* ***PRACTICAL -1 Array operation***

# THIS IS FOR ARRAY 1 PRACTICAL

import array as arr

arr1 = [5,10,20,8,40,55,12]

print(arr1)

# PRINT ELEMENT BY INDEXING NUMBER

print(arr1[4])

# LENGTH OF ARRAY

print(len(arr1))

# APPEND IS USE TO ADD THE ELEMENT AT THE END OF ARRAY

arr1.append(3)

print(arr1)

# EXTEND IS USE TO ADD

B=[16,9,7,24,40,15]

arr1.extend(B)

print(arr1)

# INSERT IS USE TO ADD AN ELEMENTS AT ANY POSITION OF ARRAY

arr1.insert(2,19)

print(arr1)

# SHORTING THE ELEMENT

arr1.sort()

print(arr1)

# POP IS USE TO DELETE THE ELEMENT BY INDEX NUMBER

# BY DEFAULT POP FUNCTION DELETES THE ELEMENT OF THE LAST INDEX NUMBER

arr1.pop(7)

print(arr1)

# REMOVE IS USE TO DELETE THE ELEMENT BY NAME OR BY VALUE

arr1.remove(12)

print(arr1)

c=[11, 20, 22,24]

# LOOPS IN ARRAY

b=c[0]

while b<= c[2]:

    print(b)

    b = b+1

* ***PRACTICAL -2 Array operation 2***
* # THIS IS FOR SECOND QUESTION
* def create\_array():
* try:
* # Get the length of the array from the user
* length = int(input("Enter the length of the array: "))
* if length <= 0:
* print("Please enter a positive integer for the length.")
* return
* # Initialize an empty array
* user\_array = []
* # Input elements from the user
* for i in range(length):
* element = input(f"Enter element {i + 1}: ")
* user\_array.append(element)
* # Display the created array
* print("\nArray created:")
* print(user\_array)
* except ValueError:
* print("Please enter a valid integer for the length.")
* # Example usage:
* if \_\_name\_\_ == "\_\_main\_\_":
* create\_array()
* ***PRACTICAL -3 Matrix 2×2***
* import numpy as np
* # Create a 2x2 matrix
* matrix\_a = np.array([[1, 2], [3, 4]])
* # (i) Data type
* print("Data type of the matrix:", matrix\_a.dtype)
* # (ii) Rank of the matrix
* print("Rank of the matrix:", np.linalg.matrix\_rank(matrix\_a))
* # (iii) Order of the matrix
* print("Order of the matrix:", matrix\_a.shape)
* # (iv) Convert the matrix into 1-D array
* flat\_array = matrix\_a.flatten()
* print("1-D array representation:", flat\_array)
* # Create another 2x2 matrix
* matrix\_b = np.array([[5, 6], [7, 8]])
* # (v) Perform addition, subtraction, and division operations
* matrix\_sum = matrix\_a + matrix\_b
* matrix\_diff = matrix\_a - matrix\_b
* matrix\_div = matrix\_a / matrix\_b
* print("\nMatrix Addition:")
* print(matrix\_sum)
* print("\nMatrix Subtraction:")
* print(matrix\_diff)
* print("\nMatrix Division:")
* print(matrix\_div)
* ***PRACTICAL -4 3-D Matrix operations***

import numpy as np

# Create a 3-D array

array\_3d = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])

# (i) Transpose of a matrix

transposed\_array = np.transpose(array\_3d)

print("Original 3-D array:")

print(array\_3d)

print("\nTransposed 3-D array:")

print(transposed\_array)

# (ii) Create another 3-D array for matrix multiplication

array\_3d\_multiply = np.array([[[2, 0], [1, 1]], [[1, 2], [0, 1]]])

# Perform matrix multiplication

result\_matrix\_multiply = np.matmul(array\_3d, array\_3d\_multiply)

print("\nMatrix for Multiplication:")

print(array\_3d\_multiply)

print("\nResult of Matrix Multiplication:")

print(result\_matrix\_multiply)

* ***PRACTICAL -5 Singly linked list operations***

#    Singly Linked list Q5

class Node:

    def \_\_init\_\_(self,data):

        self.data = data

        self.ref=None

class Linkedlist:

    def \_\_init\_\_(self):

        self.head = None

           # DISPLAY THE TOP OF THE NODE

    def display(self):

        if self.head is None:

            print("linked list is empty")

        else:

            n=self.head

            while n is not None:

                print(n.data,"==>",end=" ")

                n = n.ref

# FUNCTION FOR INSERTING A NEW VALUE OR NEW NODE AT FIRST POSITION

    def add\_begin(self, data):

        new\_node= Node(data)

        new\_node.ref = self.head    # ADD NEW NODE AS HEAD

        self.head = new\_node          # MOVE THE HEAD TO POINT TO THE NEW NODE

# ADD AT THE END

    def add\_end(self,data):

        new\_node= Node(data)

        if self.head is None:

            self.head=new\_node

        else:

            n=self.head

            while n.ref is not None:

                n= n.ref

            n.ref=new\_node

    #ADD AT ANY POSITION

    def add\_after(self, data,x):

        n = self.head

        while n is not None:

            if x ==n.data:

                break

            n=n.ref

        if n is None:

            print("node is not present in LL")

        else:

            new\_node=Node(data)

            new\_node.ref=n.ref

            n.ref=new\_node

    #delete ar begin

    def delete\_begin(self):

        if self.head is None:

            print("LL is empty so we can't delete nodes! ")

        else:

            self.head=self.head.ref

LL1=Linkedlist()

LL1.add\_begin(50)

LL1.add\_begin(30)

LL1.add\_begin(10)

LL1.add\_begin(45)

LL1.add\_end(140)

LL1.add\_after(400,140)

LL1.delete\_begin()

LL1.display()

* ***PRACTICAL -6 Doubly linked list operations***

# This is for Doubly linked list Q6

class Node:

    def \_\_init\_\_(self,data):

        self.data= data

        self.nref=None

        self.pref=None

class doublyLL:

    def \_\_init\_\_(self):

        self.head=None

    def display(self):

        if self.head is None:

            print("linked list1 is empty")

        else:

            n=self.head

            while n is not None:

                print(n.data,"==>",end=" ")

                n = n.ref

    def display\_reverse(self):

        if self.head is None:

            print("linked list2 is empty")

        else:

            n=self.head

            while n.nref is not None:

                n = n.nref

            while n is not None:

                print(n.data,"==>",end=" ")

                n=n.pref

    def insert\_empty(self,data):

        if self.head is None:

            new\_node=Node(data)

            self.head =new\_node

        else:

            print("linked is not empty")

    def add\_begin(self,data):

        new\_node=Node(data)

        if self.head is None:

            self.head = new\_node

        else:

            new\_node.nref=self.head

            self.head.pref=new\_node

            self.head=new\_node

    def add\_end(self,data):

        new\_node=Node(data)

        if self.head is None:

            self.head = new\_node

        else:

            n= self.head

            while n.nref is not None:

                n=n.nref

            n.nref=new\_node

            new\_node.pref=n

dl1=doublyLL()

dl1.display()

dl1.insert\_empty(10)

dl1.display\_reverse()

* ***PRACTICAL -7 Circularly linked list operations***
* class Node:
* def \_\_init\_\_(self, data):
* self.data = data
* self.next = None
* class CircularLinkedList:
* def \_\_init\_\_(self):
* self.head = None
* # (i) Traversing
* def traverse(self):
* if not self.head:
* print("Circular Linked List is empty.")
* return
* current = self.head
* while True:
* print(current.data, end=" -> ")
* current = current.next
* if current == self.head:
* break
* print()
* # (ii) Insertion
* # (i) At the beginning
* def insert\_at\_beginning(self, data):
* new\_node = Node(data)
* if not self.head:
* new\_node.next = new\_node
* self.head = new\_node
* else:
* new\_node.next = self.head
* current = self.head
* while current.next != self.head:
* current = current.next
* current.next = new\_node
* self.head = new\_node
* # (ii) At the end
* def insert\_at\_end(self, data):
* new\_node = Node(data)
* if not self.head:
* new\_node.next = new\_node
* self.head = new\_node
* else:
* current = self.head
* while current.next != self.head:
* current = current.next
* current.next = new\_node
* new\_node.next = self.head
* # (iii) Deletion
* # (i) At the beginning
* def delete\_at\_beginning(self):
* if not self.head:
* print("Circular Linked List is empty. Deletion not possible.")
* return
* elif self.head.next == self.head:
* self.head = None
* else:
* current = self.head
* while current.next != self.head:
* current = current.next
* current.next = self.head.next
* self.head = self.head.next
* # (ii) At the end
* def delete\_at\_end(self):
* if not self.head:
* print("Circular Linked List is empty. Deletion not possible.")
* return
* elif self.head.next == self.head:
* self.head = None
* else:
* current = self.head
* prev = None
* while current.next != self.head:
* prev = current
* current = current.next
* prev.next = self.head
* # Example usage
* circular\_list = CircularLinkedList()
* circular\_list.insert\_at\_beginning(3)
* circular\_list.insert\_at\_beginning(2)
* circular\_list.insert\_at\_beginning(1)
* circular\_list.traverse()
* circular\_list.insert\_at\_end(4)
* circular\_list.insert\_at\_end(5)
* circular\_list.traverse()
* circular\_list.delete\_at\_beginning()
* circular\_list.traverse()
* circular\_list.delete\_at\_end()
* circular\_list.traverse()
* ***PRACTICAL -8 Stacks operations***

# This is for Stack Q8

class Stack:

    def \_\_init\_\_(self):

        self.items = []

    def display(self):

        if not self.is\_empty():

            print("Stack:", self.items)

        else:

            print("Stack is empty.")

    def insert(self, item):

        self.items.append(item)

        print(f"{item} has been inserted into the stack.")

    def delete(self):

        if not self.is\_empty():

            removed\_item = self.items.pop()

            print(f"{removed\_item} has been deleted from the stack.")

        else:

            print("Stack is empty. Cannot delete.")

    def is\_empty(self):

        return len(self.items) == 0

def main():

    stack = Stack()

    while True:

        print("\nMenu:")

        print("1. Display")

        print("2. Insert")

        print("3. Delete")

        print("4. Exit")

        choice = input("Enter your choice (1-4): ")

        if choice == '1':

            stack.display()

        elif choice == '2':

            item = input("Enter the item to insert: ")

            stack.insert(item)

        elif choice == '3':

            stack.delete()

        elif choice == '4':

            print("Exiting the program. Goodbye!")

            break

        else:

            print("Invalid choice. Please enter a number between 1 and 4.")

if \_\_name\_\_ == "\_\_main\_\_":

    main()

* ***PRACTICAL -9 stacks operations using linked list***

# This is for Stack using Linked List Q9

class Node:

    def \_\_init\_\_(self, data):

        self.data = data

        self.next = None

class StackLinkedList:

    def \_\_init\_\_(self):

        self.top = None

    def display\_stack(self):

        if not self.top:

            print("Stack is empty.")

        else:

            current = self.top

            while current:

                print(current.data, end=" -> ")

                current = current.next

            print()

    def push(self, data):

        new\_node = Node(data)

        new\_node.next = self.top

        self.top = new\_node

        print(f"Element {data} pushed onto the stack.")

    def pop(self):

        if not self.top:

            print("Stack is empty. Cannot pop.")

        else:

            popped\_element = self.top.data

            self.top = self.top.next

            print(f"Element {popped\_element} popped from the stack.")

# Example usage

stack\_linked\_list = StackLinkedList()

while True:

    print("\nChoose operation:")

    print("1. Display Stack")

    print("2. Insert (Push) Element")

    print("3. Delete (Pop) Element")

    print("4. Quit")

    choice = input("Enter your choice (1/2/3/4): ")

    if choice == '1':

        stack\_linked\_list.display\_stack()

    elif choice == '2':

        element = int(input("Enter element to push onto the stack: "))

        stack\_linked\_list.push(element)

    elif choice == '3':

        stack\_linked\_list.pop()

    elif choice == '4':

        print("Exiting program.")

        break

    else:

        print("Invalid choice. Please enter 1, 2, 3, or 4.")

* ***PRACTICAL -10 Queue operations like Dequeue and Enqueue***

# THIS IS FOR DEQUEUE AND ENQUEUE Q10

class Queue:

    def \_\_init\_\_(self):

        self.items = []

    def enqueue(self, item):

        self.items.append(item)

        print(f"{item} has been enqueued.")

    def dequeue(self):

        if not self.is\_empty():

            removed\_item = self.items.pop(0)

            print(f"{removed\_item} has been dequeued.")

        else:

            print("Queue is empty. Cannot dequeue.")

    def display(self):

        if not self.is\_empty():

            print("Queue:", self.items)

        else:

            print("Queue is empty.")

    def is\_empty(self):

        return len(self.items) == 0

def main():

    queue = Queue()

    while True:

        print("\nMenu:")

        print("1. Enqueue")

        print("2. Dequeue")

        print("3. Display")

        print("4. Exit")

        choice = input("Enter your choice (1-4): ")

        if choice == '1':

            item = input("Enter the item to enqueue: ")

            queue.enqueue(item)

        elif choice == '2':

            queue.dequeue()

        elif choice == '3':

            queue.display()

        elif choice == '4':

            print("Exiting the program. Goodbye!")

            break

        else:

            print("Invalid choice. Please enter a number between 1 and 4.")

if \_\_name\_\_ == "\_\_main\_\_":

    main()

* ***PRACTICAL -11 Factorial operations***

def factorial(n):

    # Base case: factorial of 0 is 1

    if n == 0:

        return 1

    # Recursive case: n! = n \* (n-1)!

    else:

        return n \* factorial(n - 1)

# Example usage

number = int(input("Enter a number to find its factorial: "))

result = factorial(number)

print(f"The factorial of {number} is: {result}")

* ***PRACTICAL -12 tower of Hanoi operations***

def tower\_of\_hanoi(n, source, auxiliary, target):

    if n > 0:

        # Move n-1 disks from source to auxiliary using target as auxiliary

        tower\_of\_hanoi(n - 1, source, target, auxiliary)

        # Move the nth disk from source to target

        print(f"Move disk {n} from {source} to {target}")

        # Move the n-1 disks from auxiliary to target using source as auxiliary

        tower\_of\_hanoi(n - 1, auxiliary, source, target)

# Example usage

number\_of\_disks = int(input("Enter the number of disks for Tower of Hanoi: "))

tower\_of\_hanoi(number\_of\_disks, 'A', 'B', 'C')

* ***PRACTICAL -13 Linear and Binary search Algorithm***

# This is for linear and binary search tree

# (i) Linear Search Algorithm

def linear\_search(arr, target):

    for i in range(len(arr)):

        if arr[i] == target:

            return i  # Return the index if the target is found

    return -1  # Return -1 if the target is not found

# (ii) Binary Search Algorithm (Assumes the array is sorted)

def binary\_search(arr, target):

    low, high = 0, len(arr) - 1

    while low <= high:

        mid = (low + high) // 2

        if arr[mid] == target:

            return mid  # Return the index if the target is found

        elif arr[mid] < target:

            low = mid + 1

        else:

            high = mid - 1

    return -1  # Return -1 if the target is not found

#  usage

arr = [2, 5, 8, 12, 16, 23, 38, 42, 55, 67]

target\_value = int(input("Enter the target value to search: "))

# Linear Search

linear\_search\_result = linear\_search(arr, target\_value)

if linear\_search\_result != -1:

    print(f"Linear Search: Target found at index {linear\_search\_result}")

else:

    print("Linear Search: Target not found in the array")

# Binary Search (Array should be sorted)

sorted\_arr = sorted(arr)

binary\_search\_result = binary\_search(sorted\_arr, target\_value)

if binary\_search\_result != -1:

    print(f"Binary Search: Target found at index {binary\_search\_result}")

else:

    print("Binary Search: Target not found in the sorted array")