**1. Array Operations: Write a program to implement**

**basic array operations:**

**a. Insert an element at a specific position in an array.**

**b. Delete an element from a specific position in an**

**array.**

**c. Search for an element in an array (linear search).**

---🡪1. # Insert, Delete, Search in an array (list in Python)

def insert\_element(arr, position, element):

    """

    Inserts an element at a given position in the array.

    Python's list.insert() is used, which shifts elements automatically.

    """

    arr.insert(position, element)

    return arr

def delete\_element(arr, position):

    if 0 <= position < len(arr):

        arr.pop(position)

    else:

        print("Invalid position")

    return arr

def linear\_search(arr, element):

    for i in range(len(arr)):

        if arr[i] == element:

            return i

    return -1

# ------- Main Program / Test -------

array = [13, 28, 32, 45, 53]

print("Original Array:", array)

#Delete element at position 3

array = delete\_element(array, 3)

print("After Deletion at position 3:", array)

# Search for 53

index = linear\_search(array, 53)

if index != -1:

    print("Element 53 found at index:", index)

else:

    print("Element 53 not found")

**2. Linked List Manipulation: Write a program to:**

**a. Create a singly linked list.**

**b. Insert a node at the beginning, end, and at a given**

**position in a linked list.**

**c. Delete a node from a given position in a linked list.**

-🡪 # Node class to define elements of linked list

class Node:

    def \_\_init\_\_(self, data):

        self.data = data      # data value

        self.next = None      # pointer to next node

# Singly Linked List Class

class SinglyLinkedList:

    def \_\_init\_\_(self):

        self.head = None

    def insert\_at\_beginning(self, data):

        new\_node = Node(data)

        new\_node.next = self.head

        self.head = new\_node

    def insert\_at\_end(self, data):

        new\_node = Node(data)

        if not self.head:

            self.head = new\_node

            return

        temp = self.head

        while temp.next:

            temp = temp.next

        temp.next = new\_node

    def insert\_at\_position(self, data, position):

        if position == 0:

            self.insert\_at\_beginning(data)

            return

        new\_node = Node(data)

        temp = self.head

        for \_ in range(position - 1):

            if not temp:

                print("Position out of range")

                return

            temp = temp.next

        new\_node.next = temp.next

        temp.next = new\_node

    def delete\_at\_position(self, position):

        if not self.head:

            print("List is empty")

            return

        temp = self.head

        if position == 0:

            self.head = temp.next

            return

        for \_ in range(position - 1):

            if not temp.next:

                print("Position out of range")

                return

            temp = temp.next

        temp.next = temp.next.next

    def display(self):

        """

        Display all elements of the linked list.

        """

        temp = self.head

        while temp:

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

            temp = temp.next

        print("None")

# ----------- Main Program / Test -----------

ll = SinglyLinkedList()

ll.insert\_at\_end(10)

ll.insert\_at\_beginning(5)

ll.insert\_at\_position(70, 2)

ll.insert\_at\_end(20)

ll.delete\_at\_position(2)

print("Linked List after deleting node at position 2:")

ll.display()

**3. Stack Application: Write a program to:**

**a. Implement a stack using an array.**

**b. Convert an infix expression to postfix notation using**

**a stack.**

-🡪 part A

class Stack:

    def \_\_init\_\_(self, size):

        self.size = size        # Maximum size of stack

        self.stack = []         # Initialize an empty list

        self.top = -1           # Keep track of the top index

    # Push operation

    def push(self, value):

        if self.top >= self.size - 1:

            print("Stack Overflow! Cannot push:", value)

        else:

            self.stack.append(value)

            self.top += 1

            print(f"Pushed {value} into stack")

    # Pop operation

    def pop(self):

        if self.top == -1:

            print("Stack Underflow! Nothing to pop")

            return None

        else:

            value = self.stack.pop()

            self.top -= 1

            print(f"Popped {value} from stack")

            return value

    # Peek operation

    def peek(self):

        if self.top == -1:

            print("Stack is empty, no top element")

            return None

        else:

            return self.stack[self.top]

    # Check if stack is empty

    def isEmpty(self):

        return self.top == -1

    # Display stack

    def display(self):

        if self.isEmpty():

            print("Stack is empty")

        else:

            print("Stack elements (top → bottom):", self.stack[::-1])

# --------- Demonstration ---------

stack = Stack(5)   # Stack of size 5

stack.push(10)

stack.push(20)

stack.push(30)

stack.display()

print("Top element:", stack.peek())

stack.pop()

stack.display()

stack.push(40)

stack.push(50)

stack.push(60)

stack.push(70)   # This should give overflow error

stack.display()

part B

# Function to define precedence of operators

def precedence(op):

    if op == '+' or op == '-':

        return 1

    if op == '\*' or op == '/':

        return 2

    if op == '^':

        return 3

    return 0

# Function to convert infix expression to postfix

def infix\_to\_postfix(expression):

    result = ""      # final postfix expression

    stack = []       # stack to hold operators

    for char in expression:

        # If character is an operand, add to result

        if char.isalnum():   # (A-Z, a-z, 0-9)

            result += char

        # If character is '(', push to stack

        elif char == '(':

            stack.append(char)

        # If character is ')', pop until '(' is found

        elif char == ')':

            while stack and stack[-1] != '(':

                result += stack.pop()

            stack.pop()  # remove '(' from stack

        # Operator encountered

        else:

            while stack and precedence(stack[-1]) >= precedence(char):

                result += stack.pop()

            stack.append(char)

    # Pop remaining operators from stack

    while stack:

        result += stack.pop()

    return result

# --------- Demonstration ---------

exp1 = "A+B\*C"

exp2 = "(A+B)\*C"

exp3 = "A+B\*(C^D-E)^(F+G\*H)-I"

print("Infix:", exp1, " → Postfix:", infix\_to\_postfix(exp1))

print("Infix:", exp2, " → Postfix:", infix\_to\_postfix(exp2))

print("Infix:", exp3, " → Postfix:", infix\_to\_postfix(exp3))

5.**Binary Search Tree: Write a program to:**

**a. Create a binary search tree..**

**b. Insert nodes into a binary search tree.**

**c. Search for a node in a binary search tree.**

🡪# Node structure for BST

class Node:

    def \_\_init\_\_(self, key):

        self.key = key

        self.left = None

        self.right = None

# BST implementation

class BinarySearchTree:

    def \_\_init\_\_(self):

        self.root = None

    # Insert a node

    def insert(self, root, key):

        if root is None:

            return Node(key)

        if key < root.key:

            root.left = self.insert(root.left, key)

        elif key > root.key:

            root.right = self.insert(root.right, key)

        return root

    # Search a node

    def search(self, root, key):

        if root is None:

            return None

        if root.key == key:

            return root

        if key < root.key:

            return self.search(root.left, key)

        else:

            return self.search(root.right, key)

    # Inorder Traversal (L → Root → R)

    def inorder(self, root):

        if root:

            self.inorder(root.left)

            print(root.key, end=" ")

            self.inorder(root.right)

# --------- Demonstration ---------

bst = BinarySearchTree()

# Insert nodes

root = None

for value in [50, 30, 70, 20, 40, 60, 80]:

    root = bst.insert(root, value)

print("Inorder Traversal of BST (sorted):")

bst.inorder(root)

print()

# Search nodes

key = 60

if bst.search(root, key):

    print(f"Node {key} found in BST")

else:

    print(f"Node {key} not found in BST")

key = 25

if bst.search(root, key):

    print(f"Node {key} found in BST")

else:

    print(f"Node {key} not found in BST")

**6. Tree Traversal: Write a program to:**

**a. Implement pre-order,**

**b. in-order,**

**c. Post-order traversal of a binary tree**

🡪# Node structure

class Node:

    def \_\_init\_\_(self, key):

        self.key = key

        self.left = None

        self.right = None

# BST with traversal methods

class BinarySearchTree:

    def \_\_init\_\_(self):

        self.root = None

    # Insert a node

    def insert(self, root, key):

        if root is None:

            return Node(key)

        if key < root.key:

            root.left = self.insert(root.left, key)

        elif key > root.key:

            root.right = self.insert(root.right, key)

        return root

    # Inorder Traversal (L → Root → R)

    def inorder(self, root):

        if root:

            self.inorder(root.left)

            print(root.key, end=" ")

            self.inorder(root.right)

    # Preorder Traversal (Root → L → R)

    def preorder(self, root):

        if root:

            print(root.key, end=" ")

            self.preorder(root.left)

            self.preorder(root.right)

    # Postorder Traversal (L → R → Root)

    def postorder(self, root):

        if root:

            self.postorder(root.left)

            self.postorder(root.right)

            print(root.key, end=" ")

# --------- Demonstration ---------

bst = BinarySearchTree()

root = None

# Insert nodes

for value in [50, 30, 70, 20, 40, 60, 80]:

    root = bst.insert(root, value)

print("Inorder Traversal (L → Root → R):")

bst.inorder(root)

print()

print("Preorder Traversal (Root → L → R):")

bst.preorder(root)

print()

print("Postorder Traversal (L → R → Root):")

bst.postorder(root)

print()

**8.Sorting Algorithms: Write programs to implement**

**and compare the following sorting algorithms:**

**a. Bubble sort**

**b. Insertion sort**

**c. Selection sort**

🡪import time

# ---------- Bubble Sort ----------

def bubble\_sort(arr):

    n = len(arr)

    comparisons, swaps = 0, 0

    for i in range(n):

        for j in range(0, n-i-1):

            comparisons += 1

            if arr[j] > arr[j+1]:

                arr[j], arr[j+1] = arr[j+1], arr[j]

                swaps += 1

    return arr, comparisons, swaps

# ---------- Insertion Sort ----------

def insertion\_sort(arr):

    comparisons, swaps = 0, 0

    for i in range(1, len(arr)):

        key = arr[i]

        j = i-1

        while j >= 0 and arr[j] > key:

            comparisons += 1

            arr[j+1] = arr[j]

            swaps += 1

            j -= 1

        comparisons += 1  # last failed comparison

        arr[j+1] = key

    return arr, comparisons, swaps

# ---------- Selection Sort ----------

def selection\_sort(arr):

    comparisons, swaps = 0, 0

    n = len(arr)

    for i in range(n):

        min\_idx = i

        for j in range(i+1, n):

            comparisons += 1

            if arr[j] < arr[min\_idx]:

                min\_idx = j

        arr[i], arr[min\_idx] = arr[min\_idx], arr[i]

        swaps += 1

    return arr, comparisons, swaps

# ---------- Demonstration ----------

arr = [75,16, 80, 10,18,6]

print("Original Array:", arr)

# Bubble Sort

sorted\_arr, comp, swp = bubble\_sort(arr.copy())

print("\nBubble Sort →", sorted\_arr)

print("Comparisons:", comp, "| Swaps:", swp)

# Insertion Sort

sorted\_arr, comp, swp = insertion\_sort(arr.copy())

print("\nInsertion Sort →", sorted\_arr)

print("Comparisons:", comp, "| Swaps:", swp)

# Selection Sort

sorted\_arr, comp, swp = selection\_sort(arr.copy())

print("\nSelection Sort →", sorted\_arr)

print("Comparisons:", comp, "| Swaps:", swp)

**9.Searching Algorithms: Write programs to implement**

**and compare:**

**a. Linear search**

**b. Binary search (on a sorted array)**

**🡪**# ---------- Linear Search ----------

def linear\_search(arr, target):

    comparisons = 0

    for i in range(len(arr)):

        comparisons += 1

        if arr[i] == target:

            return i, comparisons

    return -1, comparisons

# ---------- Binary Search ----------

def binary\_search(arr, target):

    left, right = 0, len(arr) - 1

    comparisons = 0

    while left <= right:

        mid = (left + right) // 2

        comparisons += 1

        if arr[mid] == target:

            return mid, comparisons

        elif arr[mid] < target:

            left = mid + 1

        else:

            right = mid - 1

    return -1, comparisons

# ---------- Demonstration ----------

arr = [11, 22, 25, 64, 90]  # Sorted array for binary search

target = 25

print("Array:", arr)

print("Target:", target)

# Linear Search

pos, comp = linear\_search(arr, target)

print("\nLinear Search → Position:", pos, "| Comparisons:", comp)

# Binary Search

pos, comp = binary\_search(arr, target)

print("Binary Search → Position:", pos, "| Comparisons:", comp)