



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. Pravin Dharamshi Manodra.

Roll No: 321

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)

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.	

Git repository link: [DS Practical](#)

Practical -1

Aim:

- a. Write a program to store the elements in 1-D array and provide an option to perform the operations like searching, sorting, merging, and reversing the elements.**

Theory:

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 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:

```
practical1a.py - C:\Users\Admin\Desktop\ds\practical1a.py (3.8.1)
File Edit Format Run Options Window Help

arr = [45,3,73,9,23,10,35,22]

def linear_search(arr,n):
    for i in range(len(arr)):
        if arr[i] == n:
            return i
    return -1

print(linear_search(arr,9))

def bubble_sort(arr):
    len_arr=len(arr)
    for i in range(len_arr):
        for j in range(0, len_arr-i-1):
            if arr[j] > arr[j+1]:
                arr[j],arr[j+1] = arr[j+1],arr[j]
    return arr

print(bubble_sort(arr))

def merge_arr(arr,element):
    new_arr =arr.append(element)
    return arr
print(merge_arr(arr,14))

def reverse(arr):
    new_arr=arr.reverse()
    return arr
print (reverse(arr))
```

Output:

```
Python 3.8.1 Shell
File Edit Shell Debug Options Window Help
Python 3.8.1 (tags/v3.8.1:1b293b6, Dec 18 2019, 22:39:24)
Type "help", "copyright", "credits" or "license()" for m
>>>
===== RESTART: C:\Users\Admin\Desktop\ds\pract
3
[3, 9, 10, 22, 23, 35, 45, 73]
[3, 9, 10, 22, 23, 35, 45, 73, 14]
[14, 73, 45, 35, 23, 22, 10, 9, 3]
>>> |
```

b.

Aim: 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 $A=[a_{ij}]$ be a matrix of order $m \times n$, 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 A^T .

Code:

```
practical1b.py - C:\Users\Admin\Desktop\ds\practical1b.py (3.8.1)
File Edit Format Run Options Window Help

X = [[12,7,3],
     [4,5,6],
     [7,8,9]]

Y = [[5,8,1],
     [6,7,3],
     [4,5,9]]

result = [[0,0,0],
          [0,0,0],
          [0,0,0]]

def addition(X,Y):
    print("Addition")
    for i in range(len(X)):
        for j in range(len(X[0])):
            result[i][j] = X[i][j] + Y[i][j]

    for r in result:
        print(r)

addition(X,Y)

def multiplication(X,Y):
    print("Multiplication ")
    for i in range(len(X)):
        for j in range(len(Y[0])):
            for k in range(len(Y)):
                result[i][j] += X[i][k] * Y[k][j]

    for r in result:
        print(r)

multiplication(X,Y)
```

```
practical1b.py - C:\Users\Admin\Desktop\ds\practical1b.py (3.8.1)
File Edit Format Run Options Window Help

    print("Addition")
    for i in range(len(X)):
        for j in range(len(X[0])):
            result[i][j] = X[i][j] + Y[i][j]

    for r in result:
        print(r)

addition(X,Y)

def multiplication(X,Y):
    print("Multiplication ")
    for i in range(len(X)):
        for j in range(len(Y[0])):
            for k in range(len(Y)):
                result[i][j] += X[i][k] * Y[k][j]

    for r in result:
        print(r)

multiplication(X,Y)

def transpose(X):
    for i in range(len(X)):
        for j in range(len(X[0])):
            result[j][i] = X[i][j]

    for r in result:
        print(r)

print("Transpose of X")
transpose(X)
print("Transpose of Y")
transpose(Y)
```

Output:

```
Python 3.8.1 Shell
File Edit Shell Debug Options Window Help
Python 3.8.1 (tags/v3.8.1:1b293b6, Dec 18 2019, 22:39:24) [MSC v.1916 32 bit
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:\Users\Admin\Desktop\ds\practical1b.py =====
Addition
[17, 15, 4]
[10, 12, 9]
[11, 13, 18]
Multiplication
[131, 175, 64]
[84, 109, 82]
[130, 170, 130]
Transpose of X
[12, 4, 7]
[7, 5, 8]
[3, 6, 9]
Transpose of Y
[5, 6, 4]
[8, 7, 5]
[1, 3, 9]
>>>
```

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

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 j th 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:

practical2.py - C:\Users\Admin\Desktop\ds\practical2.py (3.8.1)

File Edit Format Run Options Window Help

```
class Node:

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

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

class DlinkedList:

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

    def __len__(self):
        return self.size

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

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

    def add_head(self, e):
        temp = self.head
        self.head = Node(e)
        self.head.next = temp
```

practical2.py - C:\Users\Admin\Desktop\ds\practical2.py (3.8.1)

File Edit Format Run Options Window Help

```
        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 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 find_second_last_element(self):
    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 get_node_at(self, index):
    element_node = self.head
    counter = 0
    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_prev_node_at(self, position):
    if position == 0:
        print('No previous element')
        return None
    return self.get_node_at(position).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)
        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

#List 1
print('List 1')
list1 = DlinkedList()
list1.add_head(2)
list1.add_head(1)
list1.add_tail(3)
list1.add_tail(4)
list1.add_tail(5)
list1.add_tail(7)
list1.display()
list1.remove_head()
list1.remove_tail()
print('Head and Tail elements removed')
list1.display()

print('Added element between the list')
list1.add_between_list(2,6)
list1.display()

print('Removed element from between the list')
list1.remove_between_list(2)

```

```

list1 = DlinkedList()
list1.add_head(2)
list1.add_head(1)
list1.add_tail(3)
list1.add_tail(4)
list1.add_tail(5)
list1.add_tail(7)
list1.display()
list1.remove_head()
list1.remove_tail()
print('Head and Tail elements removed')
list1.display()

print('Added element between the list')
list1.add_between_list(2,6)
list1.display()

print('Removed element from between the list')
list1.remove_between_list(2)
list1.display()

#List 2
list2 = DlinkedList()
list2.add_head(7)
list2.add_head(6)
list2.add_tail(8)
list2.add_tail(9)
print('List 2')
list2.display()

#merging lists
list1.merge(list2)
print('List after merging')
list1.display()

```

Output:

```
Python 3.8.1 Shell
File Edit Shell Debug Options Window Help
===== RESTART: C:\Users\Admin\Desktop

List 1
1
2
3
4
5
7
Removing
Head and Tail elements removed
2
3
4
5
Added element between the list
2
3
6
4
5
Removed element from between the list
2
3
4
5
List 2
6
7
8
9

List after merging
2
3
4
5
6
7
8
9
>>>
```

Practical – 3

Implement the following for Stack:

a.

Aim: Perform Stack operations using Array implementation.

Theory:

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:

```
*practical3.py - C:\Users\Admin\Desktop\ds\practical3.py (3.8.1)*
File Edit Format Run Options Window Help
class Stack:
    def __init__(self):
        self.stack_arr = [2,13,3,22,43,15]

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

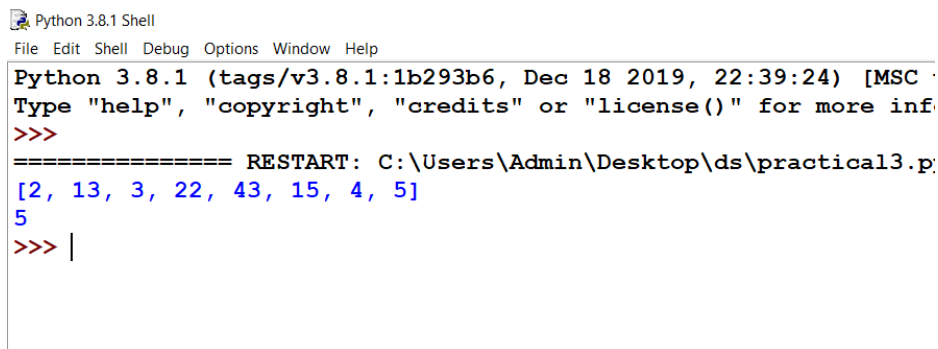
    def pop(self):
        if len(self.stack_arr) == 0:
            print('Stack is empty!')
            return None
        else:
            self.stack_arr.pop()

    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()
print(stack.get_head())
```

Output:



```
Python 3.8.1 Shell
File Edit Shell Debug Options Window Help
Python 3.8.1 (tags/v3.8.1:1b293b6, Dec 18 2019, 22:39:24) [MSC
Type "help", "copyright", "credits" or "license()" for more inf
>>>
===== RESTART: C:\Users\Admin\Desktop\ds\practical3.p
[2, 13, 3, 22, 43, 15, 4, 5]
5
>>> |
```

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 \rightarrow 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 (n-1) 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:

practical3b.py - C:\Users\Admin\Desktop\ds\practical3b.py (3.8.1)

File Edit Format Run Options Window Help

```
def hanoi(disks, source, auxiliary, target):
    if disks == 1:
        print('Move disk 1 from peg {} to peg {}'.format(source, target))
        return

    hanoi(disks - 1, source, target, auxiliary)
    print('Move disk {} from peg {} to peg {}'.format(disks, source, target))
    hanoi(disks - 1, auxiliary, source, target)

disks = int(input('Enter number of disks: '))
hanoi(disks, '1', '2', '3')
```

Output:

Python 3.8.1 Shell

File Edit Shell Debug Options Window Help

Python 3.8.1 (tags/v3.8.1:1b293b6, Dec 18 2019, 22:39:24) [MSC v.1

Type "help", "copyright", "credits" or "license()" for more inform

>>>

===== RESTART: C:\Users\Admin\Desktop\ds\practical3b.py :

Enter number of disks: 4

Move disk 1 from peg 1 to peg 2.

Move disk 2 from peg 1 to peg 3.

Move disk 1 from peg 2 to peg 3.

Move disk 3 from peg 1 to peg 2.

Move disk 1 from peg 3 to peg 1.

Move disk 2 from peg 3 to peg 2.

Move disk 1 from peg 1 to peg 2.

Move disk 4 from peg 1 to peg 3.

Move disk 1 from peg 2 to peg 3.

Move disk 2 from peg 2 to peg 1.

Move disk 1 from peg 3 to peg 1.

Move disk 3 from peg 2 to peg 3.

Move disk 1 from peg 1 to peg 2.

Move disk 2 from peg 1 to peg 3.

Move disk 1 from peg 2 to peg 3.

>>> |

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:

```
*practical3c.py - C:\Users\Admin\Desktop\ds\practical3c.py (3.8.1)*
File Edit Format Run Options Window Help

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

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

    def _len_(self):
        return self.size

    def get_head(self):
        return self.head

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

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

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

```



```

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)

```

Output:

```
Python 3.8.1 Shell
File Edit Shell Debug Options Window Help
Python 3.8.1 (tags/v3.8.1:1b293b6, Dec 18 2019, 22:01:04) [AMD64]
Type "help", "copyright", "credits" or "license()" >
>>>
===== RESTART: C:\Users\Admin\Desktop\ds\I
Enter the order for polynomial : 3
Enter coefficient for power 3 : 2
Enter coefficient for power 2 : 3
Enter coefficient for power 1 : 2
Enter coefficient for power 0 : 1
Enter coefficient for power 3 : 4
Enter coefficient for power 2 : 2
Enter coefficient for power 1 : 3
Enter coefficient for power 0 : 1
6
5
5
2
>>> |
```

d.

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

Theory:

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

Factors:

Factors are the numbers you multiply to get another number. For instance, factors of 15 are 3 and 5, because $3 \times 5 = 15$. 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:

practical3d.py - C:\Users\Admin\Desktop\ds\practical3d.py (3.8.1)

File Edit Format Run Options Window Help

```
# using recursion
def recur_factorial(n):
    if n == 1:
        return n
    else:
        return n*recur_factorial(n-1)

num =12

if num < 0:
    print("Number is negative")
elif num == 0:
    print("The factorial is 1")
else:
    print("The factorial of", num, "is", recur_factorial(num))

#using iteration
def print_factors(x):
    print("The factors of",x,"are:")
    for i in range(1, x + 1):
        if x % i == 0:
            print(i)

num = 12

print_factors(num)
```

Output:

Python 3.8.1 Shell

File Edit Shell Debug Options Window Help

```
Python 3.8.1 (tags/v3.8.1:1b293b6, Dec 18 2019, 22:3
Type "help", "copyright", "credits" or "license()" f
>>>
===== RESTART: C:\Users\Admin\Desktop\ds\p
The factorial of 12 is 479001600
The factors of 12 are:
1
2
3
4
6
12
>>> |
```

Practical – 4

Aim: Perform Queues operations using Circular Array implementation.

Theory:

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:

```
class CircularQueue():

    def __init__(self, size): # initializing the class
        self.size = size
        self.queue = [None for i in range(size)]
        self.front = self.rear = -1

    def enqueue(self, data):
        if ((self.rear + 1) % self.size == self.front):
            print(" Queue is Full\n")
        elif (self.front == -1):
            self.front = 0
            self.rear = 0
            self.queue[self.rear] = data
        else:
            self.rear = (self.rear + 1) % self.size
            self.queue[self.rear] = data

    def dequeue(self):
        if (self.front == -1):
            print ("Queue is Empty\n")
        elif (self.front == self.rear):
            temp=self.queue[self.front]
            self.front = -1
            self.rear = -1
            return temp
        else:
            temp = self.queue[self.front]
            self.front = (self.front + 1) % self.size
            return temp

    def display(self):
        if(self.front == -1):
            print ("Queue is Empty")
        elif (self.rear >= self.front):
```

```

        print("Elements in the circular queue are:",end = " ")
        for i in range(self.front, self.rear + 1):
            print(self.queue[i], end = " ")
        print ()
    else:
        print ("Elements in Circular Queue are:", end = " ")
        for i in range(self.front, self.size):
            print(self.queue[i], end = " ")
        for i in range(0, self.rear + 1):
            print(self.queue[i], end = " ")
        print ()
    if ((self.rear + 1) % self.size == self.front):
        print("Queue is Full")

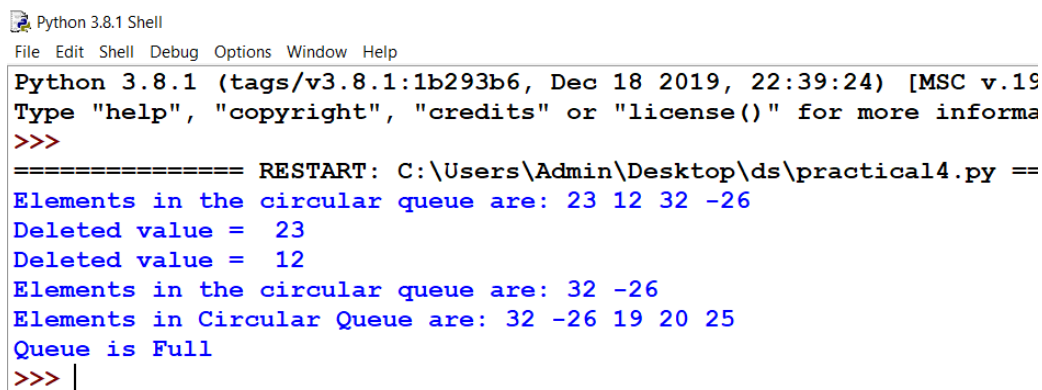
# Driver Code
ob = CircularQueue(5)
ob.enqueue(23)
ob.enqueue(12)
ob.enqueue(32)
ob.enqueue(-26)
ob.display()

print ("Deleted value = ", ob.dequeue())

print ("Deleted value = ", ob.dequeue())
ob.display()
ob.enqueue(19)
ob.enqueue(20)
ob.enqueue(25)
ob.display()

```

Output:



```

Python 3.8.1 (tags/v3.8.1:1b293b6, Dec 18 2019, 22:39:24) [MSC v.19
Type "help", "copyright", "credits" or "license()" for more informa
>>>
===== RESTART: C:\Users\Admin\Desktop\ds\practical4.py ==
Elements in the circular queue are: 23 12 32 -26
Deleted value = 23
Deleted value = 12
Elements in the circular queue are: 32 -26
Elements in Circular Queue are: 32 -26 19 20 25
Queue is Full
>>> |

```

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:

```
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()
```

Output:

```
Python 3.8.1 Shell
File Edit Shell Debug Options Window Help
Python 3.8.1 (tags/v3.8.1:1b293b6, Dec 18 2019, 22:39:24) [MSC v.1
Type "help", "copyright", "credits" or "license()" for more inform
>>>
===== RESTART: C:\Users\Admin\Desktop\ds\practical5.py =
Press 1 for linear search
Press 2 for binary search
Press 3 to exit
1
Enter number to search:5
Number not found
>>>
===== RESTART: C:\Users\Admin\Desktop\ds\practical5.py =
Press 1 for linear search
Press 2 for binary search
Press 3 to exit
2
Enter number to search:15
Position: 3
>>> |
```

Practical – 6

Aim: WAP to sort a list of elements. Give user the option to perform sorting using Insertion sort, 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 $O(n^2)$ 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 'inserted 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:

```
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:
```



```

        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()

```

Output:

```

Python 3.8.1 Shell
File Edit Shell Debug Options Window Help
Python 3.8.1 (tags/v3.8.1:1b293b6, Dec 18 2019, 22:39:24) [MSC v.1916 32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:\Users\Admin\Desktop\ds\practical6.py =====
Press 1 for bubble sort
Press 2 for insertion sort
Press 3 for selection sort
Press 4 to exit
List: [26, 74, 12, 3, 48, 2, 37, 15]
1
Sorted list [2, 3, 12, 15, 26, 37, 48, 74]
Press 1 for bubble sort
Press 2 for insertion sort
Press 3 for selection sort
Press 4 to exit
List: [2, 3, 12, 15, 26, 37, 48, 74]
2
Sorted list [2, 3, 12, 15, 26, 37, 48, 74]
Press 1 for bubble sort
Press 2 for insertion sort
Press 3 for selection sort
Press 4 to exit
List: [2, 3, 12, 15, 26, 37, 48, 74]
3
Sorted list [2, 3, 12, 15, 26, 37, 48, 74]
Press 1 for bubble sort
Press 2 for insertion sort
Press 3 for selection sort
Press 4 to exit
List: [2, 3, 12, 15, 26, 37, 48, 74]
4
>>> |

```

Practical – 7

Implement the following for Hashing:

a.

Aim: Write a program to implement the collision technique.

Theory:

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:

```
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))
```

Output:

```
Python 3.8.1 Shell
File Edit Shell Debug Options Window Help
Python 3.8.1 (tags/v3.8.1:1b293b6, Dec 18 2019, 22:39:24) [MSC v.1916 32 b
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:\Users\Admin\Desktop\ds\practical7a.py =====
Before : [None, None, None, None]
Collision detected
Collision detected
After: [None, 1, None, 23]
>>> |
```

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:

```

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:

```

Python 3.8.1 Shell
File Edit Shell Debug Options Window Help
Python 3.8.1 (tags/v3.8.1:1b293b6, Dec 18 2019, 22:...)
Type "help", "copyright", "credits" or "license()" :
>>>
===== RESTART: C:\Users\Admin\Desktop\ds\
Before : [None, None, None, None]
hash value for 23 is : 3
hash value for 43 is : 3
Collission detected for 43
hash value for 1 is : 1
hash value for 87 is : 3
Collission detected for 87
After: [43, 1, 87, 23]
>>> |

```

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 sub tree by recursively calling the pre-order function.
- Traverse the right sub tree 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 sub tree by recursively calling the in-order function.
- Access the data part of the current node.
- Traverse the right sub tree 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 sub tree and less than all keys in its right sub tree, in-order traversal retrieves the keys in ascending sorted order.

Post-order (LRN):

- Traverse the left sub tree by recursively calling the post-order function.
- Traverse the right sub tree 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:

```

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

# Insert Node
def insert(self, data):
    if self.val:
        if data < self.val:
            if self.left is None:
                self.left = Node(data)
            else:
                self.left.insert(data)
        elif data > self.val:
            if self.right is None:
                self.right = Node(data)
            else:
                self.right.insert(data)
        else:
            self.val = data

# Print the Tree
def PrintTree(self):
    if self.left:
        self.left.PrintTree()
    print( self.val),
    if self.right:
        self.right.PrintTree()

# Inorder tree traversal
def printInorder(root):
    if root:
        printInorder(root.left)
        print(root.val)

```

```

        printInorder(root.right)

# Postorder tree traversal
def printPostorder(root):
    if root:
        printPostorder(root.left)
        printPostorder(root.right)
        print(root.val)

# Preorder tree traversal
def printPreorder(root):
    if root:
        print(root.val)
        printPreorder(root.left)
        printPreorder(root.right)

# Driver code
root = Node(4)
root.insert(2)
root.insert(3)
root.insert(1)
root.insert(5)
root.insert(6)

print("Binary tree is")
root.PrintTree()
print("Preorder traversal of binary tree is")
printPreorder(root)
print("Inorder traversal of binary tree is")
printInorder(root)
print("Postorder traversal of binary tree is")
printPostorder(root)

```

Output:

Python 3.8.1 Shell

File Edit Shell Debug Options Window Help

```
Python 3.8.1 (tags/v3.8.1:1b293b6, Dec 18 2019, 22:39:24) |
Type "help", "copyright", "credits" or "license()" for more
>>>
```

```
===== RESTART: C:\Users\Admin\Desktop\ds\practica
```

```
Binary tree is
```

```
1
2
3
4
5
6
```

```
Preorder traversal of binary tree is
```

```
4
2
1
3
5
6
```

```
Inorder traversal of binary tree is
```

```
1
2
3
4
5
6
```

```
Postorder traversal of binary tree is
```

```
1
3
2
6
5
4
```

```
>>> |
```