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./___ROHAN YADAV ____

Roll No: _346______ 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: __346_________**

Subject: Data Structures

INDEX

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

Practical 1(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:

```
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]
Armod = ArrayModification()
print(Armod.linear_search(lst,3))
```

```
print(Arrmod.merge(lst,lst2))
print(Arrmod.insertion_sort(lst))
print(Arrmod.reverse(lst))
```

Github link practical 1(a)

https://github.com/rohanyadav75/Ds-practical-/blob/master/Prac%201(a).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:

1. Input the order of the matrix.

2. Input the matrix 1 elements.

3. Input the matrix 2 elements.

4. Repeat from I = 0 to m

5. Repeat from j = 0 to n

6. Mat3[i][j] = mat1[i][j] + mat2[i][j]

7. Print mat3.

Matrix multiplication:

1. Input the order of the matrix1 (m * n).

2. Input the order of matrix2 (p * q).

3. Input the matrix 1 elements.

4. Input the matrix 2 elements.

5. Repeat from I = 0 to m

6. Repeat from j = 0 to q

7. Repeat from k = 0 to p

8. Sum=sum+ mat1[c][k] * mat2[k][d];

9. Mat3[c][d]=sum

10. 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.

```
Mat1
=
[[3,
4, -
6],
            [12,71,24],
            [21,3,21]]
```

```
Mat2 = [[2, 16, -16],
           [1,7,-3],
           [-1,3,3]]
Mat3 = [[0,0,0,],
     [0,0,0,],
     [0,0,0,]]

# Matrix Addition
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)

# Matrix Multiplication

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)
```

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

Github link practical 1(b)

https://github.com/rohanyadav75/Ds-practical-/blob/master/Prac%201(b)%20.py

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:

```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(list1.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")
            return None
        last = list1.get_tail()
        print(last.element)
        while last.previous:
            if type(last.previous.element) ==
type(list1.head):
                print(last.previous.element.element)
                if last.previous == self.head:
                    return None
                else:
```

```
                last = last.previous
            print(last.previous.element)
            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



    else:
        print("Size not sufficient")

    return None




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

```
            print("Empty Singly linked list")
        elif self.size == 1:
            self.head == None
            self.size -= 1
        else:
            Node = self.find_second_last_element()
            if Node:
                Node.next = None
                self.size -= 1

    def get_node_at(self,index):
        element_node = self.head
        counter = 0
        if index == 0:
            return element_node.element
        if index > self.size-1:
            print("Index out of bound")
            return None
        while(counter < index):
            element_node = element_node.next
            counter += 1
        return element_node

    def get_previous_node_at(self,index):
        if index == 0:
            print('No previous value')
            return None
        return list1.get_node_at(index).previous
```

```python
def remove_between_list(self,position):
    if position > self.size-1:
        print("Index out of bound")
    elif position == self.size-1:
        self.remove_tail()
    elif position == 0:
        self.remove_head()
    else:
        prev_node = self.get_node_at(position-1)
        next_node = self.get_node_at(position+1)
        prev_node.next = next_node
        next_node.previous = prev_node
        self.size -= 1

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

```
            element_node.next = current_node
            current_node.previous = element_node
            self.size += 1

    def search (self,search_value):
        index = 0
        while (index < self.size):
            value = self.get_node_at(index)
            if type(value.element) == type(list1.head):
                print("Searching at " + str(index) + " and value is " + str(value.element.element))
            else:
                print("Searching at " + str(index) + " and value is " + str(value.element))
            if value.element == search_value:
                print("Found value at " + str(index) + " location")
                return True
            index += 1
        print("Not Found")
        return False

    def merge(self,linkedlist_value):
        if self.size > 0:
            last_node = self.get_node_at(self.size-1)
            last_node.next = linkedlist_value.head
            linkedlist_value.head.previous = last_node
            self.size = self.size + linkedlist_value.size
```

```
        else:
            self.head = linkedlist_value.head
            self.size = linkedlist_value.size

l1 = Node('element 1')
list1 = LinkedList()
list1.add_head(l1)
list1.add_tail('element 2')
list1.add_tail('element 3')
list1.add_tail('element 4')
list1.get_head().element.element
list1.add_between_list(2,'element between')
list1.remove_between_list(2)

list2 = LinkedList()
l2 = Node('element 5')
list2.add_head(l2)
list2.add_tail('element 6')
list2.add_tail('element 7')
list2.add_tail('element 8')
list1.merge(list2)
list1.get_previous_node_at(3).element
list1.reverse_display()
list1.search('element 6')
```

Github link practical 2 :

https://github.com/rohanyadav75/Ds-practical-/blob/master/Prac%202.py

Practical no 3(a):

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.

1. Increment the variable Top so that it can now refere to the next

Memory location.

2. 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:
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()
```

Github link 3 (a)

https://github.com/rohanyadav75/Ds-practical-/blob/master/Prac%203%20(a).%20Py

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:

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.

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

(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.

CODE:

```
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]
```

```python
    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 towerOfHanoi(n, fromrod,to,temp):
    if n == 1:
        fromrod.pop()
        to.push('disk 1')
        if to.display() != None:
            print(to.display())

    else:

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

towerOfHanoi(n, A, C, B)

Github link 3 (b)
```

https://github.com/rohanyadav75/Ds-practical-/blob/master/Prac%203(b).py

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 :

```
class Node:

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

```python
        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")
            return None
        last = my_list.get_tail()
        print(last.element)
        while last.previous:
            if type(last.previous.element) ==
type(my_list.head):
                print(last.previous.element.element)
                if last.previous == self.head:
                    return None
                else:
                    last = last.previous
            print(last.previous.element)
            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
```

```python
        if self.size >= 2:
            first = self.head
            temp_counter = self.size -2
            while temp_counter > 0:
                first = first.next
                temp_counter -= 1
            return first


        else:
            print("Size not sufficient")

        return None



    def remove_tail(self):
        if self.is_empty():
            print("Empty Singly linked list")
        elif self.size == 1:
            self.head == None
            self.size -= 1
        else:
            Node = self.find_second_last_element()
            if Node:
                Node.next = None
```

```
            self.size -= 1

    def get_node_at(self,index):
        element_node = self.head
        counter = 0
        if index == 0:
            return element_node.element
        if index > self.size-1:
            print("Index out of bound")
            return None
        while(counter < index):
            element_node = element_node.next
            counter += 1
        return element_node

    def get_previous_node_at(self,index):
        if index == 0:
            print('No previous value')
            return None
        return my_list.get_node_at(index).previous

    def remove_between_list(self,position):
        if position > self.size-1:
            print("Index out of bound")
        elif position == self.size-1:
            self.remove_tail()
        elif position == 0:
            self.remove_head()
```

```
        else:
            prev_node = self.get_node_at(position-1)
            next_node = self.get_node_at(position+1)
            prev_node.next = next_node
            next_node.previous = prev_node
            self.size -= 1

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

    def search (self,search_value):
        index = 0
        while (index < self.size):
            value = self.get_node_at(index)
```

```
            if value.element == search_value:
                return value.element
            index += 1
        print("Not Found")
        return False

    def merge(self,linkedlist_value):
        if self.size > 0:
            last_node = self.get_node_at(self.size-1)
            last_node.next = linkedlist_value.head
            linkedlist_value.head.previous = last_node
            self.size = self.size + linkedlist_value.size

        else:
            self.head = linkedlist_value.head
            self.size = linkedlist_value.size




my_list = LinkedList()
order = int(input('Enter the order for polynomial : '))
my_list.add_head(Node(int(input(f"Enter coefficient for power
{order} : "))))
for i in reversed(range(order)):
    my_list.add_tail(int(input(f"Enter coefficient for power
{i} : ")))

my_list2 = LinkedList()
```

```
my_list2.add_head(Node(int(input(f"Enter coefficient for power
{order} : "))))
for i in reversed(range(order)):
    my_list2.add_tail(int(input(f"Enter coefficient for power
{i} : ")))


for i in range(order + 1):
    print(my_list.get_node_at(i).element +
my_list2.get_node_at(i).element)
```

Github link 3 (c)

https://github.com/rohanyadav75/Ds-practical-/blob/master/Prac%203(c).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:

```
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)
```

GITHUB LINK PRACTICAL 3 (D)

https://github.com/rohanyadav75/Ds-practical-/blob/master/Prac%203(d).py

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

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 :

```
Class ArrayQueue:
   """FIFO queue implementation using a Python list as underlying storage."""
   DEFAULT_CAPACITY = 10         # moderate capacity for all new queues

   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        # help garbage collection
    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.
    """
    If self.is_empty():
        Raise Empty('Queue is empty')
    Back = (self._front + self._size – 1) % len(self._data)
```

```
    Answer = self._data[back]
    Self._data[back] = None         # help garbage collection
    Self._front = self._front
    Self._size -= 1
    Self._back = (self._front + self._size – 1) % len(self._data)
    Return answer

Def enqueueEnd(self, e):
    “””Add an element to the back of queue.”””
    If self._size == len(self._data):
        Self._resize(2 * len(self.data))     # double the array size
    Avail = (self._front + self._size) % len(self._data)
    Self._data[avail] = e
    Self._size += 1
    Self._back = (self._front + self._size – 1) % len(self._data)

Def enqueueStart(self, e):
    “””Add an element to the start of queue.”””
    If self._size == len(self._data):
        Self._resize(2 * len(self._data))     # double the array size
    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):                 # we assume cap >= len(self)
```

```
        """Resize to a new list of capacity >= len(self)."""
        Old = self._data                     # keep track of existing list
        Self._data = [None] * cap            # allocate list with new capacity
        Walk = self._front
        For k in range(self._size):          # only consider existing elements
           Self._data[k] = old[walk]         # intentionally shift indices
           Walk = (1 + walk) % len(old)      # use old size as modulus
        Self._front = 0                      # front has been realigned
        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]}")

Github link practical 4

https://github.com/rohanyadav75/Ds-practical-/blob/master/Prac%204.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:

```
def
linear_search(lst,n):
    for i in range(len(lst)):
        if lst[i] == n:
            return print('Position:',i)
    return print("Number not found")

def binary_search(lst,n,start,end):
    if start <= end:
        mid = (end + start) // 2
        if lst[mid] == n:
            return print('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 print("Number not found")
```

```
def run():
    while True:
        print("Press 1 for linear search")
        print("Press 2 for binary search")
        print("Press 3 to exit")
        c = int(input())
        if c == 1:
            n = int(input("Enter number to search:"))
            linear_search(lst,n)
            break
        elif c == 2:
            s_lst = sorted(lst)
            n = int(input("Enter number to search:"))

    binary_search(s_lst,n,0,len(s_lst)-1)
            break
        else:
            break



lst = [26,74,12,3,48,2,37,15]
run()
```

GITHUB LINK PRACTICAL (5)

https://github.com/rohanyadav75/Ds-practical-/blob/master/Prac%205.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(n2

) 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(n2

), where n is the number of items.

CODE:

```
Def bubble_sort(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]
      Return lst


Def insertion_sort(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 selection_sort(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 run():
    While True:
        Print("Press 1 for bubble sort")
        Print("Press 2 for insertion sort")
        Print("Press 3 for selection sort")
        Print("Press 4 to exit")
        Print("List:",lst)
        C = int(input())
        If c == 1:
            Print("Sorted list",bubble_sort(lst))
        Elif c == 2:
            Print("Sorted list",insertion_sort(lst))
        Elif c == 3:
            Print("Sorted list",selection_sort(lst))
        Else:
            Break

Lst = [26,74,12,3,48,2,37,15]
Run()
```

Github link practical 6:

https://github.com/rohanyadav75/Ds-practical-/blob/master/Prac%206.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:

```
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))
```

Github link practical 7(a)

https://github.com/rohanyadav75/Ds-practical-/blob/master/Prac%207(a).py

Practical No 7(B)

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

Theory:

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.

CODE:

```
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
   List_full = False
   Old_list_index=list_index
   If list_index == size_list – 1:
      List_index = 0
   Else:
      List_index += 1


   While main_list[list_index]:
      If list_index+1 == size_list:
         List_index = 0
      Else:
         List_index += 1
      If list_index == old_list_index:
```

```
            List_full = True
            Break
    If list_full == True:
         Print("list was full. Could not saved")
```

```
Def search_list(key,main_list):
    #for I in range(size_list):
    Val = hash_function(key)
    If main_list[val] == key:
        Print("list found",val)
    Else:
        Print("not found")
```

```
List_values = [1,3,8,6,5,14]
```

```
Main_list = [None for x in range(size_list)]
Print(main_list)
Create_hash_table(list_values,main_list)
Print(main_list)
Search_list(5,main_list)
```

GITHUB LINK PRACTICAL 7(B)

https://github.com/rohanyadav75/Ds-practical-/blob/master/Prac%207(b).py

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)

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

2. Visit the root.

3. 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)

1. Visit the root.

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

3. 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

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

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

3. 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:

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

```python
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)
```

```python
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()
```

GITHUB LINK PRACTICAL 8

https://github.com/rohanyadav75/Ds-practical-/blob/master/Prac%208.py