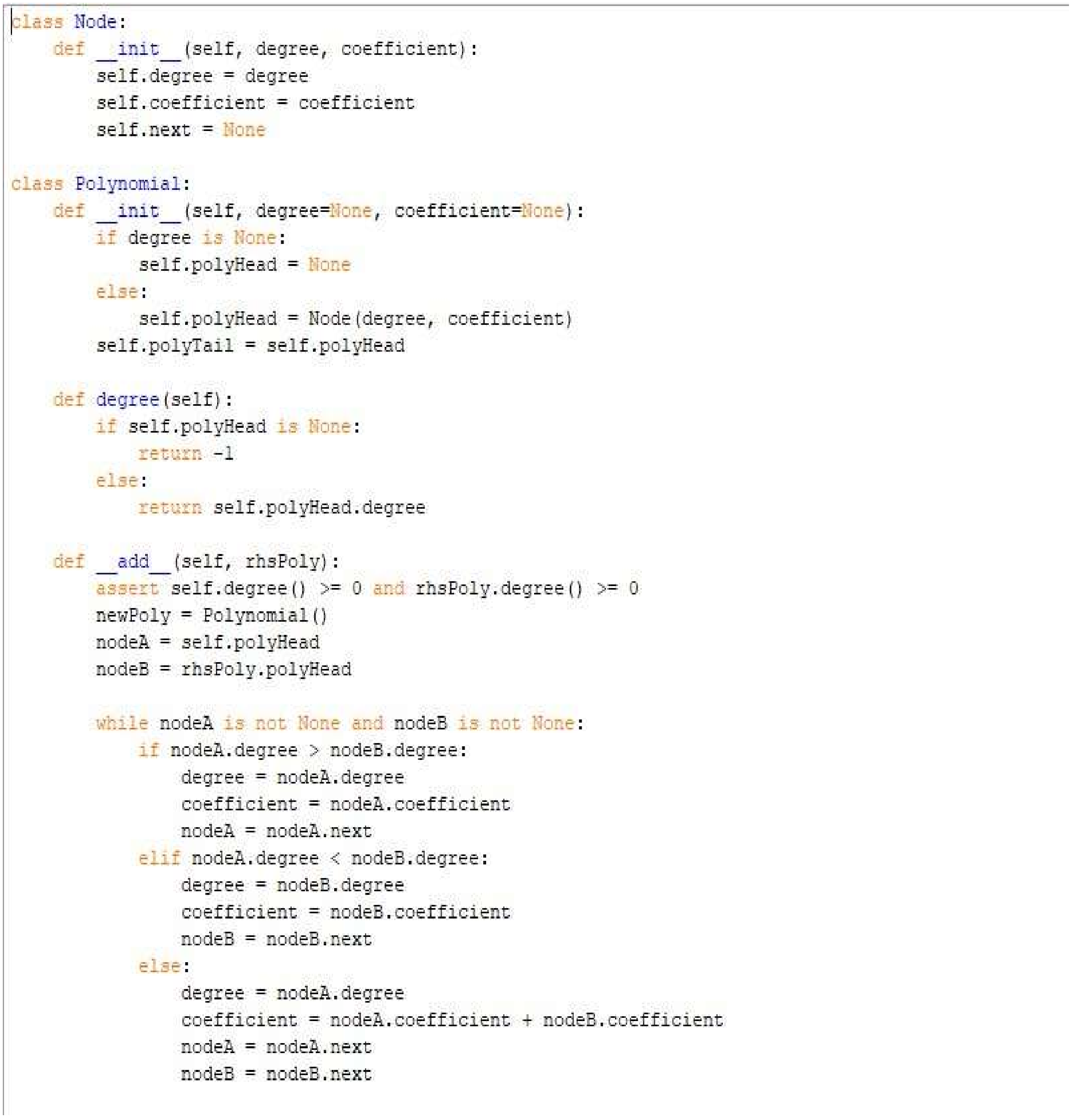
Github Repository Link :
https://github.com/NitinSingh1071/DS/blob/main/Practical3c.py

Theory :

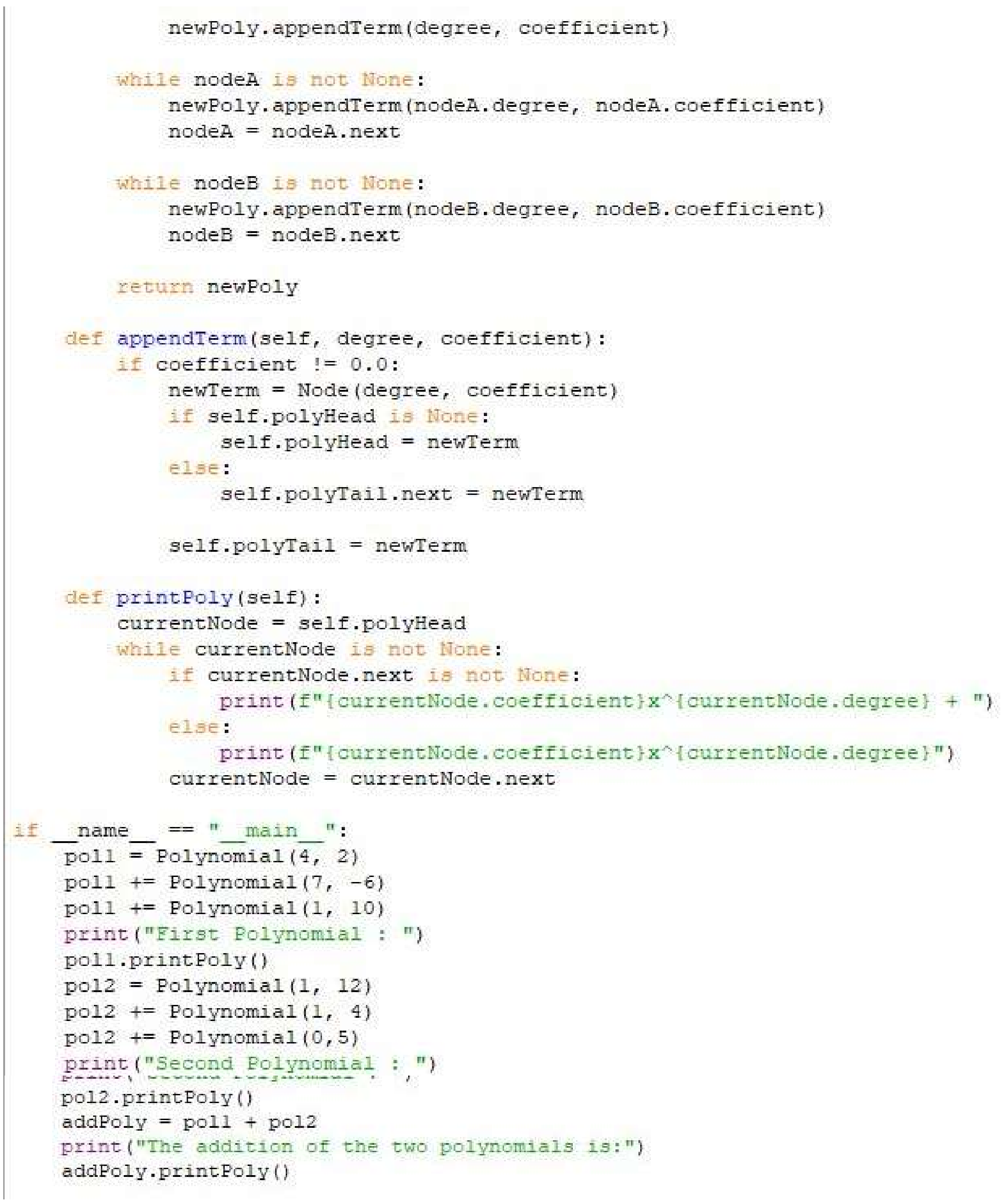
A linked list is a sequence of data elements, which are connected together via links. Each data element contains a connection to another data element in form of a pointer. Python does not have linked lists in its standard library. We implement the concept of linked lists using the concept of nodes as discussed in the previous chapter. We have already seen how we create a node class and how to traverse the elements of a node. In this chapter we are going to study the types of linked lists known as singly linked lists. In this type of data structure there is only one link between any two data elements. We create such a list and create additional methods to insert, update and remove elements from the list

Nitin Singh SYCS 4059

Code :

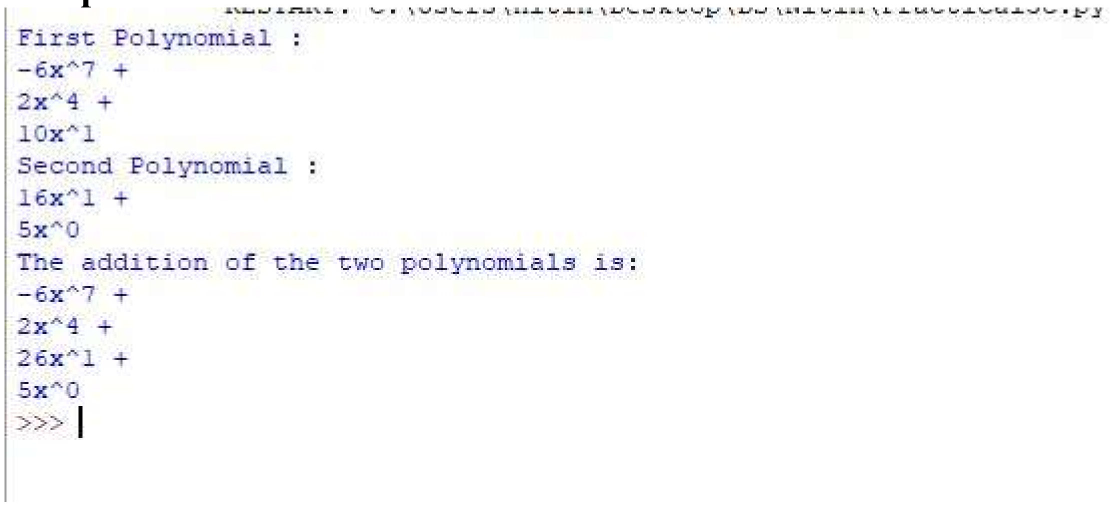


```
class Node:
    def __init__(self, degree, coefficient):
        self.degree = degree
        self.coefficient = coefficient
        self.next = None

class Polynomial:
    def __init__(self, degree=None, coefficient=None):
        if degree is None:
            self.polyHead = None
        else:
            self.polyHead = Node(degree, coefficient)
        self.polyTail = self.polyHead

    def degree(self):
        if self.polyHead is None:
            return -1
        else:
            return self.polyHead.degree

    def __add__(self, rhsPoly):
        assert self.degree() >= 0 and rhsPoly.degree() >= 0
        newPoly = Polynomial()
        nodeA = self.polyHead
        nodeB = rhsPoly.polyHead

        while nodeA is not None and nodeB is not None:
            if nodeA.degree > nodeB.degree:
                degree = nodeA.degree
                coefficient = nodeA.coefficient
                nodeA = nodeA.next
            elif nodeA.degree < nodeB.degree:
                degree = nodeB.degree
                coefficient = nodeB.coefficient
                nodeB = nodeB.next
            else:
                degree = nodeA.degree
                coefficient = nodeA.coefficient + nodeB.coefficient
                nodeA = nodeA.next
                nodeB = nodeB.next
```



```
            newPoly.appendTerm(degree, coefficient)

        while nodeA is not None:
            newPoly.appendTerm(nodeA.degree, nodeA.coefficient)
            nodeA = nodeA.next

        while nodeB is not None:
            newPoly.appendTerm(nodeB.degree, nodeB.coefficient)
            nodeB = nodeB.next

        return newPoly

    def appendTerm(self, degree, coefficient):
        if coefficient != 0.0:
            newTerm = Node(degree, coefficient)
            if self.polyHead is None:
                self.polyHead = newTerm
            else:
                self.polyTail.next = newTerm

            self.polyTail = newTerm

    def printPoly(self):
        currentNode = self.polyHead
        while currentNode is not None:
            if currentNode.next is not None:
                print(f"{currentNode.coefficient}x^{currentNode.degree} + ")
            else:
                print(f"{currentNode.coefficient}x^{currentNode.degree}")
            currentNode = currentNode.next

if __name__ == "__main__":
    pol1 = Polynomial(4, 2)
    pol1 += Polynomial(7, -6)
    pol1 += Polynomial(1, 10)
    print("First Polynomial : ")
    pol1.printPoly()
    pol2 = Polynomial(1, 12)
    pol2 += Polynomial(1, 4)
    pol2 += Polynomial(0,5)
    print("Second Polynomial : ")
    pol2.printPoly()
    addPoly = pol1 + pol2
    print("The addition of the two polynomials is:")
    addPoly.printPoly()
```

Output :



```
[illegible]
First Polynomial :
-6x^7 +
2x^4 +
10x^1
Second Polynomial :
16x^1 +
5x^0
The addition of the two polynomials is:
-6x^7 +
2x^4 +
26x^1 +
5x^0
>>>
```

Practical No. 3.D

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

Github Repository Link : https://github.com/NitinSingh1071/DS/blob/main/Practical3d.py

Theory :

Factorial of Numbers:

The factorial of a number is the product of all the integers from 1 to that number.

For example, the factorial of 6 is 1*2*3*4*5*6 = 720. Factorial is not defined for negative numbers and the factorial of zero is one, 0! = 1.

Iterative Solution:

Factorial can also be calculated iteratively . Here we have shown the iterative approach using both for and while loop.

Recursive Solution:

Factorial can also be calculated recursively. Recursive is calling the function inside of a function.

Factors of the Numbers:

In order to find factors of a number, we have to run a loop over all numbers from 1 to itself and see if it is divisible. If i is able to divide num completely, it is added in the list. Finally the list is displayed as the factors of given number

Iterative Solution:

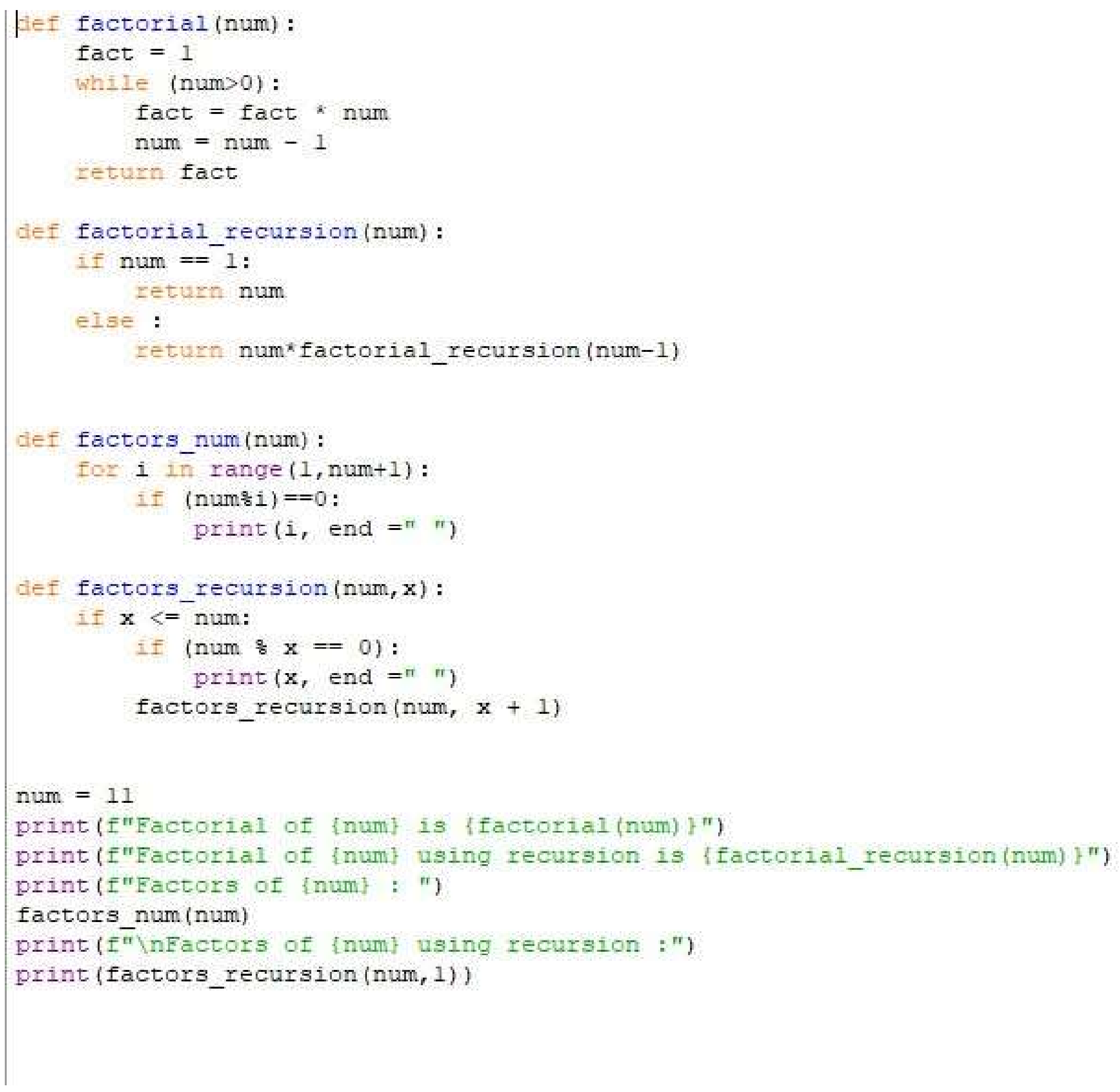
Factors can also be calculated iteratively . Here we have shown the iterative approach using both for and while loop.

Recursive Solution:

Factorial can also be calculated recursively. Recursive is calling the function inside of a function.

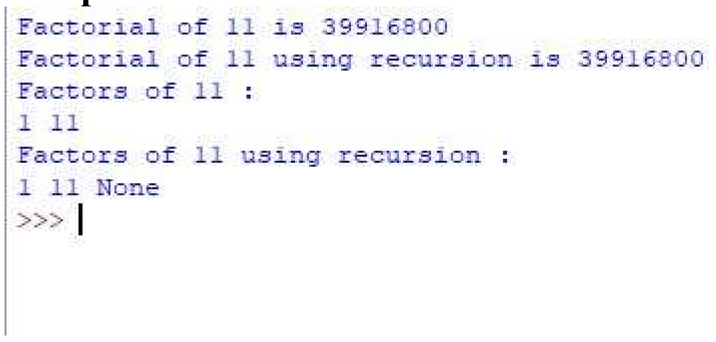
Code :

Nitin Singh SYCS 4059



```
def factorial(num):
    fact = 1
    while (num>0):
        fact = fact * num
        num = num - 1
    return fact

def factorial_recursion(num):
    if num == 1:
        return num
    else :
        return num*factorial_recursion(num-1)


def factors_num(num):
    for i in range(1,num+1):
        if (num%i)==0:
            print(i, end =" ")

def factors_recursion(num,x):
    if x <= num:
        if (num % x == 0):
            print(x, end =" ")
        factors_recursion(num, x + 1)


num = 11
print(f"Factorial of {num} is {factorial(num)}")
print(f"Factorial of {num} using recursion is {factorial_recursion(num)}")
print(f"Factors of {num} : ")
factors_num(num)
print(f"\nFactors of {num} using recursion :")
print(factors_recursion(num,1))
```

Output :



```
Factorial of 11 is 39916800
Factorial of 11 using recursion is 39916800
Factors of 11 :
1 11
Factors of 11 using recursion :
1 11 None
>>> 
```

Practical No. 4

Aim : Perform Queues operations using Circular Array implementation.

Github Repository Link :
https://github.com/NitinSingh1071/DS/blob/main/practical4.py

Theory :

In the circular queue, the first index comes right after the last index. You can think of a circular queue as shown in the following figure. Circular queue will be full when front = -1 and rear = max-1. Implementation of circular queue is similar to that of a linear queue. Only the logic part that is implemented in the case of insertion and deletion is different from that in a linear queue.

There are three scenario of inserting an element in a queue.

If (rear + 1)%maxsize = front, the queue is full. In that case, overflow occurs and therefore, insertion can not be performed in the queue.

If rear != max - 1, then rear will be incremented to the mod(maxsize) and the new value will be inserted at the rear end of the queue.

If front != 0 and rear = max - 1, then it means that queue is not full therefore, set the value of rear to 0 and insert the new element there.

To delete an element from the circular queue, we must check for the three following conditions.

If front = -1, then there are no elements in the queue and therefore this will be the case of an underflow condition.

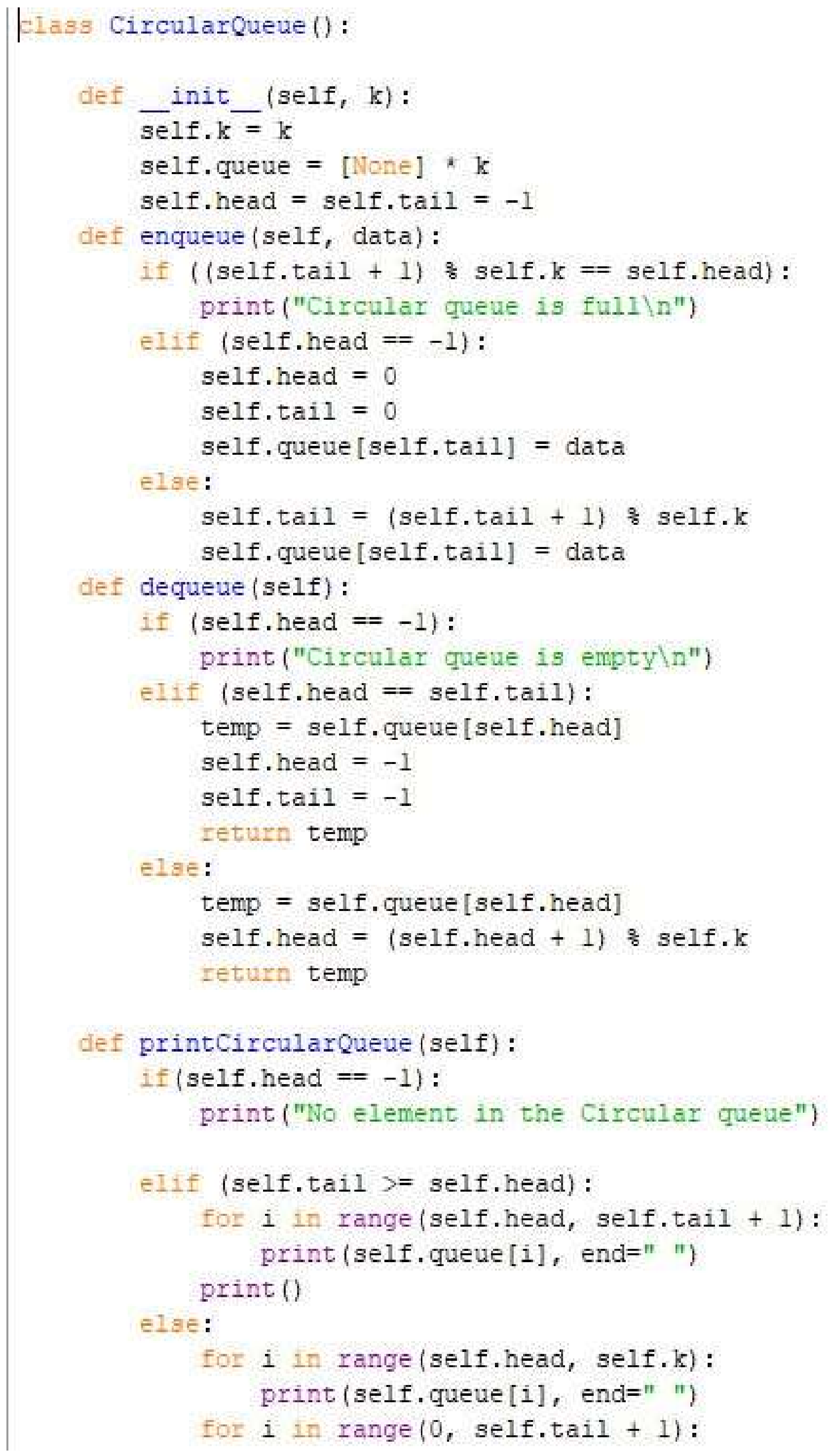
If there is only one element in the queue, in this case, the condition rear = front holds and therefore, both are set to -1 and the queue is deleted completely.

If front = max -1 then, the value is deleted from the front end the value of front is set to 0.

Otherwise, the value of front is incremented by 1 and then delete the element at the front end.

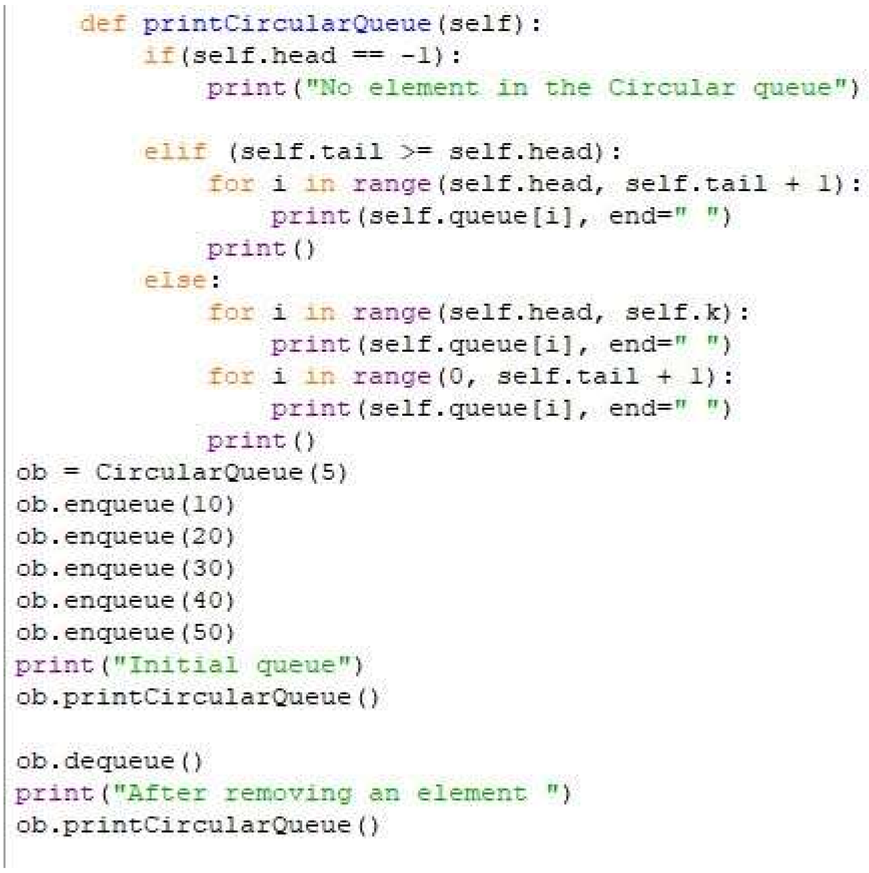
Nitin Singh SYCS 4059

Code :



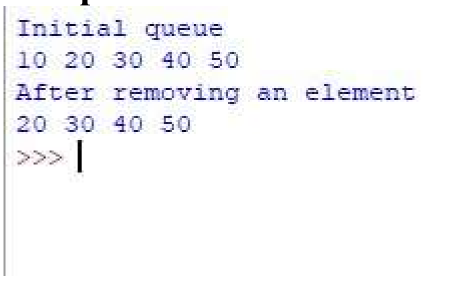
```
class CircularQueue():

    def __init__(self, k):
        self.k = k
        self.queue = [None] * k
        self.head = self.tail = -1
    def enqueue(self, data):
        if ((self.tail + 1) % self.k == self.head):
            print("Circular queue is full\n")
        elif (self.head == -1):
            self.head = 0
            self.tail = 0
            self.queue[self.tail] = data
        else:
            self.tail = (self.tail + 1) % self.k
            self.queue[self.tail] = data
    def dequeue(self):
        if (self.head == -1):
            print("Circular queue is empty\n")
        elif (self.head == self.tail):
            temp = self.queue[self.head]
            self.head = -1
            self.tail = -1
            return temp
        else:
            temp = self.queue[self.head]
            self.head = (self.head + 1) % self.k
            return temp

    def printCircularQueue(self):
        if(self.head == -1):
            print("No element in the Circular queue")

        elif (self.tail >= self.head):
            for i in range(self.head, self.tail + 1):
                print(self.queue[i], end=" ")
            print()
        else:
            for i in range(self.head, self.k):
                print(self.queue[i], end=" ")
            for i in range(0, self.tail + 1):
```

Nitin Singh SYCS 4059



```
    def printCircularQueue(self):
        if(self.head == -1):
            print("No element in the Circular queue")

        elif (self.tail >= self.head):
            for i in range(self.head, self.tail + 1):
                print(self.queue[i], end=" ")
            print()
        else:
            for i in range(self.head, self.k):
                print(self.queue[i], end=" ")
            for i in range(0, self.tail + 1):
                print(self.queue[i], end=" ")
            print()
ob = CircularQueue(5)
ob.enqueue(10)
ob.enqueue(20)
ob.enqueue(30)
ob.enqueue(40)
ob.enqueue(50)
print("Initial queue")
ob.printCircularQueue()

ob.dequeue()
print("After removing an element ")
ob.printCircularQueue()
```

Output :



```
Initial queue
10 20 30 40 50
After removing an element
20 30 40 50
>>> 
```

Practical No. 5

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

Github Repository Link :
https://github.com/NitinSingh1071/DS/blob/main/practical5.py

Theory :

Searching is a very basic necessity when you store data in different data structures. The simplest approach is to go across every element in the data structure and match it with the value you are searching for. This is known as Linear search. It is inefficient and rarely used, but creating a program for it gives an idea about how we can implement some advanced search algorithms.

Linear Search:

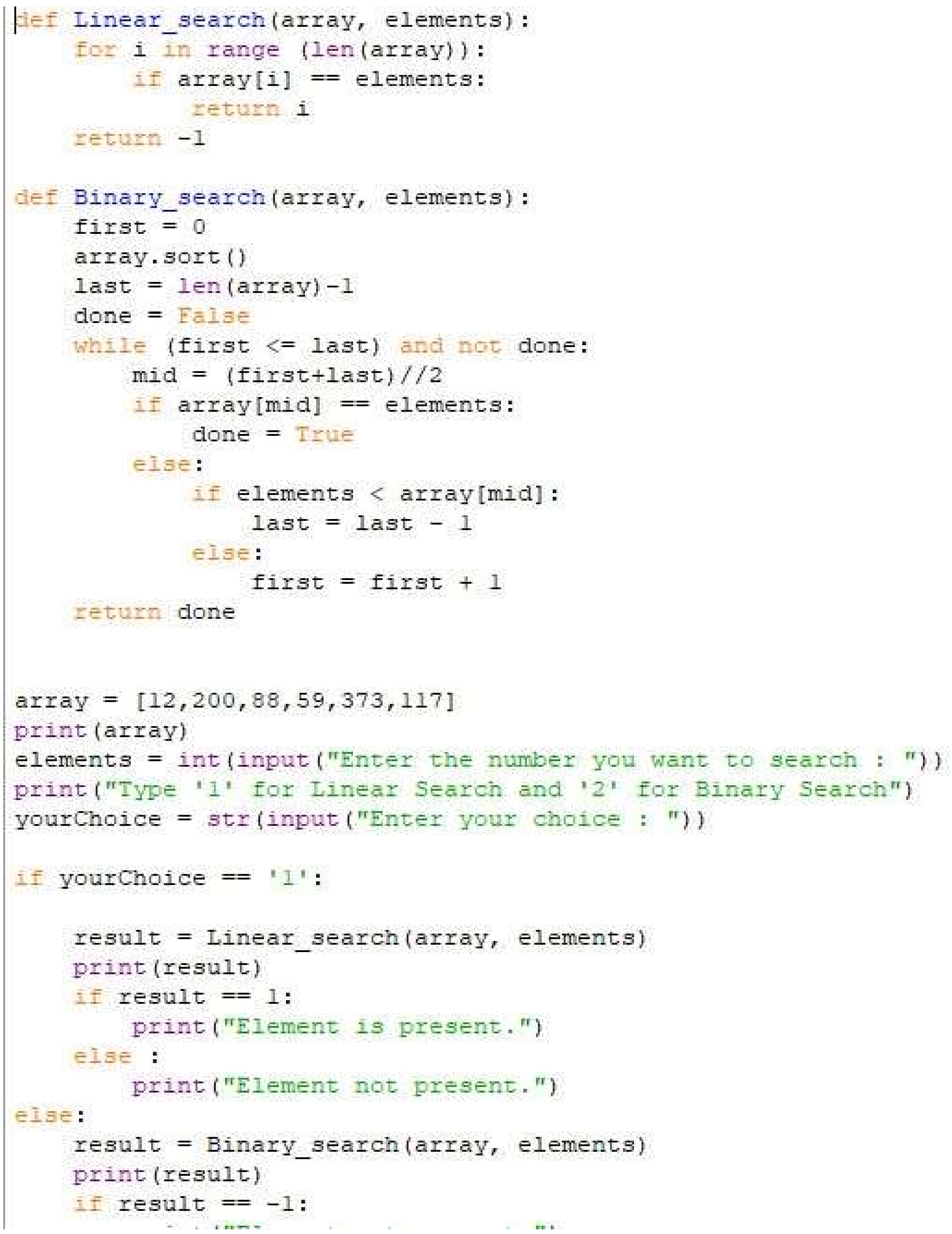
In this type of search, a sequential search is made over all items one by one. Every item is checked and if a match is found then that particular item is returned, otherwise the search continue still the end of the data structure.

Binary Search:

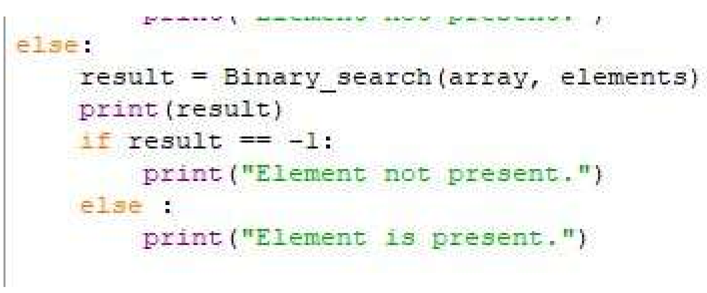
Search a sorted array by repeatedly dividing the search interval in half. Begin with an interval covering the whole array. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise narrow it to the upper half. Repeatedly check until the value is found or the interval is empty.

Code :

Nitin Singh SYCS 4059



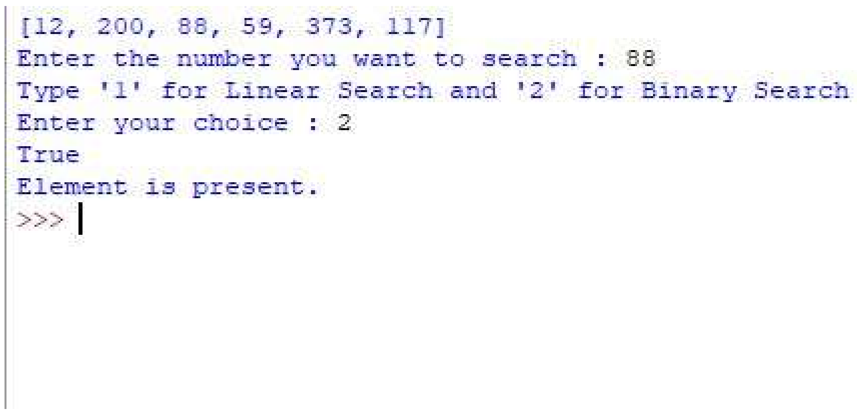
```
def Linear_search(array, elements):
    for i in range (len(array)):
        if array[i] == elements:
            return i
    return -1

def Binary_search(array, elements):
    first = 0
    array.sort()
    last = len(array)-1
    done = False
    while (first <= last) and not done:
        mid = (first+last)//2
        if array[mid] == elements:
            done = True
        else:
            if elements < array[mid]:
                last = last - 1
            else:
                first = first + 1
    return done


array = [12,200,88,59,373,117]
print(array)
elements = int(input("Enter the number you want to search : "))
print("Type '1' for Linear Search and '2' for Binary Search")
yourChoice = str(input("Enter your choice : "))

if yourChoice == '1':

    result = Linear_search(array, elements)
    print(result)
    if result == 1:
        print("Element is present.")
    else :
        print("Element not present.")
else:
    result = Binary_search(array, elements)
    print(result)
    if result == -1:
        [illegible]
```



```
        [illegible]
else:
    result = Binary_search(array, elements)
    print(result)
    if result == -1:
        print("Element not present.")
    else :
        print("Element is present.")
```

Nitin Singh SYCS 4059

Output :



```
[12, 200, 88, 59, 373, 117]
Enter the number you want to search : 88
Type '1' for Linear Search and '2' for Binary Search
Enter your choice : 2
True
Element is present.
>>> 
```

Practical No. 6

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

Github Repository Link :
https://github.com/NitinSingh1071/DS/blob/main/practical6.py

Theory :

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.

The importance of sorting lies in the fact that data searching can be optimized to a very high level, if data is stored in a sorted manner. Sorting is also used to represent data in more readable formats. Below we see five such implementations of sorting in python.

Bubble Sort

It is a comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order.

Insertion Sort

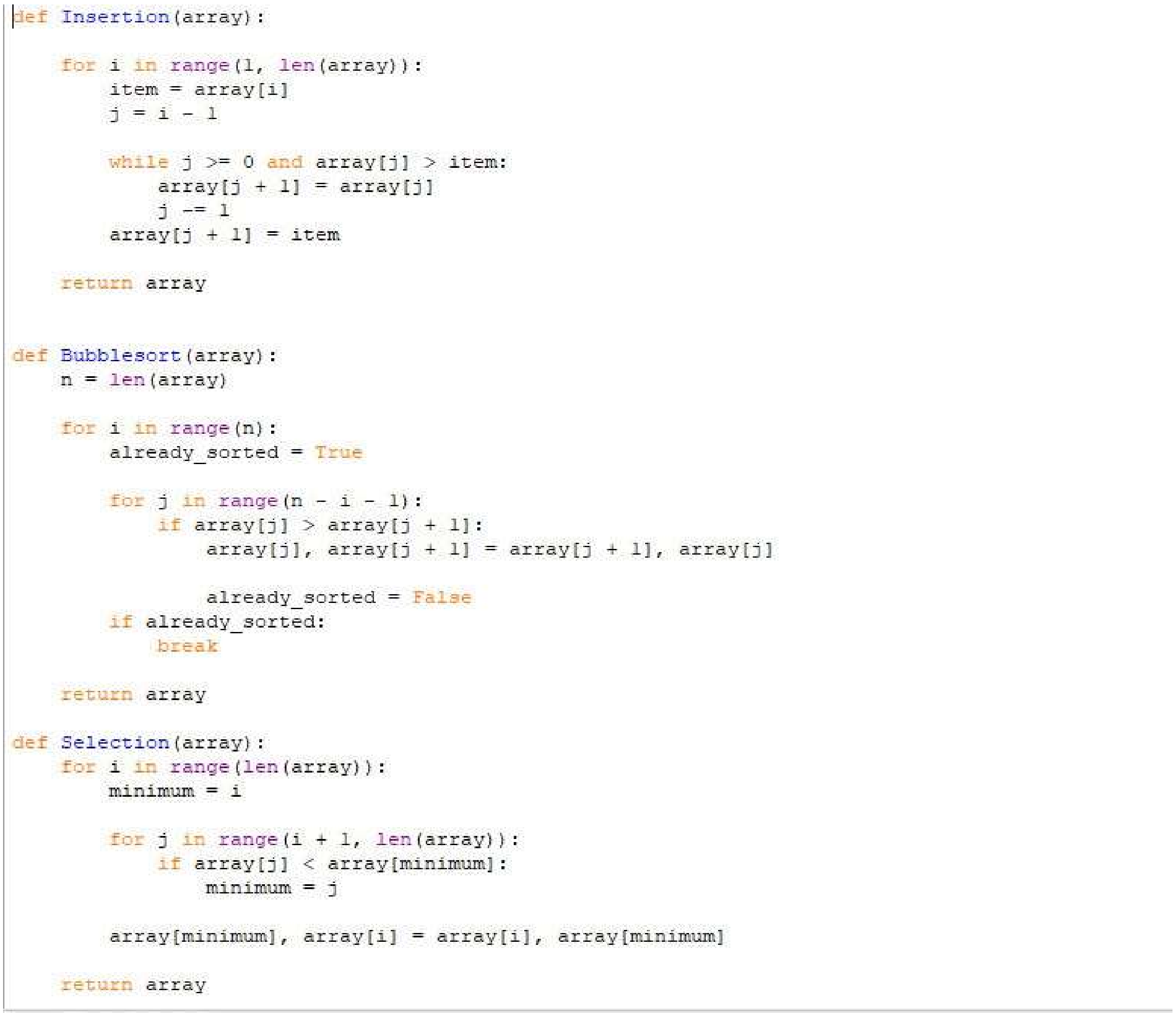
Insertion sort involves finding the right place for a given element in a sorted list. So in beginning we compare the first two elements and sort them by comparing them. Then we pick the third element and find its proper position among the previous two sorted elements. This way we gradually go on adding more elements to the already sorted list by putting them in their proper position.

Selection Sort

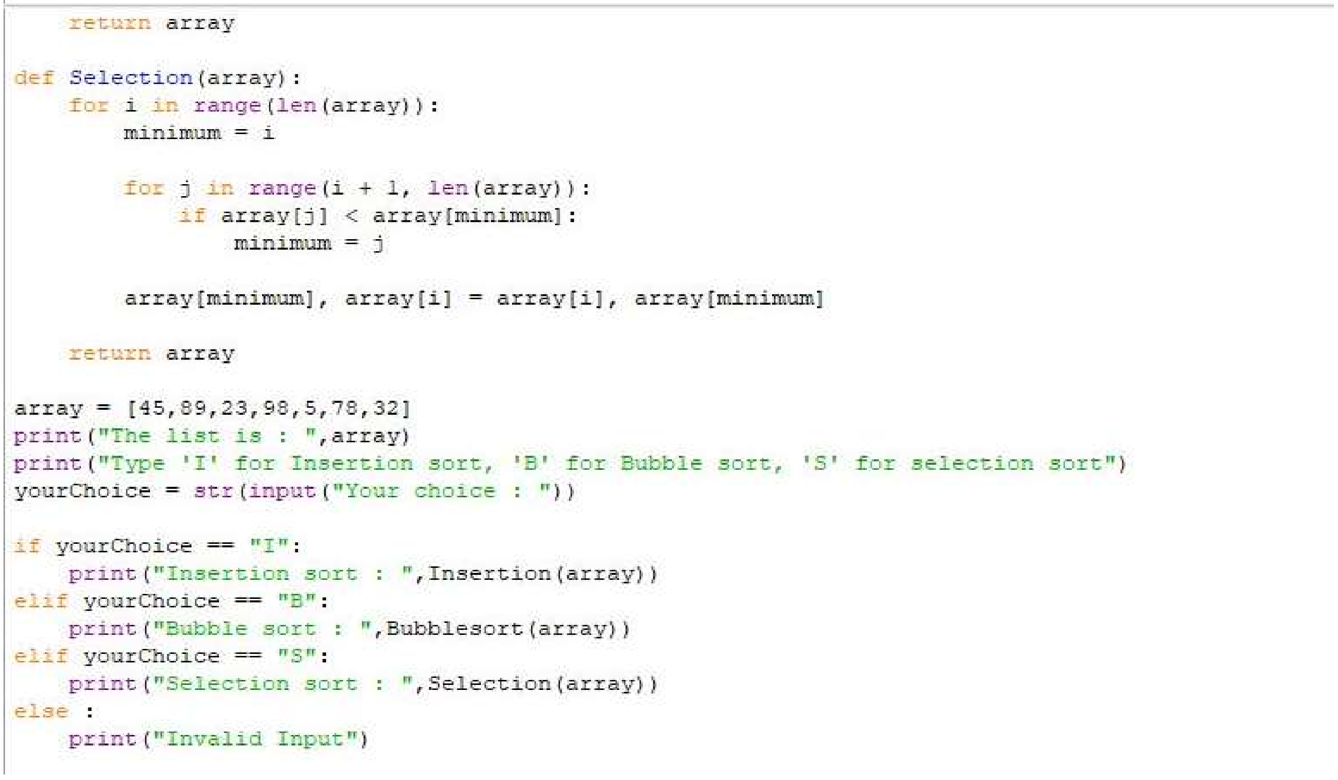
In selection sort we start by finding the minimum value in a given list and move it to a sorted list. Then we repeat the process for each of the remaining elements in the unsorted list. The next element entering the sorted list is compared with the existing elements and placed at its correct position. So at the end all the elements from the unsorted list are sorted.

Nitin Singh SYCS 4059

Code :



```
def Insertion(array):

    for i in range(1, len(array)):
        item = array[i]
        j = i - 1

        while j >= 0 and array[j] > item:
            array[j + 1] = array[j]
            j -= 1
        array[j + 1] = item

    return array


def Bubblesort(array):
    n = len(array)

    for i in range(n):
        already_sorted = True

        for j in range(n - i - 1):
            if array[j] > array[j + 1]:
                array[j], array[j + 1] = array[j + 1], array[j]

                already_sorted = False
        if already_sorted:
            break

    return array

def Selection(array):
    for i in range(len(array)):
        minimum = i

        for j in range(i + 1, len(array)):
            if array[j] < array[minimum]:
                minimum = j

        array[minimum], array[i] = array[i], array[minimum]

    return array
```

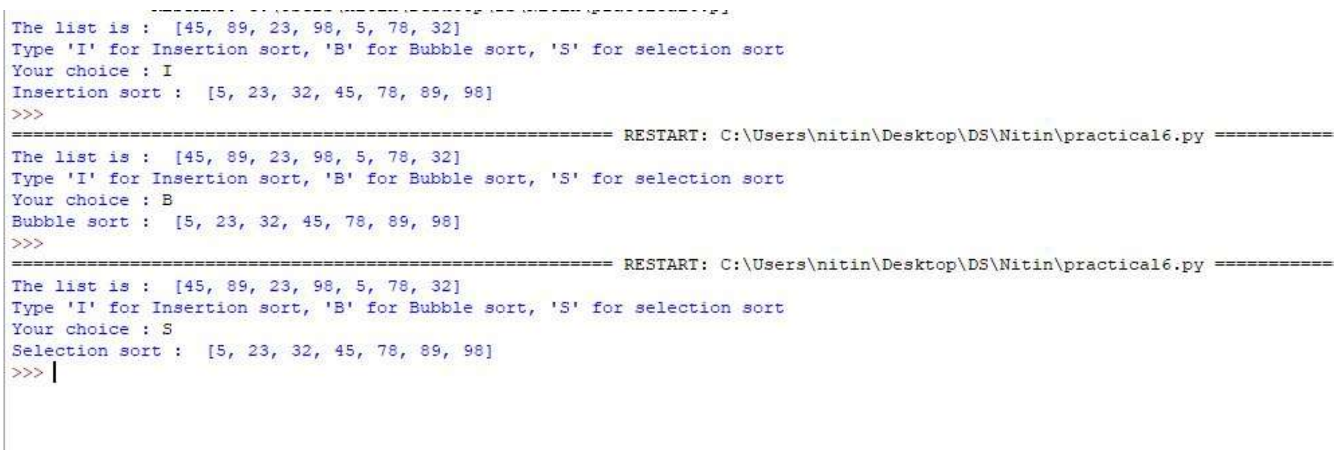


```
    return array

def Selection(array):
    for i in range(len(array)):
        minimum = i

        for j in range(i + 1, len(array)):
            if array[j] < array[minimum]:
                minimum = j

        array[minimum], array[i] = array[i], array[minimum]

    return array
```

```
array = [45,89,23,98,5,78,32]
print("The list is : ",array)
print("Type 'I' for Insertion sort, 'B' for Bubble sort, 'S' for selection sort")
yourChoice = str(input("Your choice : "))

if yourChoice == "I":
    print("Insertion sort : ",Insertion(array))
elif yourChoice == "B":
    print("Bubble sort : ",Bubblesort(array))
elif yourChoice == "S":
    print("Selection sort : ",Selection(array))
else :
    print("Invalid Input")
```

Nitin Singh SYCS 4059

Output :



```
The list is :  [45, 89, 23, 98, 5, 78, 32]
Type 'I' for Insertion sort, 'B' for Bubble sort, 'S' for selection sort
Your choice : I
Insertion sort :  [5, 23, 32, 45, 78, 89, 98]
>>>
========================================================= RESTART: C:\Users\nitin\Desktop\DS\Nitin\practical6.py ===========
The list is :  [45, 89, 23, 98, 5, 78, 32]
Type 'I' for Insertion sort, 'B' for Bubble sort, 'S' for selection sort
Your choice : B
Bubble sort :  [5, 23, 32, 45, 78, 89, 98]
>>>
========================================================= RESTART: C:\Users\nitin\Desktop\DS\Nitin\practical6.py ===========
The list is :  [45, 89, 23, 98, 5, 78, 32]
Type 'I' for Insertion sort, 'B' for Bubble sort, 'S' for selection sort
Your choice : S
Selection sort :  [5, 23, 32, 45, 78, 89, 98]
>>>
```

Practical No. 7.A

Aim : Write a program to implement the collision technique.

Github Repository Link :
https://github.com/NitinSingh1071/DS/blob/main/Practical7a.py

Theory :

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

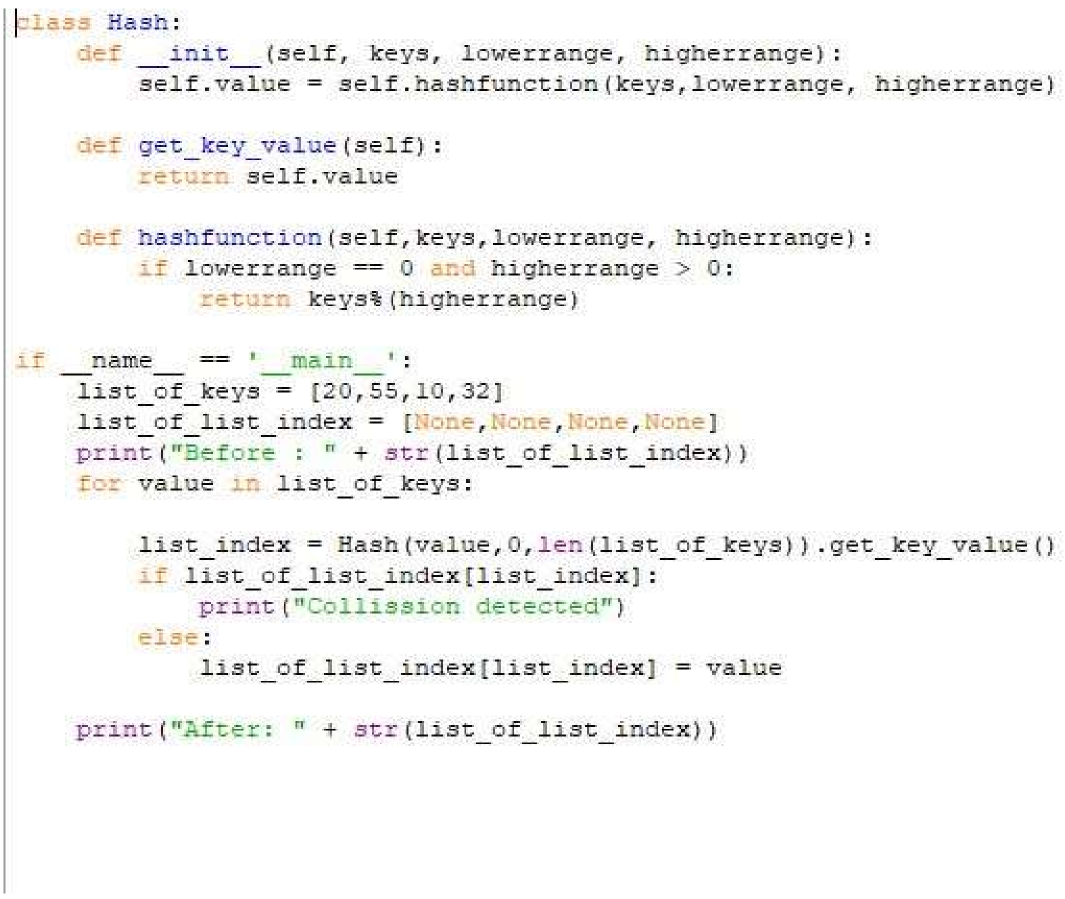
The most popular resolution techniques are,

Separate chaining

Open addressing

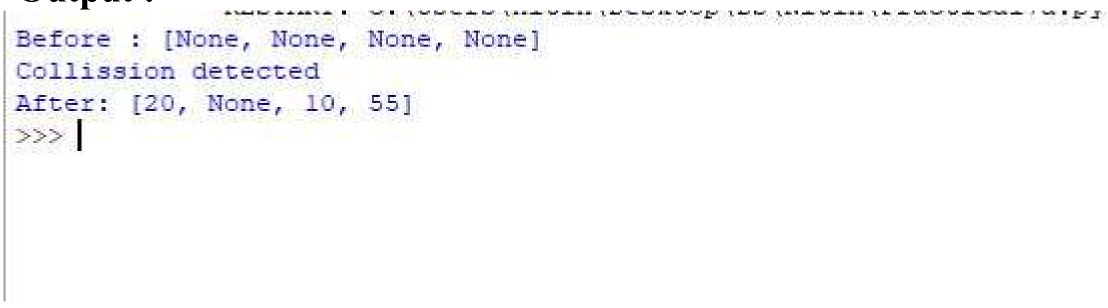
Code :

Nitin Singh SYCS 4059



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

    def get_key_value(self):
        return self.value

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

if __name__ == '__main__':
    list_of_keys = [20,55,10,32]
    list_of_list_index = [None,None,None,None]
    print("Before : " + str(list_of_list_index))
    for value in list_of_keys:

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



```
Before : [None, None, None, None]
Collission detected
After: [20, None, 10, 55]
>>> 
```

Nitin Singh SYCS 4059

Practical No. 7.B

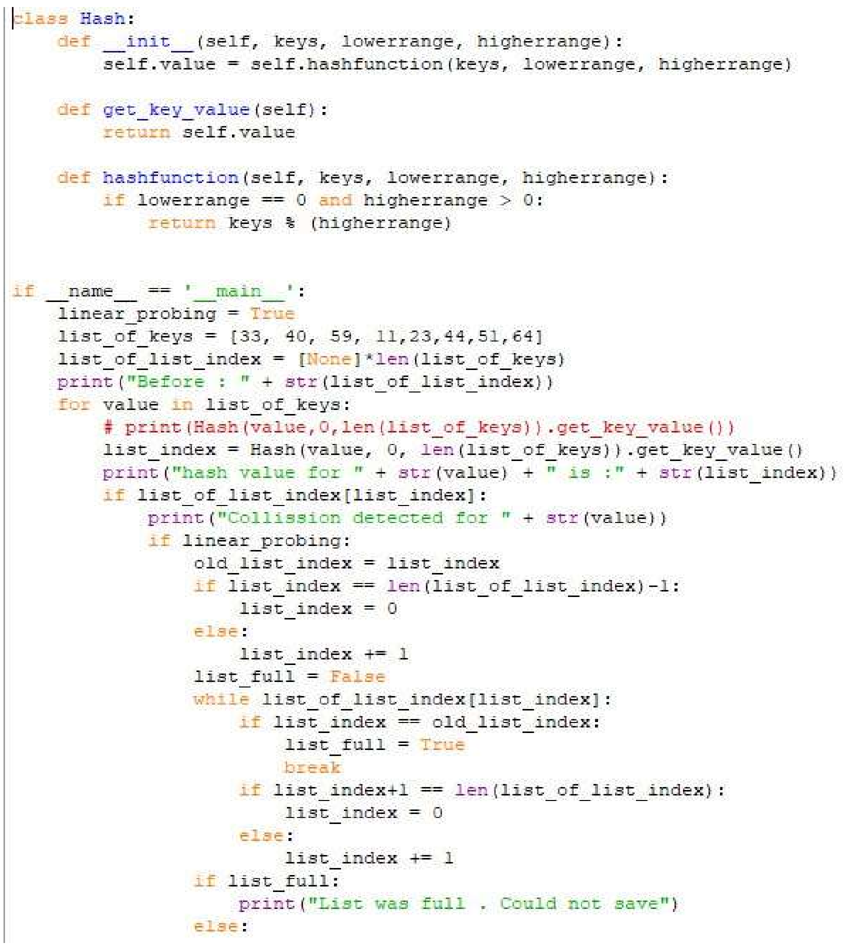
Aim : Write a program to implement the concept of linear probing.

Github Repository Link :
https://github.com/NitinSingh1071/DS/blob/main/Practical7b.py

Theory :

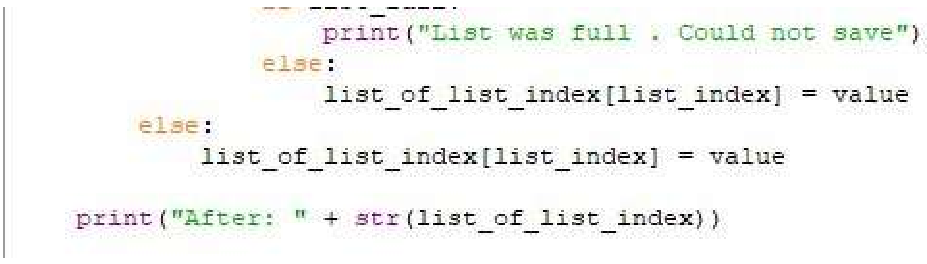
Linear probing is a component of open addressing schemes for using a hash table to solve the dictionary problem. In the dictionary problem, a data structure should maintain a collection of key–value pairs subject to operations that insert or delete pairs from the collection or that search for the value associated with a given key. In open addressing solutions to this problem, the data structure is an array *T* (the hash table) whose cells *T*. (when nonempty) each store a single key–value pair. A hash function is used to map each key into the cell of *T* where that key should be stored, typically scrambling the keys so that keys with similar values are not placed near each other in the table. A hash collision occurs when the hash function maps a key into a cell that is already occupied by a different key. Linear probing is a strategy for resolving collisions, by placing the new key into the closest following empty cell.

Code :



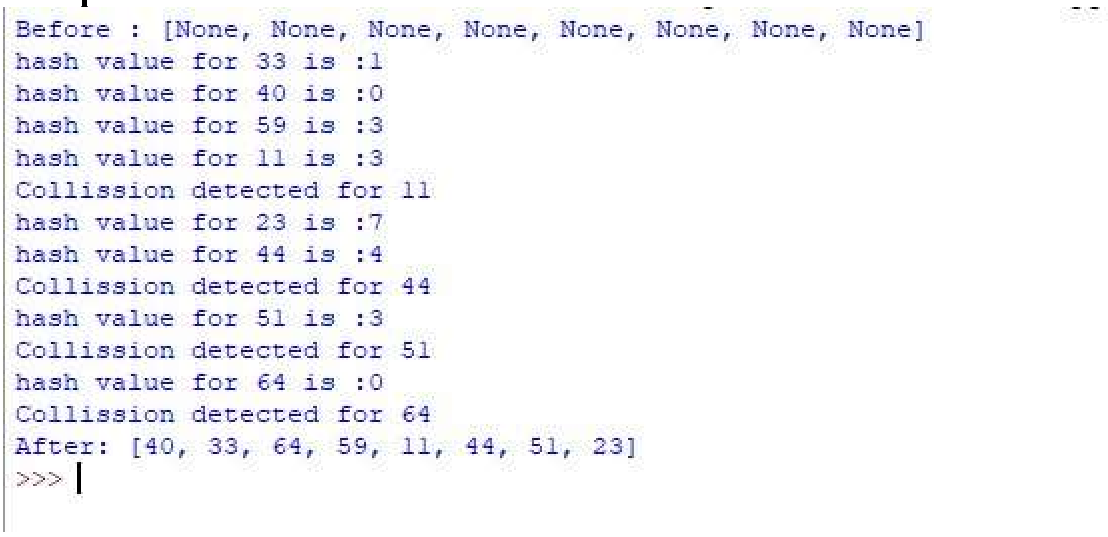
```
class Hash:
    def __init__(self, keys, lowerrange, higherrange):
        self.value = self.hashfunction(keys, lowerrange, higherrange)

    def get_key_value(self):
        return self.value

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


if __name__ == '__main__':
    linear_probing = True
    list_of_keys = [33, 40, 59, 11,23,44,51,64]
    list_of_list_index = [None]*len(list_of_keys)
    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:
```

Nitin Singh SYCS 4059



```
                    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 :



```
Before : [None, None, None, None, None, None, None, None]
hash value for 33 is :1
hash value for 40 is :0
hash value for 59 is :3
hash value for 11 is :3
Collission detected for 11
hash value for 23 is :7
hash value for 44 is :4
Collission detected for 44
hash value for 51 is :3
Collission detected for 51
hash value for 64 is :0
Collission detected for 64
After: [40, 33, 64, 59, 11, 44, 51, 23]
>>>
```

Practical No. 8

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

Github Repository Link :
https://github.com/NitinSingh1071/DS/blob/main/practical8.py

Theory :

A tree whose elements have at most 2 children is called a binary tree. Since each element in a binary tree can have only 2 children, we typically name them the left and right child. The topmost node is called root of the tree. The elements that are directly under an element are called its children. The element directly above something is called its parent. Finally, elements with no children are called leaves.

Unlike linear data structures (Array, Linked List, Queues, Stacks, etc) which have only one logical way to traverse them, trees can be traversed in different ways. Following are the generally used ways for traversing trees.

Uses of Inorder

In case of binary search trees (BST), Inorder traversal gives nodes in non-decreasing order. To get nodes of BST in non-increasing order, a variation of Inorder traversal where Inorder traversal s reversed can be used.

Uses of Preorder

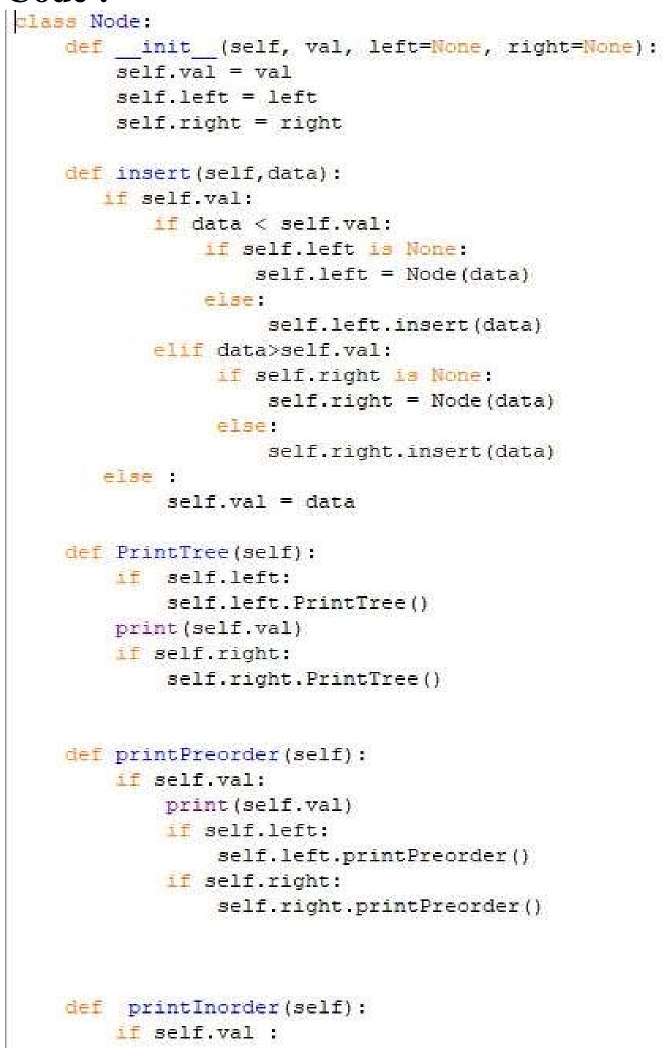
Preorder traversal is used to create a copy of the tree. Preorder traversal is also used to get prefix expression on of an expression tree.

Uses of Postorder

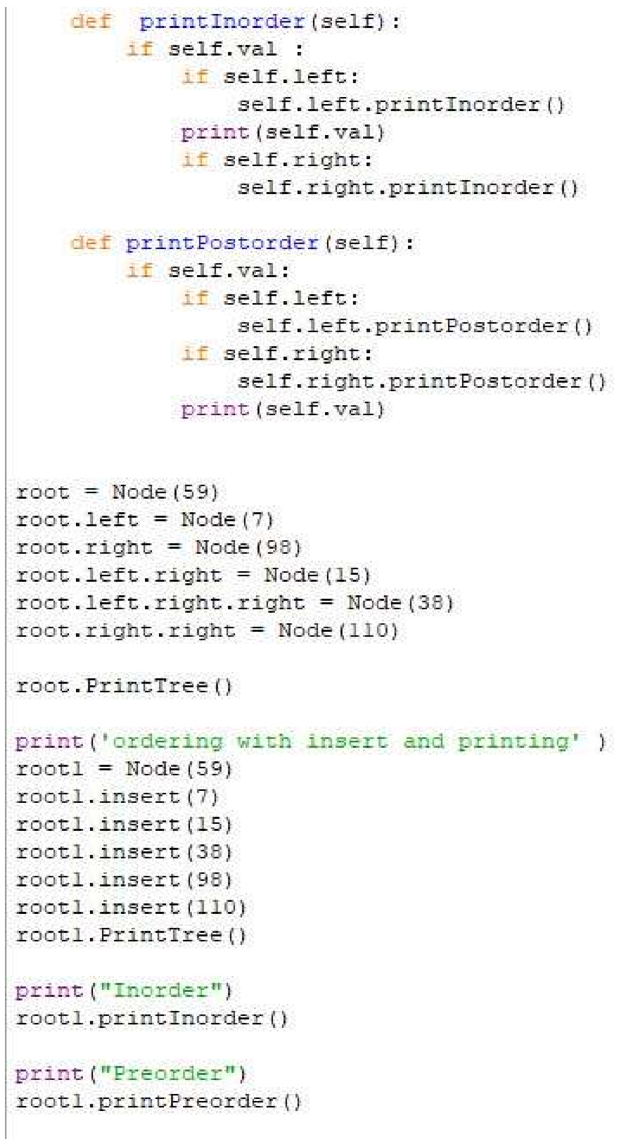
Postorder traversal is used to delete the tree. Please see the question for deletion of tree for details. Postorder traversal is also useful to get the postfix expression of an expression tree.

Nitin Singh SYCS 4059

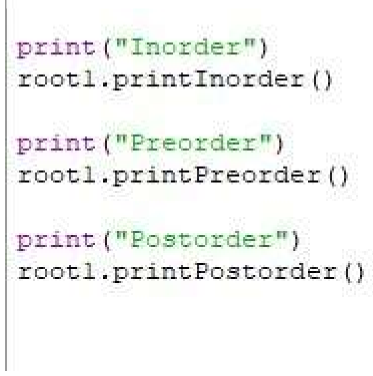
Code :



```
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 :
```



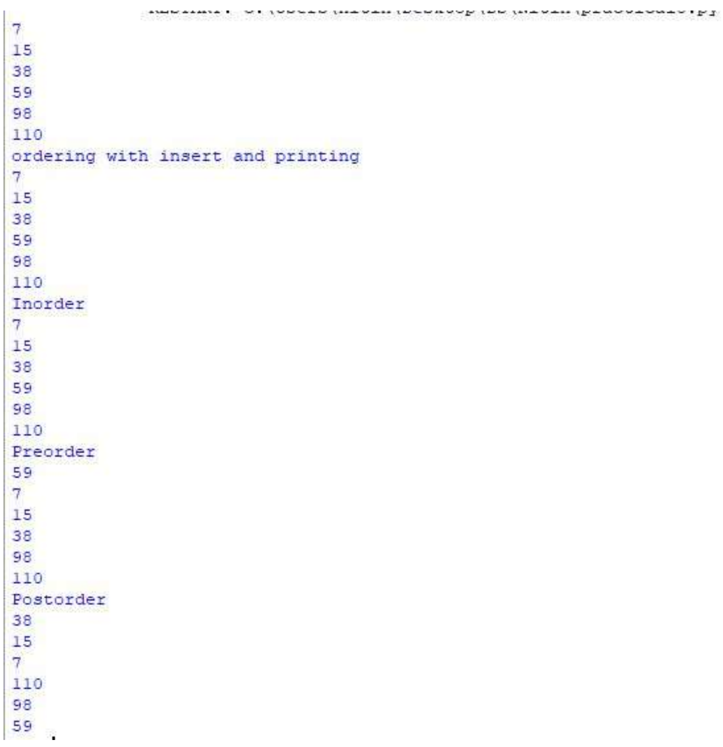
```
    def  printInorder(self):
        if self.val :
            if self.left:
                self.left.printInorder()
            print(self.val)
            if self.right:
                self.right.printInorder()

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


root = Node(59)
root.left = Node(7)
root.right = Node(98)
root.left.right = Node(15)
root.left.right.right = Node(38)
root.right.right = Node(110)

root.PrintTree()

print('ordering with insert and printing' )
root1 = Node(59)
root1.insert(7)
root1.insert(15)
root1.insert(38)
root1.insert(98)
root1.insert(110)
root1.PrintTree()

print("Inorder")
root1.printInorder()

print("Preorder")
root1.printPreorder()
```



```
print("Inorder")
root1.printInorder()

print("Preorder")
root1.printPreorder()

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

Nitin Singh SYCS 4059

Output :



```
7
15
38
59
98
110
ordering with insert and printing
7
15
38
59
98
110
Inorder
7
15
38
59
98
110
Preorder
59
7
15
38
98
110
Postorder
38
15
7
110
98
59
```