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. Shravan Shivanand Kamat

Roll No: 372 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: 372**

Subject: Data Structures

INDEX

Github Repository Link: https://github.com/cryogen78/DS

Practical 1. Implement the following for Array:

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:

What is searching?
Searching is the process of finding a given value position in a list of values. It decides whether a search key is present in the data or not. It is the algorithmic process of finding a particular item in a collection of items.

What are some searching algorithms?
Linear Search :
A simple approach is to do a linear search, i.e

- Start from the leftmost element of arr[] and one by one compare x with each element of arr[]
- If x matches with an element, return the index.
- If x doesn’t match with any of elements, return -1.

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.

Sorting:

Sorting means arranging the elements of a list in a specific order.

There are many sorting algorithms like:

- Bubble sort
- Merge sort
- Selection Sort
- Insertion Sort
- Quick sort etc

Merging:

Merging means to join two list.

Reversing:

Reversing means the to arrange the elements of the list in reverse order.

In the Code below one of the searching and sorting techniques have been applied.

Code and Output:

prac1a.py - D:\studies\data structures\prac1a.py (3.7.7)

```
# To store the elements in 1-D array and provide an option to perform the operat
#3005 (New Rollno:372) Shravan SYIT
class Array:

    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,lst1,lst2):
        return Array.insertion_sort(lst1 + lst2)

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

lst = [5,31,44,94,108,109]
Arrmod = Array()
print(Arrmod.linear_search(lst,3))
```

Python 3.7.7 Shell

```
Python 3.7.7 (tags/v3.7.7:d7c567b08f, Mar 10 2020, 10:41:24) [MS(
(AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more info:
>>>
================ RESTART: D:\studies\data structures\prac1a.py =
-1
>>>
```

Practical 1. Implement the following for Array:

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

Theory

Matrix Addition: Matrix addition is the operation of adding two matrices by adding the corresponding entries together. The matrix can be added only when the number of rows and columns of the first matrix is equal to the number of rows and columns of the second matrix.

Matrix Multiplication: We can multiply two matrices if, and only if, the number of columns in the first matrix equals the number of rows in the second matrix. Otherwise, the product of two matrices is undefined.

Matrix Transpose: If A=[aij] be a matrix of order m x n, then the matrix obtained by interchanging the rows and columns of A is known as Transpose of matrix A. Transpose of matrix A is represented by AT.

Code and Output:

prac1b.py - D:\studies\data structures\prac1b.py (3.7.7)

File Edit Format Run Options Window Help

```python
#Write a program to perform the Matrix addition, Multiplication and Transpose Operation.
#3005 (New Rollno:372) Shravan SYIT
Mat1 = [[5, 8, -5],
      [31,44,94],
      [16,108,109]]

Mat2 = [[5, 1, -3],
           [2,6,-8],
           [-1,3,3]]
Mat3 = [[0,0,0,],
     [0,0,0,],
     [0,0,0,]]

for i in range(len(Mat1)):
    for j in range(len(Mat2[0])):
        for k in range(len(Mat2)):
            Mat3[i][j] += Mat1[i][k] + Mat2[k][j]

print(Mat3)

Mat3 = [[0, 0, 0, 0],
        [0, 0, 0, 0],
        [0, 0, 0, 0]]

for i in range(len(Mat1)):
    for j in range(len(Mat2[0])):
        for k in range(len(Mat2)):
            Mat3[i][j] += Mat1[i][k] * Mat2[k][j]

print(Mat3)

for i in map(list, zip(*Mat1)):
    print(i)
```

Python 3.7.7 Shell

File Edit Shell Debug Options Window Help

```
Python 3.7.7 (tags/v3.7.7:d7c567b08f, Mar 10 2020, 10:41:24) [MS
(AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more info
>>>
================= RESTART: D:\studies\data structures\prac1b.py
[[14, 18, 0], [175, 179, 161], [239, 243, 225]]
[[46, 38, -94, 0], [149, 577, -163, 0], [187, 991, -585, 0]]
[5, 31, 16]
[8, 44, 108]
[-5, 94, 109]
>>> 
```

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

Singly Linked List:
A singly linked list, in its simplest form, is a collection of nodes that collectively form a linear sequence. Each node stores a reference to an object that is an element of the sequence, as well as a reference to the next node of the list
Doubly Linked List:
In a singly linked list, each node maintains a reference to the node that is immediately after it. However, there are limitations that stem from the asymmetry of a singly linked list. To provide greater symmetry, we define a linked list in which each node keeps an explicit reference to the node before it and a reference to the node after it.
Such a structure is known as a doubly linked list. These lists allow a greater variety of O (1)-time update operations, including insertions and deletions at arbitrary positions within the list. We continue to use the term “next” for the reference to the node that follows another, and we introduce the term “prey” for the reference to the node that precedes it. With array-based sequences, an integer index was a convenient means for describing a position within a sequence. However, an index is not convenient for linked lists as there is no efficient way to find the jth element; it would seem to require a traversal of a portion of the list.
When working with a linked list, the most direct way to describe the location of an operation is by identifying a relevant node of the list. However, we prefer to encapsulate the inner workings of our data structure to avoid having users directly access nodes of a list.

Code:

prac2.py - D:\studies\data structures\prac2.py (3.7.7)

File Edit Format Run Options Window Help

```python
#Implement Linked List. Include options for insertion, deletion and search of a number, reverse the list and concatenate two linked lists
#3005 (New Rollno:372) Shravan SYIT
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
```



```python
    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

    def get_head(self) -> Node:
        return self.start

    def remove_head(self):
        if self.is_empty():
            print("Doubly Linked List is empty")
            return
        elif self.length == 1:
            self.start = None
            self.end = None
        else:
            self.start = self.start.next
        self.length -= 1

    def add_tail(self, e):
        new_value = Node(e)
        if self.get_tail() is not None:
            old_tail = self.end
            self.end = new_value
            self.end.prev = old_tail
            old_tail.next = self.end
            self.length += 1
        else:
            self.add_head(e)

    def get_tail(self) -> Node:
        return self.end

    def remove_tail(self):
        if self.is_empty():
```

```
def remove_tail(self):
    if self.is_empty():
        print("Doubly Linked List is empty")
        return
    elif self.length == 1:
        self.start = None
        self.end = None
    else:
        self.end = self.end.prev
    self.length -= 1

def add_between_list(self, index, element):
    if index > self.length or index < 0:
        print("Index out of bounds")
    elif index == self.length:
        self.add_tail(element)
    elif index == 0:
        self.add_head(element)
    else:
        prev_node = self.get_node_at(index - 1)
        current_node = self.get_node_at(index)
        new_value = Node(element)
        prev_node.next = new_value
        new_value.next = current_node
        new_value.prev = prev_node
        current_node.prev = new_value
        self.length += 1

def get_node_at(self, index, direction="auto") -> Node:
    if index < 0 or index >= self.length:
        print("Index out of bounds")
    else:
        element_node = self.start
        counter = 0
        while counter < index:
            element_node = element_node.next
            counter += 1
        return element_node

def remove_between_list(self, index):
```

```
    def remove_between_list(self, index):
        if index < 0 or index >= self.length:
            print("Index out of bounds")
        elif index == self.length - 1:
            self.remove_tail()
        elif index == 0:
            self.remove_head()
        else:
            prev_node = self.get_node_at(index - 1)
            next_node = self.get_node_at(index + 1)
            prev_node.next = next_node
            next_node.prev = prev_node
            self.length -= 1

    def merge(self, linked_list_value):
        if not self.is_empty():
            last_node = self.get_tail()
            last_node.next = linked_list_value.start
            if not linked_list_value.is_empty():
                linked_list_value.start.prev = last_node
                self.end = linked_list_value.end
            self.length = self.length + linked_list_value.length
        else:
            self.start = linked_list_value.start
            self.end = linked_list_value.end
            self.length = linked_list_value.length

    def reverse(self):
        if self.is_empty():
            print("Doubly Linked List is empty")
            return
        elif self.length == 1:
            return
        elif self.length == 2:
            temp = self.start
            self.start = self.end
            self.end = temp
            self.start.prev = None
            self.start.next = self.end
            self.end.prev = self.start
```

```
            self.end.prev = self.start
            self.end.next = None
        elif self.length == 3:
            mid = self.__get_node_from_start(1)
            temp = self.start
            self.start = self.end
            self.end = temp
            self.start.prev = None
            self.start.next = mid
            mid.prev = self.start
            mid.next = self.end
            self.end.prev = mid
            self.end.next = None
        elif self.length > 3:
            temp_lst = []
            first = self.start
            temp_lst.append(first)
            first = first.next
            while first:
                temp_lst.append(first)
                first = first.next
            temp_lst.reverse()
            for i in range(0, len(temp_lst)):
                if i == 0:
                    self.start = temp_lst[i]
                    self.start.prev = None
                    self.start.next = temp_lst[i + 1]
                elif i == 1:
                    temp_lst[i].prev = self.start
                    temp_lst[i].next = temp_lst[i + 1]
                elif i == len(temp_lst) - 1:
                    self.end = temp_lst[i]
                    self.end.prev = temp_lst[i - 1]
                    self.end.next = None
                else:
                    temp_lst[i].prev = temp_lst[i - 1]
                    temp_lst[i].next = temp_lst[i + 1]
            temp_lst[len(temp_lst) - 2].prev = temp_lst[len(temp_lst) - 2 - 1]
            temp_lst[len(temp_lst) - 2].next = self.end
```

```
one = LinkedList()
print(one.is_empty())
one.add_between_list(0, "zero")
one.add_between_list(1, "one")
one.add_between_list(2, "two")
one.display()
print(one.is_empty())
one.remove_between_list(1)
one.display()
one.add_between_list(1, "one")
one.display()

two = LinkedList()
two.add_head("first_name")
two.add_tail("last_name")
print("Head: ", end='')
two.get_head().display()
print("Tail: ", end='')
two.get_tail().display()
two.remove_tail()
two.get_tail().display()
two.remove_head()
try:
    two.get_head().display()
except AttributeError as e:
    print(e, "\nNo head found")
two.display()

two = LinkedList()
two.add_between_list(0, "zero")
two.add_between_list(1, "one")
two.add_between_list(2, "two")
two.add_between_list(3, "three")
two.display()
two.get_node_at(3).display()
two.get_node_at(3, "start").display()
two.get_node_at(3, "end").display()

three = LinkedList()
three.add_between_list(0, "0")
```

```
three = LinkedList()
three.add_between_list(0, "0")
three.add_between_list(1, "1")
three.add_between_list(2, "2")
three.add_between_list(3, "4")
two.display()
print("Head: ", end='')
two.get_head().display()
print("Tail: ", end='')
two.get_tail().display()
three.display()
print("Head: ", end='')
three.get_head().display()
print("Tail: ", end='')
three.get_tail().display()
two.merge(three)
two.display()
print("Head: ", end='')
two.get_head().display()
print("Tail: ", end='')
two.get_tail().display()
print(f"Size: {two.get_size()}")
two.display()
two.reverse()
two.display()
```

Output:

```
================= RESTART: D:\studies\data structures\prac2.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 3. Implement the following for Stack:

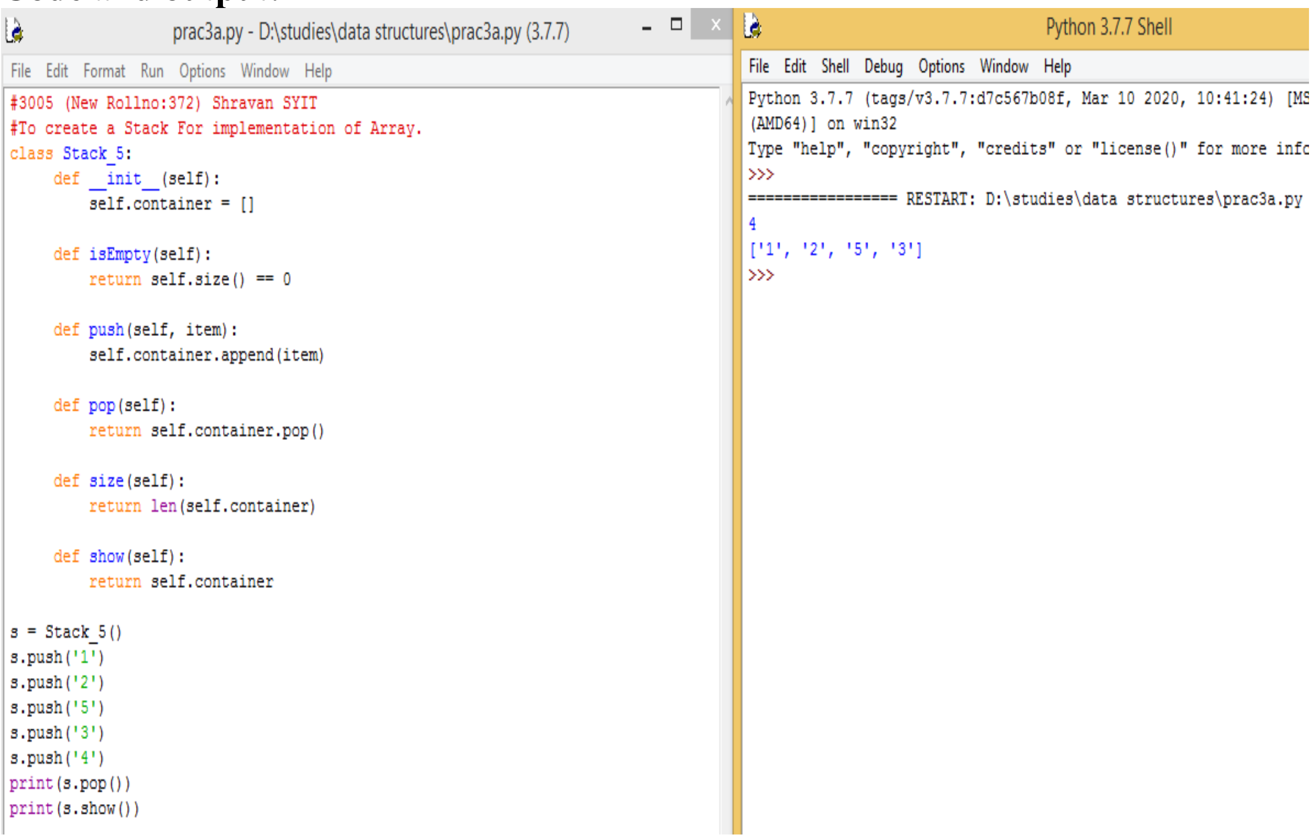
A.

Aim: Perform Stack operations using Array implementation.

Theory

Stack:
A stack is a collection of objects that are inserted and removed according to the last-in, first-out (LIFO) principle. A user may insert objects into a stack at any time, but may only access or remove the most recently inserted object that remains (at the so-called “top” of the stack). We can implement a stack quite easily by storing its elements in a Python list. The list class already supports adding an element to the end with the append method, and removing the last element with the pop method, so it is natural to align the top of the stack at the end of the list. Stack is an abstract data type (ADT) such that an instance S supports the following two methods:
S.push(e): Add element e to the top of stack S.
S.pop(): Remove and return the top element from the stack S;
an error occurs if the stack is empty.

Code and output:




```python
#3005 (New Rollno:372) Shravan SYIT
#To create a Stack For implementation of Array.
class Stack_5:
    def __init__(self):
        self.container = []

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

    def push(self, item):
        self.container.append(item)

    def pop(self):
        return self.container.pop()

    def size(self):
        return len(self.container)

    def show(self):
        return self.container

s = Stack_5()
s.push('1')
s.push('2')
s.push('5')
s.push('3')
s.push('4')
print(s.pop())
print(s.show())
```

```
Python 3.7.7 (tags/v3.7.7:d7c567b08f, Mar 10 2020, 10:41:24) [MS
(AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more info
>>>
================= RESTART: D:\studies\data structures\prac3a.py
4
['1', '2', '5', '3']
>>>
```

Practical 3. Implement the following for Stack:

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:
1) Only one disk can be moved at a time.
2) 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.
3) No disk may be placed on top of a smaller disk.

To write an algorithm for Tower of Hanoi, first we need to learn how to solve this problem with lesser number 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 (nth 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. Each peg is a Stack object.

Code:

prac3b.py - D:\studies\data structures\prac3b.py (3.7.7)

File Edit Format Run Options Window Help

```
#3005 (New Rollno:372) Shravan SYIT
#Implementation of the Tower Of Hanoi.
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)

A = Stack()
B = Stack()
C = Stack()
def Hanoi(n, fromrod,to,temp):
    if n == 1:
        fromrod.pop()
        to.push('disk 1')
        if to.display() != None:
            print(to.display())
```

```
A = Stack()
B = Stack()
C = Stack()
def Hanoi(n, fromrod,to,temp):
    if n == 1:
        fromrod.pop()
        to.push('disk 1')
        if to.display() != None:
            print(to.display())

    else:

        Hanoi(n-1, fromrod, temp, to)
        fromrod.pop()
        to.push(f'disk {n}')
        if to.display() !=  None:
            print(to.display())
        Hanoi(n-1, temp, to, fromrod)

n = int(input('Enter the number of the disk in rod A : '))
for i in  range(n):
    A.push(f'disk {i+1} ')

Hanoi(n, A, C, B)
```

Output:

```
================= RESTART: D:\studies\data structures\prac3b.py ==
Enter the number of the disk in rod A : 5
['disk 1']
['disk 2']
['disk 2', 'disk 1']
['disk 3']
['disk 1 ', 'disk 2 ', 'disk 1']
['disk 3', 'disk 2']
['disk 3', 'disk 2', 'disk 1']
['disk 4']
['disk 4', 'disk 1']
['disk 1 ', 'disk 2']
['disk 1 ', 'disk 2', 'disk 1']
['disk 4', 'disk 3']
['disk 1']
['disk 4', 'disk 3', 'disk 2']
['disk 4', 'disk 3', 'disk 2', 'disk 1']
['disk 5']
['disk 1']
['disk 5', 'disk 2']
['disk 5', 'disk 2', 'disk 1']
['disk 3']
['disk 4', 'disk 1']
['disk 3', 'disk 2']
['disk 3', 'disk 2', 'disk 1']
['disk 5', 'disk 4']
['disk 5', 'disk 4', 'disk 1']
['disk 2']
['disk 2', 'disk 1']
['disk 5', 'disk 4', 'disk 3']
['disk 1']
['disk 5', 'disk 4', 'disk 3', 'disk 2']
['disk 5', 'disk 4', 'disk 3', 'disk 2', 'disk 1']
>>> |
```

Practical 3. Implement the following for Stack:

C.

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

Theory:

Different operations can be performed on the polynomials like addition, subtraction, multiplication, and division. A polynomial is an expression within which a finite number of constants and variables are combined using addition, subtraction, multiplication, and exponents. Adding and subtracting polynomials is just adding and subtracting their like terms. The sum of two monomials is called a binomial and the sum of three monomials is called a trinomial. The sum of a finite number of monomials in x is called a polynomial in x. The coefficients of the monomials in a polynomial are called the coefficients of the polynomial. If all the coefficients of a polynomial are zero, then the polynomial is called the zero polynomial.

Two polynomials can be added by using arithmetic operator plus (+). Adding polynomials is simply "combining like terms" and then add the like terms.

Every Polynomial in the program is a Doubly Linked List object. The corresponding terms are added and displayed in the form of an expression.

Code:

prac3c.py - D:\studies\data structures\prac3c.py (3.7.7)

File Edit Format Run Options Window Help

```
#3005 (New Rollno:372) Shravan SYIT
class DoublyNode:
    def __init__(self, element, next=None, prev=None):
        self.element = element
        self.next = next
        self.prev = prev

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


class DoublyLinkedList:
    def __init__(self):
        self.__head = None
        self.__tail = None
        self.__size = 0

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

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

    def __display_backward(self):
        if self.is_empty():
            print("Doubly Linked List is empty")
            return
        last = self.__tail
        print("The List: ", end='')
        print("[" + last.element, end='')
        last = last.prev
        while last:
            print(", " + last.element, end='')
            last = last.prev
        print("]")

    def __display_forward(self):
        if self.is_empty():
            print("Doubly Linked List is empty")
            return
```

```
        first = self.__head
        print("The List: ", end='')
        print("[" + first.element, end='')
        first = first.next
        while first:
            print(", " + first.element, end='')
            first = first.next
        print("]")

    def display(self, direction="start"):
        if direction == "end":
            self.__display_backward()
        elif direction == "start":
            self.__display_forward()
        else:
            self.__display_forward()

    def add_head(self, e):
        old_head = self.__head
        self.__head = DoublyNode(e)
        self.__head.next = old_head
        if old_head is not None:
            old_head.prev = self.__head
        if self.__tail is None:
            self.__tail = self.__head
        self.__size += 1

    def get_head(self) -> DoublyNode:
        return self.__head

    def remove_head(self):
        if self.is_empty():
            print("Doubly Linked List is empty")
            return
        elif self.__size == 1:
            self.__head = None
            self.__tail = None
        else:
            self.__head = self.__head.next
        self.__size -= 1
```

```
    def add_tail(self, e):
        new_value = DoublyNode(e)
        if self.get_tail() is not None:
            old_tail = self.__tail
            self.__tail = new_value
            self.__tail.prev = old_tail
            old_tail.next = self.__tail
            self.__size += 1
        else:
            self.add_head(e)

    def get_tail(self) -> DoublyNode:
        return self.__tail

    def remove_tail(self):
        if self.is_empty():
            print("Doubly Linked List is empty")
            return
        elif self.__size == 1:
            self.__head = None
            self.__tail = None
        else:
            self.__tail = self.__tail.prev
        self.__size -= 1

    def add_between_list(self, index, element):
        if index > self.__size or index < 0:
            print("Index out of bounds")
        elif index == self.__size:
            self.add_tail(element)
        elif index == 0:
            self.add_head(element)
        else:
            prev_node = self.get_node_at(index - 1)
            current_node = self.get_node_at(index)
            new_value = DoublyNode(element)
            prev_node.next = new_value
            new_value.next = current_node
            new_value.prev = prev_node
            current_node.prev = new_value
```

```
                self.__size += 1

    def __get_node_from_start(self, index):
        element_node = self.__head
        counter = 0
        while counter < index:
            element_node = element_node.next
            counter += 1
        return element_node

    def __get_node_from_end(self, index):
        element_node = self.__tail
        counter = self.__size
        while counter > index + 1:
            element_node = element_node.prev
            counter -= 1
        return element_node

    def get_node_at(self, index, direction="auto") -> DoublyNode:
        if index < 0 or index >= self.__size:
            print("Index out of bounds")
        elif direction == "start":
            return self.__get_node_from_start(index)
        elif direction == "end":
            return self.__get_node_from_end(index)
        elif self.__size == 1 or index <= self.__size // 2:
            return self.__get_node_from_start(index)
        elif index > self.__size // 2:
            return self.__get_node_from_end(index)

    def remove_between_list(self, index):
        if index < 0 or index >= self.__size:
            print("Index out of bounds")
        elif index == self.__size - 1:
            self.remove_tail()
        elif index == 0:
            self.remove_head()
        else:
            prev_node = self.get_node_at(index - 1)
            next_node = self.get_node_at(index + 1)
```

```
            next_node.prev = prev_node
            self.__size -= 1

    def search(self, search_value) -> bool:
        index = 0
        while index < self.__size:
            value = self.get_node_at(index)
            if value.element == search_value:
                print("Found value '" + str(search_value) + "' at location: " + str(index))
                return True
            index += 1
        print("Couldn't find value '" + str(search_value) + "'")
        return False

    def merge(self, linked_list_value):
        if not self.is_empty():
            last_node = self.get_tail()
            last_node.next = linked_list_value.__head
            if not linked_list_value.is_empty():
                linked_list_value.__head.prev = last_node
                self.__tail = linked_list_value.__tail
            self.__size = self.__size + linked_list_value.__size
        else:
            self.__head = linked_list_value.__head
            self.__tail = linked_list_value.__tail
            self.__size = linked_list_value.__size

    def reverse(self):
        if self.is_empty():
            print("Doubly Linked List is empty")
            return
        elif self.__size == 1:
            return
        elif self.__size == 2:
            temp = self.__head
            self.__head = self.__tail
            self.__tail = temp
            self.__head.prev = None
            self.__head.next = self.__tail
            self.__tail.prev = self.__head
```

```
            self.__tail.next = None
        elif self.__size == 3:
            mid = self.__get_node_from_start(1)
            temp = self.__head
            self.__head = self.__tail
            self.__tail = temp
            self.__head.prev = None
            self.__head.next = mid
            mid.prev = self.__head
            mid.next = self.__tail
            self.__tail.prev = mid
            self.__tail.next = None
        elif self.__size > 3:
            temp_lst = []
            first = self.__head
            temp_lst.append(first)
            first = first.next
            while first:
                temp_lst.append(first)
                first = first.next
            temp_lst.reverse()
            for i in range(0, len(temp_lst)):
                if i == 0:
                    self.__head = temp_lst[i]
                    self.__head.prev = None
                    self.__head.next = temp_lst[i + 1]
                elif i == 1:
                    temp_lst[i].prev = self.__head
                    temp_lst[i].next = temp_lst[i + 1]
                elif i == len(temp_lst) - 1:
                    self.__tail = temp_lst[i]
                    self.__tail.prev = temp_lst[i - 1]
                    self.__tail.next = None
                else:
                    temp_lst[i].prev = temp_lst[i - 1]
                    temp_lst[i].next = temp_lst[i + 1]
            temp_lst[len(temp_lst) - 2].prev = temp_lst[len(temp_lst) - 2 - 1]
            temp_lst[len(temp_lst) - 2].next = self.__tail

    @staticmethod
```

```
    def test_main():
        dll = DoublyLinkedList()
        print(dll.is_empty())
        dll.add_between_list(0, "zero")
        dll.add_between_list(1, "one")
        dll.add_between_list(2, "two")
        dll.display()
        print(dll.is_empty())
        print("After removing")
        dll.remove_between_list(1)
        dll.display()
        print("After adding again")
        dll.add_between_list(1, "one")
        dll.display()

        dll2 = DoublyLinkedList()
        dll2.add_head("first_name")
        dll2.add_tail("last_name")
        print("Head: ", end='')
        dll2.get_head().display()
        print("Tail: ", end='')
        dll2.get_tail().display()
        dll2.remove_tail()
        dll2.get_tail().display()
        dll2.remove_head()
        try:
            dll2.get_head().display()
        except AttributeError as e:
            print(e, "\nNo head found")
        dll2.display()

        dll3 = DoublyLinkedList()
        dll3.add_between_list(0, "zero")
        dll3.add_between_list(1, "one")
        dll3.add_between_list(2, "two")
        dll3.add_between_list(3, "three")
        dll3.display()
        dll3.search("two")
        dll3.search("four")
        dll3.get_node_at(3).display()
```

```
        dll3.get_node_at(3, "start").display()
        dll3.get_node_at(3, "end").display()

        dll4 = DoublyLinkedList()
        dll4.add_between_list(0, "0")
        dll4.add_between_list(1, "1")
        dll4.add_between_list(2, "2")
        dll4.add_between_list(3, "4")
        print("The Two Lists: ")
        dll3.display()
        print("Head: ", end='')
        dll3.get_head().display()
        print("Tail: ", end='')
        dll3.get_tail().display()
        dll4.display()
        print("Head: ", end='')
        dll4.get_head().display()
        print("Tail: ", end='')
        dll4.get_tail().display()
        dll3.merge(dll4)
        print("After Merging")
        dll3.display()
        print("Head: ", end='')
        dll3.get_head().display()
        print("Tail: ", end='')
        dll3.get_tail().display()
        print(f"Size: {dll3.get_size()}")
        dll3.display()
        dll3.reverse()
        print("After Reversing")
        dll3.display()
        print("Reverse Display")
        dll3.display("end")


if __name__ == "__main__":
    polynomial_addition_1 = DoublyLinkedList()
    order = int(input('Enter the order of the polynomials : '))
    print("Polynomial 1:")
    polynomial_addition_1.add_head(int(input(f"Enter coefficient of power {order} : ")))
```

```
    for i in reversed(range(order)):
        polynomial_addition_1.add_tail(int(input(f"Enter coefficient of power {i} : ")))
    polynomial_addition_2 = DoublyLinkedList()
    print("Polynomial 2:")
    polynomial_addition_2.add_head(int(input(f"Enter coefficient of power {order} : ")))
    for i in reversed(range(order)):
        polynomial_addition_2.add_tail(int(input(f"Enter coefficient of power {i} : ")))
    superscript_map = {"0": "⁰", "1": "¹", "2": "²", "3": "³", "4": "⁴", "5": "⁵", "6": "⁶",
                       "7": "⁷", "8": "⁸", "9": "⁹"}
    trans = str.maketrans(''.join(superscript_map.keys()), ''.join(superscript_map.values()))
    addend_1 = []
    addend_2 = []
    addend_str_1 = ""
    addend_str_2 = ""
    result = []
    result_str = ""
    for i in reversed(range(order + 1)):
        addend_1.append(polynomial_addition_1.get_node_at(i).element)
        addend_2.append(polynomial_addition_2.get_node_at(i).element)
        result.append(polynomial_addition_1.get_node_at(i).element + polynomial_addition_2.get_node_at(i).element)
    for ele in reversed(range(order + 1)):
        addend_str_1 = addend_str_1 + "%+d" % (addend_1[ele]) + "x" + (str(ele).translate(trans))
        addend_str_2 = addend_str_2 + "%+d" % (addend_2[ele]) + "x" + (str(ele).translate(trans))
        result_str = result_str + "%+d" % (result[ele]) + "x" + (str(ele).translate(trans))

    addend_str_1 = addend_str_1.replace("¹", "")
    addend_str_1 = addend_str_1.lstrip("+")
    addend_str_1 = addend_str_1.rstrip("x⁰")

    sign = addend_str_2[0]
    addend_str_2 = addend_str_2.replace("¹", "")
    addend_str_2 = addend_str_2.lstrip("+")
    addend_str_2 = addend_str_2.lstrip("-")
    addend_str_2 = addend_str_2.rstrip("x⁰")

    result_str = result_str.replace("¹", "")
    result_str = result_str.lstrip("+")
    result_str = result_str.rstrip("x⁰")

    print(addend_str_1+" "+sign+" "+addend_str_2+" = "+result_str)
```

Output:

```
================= RESTART: D:\studies\data structures\prac3c.py =======
Enter the order of the polynomials : 5
Polynomial 1:
Enter coefficient of power 5 : 4
Enter coefficient of power 4 : 3
Enter coefficient of power 3 : 2
Enter coefficient of power 2 : 1
Enter coefficient of power 1 : 3
Enter coefficient of power 0 : 1
Polynomial 2:
Enter coefficient of power 5 : 5
Enter coefficient of power 4 : 4
Enter coefficient of power 3 : 3
Enter coefficient of power 2 : 2
Enter coefficient of power 1 : 1
Enter coefficient of power 0 : 0
4x⁵+3x⁴+2x³+1x²+3x+1 + 5x⁵+4x⁴+3x³+2x²+1x+0 = 9x⁵+7x⁴+5x³+3x²+4x+1
>>>
```

Practical 3. Implement the following for Stack:

D.

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

Theory

Factorial:

The factorial of a number is the product of all the integers from 1 to that number. For example, the factorial of 6 (denoted as 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.

You can find it using recursion as well as iteration to calculate the factorial of a number.

Factorial:

Factors are the numbers you multiply to get another number. For instance, factors of 15 are 3 and 5, because 3×5 = 15. Some numbers have more than one factorization (more than one way of being factored). For instance, 12 can be factored as 1×12, 2×6, or 3×4. A number that can only be factored as 1 time itself is called "prime".

You can find it using recursion as well as iteration to calculate the factors of a number.

Code:

prac3d.py - D:\studies\data structures\prac3d.py (3.7.7)

File Edit Format Run Options Window Help

```
#3005 (New Rollno:372) Shravan SYIT
class Fact:
    def __init__(self, number):
        self.number = number

    def factorial_iteration(self):
        factorial = 1
        if self.number < 0:
            raise Exception("Factorial of Negative number doesn't exist")
        elif self.number == 0:
            return factorial
        else:
            for i in range(1, self.number + 1):
                factorial = factorial * i
            return factorial

    def factorial_recursion(self, number=None):
        if number is None:
            number = self.number
        if number < 0:
            raise Exception("Factorial of Negative number doesn't exist")
        elif number == 0 or number == 1:
            return 1
        else:
            return number * self.factorial_recursion(number - 1)

    def factors_iteration(self):
        factors = []
        for i in range(1, self.number + 1):
            if self.number % i == 0:
                factors.append(i)
        return factors

    def factors_recursion(self, number=None, lst=None):
        factorial = self.number
        if number is None and lst is None:
            number = self.number
            lst = []
        if number == 0 or number < 0:
            return lst
```

```
    def factors_recursion(self, number=None, lst=None):
        factorial = self.number
        if number is None and lst is None:
            number = self.number
            lst = []
        if number == 0 or number < 0:
            return lst
        elif number == 1:
            lst.append(1)
            return sorted(lst)
        elif factorial % number == 0:
            lst.append(number)
            return self.factors_recursion(number - 1, lst)
        else:
            return self.factors_recursion(number - 1, lst)


if __name__ == '__main__':
    fact = Fact(int(input("Enter a number: ")))
    print(fact.factorial_iteration())
    print(fact.factorial_recursion())
    print(fact.factors_iteration())
    print(fact.factors_recursion())
```

Output:

```
================= RESTART: D:\studies\data structures\prac3d.py ====
Enter a number: 5
120
120
[1, 5]
[1, 5]
>>>
```

Practical 4.
Aim: Perform Queues operations using Circular Array implementation.

Theory:
Queue
the queue abstract data type defines a collection that keeps objects in a sequence, where element access and deletion are restricted to the first element in the queue, and element insertion is restricted to the back of the sequence. This restriction enforces the rule that items are inserted and deleted in a queue according to the first-in, first-out (FIFO) principle. The queue abstract data type (ADT) supports the following two fundamental methods for a queue Q:
Q.enqueue(e): Add element e to the back of queue Q.
Q.dequeue( ): Remove and return the first element from queue Q;
an error occurs if the queue is empty.
For the stack ADT, we created a very simple adapter class that used a Python list as the underlying storage.
Double Ended Queue
We next consider a queue-like data structure that supports insertion and deletion at both the front and the back of the queue. Such a structure is called a double ended queue, or deque, which is usually pronounced “deck” to avoid confusion with the dequeue method of the regular queue ADT, which is pronounced like the abbreviation “D.Q.”
The deque abstract data type is more general than both the stack and the queue ADTs.

Code:

prac4.py - D:\studies\data structures\prac4.py (3.7.7)

File Edit Format Run Options Window Help

```
#3005 (New Rollno:372) Shravan SYIT
class ArrayQueue:
    default_capacity = 5

    def __init__(self):

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

    def len(self):

        return self._size

    def is_empty(self):

        return self._size == 0

    def first(self):

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

    def dequeue(self):

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

    def enqueue(self, e):
        if self._size == len(self._data):
            self._resize(2 * len(self._data))
        avail = (self._front + self._size) % len(self._data)
        self._data[avail] = e
        self._size += 1
```

```
    def _resize(self, cap):

        old = self._data
        self._data = [None] * cap
        walk = self._front
        for k in range(self._size):
            self._data[k] = old[walk]
            walk = (1 + walk) % len(old)
        self._front = 0


class DEQueue(ArrayQueue):
    def __init__(self):
        ArrayQueue.__init__(self)
        self._back = 0

    def last(self):
        if self.is_empty():
            raise Exception('Queue is empty')
        return self._data[self._back]

    def enqueue_front(self, e):
        self.enqueue(e)
        self._back = (self._front + self._size - 1) % len(self._data)

    def enqueue_back(self, e):
        if self._size == len(self._data):
            self._resize(2 * len(self._data))
        self._front = (self._front - 1) % len(self._data)
        self._data[self._front] = e
        self._size += 1
        self._back = (self._front + self._size - 1) % len(self._data)

    def dequeue_front(self):
        if self.is_empty():
            raise Exception('Queue is empty')
        back = (self._front + self._size - 1) % len(self._data)
        answer = self._data[back]
        self._data[back] = None
        self._size -= 1
```

```
        self._back = (self._front + self._size - 1) % len(self._data)
        return answer

    def dequeue_back(self):
        answer = self.dequeue()
        self._back = (self._front + self._size - 1) % len(self._data)
        return answer

    def _resize(self, cap):
        ArrayQueue._resize(self, cap)
        self._back = (self._front + self._size - 1) % len(self._data)

    def show_all(self):
        print(f"Size: {self._size}", end=', ')
        print(f"Front: {self._front}", end=', ')
        print(f"Back: {self._back}", end=', ')
        try:
            print("First: ", self.first(), end=', ')
            print("Last: ", self.last(), end=',\n')
        except:
            print("Queue is empty")
        print(f"Queue: {self._data}")


if __name__ == "__main__":
    deq = DEQueue()
    for i in range(1, 4):
        deq.enqueue_front("f_" + str(i))
        deq.show_all()
    for i in range(1, 4):
        deq.dequeue_front()
        deq.show_all()
    for i in range(1, 4):
        deq.enqueue_back("b_" + str(i))
        deq.show_all()
    for i in range(1, 4):
        deq.dequeue_back()
        deq.show_all()
    for i in range(1, 13):
        deq.enqueue_front("f_" + str(i))
```

```
if __name__ == "__main__":
    deq = DEQueue()
    for i in range(1, 4):
        deq.enqueue_front("f_" + str(i))
        deq.show_all()
    for i in range(1, 4):
        deq.dequeue_front()
        deq.show_all()
    for i in range(1, 4):
        deq.enqueue_back("b_" + str(i))
        deq.show_all()
    for i in range(1, 4):
        deq.dequeue_back()
        deq.show_all()
    for i in range(1, 13):
        deq.enqueue_front("f_" + str(i))
        deq.show_all()
    for i in range(1, 11):
        deq.enqueue_back("b_" + str(i))
        deq.show_all()
    for i in range(1, 13):
        deq.dequeue_front()
        deq.show_all()
    for i in range(1, 11):
        deq.dequeue_back()
        deq.show_all()
```

Output:

```
================= RESTART: D:\studies\data structures\prac4.py =================
Size: 1, Front: 0, Back: 0, First:  f_1, Last:  f_1,
Queue: ['f_1', None, None, None, None]
Size: 2, Front: 0, Back: 1, First:  f_1, Last:  f_2,
Queue: ['f_1', 'f_2', None, None, None]
Size: 3, Front: 0, Back: 2, First:  f_1, Last:  f_3,
Queue: ['f_1', 'f_2', 'f_3', None, None]
Size: 2, Front: 0, Back: 1, First:  f_1, Last:  f_2,
Queue: ['f_1', 'f_2', None, None, None]
Size: 1, Front: 0, Back: 0, First:  f_1, Last:  f_1,
Queue: ['f_1', None, None, None, None]
Size: 0, Front: 0, Back: 4, Queue is empty
Queue: [None, None, None, None, None]
Size: 1, Front: 4, Back: 4, First:  b_1, Last:  b_1,
Queue: [None, None, None, None, 'b_1']
Size: 2, Front: 3, Back: 4, First:  b_2, Last:  b_1,
Queue: [None, None, None, 'b_2', 'b_1']
Size: 3, Front: 2, Back: 4, First:  b_3, Last:  b_1,
Queue: [None, None, 'b_3', 'b_2', 'b_1']
Size: 2, Front: 3, Back: 4, First:  b_2, Last:  b_1,
Queue: [None, None, None, 'b_2', 'b_1']
Size: 1, Front: 4, Back: 4, First:  b_1, Last:  b_1,
Queue: [None, None, None, None, 'b_1']
Size: 0, Front: 0, Back: 4, Queue is empty
Queue: [None, None, None, None, None]
Size: 1, Front: 0, Back: 0, First:  f_1, Last:  f_1,
Queue: ['f_1', None, None, None, None]
Size: 2, Front: 0, Back: 1, First:  f_1, Last:  f_2,
Queue: ['f_1', 'f_2', None, None, None]
Size: 3, Front: 0, Back: 2, First:  f_1, Last:  f_3,
Queue: ['f_1', 'f_2', 'f_3', None, None]
Size: 4, Front: 0, Back: 3, First:  f_1, Last:  f_4,
Queue: ['f_1', 'f_2', 'f_3', 'f_4', None]
Size: 5, Front: 0, Back: 4, First:  f_1, Last:  f_5,
Queue: ['f_1', 'f_2', 'f_3', 'f_4', 'f_5']
Size: 6, Front: 0, Back: 5, First:  f_1, Last:  f_6,
Queue: ['f_1', 'f_2', 'f_3', 'f_4', 'f_5', 'f_6', None, None, None, None]
```

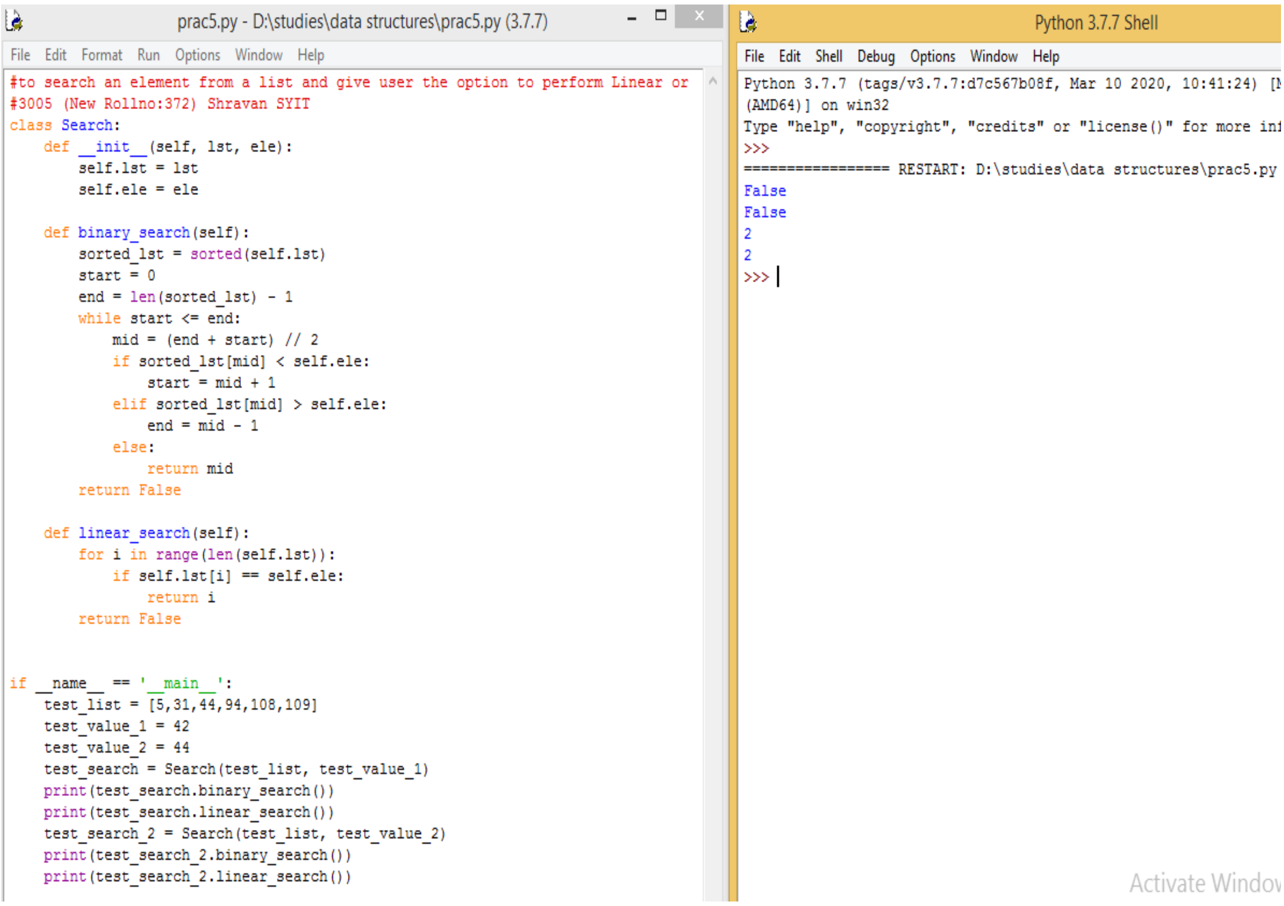
Practical 5.

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

Theory

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.
Linear Search: A Linear Search is the most basic type of searching algorithm. A Linear Search sequentially moves through your collection (or data structure) looking for a matching value. In other words, it looks down a list, one item at a time, without jumping.

Code and output:




```python
#to search an element from a list and give user the option to perform Linear or
#3005 (New Rollno:372) Shravan SYIT
class Search:
    def __init__(self, lst, ele):
        self.lst = lst
        self.ele = ele

    def binary_search(self):
        sorted_lst = sorted(self.lst)
        start = 0
        end = len(sorted_lst) - 1
        while start <= end:
            mid = (end + start) // 2
            if sorted_lst[mid] < self.ele:
                start = mid + 1
            elif sorted_lst[mid] > self.ele:
                end = mid - 1
            else:
                return mid
        return False

    def linear_search(self):
        for i in range(len(self.lst)):
            if self.lst[i] == self.ele:
                return i
        return False


if __name__ == '__main__':
    test_list = [5,31,44,94,108,109]
    test_value_1 = 42
    test_value_2 = 44
    test_search = Search(test_list, test_value_1)
    print(test_search.binary_search())
    print(test_search.linear_search())
    test_search_2 = Search(test_list, test_value_2)
    print(test_search_2.binary_search())
    print(test_search_2.linear_search())
```

```
Python 3.7.7 (tags/v3.7.7:d7c567b08f, Mar 10 2020, 10:41:24) [M
(AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more inf
>>>
================ RESTART: D:\studies\data structures\prac5.py
False
False
2
2
>>>
```

Practical 6.

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

Theory

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(n2) where n is the number of items.

Insertion Sort : This is an in-place comparison-based sorting algorithm. Here, a sub-list is maintained which is always sorted. For example, the lower part of an array is maintained to be sorted. An element which is to be 'insert'ed in this sorted sub-list, has to find its appropriate place and then it has to be inserted there. Hence the name, insertion sort.

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.

Code:

prac6.py - D:\studies\data structures\prac6.py (3.7.7)

File Edit Format Run Options Window Help

```python
#to sort a list of elements. Give user the option to perform sorting using Insertion sort, Bubble sort or Selection sort.
#3005 (New Rollno:372) Shravan SYIT
class Sort:
    def __init__(self, lst):
        self.lst = lst

    def bubble_sort(self):
        sorted_lst = self.lst.copy()
        for i in range(len(sorted_lst) - 1):
            for j in range(0, len(sorted_lst) - i - 1):
                if sorted_lst[j] > sorted_lst[j + 1]:
                    sorted_lst[j], sorted_lst[j + 1] = sorted_lst[j + 1], sorted_lst[j]
        return sorted_lst

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

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


if __name__ == '__main__':
    test_list = [44,108,31,109,94]
    test_sort = Sort(test_list)
    print(test_sort.bubble_sort())
```

```python
if __name__ == '__main__':
    test_list = [44,108,31,109,94]
    test_sort = Sort(test_list)
    print(test_sort.bubble_sort())
    print(test_sort.selection_sort())
    print(test_sort.insertion_sort())
```

Output:

```
================= RESTART: D:\studies\data structures\prac6.py ===
[31, 44, 94, 108, 109]
[31, 44, 94, 108, 109]
[31, 44, 94, 108, 109]
>>> 
```

Practical 7. Implement the following for Hashing:

A.

Aim: Write a program to implement the collision technique.

Theory:

Hash Table is a data structure which stores data in an associative manner. In a hash table, data is stored in an array format, where each data value has its own unique index value. Access of data becomes very fast if we know the index of the desired data.

Thus, it becomes a data structure in which insertion and search operations are very fast irrespective of the size of the data. Hash Table uses an array as a storage medium and uses hash technique to generate an index where an element is to be inserted or is to be located from.

Hashing:

Hashing is a technique to convert a range of key values into a range of indexes of an array. We're going to use modulo operator to get a range of key values. Consider an example of hash table of size 20, and the following items are to be stored. Item are in the (key,value) format.

In computer science, a collision or clash is a situation that occurs when two distinct pieces of data have the same hash value, checksum, fingerprint, or cryptographic digest.[1]

Due to the possible applications of hash functions in data management and computer security (in particular, cryptographic hash functions), collision avoidance has become a fundamental topic in computer science.

Collisions are unavoidable whenever members of a very large set (such as all possible person names, or all possible computer files) are mapped to a relatively short bit string. This is merely an instance of the pigeonhole principle.

The impact of collisions depends on the application. When hash functions and fingerprints are used to identify similar data, such as homologous DNA sequences or similar audio files, the functions are designed so as to maximize the probability of collision between distinct but similar data, using techniques like locality-sensitive hashing. Checksums, on the other hand, are designed to minimize the probability of collisions between similar inputs, without regard for collisions between very different inputs.

Code and Output:

prac7a.py - D:\studies\data structures\prac7a.py (3.7.7)

File Edit Format Run Options Window Help

```
#3005 (New Rollno:372) Shravan SYIT
#Implement the following Hashing Techniques.
#Write a program to implement the collision technique.

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 = [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()
        if list_of_list_index[list_index]:
            print("Collision detected")
        else:
            list_of_list_index[list_index] = value

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

Python 3.7.7 Shell

File Edit Shell Debug Options Window Help

```
Python 3.7.7 (tags/v3.7.7:d7c567b08f, Mar 10 2020, 10:41:24) [MSC
(AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more infor
>>>
================= RESTART: D:\studies\data structures\prac7a.py =
Before : [None, None, None, None]
Collision detected
Collision detected
After: [None, 1, None, 23]
>>>
```

Practical 7. Implement the following for Hashing:

B.

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

Theory:

Linear probing is a scheme in computer programming for resolving collisions in hash tables, data structures for maintaining a collection of key–value pairs and looking up the value associated with a given key. It was invented in 1954 by Gene Amdahl, Elaine M. McGraw, and Arthur Samuel and first analyzed in 1963 by Donald Knuth.

Along with quadratic probing and double hashing, linear probing is a form of open addressing. In these schemes, each cell of a hash table stores a single key–value pair. When the hash function causes a collision by mapping a new key to a cell of the hash table that is already occupied by another key, linear probing searches the table for the closest following free location and inserts the new key there. Lookups are performed in the same way, by searching the table sequentially starting at the position given by the hash function, until finding a cell with a matching key or an empty cell.

Code:

prac7b.py - D:\studies\data structures\prac7b.py (3.7.7)

File Edit Format Run Options Window Help

```
#3005 (New Rollno:372) Shravan SYIT
#Implement the following for Hashing:
#Write a program to implement the concept of linear probing.
class Hash:
    def __init__(self, keys, lowerrange, higherrange):
        self.value = self.hashfunction(keys,lowerrange, higherrange)

    def get_key_value(self):
        return self.value

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

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

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

Output:

```
================= RESTART: D:\studies\data structures\prac7b.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 8.

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

Theory

Pre-order (NLR):
Access the data part of the current node.
Traverse the left subtree by recursively calling the pre-order function.
Traverse the right subtree by recursively calling the pre-order function.
The pre-order traversal is a topologically sorted one, because a parent node is processed before any of its child nodes is done.
In-order (LNR):
Traverse the left subtree by recursively calling the in-order function.
Access the data part of the current node.
Traverse the right subtree by recursively calling the in-order function.
In a binary search tree ordered such that in each node the key is greater than all keys in its left subtree and less than all keys in its right subtree, in-order traversal retrieves the keys in ascending sorted order.
Post-order (LRN):
Traverse the left subtree by recursively calling the post-order function.
Traverse the right subtree by recursively calling the post-order function.
Access the data part of the current node:
The trace of a traversal is called a sequentialisation of the tree. The traversal trace is a list of each visited root. No one sequentialisation according to pre-, in- or post-order describes the underlying tree uniquely. Given a tree with distinct elements, either pre-order or post-order paired with in-order is sufficient to describe the tree uniquely. However, pre-order with post-order leaves some ambiguity in the tree structure.

Code:

prac8.py - D:\studies\data structures\prac8.py (3.7.7)

File Edit Format Run Options Window Help

```
#3005 (New Rollno:372) Shravan SYIT
#Write a program for inorder, postorder and preorder traversal of tree.

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)
            elif data > self.value:
                if self.right is None:
                    self.right = Node(data)
                else:
                    self.right.insert(data)
        else:
            self.value = data


if __name__ == '__main__':
    root = Node(10)
    root.left = Node(12)
    root.right = Node(5)
    print("Without any order")
    root.PrintTree()
    root_1 = Node(None)
    root_1.insert(28)
    root_1.insert(4)
    root_1.insert(13)
    root_1.insert(130)
    root_1.insert(123)
    print("Now ordering with insert")
    root_1.PrintTree()
    print("Pre order")
    root_1.Printpreorder()
    print("In Order")
    root_1.Printinorder()
    print("Post Order")
    root_1.Printpostorder()
```

Output:

```
================= RESTART: D:\studies\data structures\prac8.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
>>>
```