SRM Institute of Science and Technology

Delhi – Meerut Road, Sikri Kalan, Ghaziabad, Uttar Pradesh – 201204

Department of Computer Applications

Circular – 2023-24

B.Sc. CS 2nd Sem

Fundamentals of Data Structures and Algorithms (USA23202J)

Lab Manual

Lab 1: Recursion

Title: Understanding Recursion through Factorial Calculation

Aim: To implement and understand the concept of recursion by calculating the factorial of a given number.

Procedure:

1. Define a recursive function, `factorial(n)`, that takes an integer `n` as input.
2. Establish the base case: if `n` is 0 or 1, the factorial is 1.
3. Define the recursive step: if `n` is greater than 1, the factorial is `n` multiplied by the factorial of `n-1`.
4. Call the function with a sample input and print the result.

Source Code:

```
# factorial.py

def factorial(n):
    """
    Calculates the factorial of a non-negative integer using recursion.

    Args:
        n: A non-negative integer.

    Returns:
        The factorial of n.
    """
    # Base case: Factorial of 0 or 1 is 1
    if n == 0 or n == 1:
        return 1
    # Recursive step: n * factorial(n-1)
    else:
        return n * factorial(n - 1)

# Main part of the program to test the function
if __name__ == "__main__":
    # Get input from the user
    try:
        num = int(input("Enter a non-negative integer to calculate its 
factorial: "))
        if num < 0:
```

```
            print("Factorial is not defined for negative numbers.")
        else:
            result = factorial(num)
            print(f"The factorial of {num} is {result}")
    except ValueError:
        print("Invalid input. Please enter an integer.")
```

Input:

```
Enter a non-negative integer to calculate its factorial: 5
```

Expected Output:

```
The factorial of 5 is 120
```

Lab 2: Arrays

Title: Array Operations: Sum and Average of Elements

Aim: To implement basic array operations, specifically calculating the sum and average of elements in an array.

Procedure:

1. Initialize an array (list in Python) with a set of integer values.
2. Iterate through the array to sum all its elements.
3. Calculate the average by dividing the sum by the number of elements.
4. Print the sum and average.

Source Code:

```python
# array_operations.py

def calculate_array_stats(arr):
    """
    Calculates the sum and average of elements in a given array.

    Args:
        arr: A list of numbers.

    Returns:
        A tuple containing the sum and average (sum, average).
        Returns (0, 0) if the array is empty.
    """
    if not arr:
        return 0, 0 # Handle empty array case

    total_sum = 0
    for element in arr:
        total_sum += element

    num_elements = len(arr)
    average = total_sum / num_elements

    return total_sum, average

# Main part of the program to test the function
if __name__ == "__main__":
    # Example array
    my_array = [10, 20, 30, 40, 50]

    print(f"Original Array: {my_array}")

    # Calculate sum and average
    array_sum, array_average = calculate_array_stats(my_array)

    # Print the results
    print(f"Sum of array elements: {array_sum}")
    print(f"Average of array elements: {array_average}")

    # Another example with different values
    another_array = [2, 4, 6, 8]
    print(f"\nOriginal Array: {another_array}")
    array_sum_2, array_average_2 = calculate_array_stats(another_array)
    print(f"Sum of array elements: {array_sum_2}")
    print(f"Average of array elements: {array_average_2}")
```

Input: (No direct user input for this program, array is hardcoded)

Expected Output:

```
Original Array: [10, 20, 30, 40, 50]
Sum of array elements: 150
Average of array elements: 30.0

Original Array: [2, 4, 6, 8]
Sum of array elements: 20
Average of array elements: 5.0
```

Lab 3: Implementation of LinkedList

Title: Singly Linked List Implementation: Insertion and Traversal

Aim: To implement a singly linked list with functionalities for inserting new nodes at the end and traversing the list to print its elements.

Procedure:

1. Define a `Node` class with `data` and `next` (pointer to the next node) attributes.
2. Define a `LinkedList` class with a `head` attribute, initially `None`.
3. Implement an `append` method to add a new node to the end of the list.
4. Implement a `display` method to traverse the list from the head and print the data of each node.

Source Code:

```
# linked_list.py

class Node:
    """
    Represents a node in a singly linked list.
    Each node stores data and a reference to the next node.
    """
    def __init__(self, data):
        self.data = data
        self.next = None # Pointer to the next node

class LinkedList:
    """
    Represents a singly linked list.
    Manages the head of the list and provides methods for operations.
    """
    def __init__(self):
        self.head = None # Initially, the list is empty

    def append(self, data):
        """
        Adds a new node with the given data to the end of the linked list.

        Args:
            data: The data to be stored in the new node.
        """
        new_node = Node(data)
        if self.head is None:
            self.head = new_node # If list is empty, new node becomes the
head
            return

        last_node = self.head
        while last_node.next: # Traverse to the last node
            last_node = last_node.next
        last_node.next = new_node # Link the new node to the end

    def display(self):
        """
        Traverses the linked list from the head and prints the data of each
node.
        """
        current_node = self.head
        if current_node is None:
            print("Linked List is empty.")
```

```
            return

        elements = []
        while current_node:
            elements.append(str(current_node.data))
            current_node = current_node.next
        print("Linked List: " + " -> ".join(elements))

# Main part of the program to test the linked list implementation
if __name__ == "__main__":
    my_list = LinkedList()

    print("Appending elements to the linked list:")
    my_list.append(10)
    my_list.append(20)
    my_list.append(30)
    my_list.display() # Expected: 10 -> 20 -> 30

    my_list.append(40)
    print("\nAfter appending 40:")
    my_list.display() # Expected: 10 -> 20 -> 30 -> 40

    empty_list = LinkedList()
    print("\nTrying to display an empty list:")
    empty_list.display() # Expected: Linked List is empty.
```

Input: (No direct user input for this program, operations are hardcoded)

Expected Output:

```
Appending elements to the linked list:
Linked List: 10 -> 20 -> 30

After appending 40:
Linked List: 10 -> 20 -> 30 -> 40

Trying to display an empty list:
Linked List is empty.
```

Lab 4: Implementation of Stack and its Applications

Title: Stack Implementation using List and Parenthesis Checker Application

Aim: To implement a stack data structure using a Python list and demonstrate its application in checking for balanced parentheses in an expression.

Procedure:

1. Implement a `Stack` class with `push`, `pop`, `peek`, `is_empty`, and `size` methods. Use a Python list internally.
2. Implement a function `is_balanced_parentheses` that takes a string expression as input.
3. Use the stack to process the expression:
   - If an opening parenthesis `(`, `{`, `[` is encountered, push it onto the stack.
   - If a closing parenthesis `)`, `}`, `]` is encountered, pop from the stack. If the stack is empty or the popped element doesn't match the opening counterpart, the parentheses are unbalanced.
4. After processing the entire expression, if the stack is empty, the parentheses are balanced; otherwise, they are not.

Source Code:

```python
# stack_parenthesis_checker.py

class Stack:
    """
    A simple Stack implementation using a Python list.
    """
    def __init__(self):
        self.items = []

    def push(self, item):
        """Adds an item to the top of the stack."""
        self.items.append(item)

    def pop(self):
        """Removes and returns the item from the top of the stack.
        Raises IndexError if the stack is empty.
        """
        if self.is_empty():
            raise IndexError("pop from empty stack")
        return self.items.pop()

    def peek(self):
        """Returns the item at the top of the stack without removing it.
        Raises IndexError if the stack is empty.
        """
        if self.is_empty():
            raise IndexError("peek from empty stack")
        return self.items[-1]

    def is_empty(self):
        """Checks if the stack is empty."""
        return len(self.items) == 0

    def size(self):
        """Returns the number of items in the stack."""
        return len(self.items)

def is_balanced_parentheses(expression):
    """
```

```
    Checks if the parentheses in an expression are balanced.

    Args:
        expression: A string containing parentheses.

    Returns:
        True if parentheses are balanced, False otherwise.
    """
    s = Stack()
    mapping = {")": "(", "}": "{", "]": "["}
    open_brackets = set(mapping.values())
    close_brackets = set(mapping.keys())

    for char in expression:
        if char in open_brackets:
            s.push(char)
        elif char in close_brackets:
            if s.is_empty():
                return False # Closing bracket without a corresponding
opening bracket
            top_element = s.pop()
            if mapping[char] != top_element:
                return False # Mismatched brackets
    return s.is_empty() # True if stack is empty (all brackets matched)

# Main part of the program to test the stack and parenthesis checker
if __name__ == "__main__":
    # Test Stack operations
    my_stack = Stack()
    print("Stack operations:")
    print(f"Is stack empty? {my_stack.is_empty()}") # True
    my_stack.push(10)
    my_stack.push(20)
    print(f"Stack size: {my_stack.size()}") # 2
    print(f"Top element: {my_stack.peek()}") # 20
    print(f"Popped element: {my_stack.pop()}") # 20
    print(f"Stack size after pop: {my_stack.size()}") # 1
    print(f"Is stack empty? {my_stack.is_empty()}") # False
    my_stack.pop()
    print(f"Is stack empty after all pops? {my_stack.is_empty()}") # True

    # Test Parenthesis Checker
    print("\nParenthesis Checker:")
    expressions = [
        "([{}])",        # Balanced
        "({[()]})",      # Balanced
        "(()",           # Unbalanced (missing closing)
        "())",           # Unbalanced (extra closing)
        "[{}]",          # Balanced
        "{[}]",          # Unbalanced (mismatched)
        "",              # Balanced (empty string)
        "abc(def[ghi])", # Balanced (non-bracket chars ignored)
    ]

    for expr in expressions:
        status = "Balanced" if is_balanced_parentheses(expr) else
"Unbalanced"
        print(f"Expression: '{expr}' is {status}")
```

Input: (No direct user input for this program, expressions are hardcoded)

Expected Output:

```
Stack operations:
```

```
Is stack empty? True
Stack size: 2
Top element: 20
Popped element: 20
Stack size after pop: 1
Is stack empty? False
Is stack empty after all pops? True

Parenthesis Checker:
Expression: '([{}])' is Balanced
Expression: '({[()]})' is Balanced
Expression: '(()' is Unbalanced
Expression: '())' is Unbalanced
Expression: '[{}]' is Balanced
Expression: '{[}]' is Unbalanced
Expression: '' is Balanced
Expression: 'abc(def[ghi])' is Balanced
```

Lab 5: Queue implementation using array and pointers

Title: Queue Implementation using List (Array) and Front/Rear Pointers

Aim: To implement a queue data structure using a Python list (simulating an array) and manage front and rear pointers for enqueue and dequeue operations.

Procedure:

1. Implement a `Queue` class.
2. Initialize an empty list (`items`) and `front` and `rear` pointers (integers, e.g., -1 or 0).
3. Implement `enqueue` (add to rear): Add an element to the end of the list and update `rear`.
4. Implement `dequeue` (remove from front): Remove an element from the beginning of the list and update `front`. Handle underflow (empty queue).
5. Implement `is_empty`, `size`, and `peek` (view front element).

Source Code:

```
# queue_array_pointers.py

class Queue:
    """
    A Queue implementation using a Python list, simulating array behavior
    with conceptual front and rear pointers.
    """
    def __init__(self):
        self.items = [] # The underlying list to store queue elements
        # For a simple list-based queue, Python's list methods handle
        # the 'pointer' logic internally for efficiency.
        # Conceptually, front is always at index 0, rear is at len(items) -
1.

    def enqueue(self, item):
        """
        Adds an item to the rear of the queue.
        """
        self.items.append(item)
        print(f"Enqueued: {item}")

    def dequeue(self):
        """
        Removes and returns the item from the front of the queue.
        Raises IndexError if the queue is empty.
        """
        if self.is_empty():
            raise IndexError("dequeue from empty queue")
        dequeued_item = self.items.pop(0) # Remove from the beginning (front)
        print(f"Dequeued: {dequeued_item}")
        return dequeued_item

    def peek(self):
        """
        Returns the item at the front of the queue without removing it.
        Raises IndexError if the queue is empty.
        """
        if self.is_empty():
            raise IndexError("peek from empty queue")
        return self.items[0]

    def is_empty(self):
        """
        Checks if the queue is empty.
```

```
        """
        return len(self.items) == 0

    def size(self):
        """
        Returns the number of items in the queue.
        """
        return len(self.items)

# Main part of the program to test the queue implementation
if __name__ == "__main__":
    my_queue = Queue()

    print("Queue operations:")
    print(f"Is queue empty? {my_queue.is_empty()}") # True
    print(f"Queue size: {my_queue.size()}") # 0

    my_queue.enqueue("Task A")
    my_queue.enqueue("Task B")
    my_queue.enqueue("Task C")

    print(f"Queue size: {my_queue.size()}") # 3
    print(f"Front element (peek): {my_queue.peek()}") # Task A

    my_queue.dequeue() # Task A
    print(f"Queue size after dequeue: {my_queue.size()}") # 2
    print(f"Front element (peek): {my_queue.peek()}") # Task B

    my_queue.enqueue("Task D")
    my_queue.display() # Custom display function to show current queue state

    my_queue.dequeue() # Task B
    my_queue.dequeue() # Task C
    my_queue.dequeue() # Task D

    print(f"Is queue empty? {my_queue.is_empty()}") # True

    try:
        my_queue.dequeue() # This should raise an error
    except IndexError as e:
        print(f"Error: {e}") # Expected: Error: dequeue from empty queue
```

Input: (No direct user input for this program, operations are hardcoded)

Expected Output:

```
Queue operations:
Is queue empty? True
Queue size: 0
Enqueued: Task A
Enqueued: Task B
Enqueued: Task C
Queue size: 3
Front element (peek): Task A
Dequeued: Task A
Queue size after dequeue: 2
Front element (peek): Task B
Enqueued: Task D
Queue: ['Task B', 'Task C', 'Task D']
Dequeued: Task B
Dequeued: Task C
Dequeued: Task D
Is queue empty? True
Error: dequeue from empty queue
```

Note: I've added a simple `display` method to the Queue class for better visualization in the output, although it wasn't explicitly requested in the procedure. This helps in understanding the queue's state.

Lab 6: Implementation of binary tree using Arrays

Title: Binary Tree Implementation using Arrays (List Representation)

Aim: To implement a binary tree using a Python list (array-based representation) and demonstrate insertion and traversal.

Procedure:

1. Represent the binary tree using a list where:
   - The root is at index 0.
   - For a node at index `i`, its left child is at `2*i + 1`.
   - Its right child is at `2*i + 2`.
2. Initialize the list with `None` or a placeholder for empty nodes.
3. Implement an `insert` function to add nodes at specific positions (or implicitly by their parent's index).
4. Implement a `print_tree` function to display the array representation.

Source Code:

```
# binary_tree_array.py

class BinaryTreeArray:
    """
    Implements a binary tree using an array (Python list) representation.
    - Root is at index 0.
    - Left child of node at index i is at 2*i + 1.
    - Right child of node at index i is at 2*i + 2.
    """
    def __init__(self, max_size):
        """
        Initializes the tree with a fixed maximum size.
        None indicates an empty node.
        """
        self.tree = [None] * max_size
        self.max_size = max_size
        self.size = 0 # Current number of nodes in the tree

    def insert(self, data, parent_index=0, position='root'):
        """
        Inserts data into the tree.
        For simplicity, this example allows insertion at a specific index,
        or as a child of a given parent_index (left/right).
        A more robust implementation would manage insertion based on tree
properties (e.g., BST).
        """
        if self.size >= self.max_size:
            print("Tree is full. Cannot insert more elements.")
            return False

        if position == 'root':
            if self.tree[0] is None:
                self.tree[0] = data
                self.size += 1
                print(f"Inserted {data} at root (index 0).")
                return True
            else:
                print(f"Root already exists. Cannot insert {data} as root.")
                return False
        elif position == 'left_child':
            left_child_index = 2 * parent_index + 1
```

```
            if left_child_index < self.max_size and
self.tree[left_child_index] is None:
                self.tree[left_child_index] = data
                self.size += 1
                print(f"Inserted {data} as left child of index {parent_index}
(at index {left_child_index}).")
                return True
            else:
                print(f"Cannot insert {data} as left child of {parent_index}.
Position occupied or out of bounds.")
                return False
        elif position == 'right_child':
            right_child_index = 2 * parent_index + 2
            if right_child_index < self.max_size and
self.tree[right_child_index] is None:
                self.tree[right_child_index] = data
                self.size += 1
                print(f"Inserted {data} as right child of index
{parent_index} (at index {right_child_index}).")
                return True
            else:
                print(f"Cannot insert {data} as right child of
{parent_index}. Position occupied or out of bounds.")
                return False
        else:
            print("Invalid position specified for insertion.")
            return False

    def print_tree_array(self):
        """
        Prints the array representation of the binary tree.
        """
        print("Binary Tree (Array Representation):")
        print(self.tree)
        print(f"Current number of nodes: {self.size}")

# Main part of the program to test the binary tree array implementation
if __name__ == "__main__":
    # Create a tree with a maximum size of 10 nodes
    tree = BinaryTreeArray(10)

    # Insert root
    tree.insert('A', position='root') # Root

    # Insert children of root (A at index 0)
    tree.insert('B', parent_index=0, position='left_child')  # Left child of
A (index 1)
    tree.insert('C', parent_index=0, position='right_child') # Right child of
A (index 2)

    # Insert children of B (B at index 1)
    tree.insert('D', parent_index=1, position='left_child')  # Left child of
B (index 3)
    tree.insert('E', parent_index=1, position='right_child') # Right child of
B (index 4)

    # Insert children of C (C at index 2)
    tree.insert('F', parent_index=2, position='left_child')  # Left child of
C (index 5)

    tree.print_tree_array()

    print("\nAttempting to insert where position is occupied or out of
bounds:")
    tree.insert('X', parent_index=0, position='left_child') # Try to insert
over B
```

```
    tree.insert('Y', parent_index=5, position='right_child') # Parent index
5, right child index 12 (out of bounds)
    tree.insert('Z', parent_index=4, position='left_child') # Left child of E
(index 9)

    tree.print_tree_array()
```

Input: (No direct user input for this program, operations are hardcoded)

Expected Output:

```
Inserted A at root (index 0).
Inserted B as left child of index 0 (at index 1).
Inserted C as right child of index 0 (at index 2).
Inserted D as left child of index 1 (at index 3).
Inserted E as right child of index 1 (at index 4).
Inserted F as left child of index 2 (at index 5).
Binary Tree (Array Representation):
['A', 'B', 'C', 'D', 'E', 'F', None, None, None, None]
Current number of nodes: 6

Attempting to insert where position is occupied or out of bounds:
Cannot insert X as left child of 0. Position occupied or out of bounds.
Cannot insert Y as right child of 5. Position occupied or out of bounds.
Inserted Z as left child of index 4 (at index 9).
Binary Tree (Array Representation):
['A', 'B', 'C', 'D', 'E', 'F', None, None, None, 'Z']
Current number of nodes: 7
```

Lab 7: Implement all the three type of Tree Traversals

Title: Binary Tree Traversals: Inorder, Preorder, and Postorder

Aim: To implement a binary tree using a node-based approach and demonstrate the three standard tree traversal algorithms: Inorder, Preorder, and Postorder.

Procedure:

1. Define a `TreeNode` class with `data`, `left` (left child), and `right` (right child) attributes.
2. Implement a `BinaryTree` class with a `root` attribute.
3. Implement methods for inserting nodes to build a sample tree (e.g., a simple insertion for demonstration, or a more complex BST insertion).
4. Implement three recursive functions for traversal:
   - `inorder_traversal(node)`: Left -> Root -> Right
   - `preorder_traversal(node)`: Root -> Left -> Right
   - `postorder_traversal(node)`: Left -> Right -> Root

Source Code:

```
# tree_traversals.py

class TreeNode:
    """
    Represents a node in a binary tree.
    """
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

class BinaryTree:
    """
    A simple Binary Tree class for demonstrating traversals.
    """
    def __init__(self):
        self.root = None

    def insert(self, data):
        """
        Inserts a new node into the binary tree.
        This is a simple insertion for demonstration purposes,
        not a balanced or BST-specific insertion.
        It adds nodes level by level (like a complete binary tree for
simplicity).
        For a more robust solution, a queue would be used for level-order
insertion.
        Here, we'll manually build a small tree for traversal examples.
        """
        # For this lab, we'll manually build a sample tree for clarity
        # rather than a generic insert method that balances or orders.
        pass

    def build_sample_tree(self):
        """
        Manually builds a sample binary tree for traversal demonstration.
        Tree structure:
              1
             / \
            2   3
           / \
```

```
          4   5
        """
        self.root = TreeNode(1)
        self.root.left = TreeNode(2)
        self.root.right = TreeNode(3)
        self.root.left.left = TreeNode(4)
        self.root.left.right = TreeNode(5)
        print("Sample tree built.")

    def inorder_traversal(self, node):
        """
        Performs an Inorder traversal (Left -> Root -> Right).
        """
        if node:
            self.inorder_traversal(node.left)
            print(node.data, end=" ")
            self.inorder_traversal(node.right)

    def preorder_traversal(self, node):
        """
        Performs a Preorder traversal (Root -> Left -> Right).
        """
        if node:
            print(node.data, end=" ")
            self.preorder_traversal(node.left)
            self.preorder_traversal(node.right)

    def postorder_traversal(self, node):
        """
        Performs a Postorder traversal (Left -> Right -> Root).
        """
        if node:
            self.postorder_traversal(node.left)
            self.postorder_traversal(node.right)
            print(node.data, end=" ")

# Main part of the program to test tree traversals
if __name__ == "__main__":
    tree = BinaryTree()
    tree.build_sample_tree()

    print("\nInorder Traversal:")
    tree.inorder_traversal(tree.root) # Expected: 4 2 5 1 3
    print()

    print("\nPreorder Traversal:")
    tree.preorder_traversal(tree.root) # Expected: 1 2 4 5 3
    print()

    print("\nPostorder Traversal:")
    tree.postorder_traversal(tree.root) # Expected: 4 5 2 3 1
    print()
```

Input: (No direct user input for this program, tree is hardcoded)

Expected Output:

```
Sample tree built.

Inorder Traversal:
4 2 5 1 3

Preorder Traversal:
1 2 4 5 3
```

```
Postorder Traversal:
4 5 2 3 1
```

Lab 8: Implementation of BST HeapDataStructure

Title: Binary Search Tree (BST) Implementation: Insertion and Search

Aim: To implement a Binary Search Tree (BST) with functionalities for inserting new nodes while maintaining BST properties and searching for a specific value.

Procedure:

1. Define a `Node` class for the BST, similar to a binary tree node, but with the understanding that left children are smaller and right children are larger.
2. Implement a `BST` class with a `root` attribute.
3. Implement an `insert` method:
   - If the tree is empty, the new node becomes the root.
   - Otherwise, traverse the tree to find the correct position: go left if the new data is smaller, go right if it's larger.
4. Implement a `search` method:
   - Start from the root.
   - If the current node's data matches the search value, return the node.
   - If the search value is smaller, go left; otherwise, go right.
   - If a `None` child is reached, the value is not in the tree.
5. Implement an `inorder_traversal` for verification.

Source Code:

```
# bst_implementation.py

class Node:
    """
    Represents a node in a Binary Search Tree (BST).
    """
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None

class BST:
    """
    Implements a Binary Search Tree.
    """
    def __init__(self):
        self.root = None

    def insert(self, key):
        """
        Inserts a new key into the BST.
        """
        new_node = Node(key)
        if self.root is None:
            self.root = new_node
            print(f"Inserted {key} as root.")
            return

        current = self.root
        while True:
            if key < current.key:
                if current.left is None:
                    current.left = new_node
                    print(f"Inserted {key} as left child of {current.key}.")
                    return
```

```
                current = current.left
            elif key > current.key:
                if current.right is None:
                    current.right = new_node
                    print(f"Inserted {key} as right child of {current.key}.")
                    return
                current = current.right
            else:
                print(f"Key {key} already exists in the BST. Not inserted.")
                return

    def search(self, key):
        """
        Searches for a key in the BST.

        Returns:
            The Node object if found, None otherwise.
        """
        current = self.root
        while current:
            if key == current.key:
                return current
            elif key < current.key:
                current = current.left
            else: # key > current.key
                current = current.right
        return None # Key not found

    def inorder_traversal(self, node):
        """
        Performs an Inorder traversal of the BST (Left -> Root -> Right).
        This prints elements in sorted order.
        """
        if node:
            self.inorder_traversal(node.left)
            print(node.key, end=" ")
            self.inorder_traversal(node.right)

# Main part of the program to test the BST implementation
if __name__ == "__main__":
    bst = BST()

    # Insert elements
    bst.insert(50)
    bst.insert(30)
    bst.insert(70)
    bst.insert(20)
    bst.insert(40)
    bst.insert(60)
    bst.insert(80)
    bst.insert(30) # Attempt to insert duplicate

    print("\nInorder Traversal of BST (should be sorted):")
    bst.inorder_traversal(bst.root)
    print()

    # Search for elements
    print("\nSearching for elements:")
    search_keys = [40, 90, 50, 10]
    for key in search_keys:
        node = bst.search(key)
        if node:
            print(f"Key {key} found in BST.")
        else:
            print(f"Key {key} not found in BST.")
```

Input: (No direct user input for this program, operations are hardcoded)

Expected Output:

```
Inserted 50 as root.
Inserted 30 as left child of 50.
Inserted 70 as right child of 50.
Inserted 20 as left child of 30.
Inserted 40 as right child of 30.
Inserted 60 as left child of 70.
Inserted 80 as right child of 70.
Key 30 already exists in the BST. Not inserted.

Inorder Traversal of BST (should be sorted):
20 30 40 50 60 70 80

Searching for elements:
Key 40 found in BST.
Key 90 not found in BST.
Key 50 found in BST.
Key 10 not found in BST.
```

Lab 9: Implementation of Min and Max Heap

Title: Min-Heap and Max-Heap Implementation using List

Aim: To implement both Min-Heap and Max-Heap data structures using a Python list (array-based representation) and demonstrate their basic operations (insertion and extraction).

Procedure:

1. **Heap Property:** Understand that in a Max-Heap, parent nodes are always greater than or equal to their children, and in a Min-Heap, parent nodes are always less than or equal to their children.
2. **Array Representation:** For a node at index `i`:
   - Left child: `2*i + 1`
   - Right child: `2*i + 2`
   - Parent: `(i - 1) // 2`
3. **Implement `MinHeap` class:**
   - `insert(key)`: Add to the end, then `heapify_up` (bubble up) to maintain min-heap property.
   - `extract_min()`: Remove root, replace with last element, then `heapify_down` (bubble down) to maintain min-heap property.
   - `_heapify_up(index)`: Helper to move element up.
   - `_heapify_down(index)`: Helper to move element down.
4. **Implement `MaxHeap` class:** (Similar to MinHeap, but comparison logic is reversed for `heapify_up` and `heapify_down`).

Source Code:

```
# min_max_heap.py

class MinHeap:
    """
    Implements a Min-Heap data structure.
    """
    def __init__(self):
        self.heap = []

    def _parent(self, i):
        return (i - 1) // 2

    def _left_child(self, i):
        return 2 * i + 1

    def _right_child(self, i):
        return 2 * i + 2

    def _swap(self, i, j):
        self.heap[i], self.heap[j] = self.heap[j], self.heap[i]

    def _heapify_up(self, i):
        """
        Moves the element at index i up the heap to maintain the min-heap
property.
        """
        while i > 0 and self.heap[self._parent(i)] > self.heap[i]:
            self._swap(i, self._parent(i))
            i = self._parent(i)

    def _heapify_down(self, i):
```

```
        """
        Moves the element at index i down the heap to maintain the min-heap
property.
        """
        min_index = i
        left = self._left_child(i)
        right = self._right_child(i)
        n = len(self.heap)

        if left < n and self.heap[left] < self.heap[min_index]:
            min_index = left
        if right < n and self.heap[right] < self.heap[min_index]:
            min_index = right

        if min_index != i:
            self._swap(i, min_index)
            self._heapify_down(min_index)

    def insert(self, key):
        """
        Inserts a new key into the min-heap.
        """
        self.heap.append(key)
        self._heapify_up(len(self.heap) - 1)
        print(f"Inserted {key} into Min-Heap. Current Heap: {self.heap}")

    def extract_min(self):
        """
        Removes and returns the minimum element from the min-heap (root).
        """
        if not self.heap:
            raise IndexError("Heap is empty")

        if len(self.heap) == 1:
            min_val = self.heap.pop()
            print(f"Extracted Min: {min_val}. Current Heap: {self.heap}")
            return min_val

        min_val = self.heap[0]
        self.heap[0] = self.heap.pop() # Move last element to root
        self._heapify_down(0)
        print(f"Extracted Min: {min_val}. Current Heap: {self.heap}")
        return min_val

    def peek_min(self):
        """Returns the minimum element without removing it."""
        if not self.heap:
            raise IndexError("Heap is empty")
        return self.heap[0]

    def is_empty(self):
        return len(self.heap) == 0

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

class MaxHeap:
    """
    Implements a Max-Heap data structure.
    """
    def __init__(self):
        self.heap = []

    def _parent(self, i):
        return (i - 1) // 2
```

```
    def _left_child(self, i):
        return 2 * i + 1

    def _right_child(self, i):
        return 2 * i + 2

    def _swap(self, i, j):
        self.heap[i], self.heap[j] = self.heap[j], self.heap[i]

    def _heapify_up(self, i):
        """
        Moves the element at index i up the heap to maintain the max-heap
property.
        """
        while i > 0 and self.heap[self._parent(i)] < self.heap[i]:
            self._swap(i, self._parent(i))
            i = self._parent(i)

    def _heapify_down(self, i):
        """
        Moves the element at index i down the heap to maintain the max-heap
property.
        """
        max_index = i
        left = self._left_child(i)
        right = self._right_child(i)
        n = len(self.heap)

        if left < n and self.heap[left] > self.heap[max_index]:
            max_index = left
        if right < n and self.heap[right] > self.heap[max_index]:
            max_index = right

        if max_index != i:
            self._swap(i, max_index)
            self._heapify_down(max_index)

    def insert(self, key):
        """
        Inserts a new key into the max-heap.
        """
        self.heap.append(key)
        self._heapify_up(len(self.heap) - 1)
        print(f"Inserted {key} into Max-Heap. Current Heap: {self.heap}")

    def extract_max(self):
        """
        Removes and returns the maximum element from the max-heap (root).
        """
        if not self.heap:
            raise IndexError("Heap is empty")

        if len(self.heap) == 1:
            max_val = self.heap.pop()
            print(f"Extracted Max: {max_val}. Current Heap: {self.heap}")
            return max_val

        max_val = self.heap[0]
        self.heap[0] = self.heap.pop() # Move last element to root
        self._heapify_down(0)
        print(f"Extracted Max: {max_val}. Current Heap: {self.heap}")
        return max_val

    def peek_max(self):
        """Returns the maximum element without removing it."""
        if not self.heap:
```

```
            raise IndexError("Heap is empty")
        return self.heap[0]

    def is_empty(self):
        return len(self.heap) == 0

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

# Main part of the program to test Min-Heap and Max-Heap implementations
if __name__ == "__main__":
    print("--- Min-Heap Operations ---")
    min_heap = MinHeap()
    min_heap.insert(3)
    min_heap.insert(1)
    min_heap.insert(4)
    min_heap.insert(2)
    min_heap.insert(5)

    print(f"Min-Heap size: {min_heap.size()}")
    print(f"Min element (peek): {min_heap.peek_min()}")
    min_heap.extract_min() # Should extract 1
    min_heap.extract_min() # Should extract 2
    print(f"Min-Heap size: {min_heap.size()}")
    print(f"Is Min-Heap empty? {min_heap.is_empty()}")
    min_heap.extract_min() # Should extract 3
    min_heap.extract_min() # Should extract 4
    min_heap.extract_min() # Should extract 5
    print(f"Is Min-Heap empty? {min_heap.is_empty()}")

    try:
        min_heap.extract_min()
    except IndexError as e:
        print(f"Error: {e}")

    print("\n--- Max-Heap Operations ---")
    max_heap = MaxHeap()
    max_heap.insert(3)
    max_heap.insert(1)
    max_heap.insert(4)
    max_heap.insert(2)
    max_heap.insert(5)

    print(f"Max-Heap size: {max_heap.size()}")
    print(f"Max element (peek): {max_heap.peek_max()}")
    max_heap.extract_max() # Should extract 5
    max_heap.extract_max() # Should extract 4
    print(f"Max-Heap size: {max_heap.size()}")
    print(f"Is Max-Heap empty? {max_heap.is_empty()}")
    max_heap.extract_max() # Should extract 3
    max_heap.extract_max() # Should extract 2
    max_heap.extract_max() # Should extract 1
    print(f"Is Max-Heap empty? {max_heap.is_empty()}")

    try:
        max_heap.extract_max()
    except IndexError as e:
        print(f"Error: {e}")
```

Input: (No direct user input for this program, operations are hardcoded)

Expected Output:

```
--- Min-Heap Operations ---
```

```
Inserted 3 into Min-Heap. Current Heap: [3]
Inserted 1 into Min-Heap. Current Heap: [1, 3]
Inserted 4 into Min-Heap. Current Heap: [1, 3, 4]
Inserted 2 into Min-Heap. Current Heap: [1, 2, 4, 3]
Inserted 5 into Min-Heap. Current Heap: [1, 2, 4, 3, 5]
Min-Heap size: 5
Min element (peek): 1
Extracted Min: 1. Current Heap: [2, 3, 4, 5]
Extracted Min: 2. Current Heap: [3, 5, 4]
Min-Heap size: 3
Is Min-Heap empty? False
Extracted Min: 3. Current Heap: [4, 5]
Extracted Min: 4. Current Heap: [5]
Extracted Min: 5. Current Heap: []
Is Min-Heap empty? True
Error: Heap is empty

--- Max-Heap Operations ---
Inserted 3 into Max-Heap. Current Heap: [3]
Inserted 1 into Max-Heap. Current Heap: [3, 1]
Inserted 4 into Max-Heap. Current Heap: [4, 1, 3]
Inserted 2 into Max-Heap. Current Heap: [4, 2, 3, 1]
Inserted 5 into Max-Heap. Current Heap: [5, 4, 3, 1, 2]
Max-Heap size: 5
Max element (peek): 5
Extracted Max: 5. Current Heap: [4, 2, 3, 1]
Extracted Max: 4. Current Heap: [3, 2, 1]
Max-Heap size: 3
Is Max-Heap empty? False
Extracted Max: 3. Current Heap: [2, 1]
Extracted Max: 2. Current Heap: [1]
Extracted Max: 1. Current Heap: []
Is Max-Heap empty? True
Error: Heap is empty
```

Lab 10: Implementation of Bubble and Insertion Sort

Title: Sorting Algorithms: Bubble Sort and Insertion Sort

Aim: To implement and compare the Bubble Sort and Insertion Sort algorithms for arranging elements in ascending order.

Procedure:

1. **Bubble Sort:**
   - Iterate through the array from the first element to the second to last.
   - In each pass, compare adjacent elements and swap them if they are in the wrong order (larger element bubbles up).
   - Repeat passes until no swaps are needed in a full pass.
2. **Insertion Sort:**
   - Start from the second element.
   - For each element, compare it with elements to its left.
   - Shift elements greater than the current element one position to the right to make space.
   - Insert the current element into its correct sorted position.
3. Test both algorithms with sample arrays and print the sorted results.

Source Code:

```
# bubble_insertion_sort.py

def bubble_sort(arr):
    """
    Sorts an array using the Bubble Sort algorithm.
    Time Complexity: O(n^2) in worst and average case.
    Space Complexity: O(1).
    """
    n = len(arr)
    # Traverse through all array elements
    for i in range(n - 1):
        # Last i elements are already in place
        swapped = False
        for j in range(n - 1 - i):
            # Traverse the array from 0 to n-i-1
            # Swap if the element found is greater than the next element
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                swapped = True
        # If no two elements were swapped by inner loop, then break
        if not swapped:
            break
    return arr

def insertion_sort(arr):
    """
    Sorts an array using the Insertion Sort algorithm.
    Time Complexity: O(n^2) in worst and average case, O(n) in best case.
    Space Complexity: O(1).
    """
    # Traverse through 1 to len(arr)
    for i in range(1, len(arr)):
        key = arr[i]
        # Move elements of arr[0..i-1], that are greater than key,
        # to one position ahead of their current position
        j = i - 1
        while j >= 0 and key < arr[j]:
```

```
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
    return arr

# Main part of the program to test sorting algorithms
if __name__ == "__main__":
    # Test Bubble Sort
    arr_bubble = [64, 34, 25, 12, 22, 11, 90]
    print(f"Original array for Bubble Sort: {arr_bubble}")
    bubble_sort(arr_bubble)
    print(f"Sorted array (Bubble Sort): {arr_bubble}") # Expected: [11, 12,
22, 25, 34, 64, 90]

    arr_bubble_2 = [5, 1, 4, 2, 8]
    print(f"Original array for Bubble Sort: {arr_bubble_2}")
    bubble_sort(arr_bubble_2)
    print(f"Sorted array (Bubble Sort): {arr_bubble_2}") # Expected: [1, 2,
4, 5, 8]

    print("-" * 30)

    # Test Insertion Sort
    arr_insertion = [12, 11, 13, 5, 6]
    print(f"Original array for Insertion Sort: {arr_insertion}")
    insertion_sort(arr_insertion)
    print(f"Sorted array (Insertion Sort): {arr_insertion}") # Expected: [5,
6, 11, 12, 13]

    arr_insertion_2 = [7, 8, 9, 1, 2, 3]
    print(f"Original array for Insertion Sort: {arr_insertion_2}")
    insertion_sort(arr_insertion_2)
    print(f"Sorted array (Insertion Sort): {arr_insertion_2}") # Expected:
[1, 2, 3, 7, 8, 9]
```

Input: (No direct user input for this program, arrays are hardcoded)

Expected Output:

```
Original array for Bubble Sort: [64, 34, 25, 12, 22, 11, 90]
Sorted array (Bubble Sort): [11, 12, 22, 25, 34, 64, 90]
Original array for Bubble Sort: [5, 1, 4, 2, 8]
Sorted array (Bubble Sort): [1, 2, 4, 5, 8]
------------------------------
Original array for Insertion Sort: [12, 11, 13, 5, 6]
Sorted array (Insertion Sort): [5, 6, 11, 12, 13]
Original array for Insertion Sort: [7, 8, 9, 1, 2, 3]
Sorted array (Insertion Sort): [1, 2, 3, 7, 8, 9]
```

Lab 11: Implementation of Quicksort and Mergesort

Title: Advanced Sorting Algorithms: Quicksort and Mergesort

Aim: To implement and understand the principles of Quicksort and Mergesort, two efficient comparison-based sorting algorithms.

Procedure:

1. **Quicksort:**
   - Choose a pivot element from the array.
   - Partition the array into two sub-arrays: elements smaller than the pivot and elements greater than the pivot.
   - Recursively apply Quicksort to the two sub-arrays.
2. **Mergesort:**
   - Divide the unsorted list into `n` sublists, each containing one element (a list of one element is considered sorted).
   - Repeatedly merge sublists to produce new sorted sublists until there is only one sublist remaining. This will be the sorted list.
3. Test both algorithms with sample arrays and print the sorted results.

Source Code:

```
# quick_merge_sort.py

def quick_sort(arr):
    """
    Sorts an array using the Quicksort algorithm.
    Time Complexity: O(n log n) on average, O(n^2) in worst case.
    Space Complexity: O(log n) for recursion stack on average.
    """
    if len(arr) <= 1:
        return arr
    else:
        pivot = arr[len(arr) // 2] # Choose middle element as pivot
        left = [x for x in arr if x < pivot]
        middle = [x for x in arr if x == pivot]
        right = [x for x in arr if x > pivot]
        return quick_sort(left) + middle + quick_sort(right)

def merge_sort(arr):
    """
    Sorts an array using the Mergesort algorithm.
    Time Complexity: O(n log n) in all cases.
    Space Complexity: O(n) for temporary arrays.
    """
    if len(arr) <= 1:
        return arr

    mid = len(arr) // 2
    left_half = arr[:mid]
    right_half = arr[mid:]

    left_half = merge_sort(left_half)
    right_half = merge_sort(right_half)

    return merge(left_half, right_half)

def merge(left, right):
    """
    Merges two sorted lists into a single sorted list.
```

```
    Helper function for merge_sort.
    """
    result = []
    i = j = 0

    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1

    # Append remaining elements
    result.extend(left[i:])
    result.extend(right[j:])
    return result

# Main part of the program to test sorting algorithms
if __name__ == "__main__":
    # Test Quicksort
    arr_quick = [10, 7, 8, 9, 1, 5]
    print(f"Original array for Quicksort: {arr_quick}")
    sorted_quick = quick_sort(arr_quick)
    print(f"Sorted array (Quicksort): {sorted_quick}") # Expected: [1, 5, 7,
8, 9, 10]

    arr_quick_2 = [3, 1, 4, 1, 5, 9, 2, 6]
    print(f"Original array for Quicksort: {arr_quick_2}")
    sorted_quick_2 = quick_sort(arr_quick_2)
    print(f"Sorted array (Quicksort): {sorted_quick_2}") # Expected: [1, 1,
2, 3, 4, 5, 6, 9]

    print("-" * 30)

    # Test Mergesort
    arr_merge = [38, 27, 43, 3, 9, 82, 10]
    print(f"Original array for Mergesort: {arr_merge}")
    sorted_merge = merge_sort(arr_merge)
    print(f"Sorted array (Mergesort): {sorted_merge}") # Expected: [3, 9, 10,
27, 38, 43, 82]

    arr_merge_2 = [6, 5, 12, 10, 9, 1]
    print(f"Original array for Mergesort: {arr_merge_2}")
    sorted_merge_2 = merge_sort(arr_merge_2)
    print(f"Sorted array (Mergesort): {sorted_merge_2}") # Expected: [1, 5,
6, 9, 10, 12]
```

Input: (No direct user input for this program, arrays are hardcoded)

Expected Output:

```
Original array for Quicksort: [10, 7, 8, 9, 1, 5]
Sorted array (Quicksort): [1, 5, 7, 8, 9, 10]
Original array for Quicksort: [3, 1, 4, 1, 5, 9, 2, 6]
Sorted array (Quicksort): [1, 1, 2, 3, 4, 5, 6, 9]
------------------------------
Original array for Mergesort: [38, 27, 43, 3, 9, 82, 10]
Sorted array (Mergesort): [3, 9, 10, 27, 38, 43, 82]
Original array for Mergesort: [6, 5, 12, 10, 9, 1]
Sorted array (Mergesort): [1, 5, 6, 9, 10, 12]
```

Lab 12: Linear search and Binary search

Title: Searching Algorithms: Linear Search and Binary Search

Aim: To implement and compare Linear Search and Binary Search algorithms for finding a specific element in a list.

Procedure:

1. **Linear Search:**
   - Iterate through each element of the list from the beginning.
   - Compare each element with the target value.
   - If a match is found, return its index. If the end of the list is reached without a match, return an indicator (e.g., -1).
2. **Binary Search:**
   - **Pre-requisite:** The list must be sorted.
   - Find the middle element of the list.
   - If the middle element is the target, return its index.
   - If the target is smaller, search in the left half.
   - If the target is larger, search in the right half.
   - Repeat until the element is found or the sub-array becomes empty.
3. Test both algorithms with sample lists and print the results.

Source Code:

```
# linear_binary_search.py

def linear_search(arr, target):
    """
    Performs a linear search to find the target element in the array.
    Time Complexity: O(n) in worst and average case.
    Space Complexity: O(1).
    """
    for i in range(len(arr)):
        if arr[i] == target:
            return i # Return the index if found
    return -1 # Return -1 if not found

def binary_search(arr, target):
    """
    Performs a binary search to find the target element in a sorted array.
    Time Complexity: O(log n).
    Space Complexity: O(1) (iterative) or O(log n) (recursive).
    """
    left, right = 0, len(arr) - 1

    while left <= right:
        mid = (left + right) // 2
        if arr[mid] == target:
            return mid # Return the index if found
        elif arr[mid] < target:
            left = mid + 1 # Search in the right half
        else: # arr[mid] > target
            right = mid - 1 # Search in the left half
    return -1 # Return -1 if not found

# Main part of the program to test search algorithms
if __name__ == "__main__":
    # Test Linear Search
    my_list_linear = [5, 1, 9, 2, 7, 3, 8, 4, 6]
```

```
    target1 = 7
    target2 = 10
    print(f"List for Linear Search: {my_list_linear}")

    index1 = linear_search(my_list_linear, target1)
    if index1 != -1:
        print(f"Linear Search: {target1} found at index {index1}")
    else:
        print(f"Linear Search: {target1} not found")

    index2 = linear_search(my_list_linear, target2)
    if index2 != -1:
        print(f"Linear Search: {target2} found at index {index2}")
    else:
        print(f"Linear Search: {target2} not found")

    print("-" * 30)

    # Test Binary Search (requires sorted list)
    my_list_binary = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
    target3 = 4
    target4 = 11
    print(f"List for Binary Search (must be sorted): {my_list_binary}")

    index3 = binary_search(my_list_binary, target3)
    if index3 != -1:
        print(f"Binary Search: {target3} found at index {index3}")
    else:
        print(f"Binary Search: {target3} not found")

    index4 = binary_search(my_list_binary, target4)
    if index4 != -1:
        print(f"Binary Search: {target4} found at index {index4}")
    else:
        print(f"Binary Search: {target4} not found")
```

Input: (No direct user input for this program, lists and targets are hardcoded)

Expected Output:

```
List for Linear Search: [5, 1, 9, 2, 7, 3, 8, 4, 6]
Linear Search: 7 found at index 4
Linear Search: 10 not found
------------------------------
List for Binary Search (must be sorted): [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Binary Search: 4 found at index 3
Binary Search: 11 not found
```

Lab 13: Implementation of Graph using Array

Title: Graph Representation using Adjacency Matrix (Array)

Aim: To implement a graph data structure using an adjacency matrix (2D array/list of lists) to represent connections between vertices.

Procedure:

1. Represent the graph as a 2D list (matrix) where `matrix[i][j] = 1` if there is an edge from vertex `i` to vertex `j`, and `0` otherwise.
2. Initialize the matrix with all zeros.
3. Implement a `add_edge` method to add a connection between two vertices by setting the corresponding matrix entry to 1. For an undirected graph, set both `matrix[u][v]` and `matrix[v][u]` to 1.
4. Implement a `print_graph` method to display the adjacency matrix.

Source Code:

```
# graph_adjacency_matrix.py

class Graph:
    """
    Implements a graph using an adjacency matrix representation.
    Assumes a fixed number of vertices.
    """
    def __init__(self, num_vertices):
        self.num_vertices = num_vertices
        # Initialize the adjacency matrix with all zeros
        self.adj_matrix = [[0] * num_vertices for _ in range(num_vertices)]
        print(f"Graph initialized with {num_vertices} vertices.")

    def add_edge(self, u, v, weight=1, directed=False):
        """
        Adds an edge between vertex u and vertex v.
        Args:
            u: Starting vertex (0-indexed).
            v: Ending vertex (0-indexed).
            weight: Weight of the edge (default 1 for unweighted).
            directed: If True, adds a directed edge (u -> v).
                      If False, adds an undirected edge (u <-> v).
        """
        if 0 <= u < self.num_vertices and 0 <= v < self.num_vertices:
            self.adj_matrix[u][v] = weight
            if not directed:
                self.adj_matrix[v][u] = weight # For undirected graph
            print(f"Added edge: {u} {'->' if directed else '<->'} {v}
(Weight: {weight})")
        else:
            print(f"Error: Vertices {u}, {v} are out of bounds.")

    def print_graph(self):
        """
        Prints the adjacency matrix representation of the graph.
        """
        print("\nAdjacency Matrix:")
        for row in self.adj_matrix:
            print(row)

# Main part of the program to test graph implementation
if __name__ == "__main__":
```

```python
    # Create a graph with 5 vertices
    g = Graph(5)

    # Add undirected edges
    g.add_edge(0, 1) # 0 <-> 1
    g.add_edge(0, 4) # 0 <-> 4
    g.add_edge(1, 2) # 1 <-> 2
    g.add_edge(1, 3) # 1 <-> 3
    g.add_edge(1, 4) # 1 <-> 4
    g.add_edge(2, 3) # 2 <-> 3
    g.add_edge(3, 4) # 3 <-> 4

    g.print_graph()

    print("\n--- Directed Graph Example ---")
    g_directed = Graph(3)
    g_directed.add_edge(0, 1, directed=True) # 0 -> 1
    g_directed.add_edge(1, 2, directed=True) # 1 -> 2
    g_directed.add_edge(2, 0, directed=True) # 2 -> 0
    g_directed.add_edge(0, 2, weight=5, directed=True) # 0 -> 2 with weight 5

    g_directed.print_graph()
```

Input: (No direct user input for this program, graph structure is hardcoded)

Expected Output:

```
Graph initialized with 5 vertices.
Added edge: 0 <-> 1 (Weight: 1)
Added edge: 0 <-> 4 (Weight: 1)
Added edge: 1 <-> 2 (Weight: 1)
Added edge: 1 <-> 3 (Weight: 1)
Added edge: 1 <-> 4 (Weight: 1)
Added edge: 2 <-> 3 (Weight: 1)
Added edge: 3 <-> 4 (Weight: 1)

Adjacency Matrix:
[0, 1, 0, 0, 1]
[1, 0, 1, 1, 1]
[0, 1, 0, 1, 0]
[0, 1, 1, 0, 1]
[1, 1, 0, 1, 0]

--- Directed Graph Example ---
Graph initialized with 3 vertices.
Added edge: 0 -> 1 (Weight: 1)
Added edge: 1 -> 2 (Weight: 1)
Added edge: 2 -> 0 (Weight: 1)
Added edge: 0 -> 2 (Weight: 5)

Adjacency Matrix:
[0, 1, 5]
[0, 0, 1]
[1, 0, 0]
```

Lab 14: Implementation of shortest path algorithm

Title: Shortest Path Algorithm: Dijkstra's Algorithm

Aim: To implement Dijkstra's algorithm to find the shortest path from a single source vertex to all other vertices in a weighted, directed graph.

Procedure:

1. Represent the graph using an adjacency matrix or adjacency list (adjacency matrix is used here for consistency with Lab 13).
2. Initialize `distances` to infinity for all vertices except the source (0 for source).
3. Initialize `visited` set to keep track of processed vertices.
4. Use a min-priority queue (or simply iterate to find minimum distance) to select the unvisited vertex with the smallest distance.
5. For the selected vertex, update the distances of its neighbors if a shorter path is found through the current vertex.
6. Repeat until all vertices are visited or all reachable vertices have minimum distances calculated.

Source Code:

```
# dijkstra_shortest_path.py

import sys

class Graph:
    """
    Represents a graph using an adjacency matrix for Dijkstra's algorithm.
    """
    def __init__(self, vertices):
        self.V = vertices
        self.graph = [[0 for column in range(vertices)]
                      for row in range(vertices)]

    def add_edge(self, u, v, weight):
        """Adds a directed edge with a given weight."""
        self.graph[u][v] = weight

    def print_solution(self, dist, src):
        """Prints the calculated shortest distances from the source."""
        print(f"\nShortest distances from source vertex {src}:")
        for node in range(self.V):
            if dist[node] == sys.maxsize:
                print(f"Vertex {node}: INF (unreachable)")
            else:
                print(f"Vertex {node}: {dist[node]}")

    def dijkstra(self, src):
        """
        Implements Dijkstra's shortest path algorithm for a single source.
        Args:
            src: The source vertex (0-indexed).
        """
        dist = [sys.maxsize] * self.V # Initialize distances to infinity
        dist[src] = 0                 # Distance from source to itself is 0
        spt_set = [False] * self.V    # spt_set[i] is True if vertex i is
included in shortest path tree

        for count in range(self.V):
```

```
            # Pick the minimum distance vertex from the set of vertices not
yet processed.
            # u is always equal to src in first iteration
            min_dist = sys.maxsize
            min_index = -1

            for v in range(self.V):
                if dist[v] < min_dist and not spt_set[v]:
                    min_dist = dist[v]
                    min_index = v

            # If no reachable unvisited vertex is found, break
            if min_index == -1:
                break

            u = min_index
            spt_set[u] = True # Mark the picked vertex as processed

            # Update dist value of the adjacent vertices of the picked vertex
            for v in range(self.V):
                # Only update if:
                # 1. v is not in spt_set
                # 2. There is an edge from u to v
                # 3. Total weight of path from src to v through u is smaller
than current dist[v]
                if (not spt_set[v] and self.graph[u][v] > 0 and
                        dist[u] != sys.maxsize and
                        dist[u] + self.graph[u][v] < dist[v]):
                    dist[v] = dist[u] + self.graph[u][v]

        self.print_solution(dist, src)

# Main part of the program to test Dijkstra's algorithm
if __name__ == "__main__":
    # Create a graph with 6 vertices
    # Example graph from GeeksforGeeks
    # (0) --10--> (1) --10--> (2)
    #  | \         |           |
    #  6  \        15          4
    #  |   \       |           |
    # (3) --4--> (4) --5--> (5)
    g = Graph(6)
    g.add_edge(0, 1, 10)
    g.add_edge(0, 3, 6)
    g.add_edge(0, 4, 15) # Example: direct edge to 4
    g.add_edge(1, 2, 10)
    g.add_edge(1, 4, 15)
    g.add_edge(2, 5, 4)
    g.add_edge(3, 4, 4)
    g.add_edge(4, 5, 5)

    source_vertex = 0
    print(f"Running Dijkstra's algorithm from source vertex {source_vertex}")
    g.dijkstra(source_vertex)

    print("\n--- Another Example ---")
    g2 = Graph(4)
    g2.add_edge(0, 1, 1)
    g2.add_edge(0, 2, 4)
    g2.add_edge(1, 2, 2)
    g2.add_edge(1, 3, 5)
    g2.add_edge(2, 3, 1)
    g2.dijkstra(0)
```

Input: (No direct user input for this program, graph and source are hardcoded)

Expected Output:

```
Running Dijkstra's algorithm from source vertex 0

Shortest distances from source vertex 0:
Vertex 0: 0
Vertex 1: 10
Vertex 2: 20
Vertex 3: 6
Vertex 4: 10
Vertex 5: 15

--- Another Example ---

Shortest distances from source vertex 0:
Vertex 0: 0
Vertex 1: 1
Vertex 2: 3
Vertex 3: 4
```

Lab 15: Implementation of minimum spanning tree

Title: Minimum Spanning Tree: Prim's Algorithm

Aim: To implement Prim's algorithm to find the Minimum Spanning Tree (MST) of a connected, undirected, weighted graph.

Procedure:

1. Represent the graph using an adjacency matrix.
2. Initialize `key` values (minimum edge weight to connect to MST) to infinity for all vertices, and 0 for the starting vertex.
3. Initialize `mst_set` to keep track of vertices already included in MST.
4. Initialize `parent` array to store the MST structure.
5. Repeat `V` times (where `V` is the number of vertices):
   - Select the vertex `u` not yet in `mst_set` that has the minimum `key` value.
   - Add `u` to `mst_set`.
   - For each neighbor `v` of `u`: if `v` is not in `mst_set` and the weight of edge `(u, v)` is less than `key[v]`, update `key[v]` and set `parent[v] = u`.
6. Print the edges of the MST and its total weight.

Source Code:

```
# prims_mst.py

import sys

class Graph:
    """
    Represents a graph for Prim's algorithm using an adjacency matrix.
    """
    def __init__(self, vertices):
        self.V = vertices
        self.graph = [[0 for column in range(vertices)]
                      for row in range(vertices)]

    def add_edge(self, u, v, weight):
        """Adds an undirected edge with a given weight."""
        self.graph[u][v] = weight
        self.graph[v][u] = weight # For undirected graph

    def print_mst(self, parent):
        """Prints the constructed MST edges and total weight."""
        print("\nEdges in Minimum Spanning Tree:")
        total_weight = 0
        for i in range(1, self.V):
            print(f"{parent[i]} - {i}    Weight: {self.graph[i][parent[i]]}")
            total_weight += self.graph[i][parent[i]]
        print(f"\nTotal weight of MST: {total_weight}")

    def prim_mst(self):
        """
        Implements Prim's algorithm to find the Minimum Spanning Tree.
        """
        key = [sys.maxsize] * self.V # Key values used to pick minimum weight
edge in cut
        parent = [-1] * self.V       # Array to store constructed MST
        mst_set = [False] * self.V   # To represent set of vertices not yet
included in MST
```

```
        key[0] = 0 # Make key 0 so that this vertex is picked first
        parent[0] = -1 # First node is always root of MST

        for count in range(self.V):
            # Pick the minimum key vertex from the set of vertices
            # not yet included in MST
            min_key = sys.maxsize
            min_index = -1

            for v in range(self.V):
                if key[v] < min_key and not mst_set[v]:
                    min_key = key[v]
                    min_index = v

            # If no reachable unvisited vertex is found, break
            if min_index == -1:
                break

            u = min_index
            mst_set[u] = True # Add the picked vertex to the MST set

            # Update key value and parent index of the adjacent vertices of
the picked vertex.
            # Consider only those vertices which are not yet in MST set.
            for v in range(self.V):
                # graph[u][v] is non-zero only for adjacent vertices of u
                # mst_set[v] is false for vertices not yet included in MST
                # Update the key only if graph[u][v] is smaller than key[v]
                if self.graph[u][v] > 0 and not mst_set[v] and key[v] >
self.graph[u][v]:
                    key[v] = self.graph[u][v]
                    parent[v] = u

        self.print_mst(parent)

# Main part of the program to test Prim's algorithm
if __name__ == "__main__":
    # Create a graph with 5 vertices
    # Example graph:
    # (0) --2-- (1) --3-- (2)
    #  | \       |       / |
    #  6   8     8     5   7
    #  |     \   |   /     |
    # (3) --9-- (4) -------
    g = Graph(5)
    g.add_edge(0, 1, 2)
    g.add_edge(0, 3, 6)
    g.add_edge(1, 2, 3)
    g.add_edge(1, 3, 8) # Not part of typical MST for this example
    g.add_edge(1, 4, 5)
    g.add_edge(2, 4, 7)
    g.add_edge(3, 4, 9)

    print("Running Prim's algorithm for MST...")
    g.prim_mst()

    print("\n--- Another Example ---")
    g2 = Graph(4)
    g2.add_edge(0, 1, 10)
    g2.add_edge(0, 2, 6)
    g2.add_edge(0, 3, 5)
    g2.add_edge(1, 3, 15)
    g2.add_edge(2, 3, 4)
    g2.prim_mst()
```

Input: (No direct user input for this program, graph structure is hardcoded)

Expected Output:

```
Running Prim's algorithm for MST...

Edges in Minimum Spanning Tree:
0 - 1    Weight: 2
1 - 2    Weight: 3
0 - 3    Weight: 6
1 - 4    Weight: 5

Total weight of MST: 16

--- Another Example ---

Edges in Minimum Spanning Tree:
0 - 1    Weight: 10
3 - 2    Weight: 4
0 - 3    Weight: 5

Total weight of MST: 19
```