MALAD KANDIVALI EDUCATION SOCIETY'S

NAGINDAS KHANDWALA COLLEGE OF COMMERCE, ARTS & MANAGEMENT STUDIES & SHANTABEN NAGINDASKHANDWALA 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.OMKAR RAMESHCHANDRA YADAV

Roll No: __345_____ Programme: BSc IT Semester: III

This is certified to be a bonafide record of practical works done by the above student in the college laboratory for the course **Data Structures (Course Code: 2032UISPR)** for the partial fulfilment of Third Semester of BSc IT during the academic year 2020-21.

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

External Examiner

Mr. Gangashankar Singh
(Subject-In-Charge)

Date of Examination: (College Stamp)

Class: S.Y. B.Sc. IT Sem- III **Roll No: 345**

Subject: Data Structures

INDEX

PRACTICAL 1A

Aim: 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.

Theory:

Storing Data in Arrays. Assigning values to an element in 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.

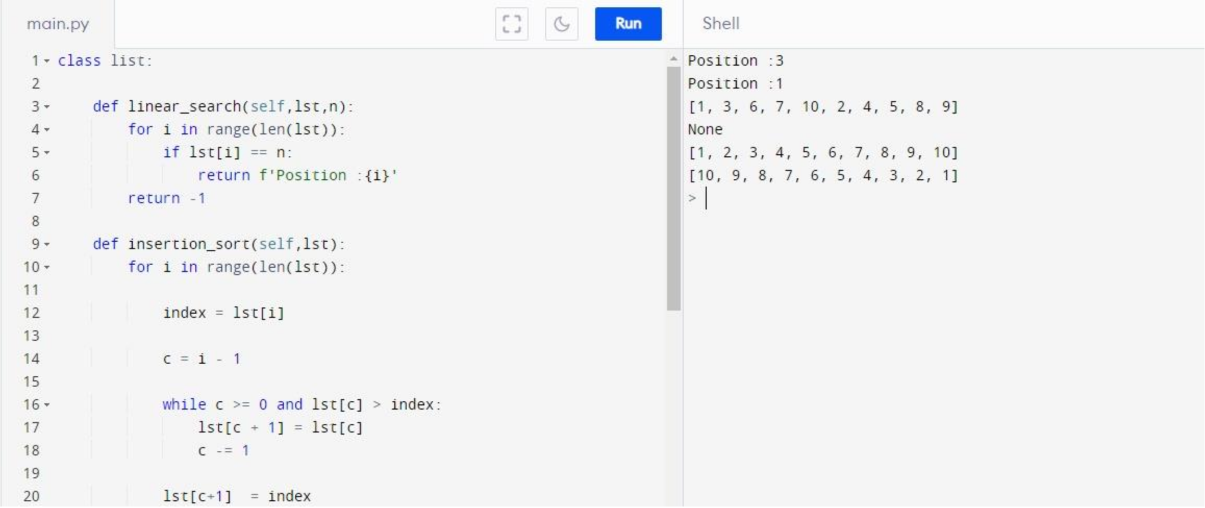
- Traverse − print all the array elements one by one.

-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 and Output:



```python
class list:

    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]

            c = i - 1

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

            lst[c+1]  = index
```

```
Position :3
Position :1
[1, 3, 6, 7, 10, 2, 4, 5, 8, 9]
None
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
> 
```

PRACTICAL 1B

Aim: Implement the following for Array:

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

Theory:

-add() − add elements of two matrices.

-subtract() − subtract elements of two matrices.

-divide() − divide elements of two matrices.

-multiply() − multiply elements of two matrices.

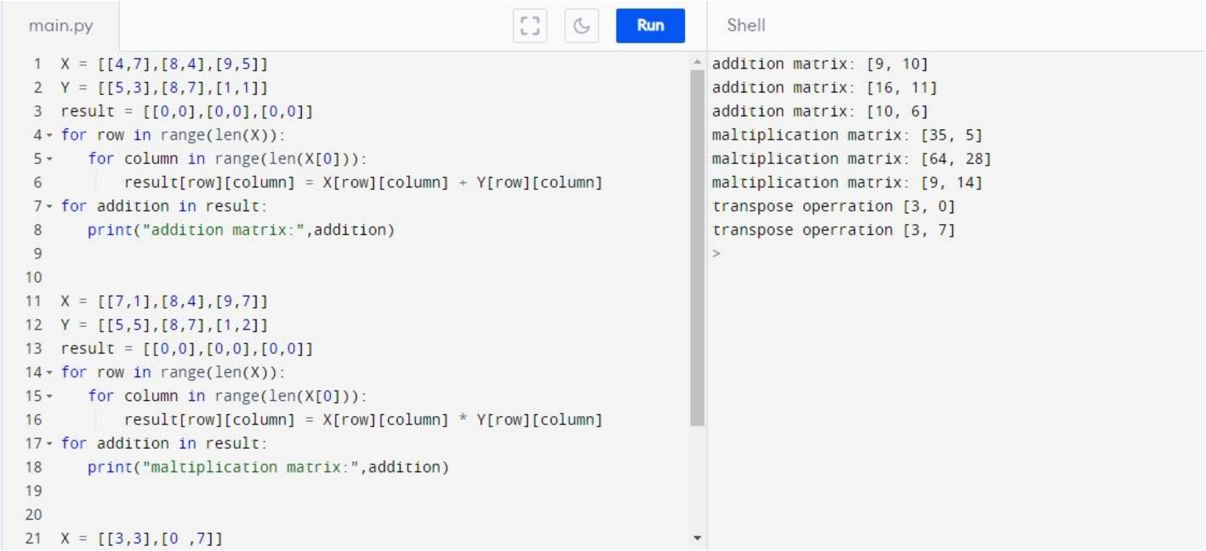
-dot() − It performs matrix multiplication, does not element wise multiplication.

-sqrt() − square root of each element of matrix.

- sum(x-axis) − add to all the elements in matrix. Second argument is optional, it is used when we want to compute the column sum if axis is 0 and row sum if axis is 1.

- “T” − It performs transpose of the specified matrix.

Code and output:



```
X = [[4,7],[8,4],[9,5]]
Y = [[5,3],[8,7],[1,1]]
result = [[0,0],[0,0],[0,0]]
for row in range(len(X)):
   for column in range(len(X[0])):
       result[row][column] = X[row][column] + Y[row][column]
for addition in result:
   print("addition matrix:",addition)


X = [[7,1],[8,4],[9,7]]
Y = [[5,5],[8,7],[1,2]]
result = [[0,0],[0,0],[0,0]]
for row in range(len(X)):
   for column in range(len(X[0])):
       result[row][column] = X[row][column] * Y[row][column]
for addition in result:
   print("maltiplication matrix:",addition)


X = [[3,3],[0 ,7]]
```

```
addition matrix: [9, 10]
addition matrix: [16, 11]
addition matrix: [10, 6]
maltiplication matrix: [35, 5]
maltiplication matrix: [64, 28]
maltiplication matrix: [9, 14]
transpose operration [3, 0]
transpose operration [3, 7]
>
```

Practical 2

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

Theory:

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

- Insertion in a Linked list: 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.

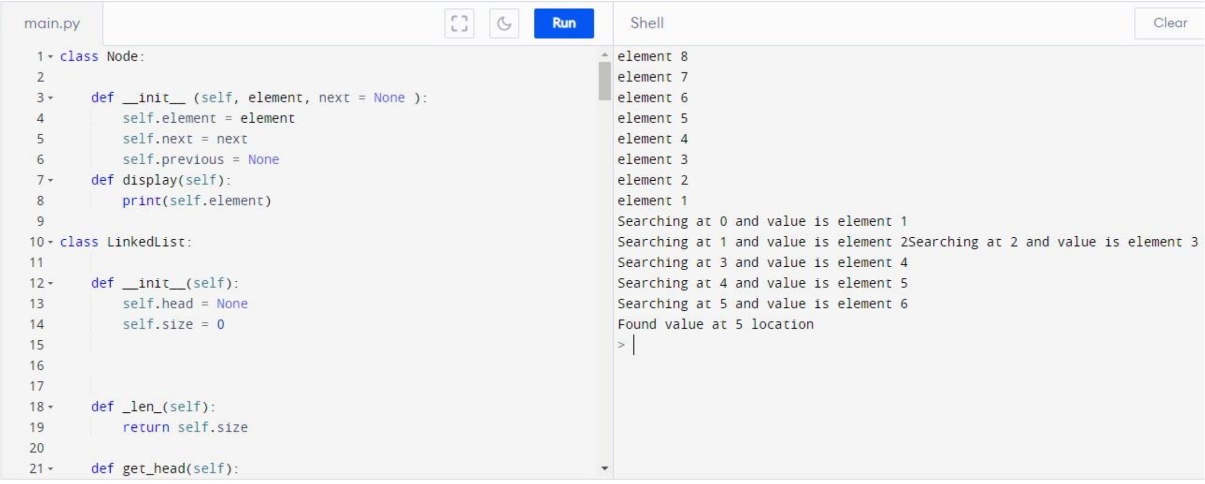
-Deleting an Item form a Linked List: We can remove an existing node using the key for that node. In the below program we locate the previous node of the node which is to be deleted. Then point the next pointer of this node to the next node of the node to be deleted.

- Searching in linked list: Searching is performed in order to find the location of a particular element in the list. Searching any element in the list needs traversing through the list and make the comparison of every element of the list with the specified element. If the element is matched with any of the list element then the location of the element is returned from the function.

- Reversing a Linked list: To reverse a Linked List recursively we need to divide the Linked List into two parts: head and remaining. Head points to the first element initially. Remaining points to the next element from the head. We traverse the Linked List recursively until the second last element.

-Concatenating Linked lists: Concatenate the two lists by traversing the first list until we reach it’s a tail node and then point the next of the tail node to the head node of the second list. Store this concatenated list in the first list

Code and Output:



```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):
```

```
element 8
element 7
element 6
element 5
element 4
element 3
element 2
element 1
Searching at 0 and value is element 1
Searching at 1 and value is element 2Searching at 2 and value is element 3
Searching at 3 and value is element 4
Searching at 4 and value is element 5
Searching at 5 and value is element 6
Found value at 5 location
> 
```

Practical 3A

Aim: Implement the following for Stack:

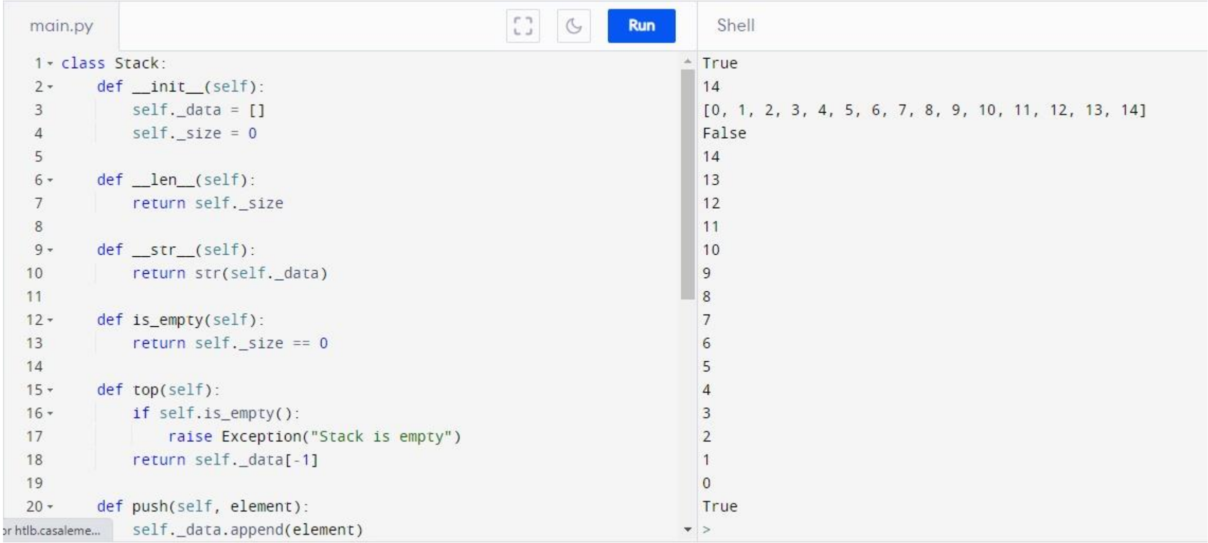
a) Perform Stack operations using Array implementation.

Theory:

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

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

Code and Output:



```python
class Stack:
    def __init__(self):
        self._data = []
        self._size = 0

    def __len__(self):
        return self._size

    def __str__(self):
        return str(self._data)

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

    def top(self):
        if self.is_empty():
            raise Exception("Stack is empty")
        return self._data[-1]

    def push(self, element):
        self._data.append(element)
```

```
True
14
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
False
14
13
12
11
10
9
8
7
6
5
4
3
2
1
0
True
>
```

Practical 3B

Aim: Implement Tower of Hanoi.

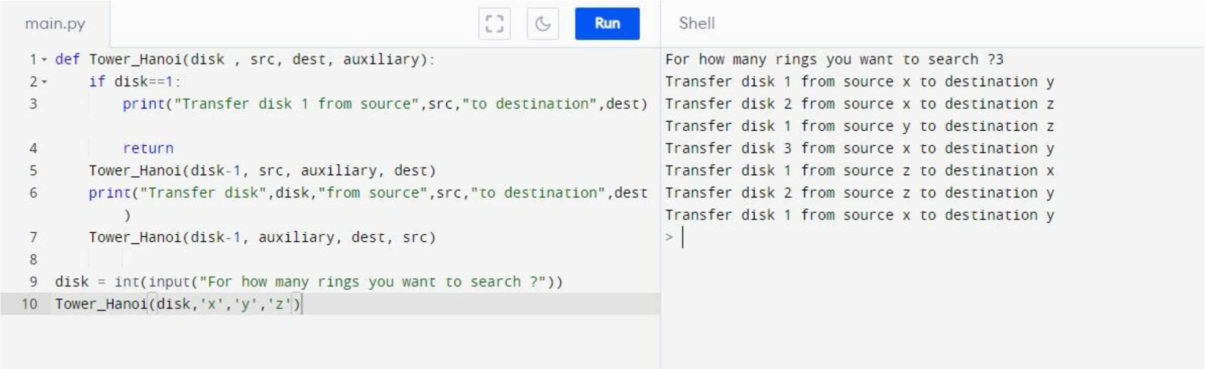
Theory:

- We are given n disks and a series of rods; we need to transfer all the disks to the final rod under the given constraints

- We can move only one disk at a time

. -Only the uppermost disk.

Code and Output:



```python
def Tower_Hanoi(disk , src, dest, auxiliary):
    if disk==1:
        print("Transfer disk 1 from source",src,"to destination",dest)

        return
    Tower_Hanoi(disk-1, src, auxiliary, dest)
    print("Transfer disk",disk,"from source",src,"to destination",dest
        )
    Tower_Hanoi(disk-1, auxiliary, dest, src)

disk = int(input("For how many rings you want to search ?"))
Tower_Hanoi(disk,'x','y','z')
```

```
For how many rings you want to search ?3
Transfer disk 1 from source x to destination y
Transfer disk 2 from source x to destination z
Transfer disk 1 from source y to destination z
Transfer disk 3 from source x to destination y
Transfer disk 1 from source z to destination x
Transfer disk 2 from source z to destination y
Transfer disk 1 from source x to destination y
>
```

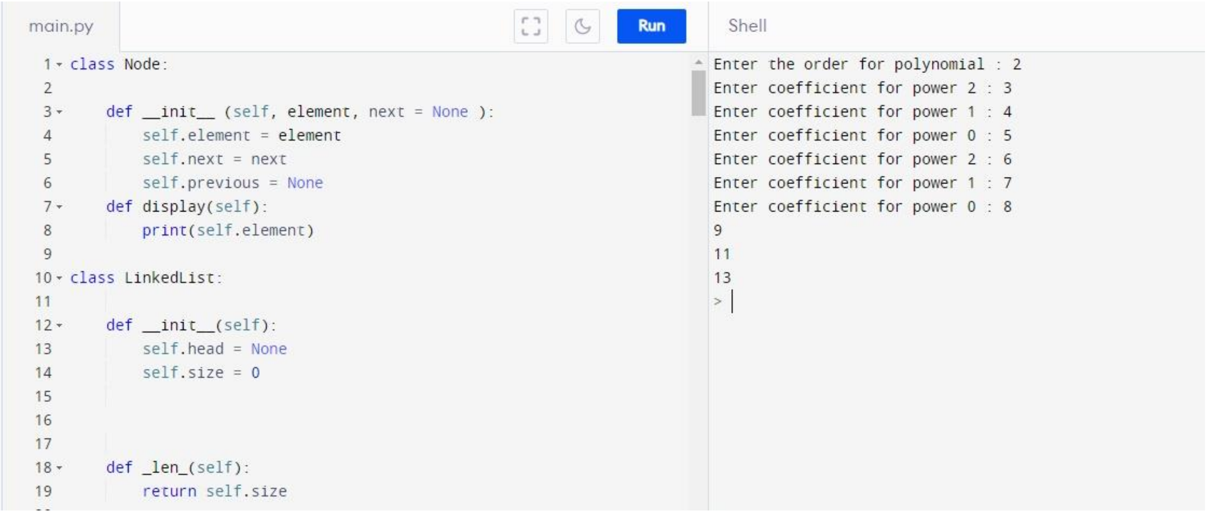
Practical 3C

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

Theory:

Polynomial is a mathematical expression that consists of variables and coefficients. for example x^2 - 4x +7. In the Polynomial linked list, the coefficients and exponents of the polynomial are defined as the data node of the list. For adding two polynomials that are stored as a linked list. We need to add the coefficients of variables with the same power. In a linked list node contains 3 members, coefficient value link to the next node a linked list that is used to store Polynomial looks like −Polynomial : 4x7 + 12x2 + 45

Code and Output:



```
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
```

```
Enter the order for polynomial : 2
Enter coefficient for power 2 : 3
Enter coefficient for power 1 : 4
Enter coefficient for power 0 : 5
Enter coefficient for power 2 : 6
Enter coefficient for power 1 : 7
Enter coefficient for power 0 : 8
9
11
13
>
```

Practical 3D

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

(i) using recursion

(ii) using iteration

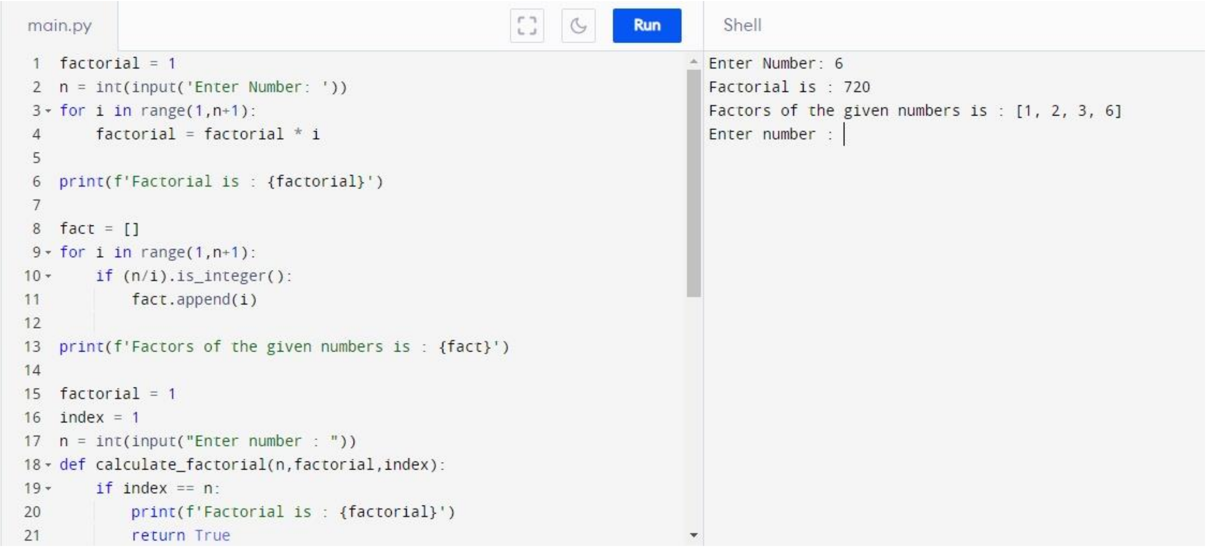
Theory:

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

• Recursion: In Python, we know that a function can call other functions. It is even possible for the function to call itself. These types of construct are termed as recursive functions.

• Iteration: Repeating identical or similar tasks without making errors is something that computers do well and people do poorly. Repeated execution of a set of statements is called iteration. Because iteration is so common, Python provides several language features to make it easier.

Code and Output:



```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
```

```
Enter Number: 6
Factorial is : 720
Factors of the given numbers is : [1, 2, 3, 6]
Enter number :
```

Practical 4

Aim: Perform Queues operations using Circular Array implementation.

Theory:

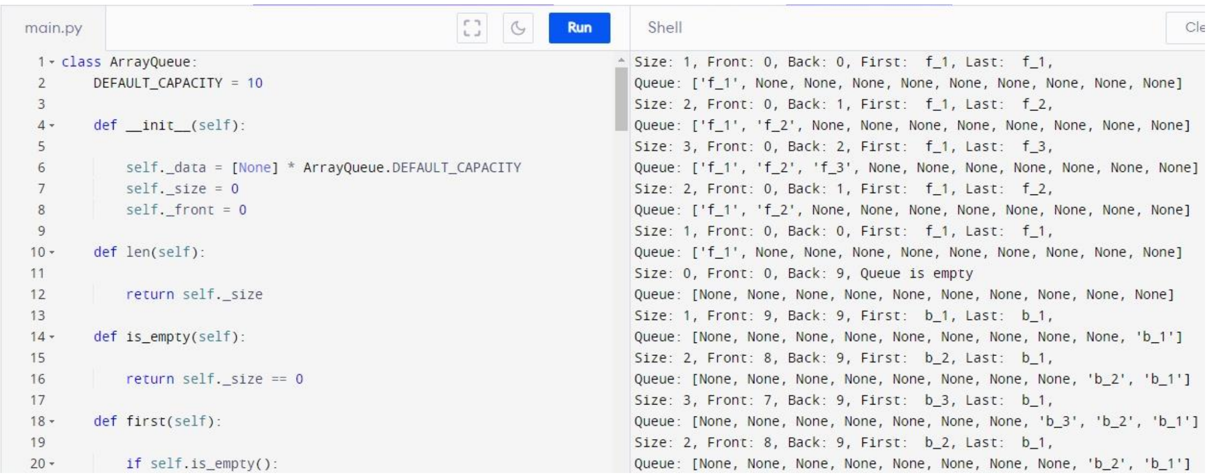
Circular queue avoids the wastage of space in a regular queue implementation using arrays. Circular Queue works by the process of circular increment i.e. when we try to increment the pointer and we reach the end of the queue, we start from the beginning of the queue. Here, the circular increment is performed by modulo division with the queue size. That is, if REAR + 1 == 5 (overflow!), REAR = (REAR + 1)%5 = 0 (start of queue) The circular queue work as follows:

two pointers FRONT and REAR FRONT track the first element of the queue REAR track the last elements of the queue initially, set value of FRONT and REAR to -1

1. 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

2. Dequeue Operation 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

Code and Output:



```
class ArrayQueue:
    DEFAULT_CAPACITY = 10

    def __init__(self):

        self._data = [None] * ArrayQueue.DEFAULT_CAPACITY
        self._size = 0
        self._front = 0

    def len(self):

        return self._size

    def is_empty(self):

        return self._size == 0

    def first(self):

        if self.is_empty():
```

```
Size: 1, Front: 0, Back: 0, First:  f_1, Last:  f_1,
Queue: ['f_1', None, None, None, None, None, None, None, None, None]
Size: 2, Front: 0, Back: 1, First:  f_1, Last:  f_2,
Queue: ['f_1', 'f_2', None, None, None, None, None, None, None, None]
Size: 3, Front: 0, Back: 2, First:  f_1, Last:  f_3,
Queue: ['f_1', 'f_2', 'f_3', None, None, None, None, None, None, None]
Size: 2, Front: 0, Back: 1, First:  f_1, Last:  f_2,
Queue: ['f_1', 'f_2', None, None, None, None, None, None, None, None]
Size: 1, Front: 0, Back: 0, First:  f_1, Last:  f_1,
Queue: ['f_1', None, None, None, None, None, None, None, None, None]
Size: 0, Front: 0, Back: 9, Queue is empty
Queue: [None, None, None, None, None, None, None, None, None, None]
Size: 1, Front: 9, Back: 9, First:  b_1, Last:  b_1,
Queue: [None, None, None, None, None, None, None, None, None, 'b_1']
Size: 2, Front: 8, Back: 9, First:  b_2, Last:  b_1,
Queue: [None, None, None, None, None, None, None, None, 'b_2', 'b_1']
Size: 3, Front: 7, Back: 9, First:  b_3, Last:  b_1,
Queue: [None, None, None, None, None, None, None, 'b_3', 'b_2', 'b_1']
Size: 2, Front: 8, Back: 9, First:  b_2, Last:  b_1,
Queue: [None, None, None, None, None, None, None, None, 'b_2', 'b_1']
```

Practical 5

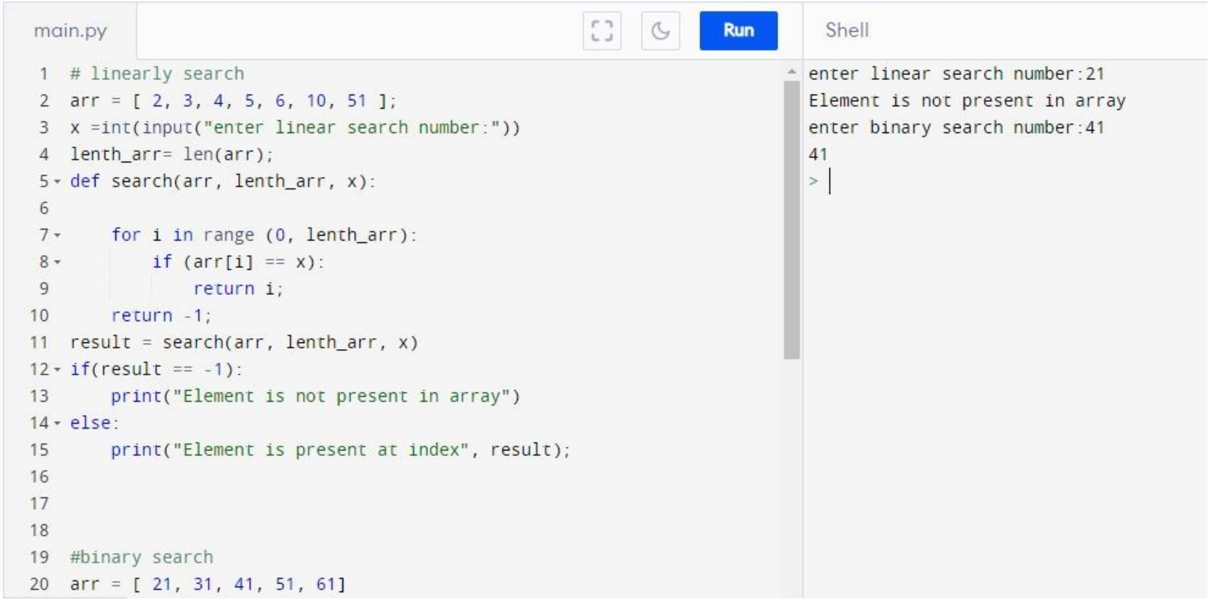
Aim: Write a program to search an element from a list. Give user the option to perform Linear or Binary search.

Theory:

• Linear Search: This linear search is a basic search algorithm which searches all the elements in the list and finds the required value. This is also known as sequential search.

• Binary Search: In computer science, a binary searcher half-interval search algorithm finds the position of a target value within a sorted array. The binary search algorithm can be classified as a dichotomies divide-and-conquer search algorithm and executes in logarithmic time.

Code and Output:



```
# linearly search
arr = [ 2, 3, 4, 5, 6, 10, 51 ];
x =int(input("enter linear search number:"))
lenth_arr= len(arr);
def search(arr, lenth_arr, x):

    for i in range (0, lenth_arr):
        if (arr[i] == x):
            return i;
    return -1;
result = search(arr, lenth_arr, x)
if(result == -1):
    print("Element is not present in array")
else:
    print("Element is present at index", result);



#binary search
arr = [ 21, 31, 41, 51, 61]
```

```
enter linear search number:21
Element is not present in array
enter binary search number:41
41
>
```

Practical 6

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

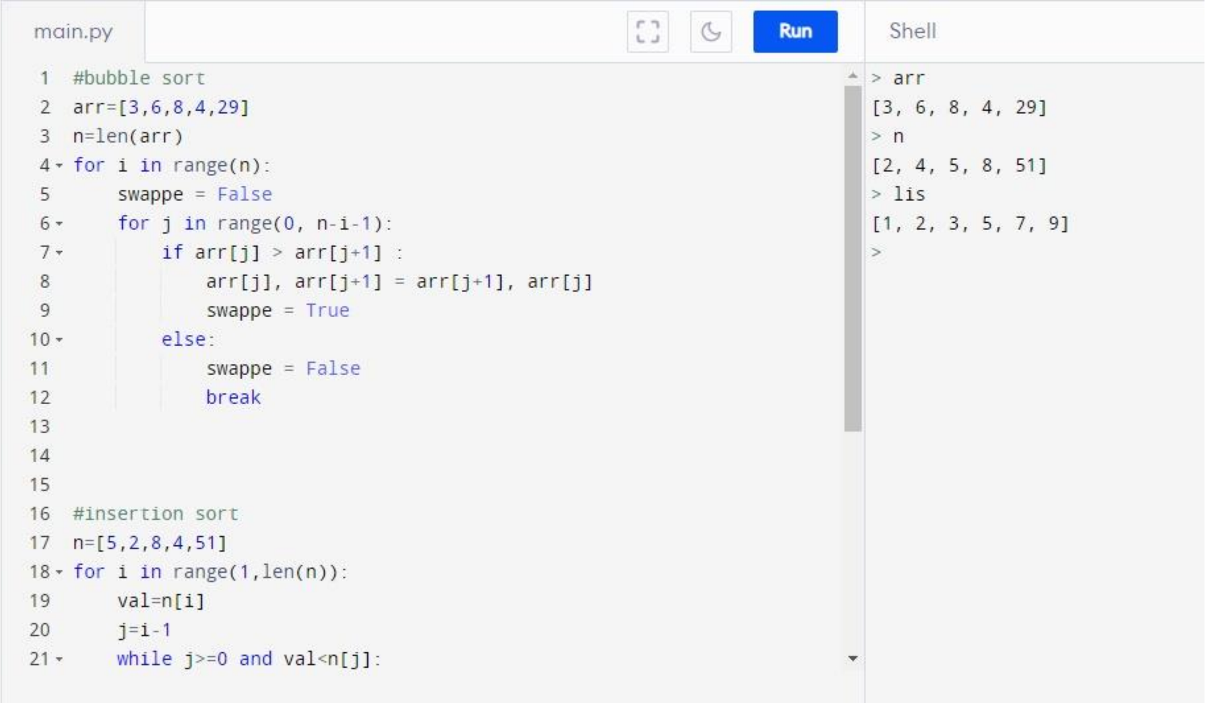
Theory:

- Bubble Sort: Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order.

- Selection Sort: The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning. The algorithm maintains two sub arrays in a given array

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

Code and Output:



```python
#bubble sort
arr=[3,6,8,4,29]
n=len(arr)
for i in range(n):
    swappe = False
    for j in range(0, n-i-1):
        if arr[j] > arr[j+1] :
            arr[j], arr[j+1] = arr[j+1], arr[j]
            swappe = True
        else:
            swappe = False
            break



#insertion sort
n=[5,2,8,4,51]
for i in range(1,len(n)):
    val=n[i]
    j=i-1
    while j>=0 and val<n[j]:
```

```
> arr
[3, 6, 8, 4, 29]
> n
[2, 4, 5, 8, 51]
> lis
[1, 2, 3, 5, 7, 9]
>
```

Practical 7A

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

Theory:

Hashing: 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

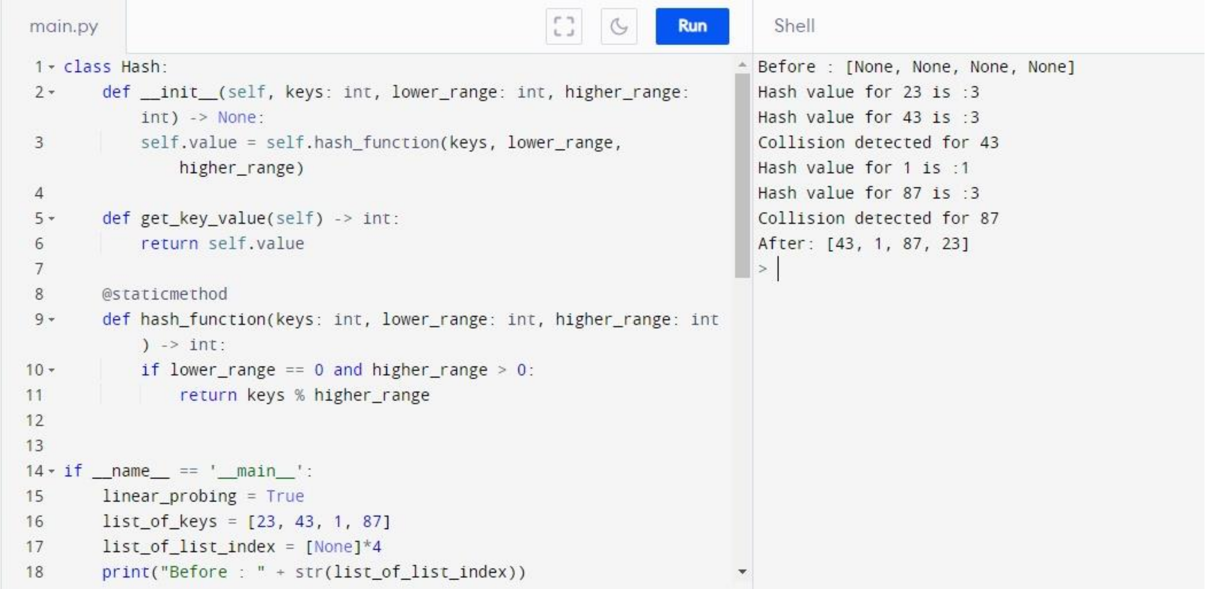
. • Collisions: A Hash Collision Attack is an attempt to find two input strings of a hash function that produce the same hash result. If two separate inputs produce the same hash output, it is called a collision.

• Collision Techniques: When one or more hash values compete with a single hash table slot, collisions occur. To resolve this, the next available empty slot is assigned to the current hash value

• Separate Chaining: The idea is to make each cell of hash table point to a linked list of records that have same hash function value.

• Open Addressing: Like separate chaining, open addressing is a method for handling collisions. In Open Addressing, all elements are stored in the hash table itself. So at any point, the size of the table must be greater than or equal to the total number of keys (Note that we can increase table size by copying old data if needed)

Code and Output:



```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))
```

```
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]
>
```

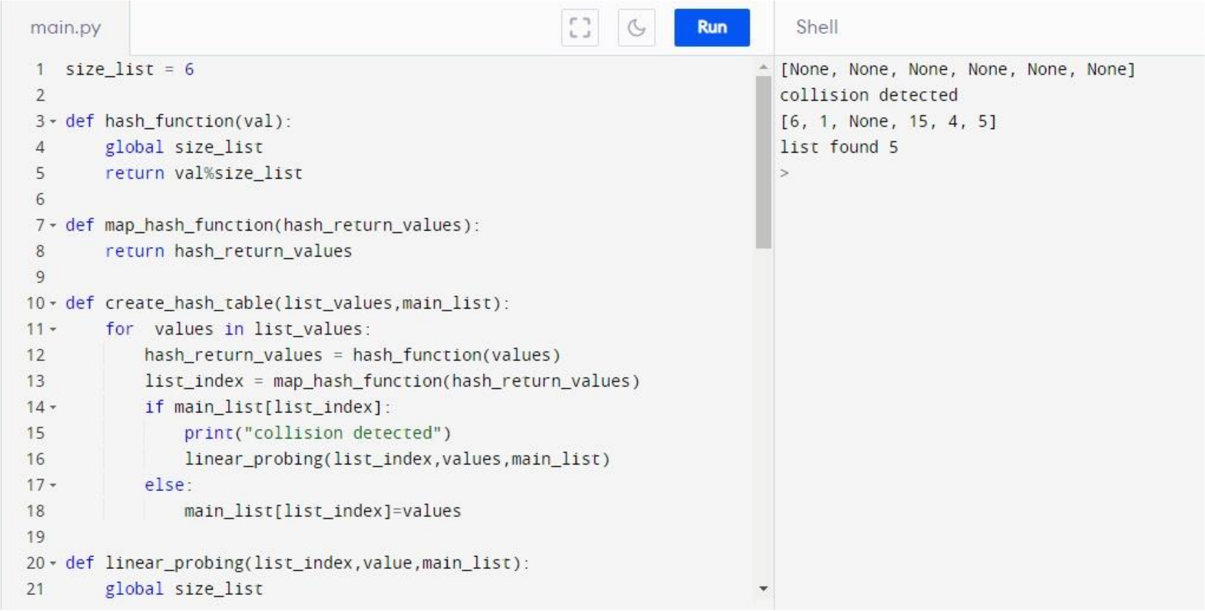
Practical 7B

Aim: Implement the following for Hashing: Write a program to implement the concept of linear probing.

Theory:

Linear probing is a scheme in computer programming for resolving collisions in hash tables, data structures for maintaining a collection of keys–value pairs and looking up the value associated with a given key. Along with quadratic probing and double hashing, linear probing is a form of open addressing.

Code and Output:



```python
size_list = 6

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

def map_hash_function(hash_return_values):
    return hash_return_values

def create_hash_table(list_values,main_list):
    for  values in list_values:
        hash_return_values = hash_function(values)
        list_index = map_hash_function(hash_return_values)
        if main_list[list_index]:
            print("collision detected")
            linear_probing(list_index,values,main_list)
        else:
            main_list[list_index]=values

def linear_probing(list_index,value,main_list):
    global size_list
```

```
[None, None, None, None, None, None]
collision detected
[6, 1, None, 15, 4, 5]
list found 5
>
```

Practical 8

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

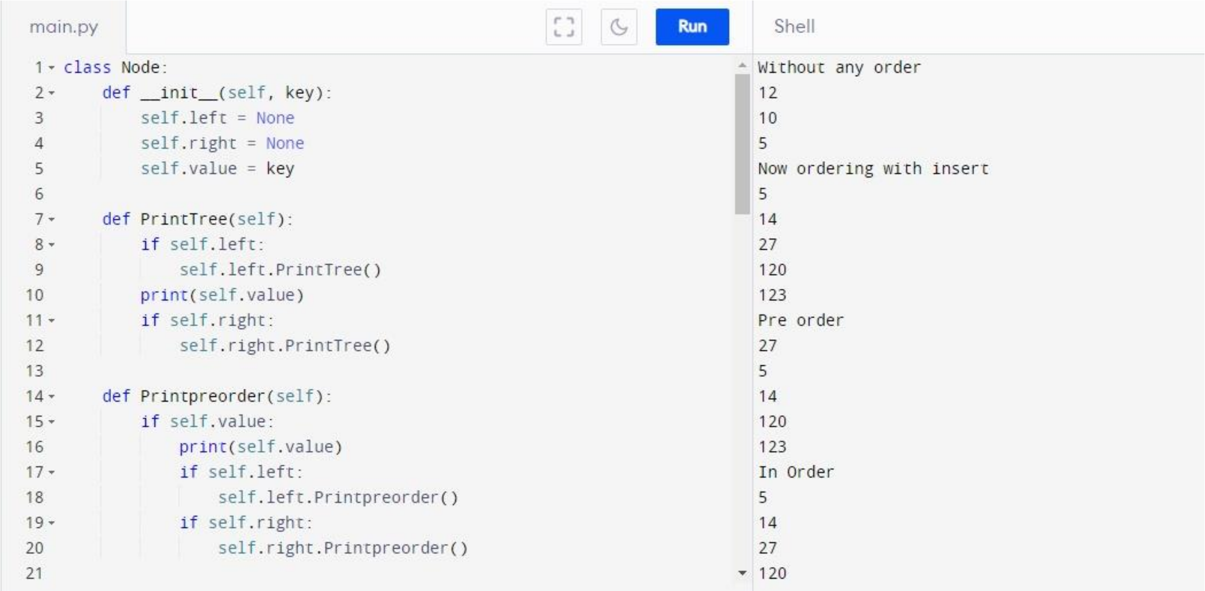
Theory:

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

• 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.

• Postorder: Postorder traversal is also useful to get the postfix expression of an expression tree

Code and Output:



```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()

```

```
Without any order
12
10
5
Now ordering with insert
5
14
27
120
123
Pre order
27
5
14
120
123
In Order
5
14
27
120
```