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: <u>Mr Manish Chaudhary</u>

Roll No:<u>313</u>   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 CS during the academic year 2020-21.

The journal work is the original study work that has been duly approved in the year 2020-21 by the undersigned.

External Examiner   Mr. Gangashankar Singh
                                                          (Subject-In-Charge)

Date of Examination: (College Stamp)

Class: S.Y. B.Sc. CS Sem- III   **Roll No: <u>313</u>**

<u>Subject: Data Structures</u>

INDEX

Sr No	Date	Topic	Sign
1	04/09/2020	Implement the following for Array: • Write a program to store the elements in 1-D array and provide an option to perform the operations like searching, sorting, merging, reversing the elements. • 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: • Perform Stack operations using Array implementation. b. • Implement Tower of Hanoi. • WAP to scan a polynomial using linked list and add two polynomials. • 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: • Write a program to implement the collision technique.	

		• 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.

Theory:

Storing Data in Arrays. Assigning values to an elementing an array is similar to assigning values to scalar variables. Simply reference an individual element of an array using the array name and the index inside parentheses, then use the assignment operator (=) followed by a value.

Following are the basic operations supported by an array.

- Traverse – print all the array elements one by one.
- Insertion – Adds an element at the given index.
- Deletion – Deletes an element at the given index.
- Search – Searches an element using the given index or by the value

Code:

File   Edit   Format   Run   Options   Window   Help

```python
class ArrayModification:

    def linear_search(self,lst,n):
        for i in range(len(lst)):
            if lst[i] == n:
                return f'Position :{i}'
        return -1

    def insertion_sort(self,lst):
        for i in range(len(lst)):

            index = lst[i]

            k = i - 1

            while k >= 0 and lst[k] > index:
                lst[k + 1] = lst[k]
                k -= 1

            lst[k+1]  = index

        return lst

    def merge(self,lst,lst2):
        lst.extend(lst2)
        print(lst)

    def reverse(self,lst):
        return lst[::-1]

lst = [2,9,1,7,3,5,2]
lst2 = [4,6,8,9,4,5]
Arrmod = ArrayModification()
print(Arrmod.linear_search(lst,3))
print(Arrmod.merge(lst,lst2))
print(Arrmod.insertion_sort(lst))
print(Arrmod.reverse(lst))
```
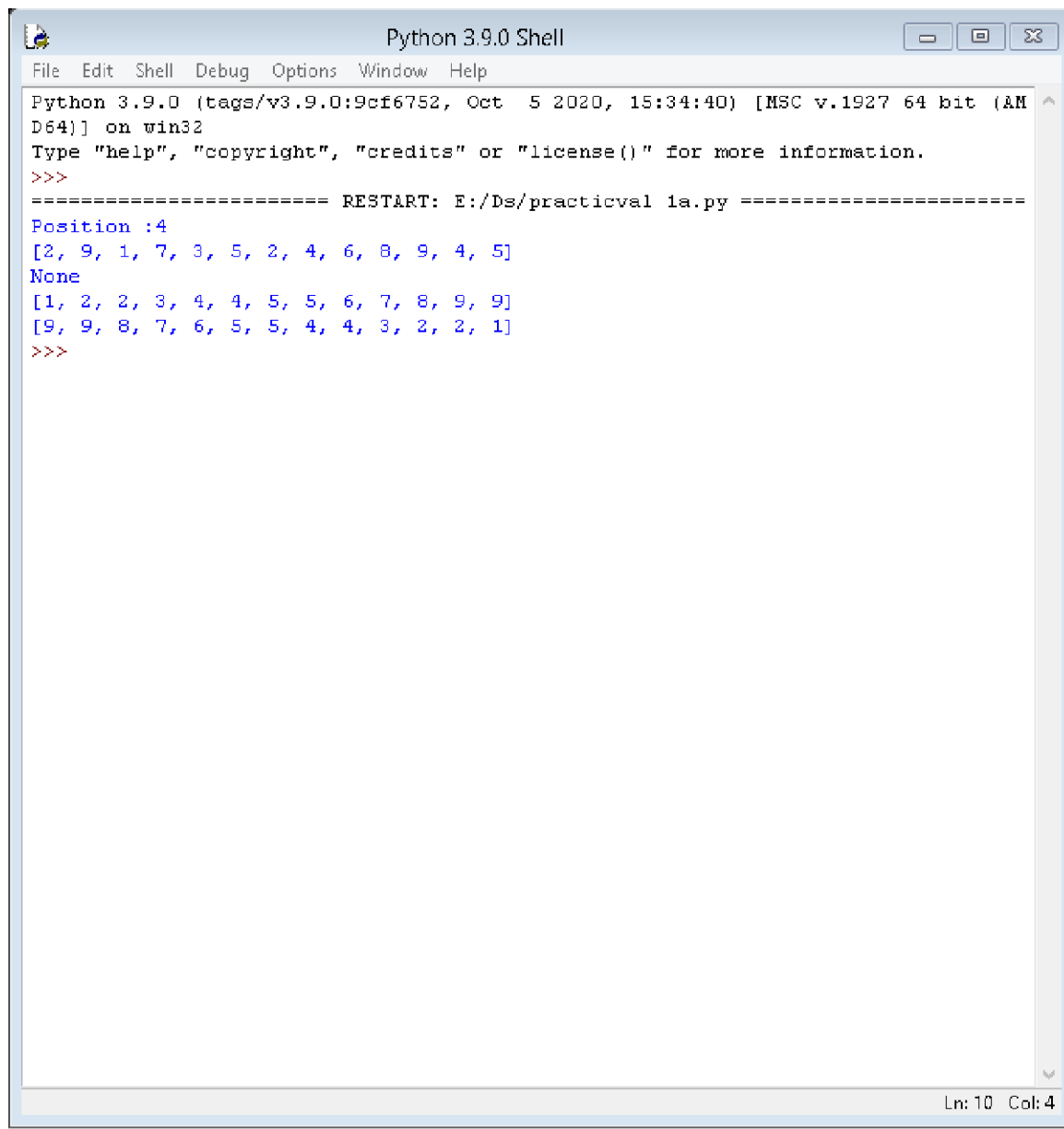
Output:

```
Python 3.9.0 Shell
File  Edit  Shell  Debug  Options  Window  Help
Python 3.9.0 (tags/v3.9.0:9cf6752, Oct  5 2020, 15:34:40) [MSC v.1927 64 bit (AM
D64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
====================== RESTART: E:/Ds/practicval 1a.py ======================
Position :4
[2, 9, 1, 7, 3, 5, 2, 4, 6, 8, 9, 4, 5]
None
[1, 2, 2, 3, 4, 4, 5, 5, 6, 7, 8, 9, 9]
[9, 9, 8, 7, 6, 5, 5, 4, 4, 3, 2, 2, 1]
>>>
```

Practical 1a Github Link:

https://github.com/manishmo143/ds-practical/blob/master/Prac1A.py

Practical No 1(B)

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

Theory:

Algorithm to perform matrix addition, matrix multiplication

Matrix addition:

- Input the order of the matrix.
- Input the matrix 1 elements.
- Input the matrix 2 elements.
- Repeat from i = 0 to m
- Repeat from j = 0 to n

6. mat3[i][j] = mat1[i][j] + mat2[i][j]

7. Print mat3.
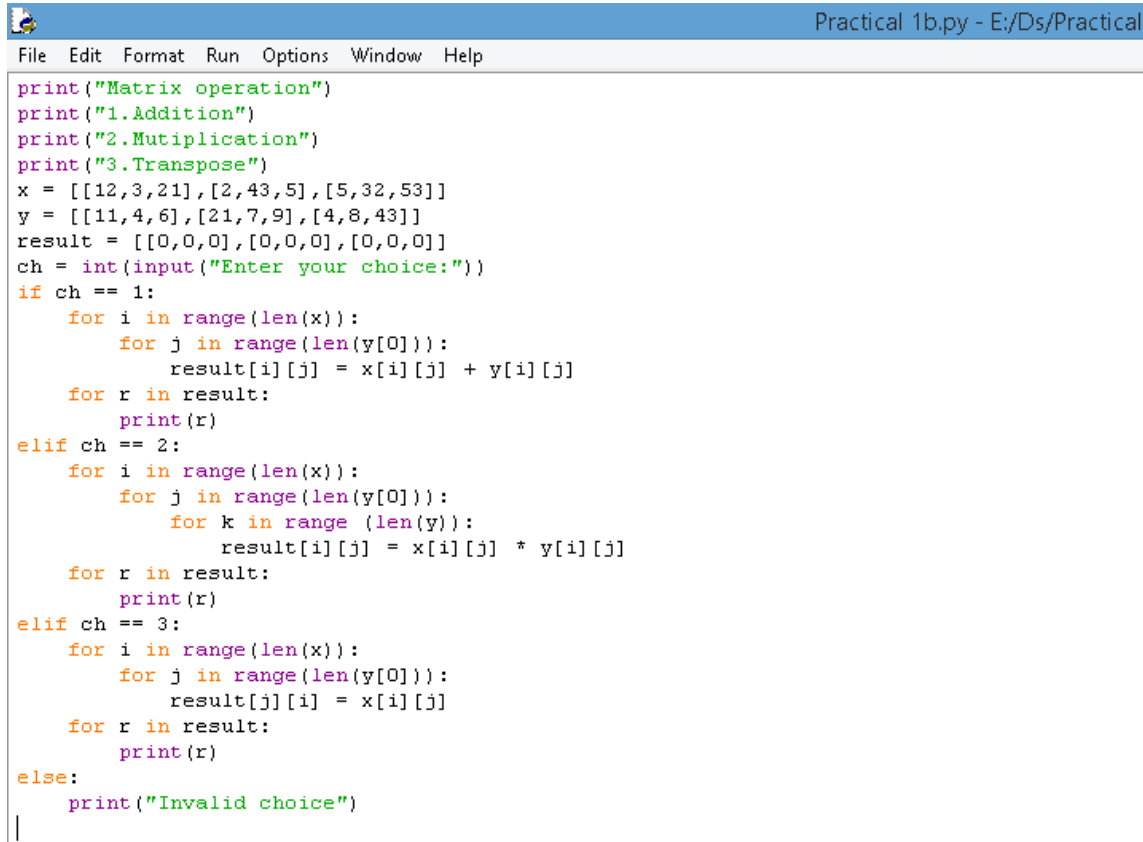
Matrix multiplication:

- Input the order of the matrix1 (m * n).
- Input the order of matrix2 (p * q).
- Input the matrix 1 elements.
- Input the matrix 2 elements.
- Repeat from i = 0 to m
- Repeat from j = 0 to q
- repeat from k = 0 to p
- sum=sum+ mat1[c][k] * mat2[k][d];
- mat3[c][d]=sum
- Print mat3.

Matrix Transpose

The transpose of a [matrix](#) is simply a flipped version of the original matrix. We can transpose a matrix by switching its rows with its columns. We denote the transpose of matrix AA by ATAT.

Code:

File Edit Format Run Options Window Help

```python
print("Matrix operation")
print("1.Addition")
print("2.Mutiplication")
print("3.Transpose")
x = [[12,3,21],[2,43,5],[5,32,53]]
y = [[11,4,6],[21,7,9],[4,8,43]]
result = [[0,0,0],[0,0,0],[0,0,0]]
ch = int(input("Enter your choice:"))
if ch == 1:
    for i in range(len(x)):
        for j in range(len(y[0])):
            result[i][j] = x[i][j] + y[i][j]
    for r in result:
        print(r)
elif ch == 2:
    for i in range(len(x)):
        for j in range(len(y[0])):
            for k in range (len(y)):
                result[i][j] = x[i][j] * y[i][j]
    for r in result:
        print(r)
elif ch == 3:
    for i in range(len(x)):
        for j in range(len(y[0])):
            result[j][i] = x[i][j]
    for r in result:
        print(r)
else:
    print("Invalid choice")
```

Output:

```
Python 3.9.0 Shell                                              –  □  ×
File  Edit  Shell  Debug  Options  Window  Help
Python 3.9.0 (tags/v3.9.0:9cf6752, Oct  5 2020, 15:34:40) [MSC v.1927 64 bit (AM
D64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
======================= RESTART: E:/Ds/Practical 1b.py =======================
Matrix operation
1.Addition
2.Mutiplication
3.Transpose
Enter your choice:1
[23, 7, 27]
[23, 50, 14]
[9, 40, 96]
>>>
======================= RESTART: E:/Ds/Practical 1b.py =======================
Matrix operation
1.Addition
2.Mutiplication
3.Transpose
Enter your choice:2
[132, 12, 126]
[42, 301, 45]
[20, 256, 2279]
>>>
======================= RESTART: E:/Ds/Practical 1b.py =======================
Matrix operation
1.Addition
2.Mutiplication
3.Transpose
Enter your choice:3
[12, 2, 5]
[3, 43, 32]
[21, 5, 53]
>>> |
```

Practical 1b Github Link:

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.

Theory:

Linked List

A linked list is a linear data structure, in which the elements are not stored at contiguous memory locations. The elements in a linked list are linked using pointers A linked list is a sequence of data structures, which are connected together via links.Linked List is a sequence of links which contains items. Each link contains a connection to another link. Linked list is the second most-used data structure after array

Linked List Representation

Linked list can be visualized as a chain of nodes, where every node points to the next node.

- Linked List contains a link element called first.
- Each link carries a data field(s) and a link field called next.
- Each link is linked with its next link using its next link.
- Last link carries a link as null to mark the end of the list

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. 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.

Insertion Operation

Adding a new node in linked list is a more than one step activity. We shall learn this with diagrams here. First, create a node using the same structure and find the location where it has to be inserted.

Deletion Operation

Deletion is also a more than one step process. We shall learn with pictorial representation. First, locate the target node to be removed, by using searching algorithms.

Reverse Operation

This operation is a thorough one. We need to make the last node to be pointed by the head node and reverse the whole linked list.

Code:

File Edit Format Run Options Window Help

```python
class Node:
    def __init__(self, element, next=None, prev=None):
        self.element = element
        self.next = next
        self.prev = prev

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

class LinkedList:
    def __init__(self):
        self.start = None
        self.end = None
        self.length = 0

    def is_empty(self) -> bool:
        return self.length == 0

    def get_size(self) -> int:
        return self.length

    def display(self):
        if self.is_empty():
            print("Doubly Linked List is empty")
            return
        first = self.start
        print("The List: ", end='')
        print("[" + first.element, end='')
        first = first.next
        while first:
            print(", " + first.element, end='')
            first = first.next
        print("]")

    def add_head(self, e):
        old_head = self.start
        self.start = Node(e)
        self.start.next = old_head
        if old_head is not None:
            old_head.prev = self.start
        if self.end is None:
            self.end = self.start
        self.length += 1
```

Ln: 255 Col: 0

Output:

```
                                                              Python 3.9.0 Shell
File  Edit  Shell  Debug  Options  Window  Help
Python 3.9.0 (tags/v3.9.0:9cf6752, Oct  5 2020, 15:34:40) [MSC v.1927 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
========================= RESTART: E:/Ds/Practical 2.py =========================
True
The List: [zero, one, two]
False
The List: [zero, two]
The List: [zero, one, two]
Head: first_name
Tail: last_name
first_name
'NoneType' object has no attribute 'display'
No head found
Doubly Linked List is empty
The List: [zero, one, two, three]
three
three
three
The List: [zero, one, two, three]
Head: zero
Tail: three
The List: [0, 1, 2, 4]
Head: 0
Tail: 4
The List: [zero, one, two, three, 0, 1, 2, 4]
Head: zero
Tail: 4
Size: 8
The List: [zero, one, two, three, 0, 1, 2, 4]
The List: [4, 2, 1, 0, three, two, one, zero]
>>> |
```

Practical 2 Github Link:

Practical No 3(A)

- **Aim:** Perform Stack operations using Array implementation. b.

Theory:

Array implementation of Stack

In array implementation, the stack is formed by using the array. All the operations regarding the stack are performed using arrays. Lets see how each operation can be implemented on the stack using array data structure.

Adding an element onto the stack (push operation)

Adding an element into the top of the stack is referred to as push operation. Push operation involves following two steps.

- Increment the variable Top so that it can now refere to the next memory location.
- Add element at the position of incremented top. This is referred to as adding new element at the top of the stack.

Stack is overflown when we try to insert an element into a completely filled stack therefore, our main function must always avoid stack overflow condition.

Deletion of an element from a stack (Pop operation)

Deletion of an element from the top of the stack is called pop operation. The value of the variable top will be incremented by 1 whenever an item is deleted from the stack. The top most element of the stack is stored in an another variable and then the top is decremented by 1. the operation returns the deleted value that was stored in another variable as the result.The underflow condition occurs when we try to delete an element from an already empty stack.

Code:

```python
class Stack:

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

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

    def pop(self):
        if len(self.stack_arr) == 0:
            print('Stack is empty!')
            return None
        else:
            self.stack_arr.pop()

    def get_head(self):
        if len(self.stack_arr) == 0:
            print('Stack is empty!')
            return None
        else:
            return self.stack_arr[-1]

    def display(self):
        if len(self.stack_arr) == 0:
            print('Stack is empty!')
            return None
        else:
            print(self.stack_arr)

stack = Stack()
stack.push(4)
stack.push(5)
stack.push(6)
stack.pop()
stack.display()
stack.get_head()
```
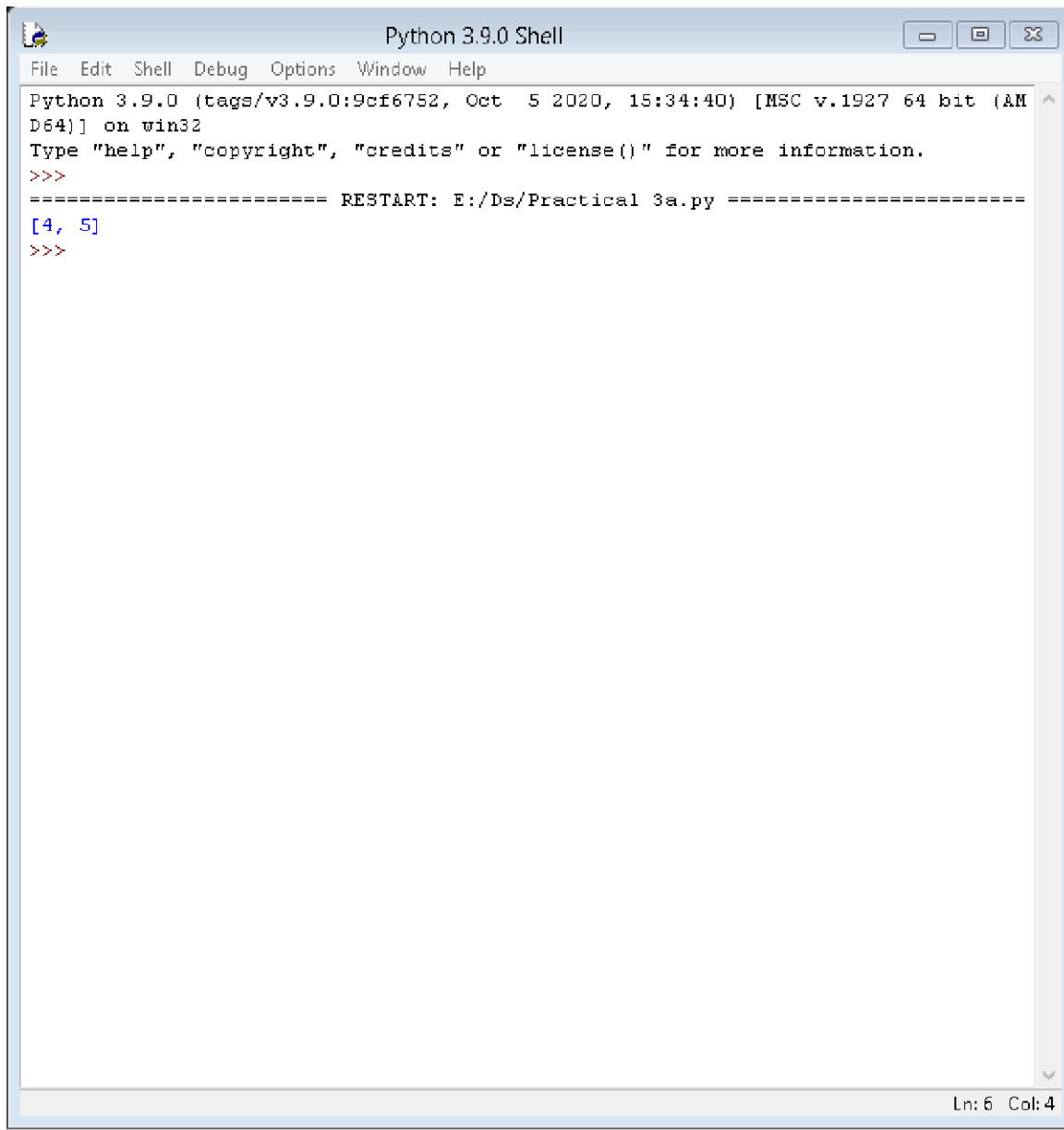
Output:

```
Python 3.9.0 Shell                                          ☐ ☐ ☒

File  Edit  Shell  Debug  Options  Window  Help
Python 3.9.0 (tags/v3.9.0:9cf6752, Oct  5 2020, 15:34:40) [MSC v.1927 64 bit (AM
D64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===================== RESTART: E:/Ds/Practical 3a.py ========================
[4, 5]
>>>
```

Practical 3a Github Link:

Practical No 3(B)

Aim: Implement Tower of Hanoi.

Theory:

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

• Only one disk can be moved at a time.

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

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

Algorithm

To write an algorithm for Tower of Hanoi, first we need to learn how to solve this problem with lesser amount of disks, say → 1 or 2. We mark three towers with name, source, destination and aux (only to help moving the disks). If we have only one disk, then it can easily be moved from source to destination peg.

If we have 2 disks −

- • First, we move the smaller (top) disk to aux peg.
- • Then, we move the larger (bottom) disk to destination peg.
- • And finally, we move the smaller disk from aux to destination peg.

So now, we are in a position to design an algorithm for Tower of Hanoi with more than two disks. We divide the stack of disks in two parts. The largest disk (n^{th} disk) is in one part and all other (n-1) disks are in the second part.

Our ultimate aim is to move disk n from source to destination and then put all other (n1) disks onto it. We can imagine to apply the same in a recursive way for all given set of disks.

Code:

Output:


```
Python 3.9.0 (tags/v3.9.0:9cf6752, Oct  5 2020, 15:34:40) [MSC v.1927 64
D64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
======================== RESTART: E:/Ds/Practical 3b.py =================
Enter the number of the disk in rod A : 3
['disk 1']
['disk 2']
['disk 2', 'disk 1']
['disk 3']
['disk 1']
['disk 3', 'disk 2']
['disk 3', 'disk 2', 'disk 1']
>>>
```

Practical 3b Github Link:

Practical No 3(c)

Aim: Write a Program to scan a polynomial using linked list and add two polynomials

Theory:

Adding two polynomials using Linked List

Given two polynomial numbers represented by a linked list. Write a function that add these lists means add the coefficients who have same variable powers.

Example:

Input:

1st number = $5x^2 + 4x^1 +$

$2x^0$ 2nd number = $5x^1 +$

$5x^0$

Output:

$5x^2 + 9x^1 +$

$7x^0$ Input:

1st number = $5x^3 + 4x^2 +$

$2x^0$ 2nd number = $5x^1 +$

$5x^0$

Output:

$5x^3 + 4x^2 + 5x^1 + 7x^0$

Code:

File   Edit   Format   Run   Options   Window   Help

```python
class Node:

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

class LinkedList:

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



    def _len_(self):
        return self.size

    def get_head(self):
        return self.head


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

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

    def reverse_display(self):
        if self.size == 0:
            print("No element")
```

```python
        last = last.previous




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

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



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

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

    def 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
```

Output:

```
*Python 3.9.0 Shell*
File  Edit  Shell  Debug  Options  Window  Help
Python 3.9.0 (tags/v3.9.0:9cf6752, Oct  5 2020, 15:34:40) [MSC
D64)] on win32
Type "help", "copyright", "credits" or "license()" for more in:
>>>
====================== RESTART: E:/Ds/Practical 3c.py ======:
Enter the order for polynomial : 1
Enter coefficient for power 1 : 2
Enter coefficient for power 0 : 3
Enter coefficient for power 1 : 4
Enter coefficient for power 0 : 5
6
8
>>>
```

Practical 3c Github Link:

https://github.com/manishmo143/ds-practical/blob/master/Prac3c.py

Practical No 3(D)

Aim: Write a Program to calculate factorial and to compute the factors of a given number a) using recursion , b)using iteration

Theory: Factorial of a non-negative integer, is multiplication of all integers smaller than or equal to n. For example factorial of 6 is 6*5*4*3*2*1 which is 720.Any recursive function can be written as an iterative function (and vise versa). Here is the math-like definition of recursion (again):

factorial(0) = 1

factorial(N) = N * factorial(N-1)

Code:

File Edit Format Run Options Window Help

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

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

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

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

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

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

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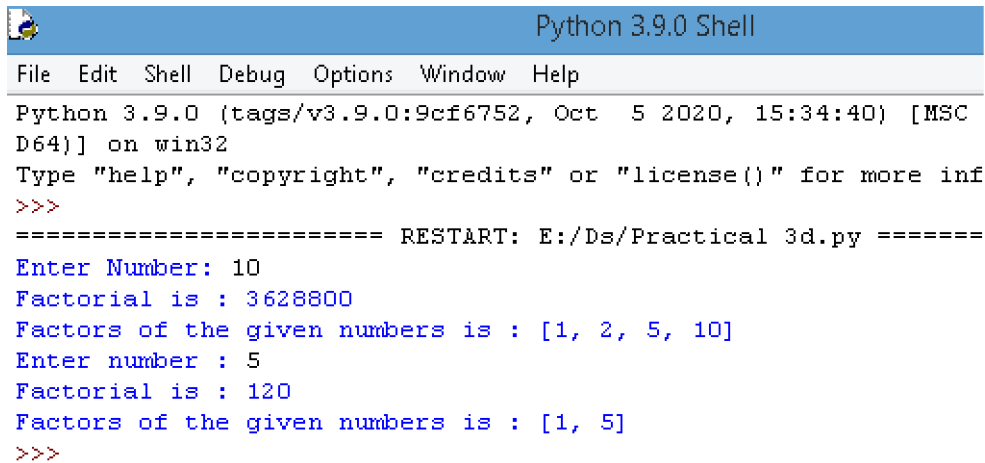
Output:

```
Python 3.9.0 Shell
File  Edit  Shell  Debug  Options  Window  Help
Python 3.9.0 (tags/v3.9.0:9cf6752, Oct  5 2020, 15:34:40) [MSC
D64)] on win32
Type "help", "copyright", "credits" or "license()" for more inf
>>>
====================== RESTART: E:/Ds/Practical 3d.py =======
Enter Number: 10
Factorial is : 3628800
Factors of the given numbers is : [1, 2, 5, 10]
Enter number : 5
Factorial is : 120
Factors of the given numbers is : [1, 5]
>>>
```

Practical 3d Github Link:

Practical No 4

Aim: Perform Queues operations using Circular Array implementation.

Theory:

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

• **Enqueue Operation**

check if the queue is full

for the first element, set value of FRONT to 0

circularly increase the REAR index by 1 (i.e. if the rear reaches the end, next it would be at the start of the queue)

add the new element in the position pointed to by REAR

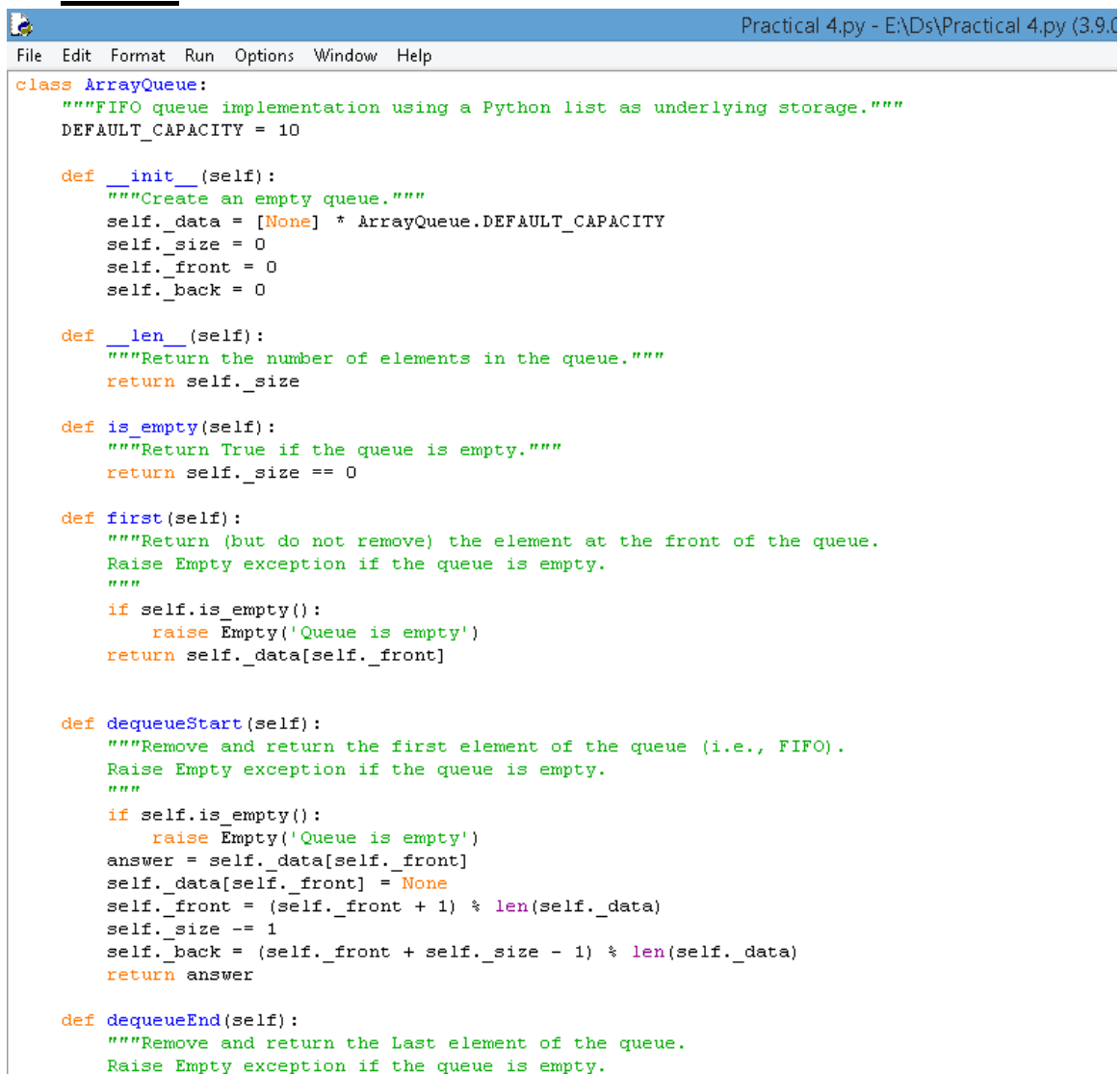## • <u>Dequeue Operation</u>

check if the queue is empty

return the value pointed by FRONT

circularly increase the FRONT index

by 1

for the last element, reset the values of FRONT and REAR to -1

<u>Code:</u>

```python
class ArrayQueue:
    """FIFO queue implementation using a Python list as underlying storage."""
    DEFAULT_CAPACITY = 10

    def __init__(self):
        """Create an empty queue."""
        self._data = [None] * ArrayQueue.DEFAULT_CAPACITY
        self._size = 0
        self._front = 0
        self._back = 0

    def __len__(self):
        """Return the number of elements in the queue."""
        return self._size

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

    def first(self):
        """Return (but do not remove) the element at the front of the queue.
        Raise Empty exception if the queue is empty.
        """
        if self.is_empty():
            raise Empty('Queue is empty')
        return self._data[self._front]

    def dequeueStart(self):
        """Remove and return the first element of the queue (i.e., FIFO).
        Raise Empty exception if the queue is empty.
        """
        if self.is_empty():
            raise Empty('Queue is empty')
        answer = self._data[self._front]
        self._data[self._front] = None
        self._front = (self._front + 1) % len(self._data)
        self._size -= 1
        self._back = (self._front + self._size - 1) % len(self._data)
        return answer

    def dequeueEnd(self):
        """Remove and return the Last element of the queue.
        Raise Empty exception if the queue is empty.
```

File   Edit   Format   Run   Options   Window   Help

```python
    def enqueueStart(self, e):
        """Add an element to the start of queue."""
        if self._size == len(self._data):
            self._resize(2 * len(self._data))
        self._front = (self._front - 1) % len(self._data)
        avail = (self._front + self._size) % len(self._data)
        self._data[self._front] = e
        self._size += 1
        self._back = (self._front + self._size - 1) % len(self._data)

    def _resize(self, cap):
        """Resize to a new list of capacity >= len(self)."""
        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
        self._back = (self._front + self._size - 1) % len(self._data)

queue = ArrayQueue()
queue.enqueueEnd(1)
print(f"First Element: {queue._data[queue._front]}, Last Element: {queue._data[queue._back]}")
queue._data
queue.enqueueEnd(2)
print(f"First Element: {queue._data[queue._front]}, Last Element: {queue._data[queue._back]}")
queue._data
queue.dequeueStart()
print(f"First Element: {queue._data[queue._front]}, Last Element: {queue._data[queue._back]}")
queue.enqueueEnd(3)
print(f"First Element: {queue._data[queue._front]}, Last Element: {queue._data[queue._back]}")
queue.enqueueEnd(4)
print(f"First Element: {queue._data[queue._front]}, Last Element: {queue._data[queue._back]}")
queue.dequeueStart()
print(f"First Element: {queue._data[queue._front]}, Last Element: {queue._data[queue._back]}")
queue.enqueueStart(5)
print(f"First Element: {queue._data[queue._front]}, Last Element: {queue._data[queue._back]}")
queue.dequeueEnd()
print(f"First Element: {queue._data[queue._front]}, Last Element: {queue._data[queue._back]}")
queue.enqueueEnd(6)
print(f"First Element: {queue._data[queue._front]}, Last Element: {queue._data[queue._back]}")
```

Output:

Python 3.9.0 Shell

File   Edit   Shell   Debug   Options   Window   Help

```
Python 3.9.0 (tags/v3.9.0:9cf6752, Oct  5 2020, 15::
D64)] on win32
Type "help", "copyright", "credits" or "license()" :
>>>
======================== RESTART: E:\Ds\Practical ·
First Element: 1, Last Element: 1
First Element: 1, Last Element: 2
First Element: 2, Last Element: 2
First Element: 2, Last Element: 3
First Element: 2, Last Element: 4
First Element: 3, Last Element: 4
First Element: 5, Last Element: 4
First Element: 5, Last Element: 3
First Element: 5, Last Element: 6
>>>
```

Practical 4 Github Link:

https://github.com/manishmo143/ds-practical/blob/master/Prac4.py

Practical No 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:

Linear search is the simplest searching algorithm that searches for an element in a list in sequential order. We start at one end and check every element until the desired element is not found.

Linear Search

Algorithm

LinearSearch(array,

key) for each item in

the array

   if item ==

     value

     return its

     index

Binary Search:

Binary Search is a searching algorithm for finding an element's position in a sorted array.In this approach, the element is always searched in the middle of a portion of an array.Binary search can be implemented only on a sorted list of items. If the elements are not sorted already, we need to sort them first.

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:

File Edit Format Run Options Window Help

```python
class Array:

    def __init__(self,array,number):
        self.lst = sorted(array)
        self.number = number
    def binary_search(self,lst,n,start,end):

        if start <= end:
            mid = (end + start) // 2
            if lst[mid] == n:

                return f'position: {mid}'
            elif lst[mid] > n:
                return binary_search(lst,n,start,mid-1)
            else:
                return binary_search(lst,n,mid + 1,end)
        else:
            return -1


    def linear_search(self,lst,n):
        for i in range(len(lst)):
            if lst[i] == n:
                return f'Position :{i}'
        return -1

    def run_search(self):
        while True:
            print('Select the searching algorithm:')
            print('1. Linear Search.')
            print('2. Binary Search.')
            print('3. quit.')
            opt = int(input('Option: '))
            if opt == 2:
                print(search.binary_search(self.lst,self.number,0,len(lst)-1))
            elif opt == 1:
                print(search.linear_search(self.lst,self.number))
            else:
                break
lst = [1,2,3,4,5,6,7,8]
number = 6
search = Array(lst,number)
search.run_search()
```

Output:

Python 3.9.0 Shell

File Edit Shell Debug Options Window Help

```
Python 3.9.0 (tags/v3.9.0:9cf6752, Oct  5 2020, 15:34:4(
D64)] on win32
Type "help", "copyright", "credits" or "license()" for i
>>>
======================= RESTART: E:\Ds\Practical 5.py
Select the searching algorithm:
1. Linear Search.
2. Binary Search.
3. quit.
Option: 1
Position :3
Select the searching algorithm:
1. Linear Search.
2. Binary Search.
3. quit.
Option: 2
position: 3
Select the searching algorithm:
1. Linear Search.
2. Binary Search.
3. quit.
Option: 3
>>>
```

Practical 5 Github Link:

https://github.com/manishmo143/ds-practical/blob/master/Prac5.py

Practical No 6

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

Theory:

Insertion Sort

Insertion sort is a simple sorting algorithm that works similar to the way you sort playing cards in your hands. The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part.

Algorithm

To sort an array of size n in ascending order:
1: Iterate from arr[1] to arr[n] over the array.

2: Compare the current element (key) to its predecessor.

3: If the key element is smaller than its predecessor, compare it to the elements before. Move the greater elements one position up to make space for the swapped element.

Bubble Sort

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

Selection sort

Selection sort is a simple sorting algorithm. This sorting algorithm is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is the entire list.

The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array. This process continues moving unsorted array boundary by one element to the right.

This algorithm is not suitable for large data sets as its average and worst case complexities are of $O(n^2)$, where **n** is the number of items.

Code:

```python
class Sorting:

    def __init__(self,lst):
        self.lst = lst

    def bubble_sort(self,lst):
        for i in range(len(lst)):
            for j in range(len(lst)):
                if lst[i] < lst[j]:
                    lst[i],lst[j] = lst[j],lst[i]
                else:
                    pass
        return lst

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

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

def run_sort(self):
    while True:
        print('Select the sorting algorithm:')
        print('1. Bubble Sort.')
        print('2. Selection Sort.')
        print('3. Insertion Sort.')
        print('4. Quit')
        opt = int(input('Option: '))
        if opt == 1:
            print(sort.bubble_sort(self.lst))
        elif opt == 2:
```

```python
                lst[i],lst[j] = lst[j],lst[i]
                else:
                    pass
        return lst

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

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

    def run_sort(self):
        while True:
            print('Select the sorting algorithm:')
            print('1. Bubble Sort.')
            print('2. Selection Sort.')
            print('3. Insertion Sort.')
            print('4. Quit')
            opt = int(input('Option: '))
            if opt == 1:
                print(sort.bubble_sort(self.lst))
            elif opt == 2:
                print(sort.selection_sort(self.lst))
            elif opt == 3:
                print(sort.insertion_sort(self.lst))
            else:
                break
lst = [4,2,3,11,9,12,1,45,]
sort = Sorting(lst)
sort.run_sort()
```
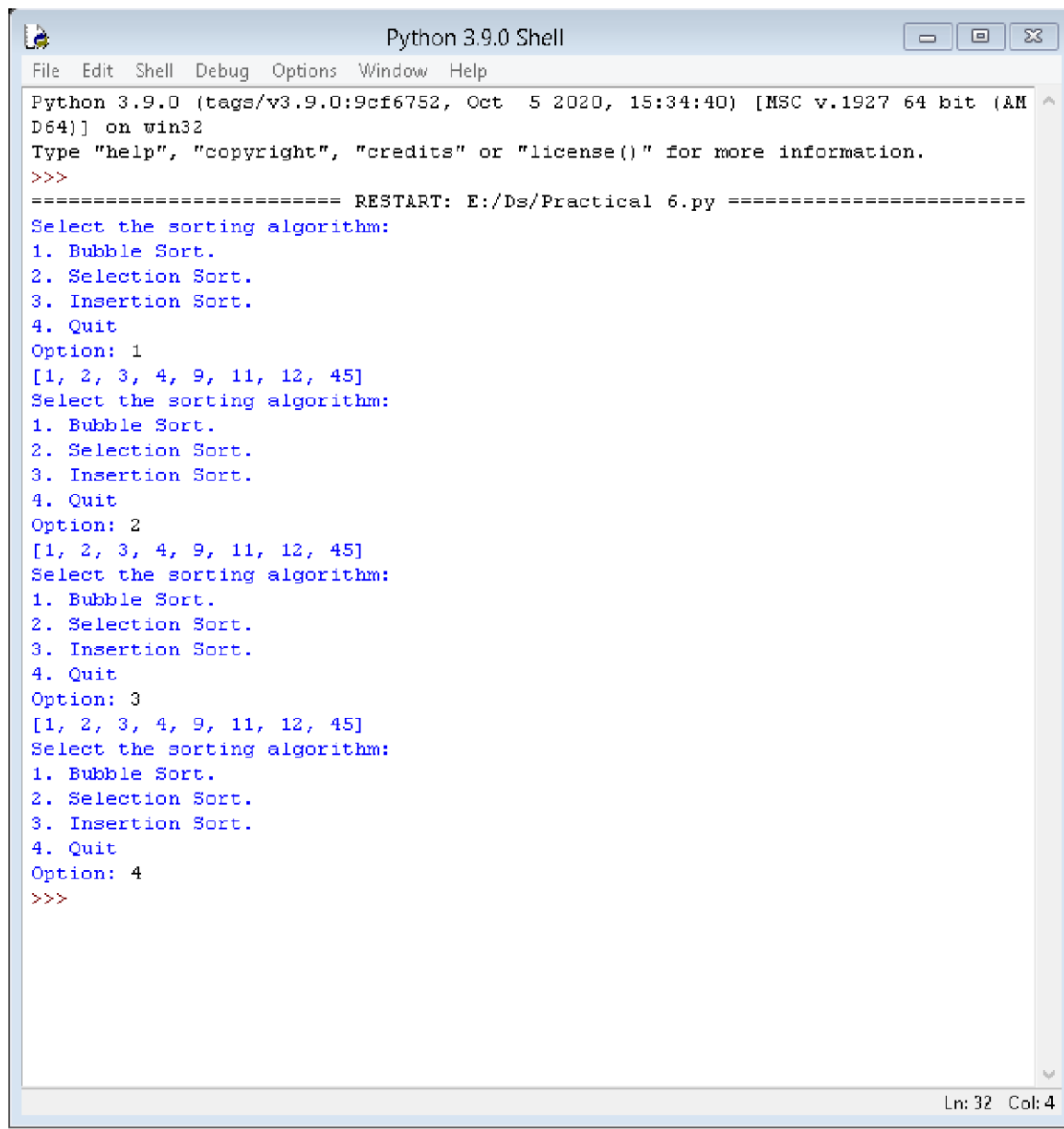
Output:

Practical 6 Github Link:

https://github.com/manishmo143/ds-practical/blob/master/Prac6.py

Practical No 7(A)

Aim: Write a program to implement the collision technique

Theory:

Hashing is a data structure that is used to store a large amount of data, which can be accessed in O(1) time by operations such as search, insert and delete

Hashing is an important Data Structure which is designed to use a special function called the Hash function which is used to map a given value with a particular key for faster access of elements. The efficiency of mapping depends of the efficiency of the hash function used.

- It should always map large keys to small keys.
- It should always generate values between 0 to m-1 where m is the size of the hash table.
- It should uniformly distribute large keys into hash table slots.

Collision Handling

If we know the keys beforehand, then we have can have perfect hashing. In perfect hashing, we do not have any collisions. However, If we do not know the keys, then we can use the following methods to avoid collisions:

- Chaining
- Open Addressing (Linear Probing, Quadratic Probing, Double Hashing)

Chaining

While hashing, the hashing function may lead to a collision that is two or more keys are mapped to the same value. Chain hashing avoids collision. The idea is to make each cell of hash table point to a linked list of records that have same hash function value.

Code:

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

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

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

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

```
Python 3.9.0 Shell                                          [ - ][ □ ][ ✕ ]

File  Edit  Shell  Debug  Options  Window  Help

Python 3.9.0 (tags/v3.9.0:9cf6752, Oct  5 2020, 15:34:40) [MSC v.1927 64 bit (AM
D64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
======================= RESTART: E:/Ds/Practical 7a.py =========================
Before : [None, None, None, None]
Hash value for 23 is :3
Hash value for 43 is :3
Collision detected for 43
Hash value for 1 is :1
Hash value for 87 is :3
Collision detected for 87
After: [43, 1, 87, 23]
>>>

                                                              Ln: 13  Col: 4
```

Practical 7a Github Link:

Practical No 7(B)

<u>Aim:</u> Write a program to implement the concept of linear probing.

<u>Theory:</u>

Linear probing is a scheme in computer programming for resolving hash collisions of values of hash functions by sequentially searching the hash table for a free location. This is accomplished using two values - one as a starting value and one as an interval between successive values in modular arithmetic. The second value, which is the same for all keys and known as the stepsize, is repeatedly added to the starting value until a free space is found, or the entire table is traversed.

As we can see, it may happen that the hashing technique is used to create an already used index of the array. In such a case, we can search the next empty location in the array by looking into the next cell until we find an empty cell. This technique is called linear probing.

<u>Code:</u>

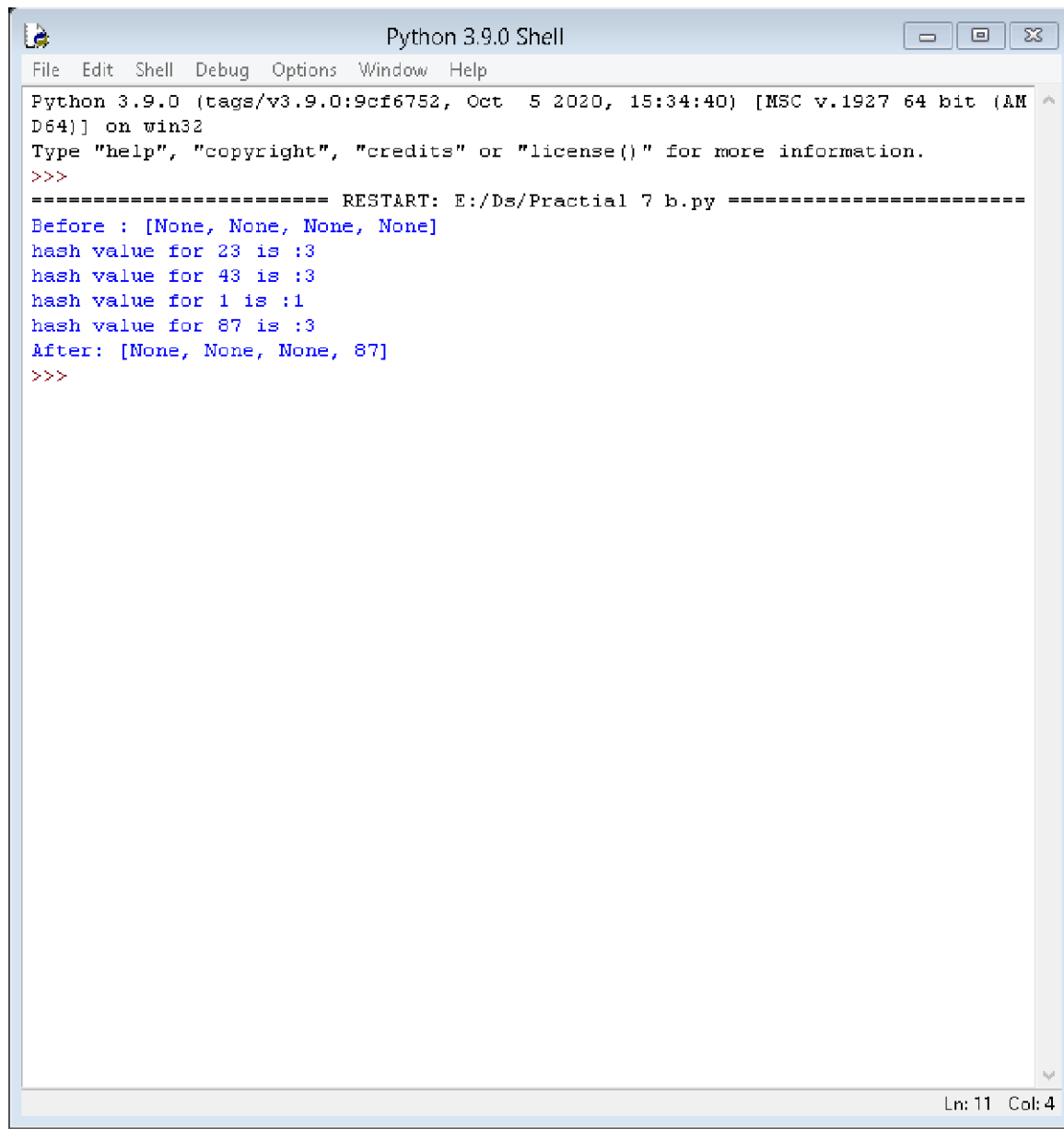File   Edit   Format   Run   Options   Window   Help

```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__':
    linear_probing = True
    list_of_keys = [23,43,1,87]
    list_of_list_index = [None,None,None,None]
    print("Before : " + str(list_of_list_index))
    for value in list_of_keys:
        #print(Hash(value,0,len(list_of_keys)).get_key_value())
        list_index = Hash(value,0,len(list_of_keys)).get_key_value()
        print("hash value for " + str(value) + " is :" + str(list_index))
        if list_of_list_index[list_index]:
            print("Collission detected for " + str(value))
            if linear_probing:
                old_list_index = list_index
                if list_index == len(list_of_list_index)-1:
                    list_index = 0
                else:
                    list_index += 1
                list_full = False
                while list_of_list_index[list_index]:
                    if list_index == old_list_index:
                        list_full = True
                        break
                    if list_index+1 == len(list_of_list_index):
                        list_index = 0
                    else:
                        list_index += 1
                if list_full:
                    print("List was full . Could not save")
                else:
                    list_of_list_index[list_index] = value

        else:
            list_of_list_index[list_index] = value
```

Output:

```
Python 3.9.0 Shell
File  Edit  Shell  Debug  Options  Window  Help

Python 3.9.0 (tags/v3.9.0:9cf6752, Oct  5 2020, 15:34:40) [MSC v.1927 64 bit (AM
D64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
======================= RESTART: E:/Ds/Practial 7 b.py =======================
Before : [None, None, None, None]
hash value for 23 is :3
hash value for 43 is :3
hash value for 1 is :1
hash value for 87 is :3
After: [None, None, None, 87]
>>>
```

Practical 7b Github Link:

Practical No 8

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

Theory:

Tree Traversals

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.

Inorder Traversal

Algorithm Inorder(tree)

- Traverse the left subtree, i.e., call Inorder(left-subtree)

- Visit the root.

- Traverse the right subtree, i.e., call Inorder(right-subtree)

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. Example: Inorder traversal for the above-given figure is 4 2 5 1 3.

Preorder Traversal :

Algorithm Preorder(tree)

- Visit the root.

- Traverse the left subtree, i.e., call Preorder(left-subtree)

- Traverse the right subtree, i.e., call Preorder(right-subtree)

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.

Example: Preorder traversal for the above given figure is 1 2 4 5 3.

Postorder Traversal

Algorithm Postorder

- Traverse the left subtree, i.e., call Postorder(left-subtree)

- Traverse the right subtree, i.e., call Postorder(right-subtree)

- Visit the root.

Uses of Postorder

Postorder traversal is used to delete the tree. Postorder traversal is also useful to get the postfix expression of an expression tree.

Example: Postorder traversal for the above given figure is 4 5 2 3 1.

Code:

File   Edit   Format   Run   Options   Window   Help

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

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

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

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

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

    def insert(self, data):
        if self.value:
            if data < self.value:
                if self.left is None:
                    self.left = Node(data)
                else:
                    self.left.insert(data)
```

Output:

```
Python 3.9.0 Shell                                               ☐ ☐ ☒

File  Edit  Shell  Debug  Options  Window  Help

Python 3.9.0 (tags/v3.9.0:9cf6752, Oct  5 2020, 15:34:40) [MSC v.1927 64 bit (AM
D64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
======================= RESTART: E:/Ds/practical 8.py =======================
Without any order
12
10
5
Now ordering with insert
4
13
28
123
130
Pre order
28
4
13
130
123
In Order
4
13
28
123
130
Post Order
13
4
123
130
28
>>>

                                                              Ln: 33  Col: 4
```

Practical 8 Github Link:

https://github.com/manishmo143/ds-
practical/blob/master/Practical8.py