Sri
SAI RAM ENGINEERING COLLEGE
An Autonomous Institution | Affiliated to Anna University & Approved by AICTE, New Delhi
Accredited by NBA and NAAC 'A+' | BIS/EOMS ISO 21001 : 2018 and BVQI 9001 : 2015 Certified and NIRF ranked institution
Sai Leo Nagar, West Tambaram, Chennai - 600 044. www.sairam.edu.in

LAB MANUAL

24AMPL302-DATA STRUCTURES AND ALGORITHM DESIGN LABORATORY

(II YEAR / III SEM)

(BATCH: 2024 – 2028)

ACADEMIC YEAR: 2025 – 2026

ODD SEMESTER

DEPARTMENT OF CSE (ARTIFICIAL INTELLIGENCE & MACHINE LEARNING)

PREFACE

“PRACTICE LEADS TO PERFECTION”

Practical work encourages and familiarizes students with the latest tools that will be required to become experts. Taking this into consideration, this manual is compiled as a preparatory note for the **DATA STRUCTURES AND ALGORITHM DESIGN LABORATORY** Laboratory programs. Sufficient details have been included to impart self-learning.

This manual is intended for the III semester CSE (AI&ML) students under Autonomous Regulations 2024. Introductory information and procedure to perform is detailed in this manual.

It is expected that this will help the students develop a broad understanding and learn quick adaptations to real-time problems.

INSTITUTE VISION

To emerge as a "Centre of excellence" offering Technical Education and Research opportunities of very high standards to students, develop the total personality of the individual and instill high levels of disciple and strive to set global standards, making our students technologically superior and ethically stronger, who in turn shall contribute to the advancement of society and humankind.

INSTITUTION MISSION

We dedicate and commit ourselves to achieve, sustain and foster unmatched excellence in Technical Education. To this end, we will pursue continuous development of infra-structure and enhance state-of-art equipment to provide our students a technologically up-to date and intellectually inspiring environment of learning, research, creativity, innovation and professional activity and inculcate in them ethical and moral values.

INSTITUTE POLICY

We at Sri Sai Ram Engineering College are committed to build a better Nation through Quality Education with team spirit. Our students are enabled to excel in all values of Life and become Good Citizens. We continually improve the System, Infrastructure and Service to satisfy the Students, Parents, Industry and Society.

DEPARTMENT VISION

To emerge as a "Centre of Excellence" in the field of Artificial Intelligence and Machine Learning by providing required skill sets, domain expertise and interactive industry interface for students and shape them to be a socially conscious and responsible citizen.

DEPARTMENT MISSION

Computer Science and Engineering (Artificial Intelligence and Machine Learning), Sri Sairam Engineering College is committed to

M1: Nurture students with a sound understanding of fundamentals, theory and practice of Artificial Intelligence and Machine Learning.

M2: Develop students with the required skill sets and enable them to take up assignments in the field of AI & ML

M3: Facilitate Industry Academia interface to update the recent trends in AI &ML

M4: Create an appropriate environment to bring out the latent talents, creativity and innovation among students to contribute to the society

Program Educational Objectives(PEOs)

To prepare the graduates to:

1. Graduates imbibe fundamental knowledge in Artificial Intelligence, Programming, Mathematical modelling and Machine Learning.
2. Graduates will be trained to gain domain expertise by applying the theory basics into practical situation through simulation and modelling techniques.
3. Graduates will enhance the capability through skill development and make them industry ready by inculcating leadership and multitasking abilities

4. Graduates will apply the gained knowledge of AI & ML in Research & Development, Innovation and contribute to the society in making things simpler.

Program Outcomes (PO's)

The graduates in Engineering will:

PO1: Engineering Knowledge: Apply knowledge of mathematics, natural science, computing, engineering fundamentals and an engineering specialization as specified in WK1 to WK4 respectively to develop to the solution of complex engineering problems.

PO2: Problem Analysis: Identify, formulate, review research literature and analyze complex engineering problems reaching substantiated conclusions with consideration for sustainable development. (WK1 to WK4)

PO3: Design/Development of Solutions: Design creative solutions for complex engineering problems and design/develop systems/components/processes to meet identified needs with consideration for the public health and safety, whole-life cost, net zero carbon, culture, society and environment as required. (WK5)

PO4: Conduct Investigations of Complex Problems: Conduct investigations of complex engineering problems using research-based knowledge including design of experiments, modelling, analysis & interpretation of data to provide valid conclusions. (WK8).

PO5: Engineering Tool Usage: Create, select and apply appropriate techniques, resources and modern engineering & IT tools, including prediction and modelling recognizing their limitations to solve complex engineering problems. (WK2 and WK6)

PO6: The Engineer and The World: Analyze and evaluate societal and environmental aspects while solving complex engineering problems for its impact on sustainability with reference to economy, health, safety, legal framework, culture and environment. (WK1, WK5, and WK7).

PO7: Ethics: Apply ethical principles and commit to professional ethics, human values, diversity and inclusion; adhere to national & international laws. (WK9)

PO8: Individual and Collaborative Team work: Function effectively as an individual, and as a member or leader in diverse/multi-disciplinary teams.

PO9: Communication: Communicate effectively and inclusively within the engineering community and society at large, such as being able to comprehend and write effective reports and design documentation,

make effective presentations considering cultural, language, and learning differences

PO10: Project Management and Finance: Apply knowledge and understanding of engineering management principles and economic decision-making and apply these to one's own work, as a member and leader in a team, and to manage projects and in multidisciplinary environments.

PO11: Life-Long Learning: Recognize the need for, and have the preparation and ability for i) independent and life-long learning ii) adaptability to new and emerging technologies and iii) critical thinking in the broadest context of technological change. (WK8)

Program Specific Outcomes (PSOs)

Computer Science and Engineering (Artificial Intelligence and Machine Learning), graduates will be able to:

1. The graduates will be in a position to design, develop, test and deploy appropriate mathematical and programming algorithms required for practical applications.
2. The graduates will have the required skills and domain expertise to provide solutions in

the field of Artificial Intelligence and Machine Learning for the Industry and society at large.

COURSE OUTCOMES

Upon completion of the course, the students will be able to

CO1	Implement and apply linear and non-linear data structures such as stacks, queues, linked lists, trees, and graphs in solving computing problems. (K3)
CO2	Apply searching and sorting techniques to solve problems efficiently, and evaluate their performance in different scenarios. (K4)
CO3	Analyze real-world problems and design suitable data structure-based solutions with optimized algorithmic approaches. (K4)

24AMPL302- DATA STRUCTURES AND ALGORITHM DESIGN LABORATORY

OBJECTIVES

- To implement fundamental data structures like linked lists, stacks, queues, trees, and graphs in Python.
- To construct and manage binary search trees and AVL trees with self-balancing capabilities.
- To develop graph traversal methods using BFS and DFS with adjacency structures.
- To apply Prim's and Dijkstra's algorithms for minimum spanning tree and shortest path problems.
- To perform sorting, and searching using efficient algorithms.
- To demonstrate hashing with collision handling using linear and quadratic probing methods.

LIST OF EXPERIMENTS

1. Implement a Python program to perform addition, subtraction, and multiplication of two polynomials represented as linked lists.
2. Write a Python program to implement a stack using lists with push(), pop(), peek(), isEmpty() and handle underflow errors.
3. Write a Python program to evaluate a postfix expression using a stack, supporting integers and the operators +, -, *, and /.
4. Implement a queue data structure in Python using a custom singly linked list. Include operations such as enqueue(), dequeue(), isEmpty(), and display().
5. Write a Python program to implement a binary search tree with operations for inserting, deleting, and searching for elements.
6. Implement an AVL tree (a self-balancing binary search tree) in Python, with node insertion and automatic balancing using rotations to maintain the AVL property.
7. Write a Python program to represent a graph using an adjacency list or matrix, and implement both Breadth-First Search (BFS) and Depth-First Search (DFS) traversal methods.

8. Implement Prim's algorithm in Python to find the Minimum Spanning Tree of a weighted, undirected graph.

9. Implement Dijkstra's algorithm to find the shortest path from a starting node to all other nodes in a graph with non-negative edge weights.

10. Write a Python program to implement Bubble sort, Selection sort, Insertion sort Algorithms.

11. Write a Python program to implement Linear Search and Binary Search, with input validation (e.g., ensuring sorted input for binary search) and returning the index of the searched element if found.

12. Write a program to implement Linear Probing and Quadratic Probing for inserting and searching elements

INDEX

	validation (e.g., ensuring sorted input for binary search) and returning the index of the searched element if found.	
12	Write a program to implement Linear Probing and Quadratic Probing for inserting and searching elements	56

1. Implement a Python program to perform addition, subtraction, and multiplication of two polynomials represented as linked lists.

AIM

To Implement a Python program to perform addition, subtraction, and multiplication of two polynomials represented as linked lists.

ALGORITHM

Polynomial Representation (Linked List)

Each term of a polynomial is represented as a node containing:

coeff: coefficient of the term

exp: exponent of the term

next: pointer to the next term

Polynomial Addition

Algorithm:

Initialize a new polynomial result.

Traverse both polynomials p1 and p2.

Compare the exponents of current terms.

Add terms with the same exponent; otherwise, copy the higher one.

Insert result into the new list

Polynomial Subraction

Similar to addition.

Subtract coefficients instead of adding when exponents match.

Negate terms from the second polynomial when exponents differ.

Polynomial Multiplication

For every term t1 in poly1, multiply it with every term t2 in poly2.

Insert the resulting term (t1.coeff * t2.coeff, t1.exp + t2.exp) into a temporary polynomial.

Add the temporary polynomial to a result polynomial.

PROGRAM

```
class Node:
    def __init__(self, coeff, exp):
        self.coeff = coeff
        self.exp = exp
        self.next = None


class Polynomial:
    def __init__(self):
        self.head = None


    def insert_term(self, coeff, exp):
        if coeff == 0:
            return


        new_node = Node(coeff, exp)
        if self.head is None or self.head.exp < exp:
            new_node.next = self.head
            self.head = new_node
        else:
            current = self.head
            while current.next and current.next.exp > exp:
                current = current.next
            if current.next and current.next.exp == exp:
```

```
            current.next.coeff += coeff
            if current.next.coeff == 0:
                current.next = current.next.next
        elif current.exp == exp:
            current.coeff += coeff
            if current.coeff == 0:
                self.head = current.next
        else:
            new_node.next = current.next
            current.next = new_node

def add(self, other):
    result = Polynomial()
    p1, p2 = self.head, other.head
    while p1 or p2:
        if p1 and (not p2 or p1.exp > p2.exp):
            result.insert_term(p1.coeff, p1.exp)
            p1 = p1.next
        elif p2 and (not p1 or p2.exp > p1.exp):
            result.insert_term(p2.coeff, p2.exp)
            p2 = p2.next
        else:
            result.insert_term(p1.coeff + p2.coeff, p1.exp)
            p1, p2 = p1.next, p2.next
    return result

def subtract(self, other):
```

```
    result = Polynomial()
    p1, p2 = self.head, other.head
    while p1 or p2:
        if p1 and (not p2 or p1.exp > p2.exp):
            result.insert_term(p1.coeff, p1.exp)
            p1 = p1.next
        elif p2 and (not p1 or p2.exp > p1.exp):
            result.insert_term(-p2.coeff,  p2.exp)
            p2 = p2.next
        else:
            result.insert_term(p1.coeff - p2.coeff, p1.exp)
            p1, p2 = p1.next, p2.next
    return result


def multiply(self, other):
    result = Polynomial()
    p1 = self.head
    while p1:
        temp = Polynomial()
        p2 = other.head
        while p2:
            temp.insert_term(p1.coeff * p2.coeff, p1.exp + p2.exp)
            p2 = p2.next
        result = result.add(temp)
        p1 = p1.next
    return result
```

```
    def display(self):
        terms = []
        current = self.head
        while current:
            if current.coeff > 0 and current != self.head:
                terms.append(f"+ {current.coeff}x^{current.exp}")
            elif current.coeff < 0:
                terms.append(f"- {-current.coeff}x^{current.exp}")
            else:
                terms.append(f"{current.coeff}x^{current.exp}")
            current = current.next
        return " ".join(terms) if terms else "0"

# Example Usage
if __name__ == "__main__":
    # First polynomial: 3x^3 + 2x^2 + x
    poly1 = Polynomial()
    poly1.insert_term(3, 3)
    poly1.insert_term(2, 2)
    poly1.insert_term(1, 1)

    # Second polynomial: 4x^2 + 2x + 1
    poly2 = Polynomial()
    poly2.insert_term(4, 2)
    poly2.insert_term(2, 1)
    poly2.insert_term(1, 0)
```

```
print("Polynomial 1:", poly1.display())
print("Polynomial 2:", poly2.display())

print("Addition:     ", poly1.add(poly2).display())
print("Subtraction: ", poly1.subtract(poly2).display())
print("Multiplication:", poly1.multiply(poly2).display())
```

OUTPUT

```
Polynomial 1: 3x^3 + 2x^2 + 1x^1
Polynomial 2: 4x^2 + 2x^1 + 1x^0
Addition:     3x^3 + 6x^2 + 3x^1 + 1x^0
Subtraction:  3x^3 - 2x^0
Multiplication: 12x^5 + 16x^4 + 11x^3 + 4x^2 + 1x^1
```

RESULT

To Implement a Python program to perform addition, subtraction, and multiplication of two polynomials represented as linked lists is verified

2. Write a Python program to I mplement a stack using lists with push(), pop(), peek(), isEmpty() and handle underflow errors.

AIM

To Write a Python program to implement a stack using lists with push(), pop(), peek(), isEmpty() and handle underflow errors.

```
class StackUnderflowError(Exception):
    """Exception raised when stack operations are performed on an empty stack."""
    Pass
```

PROGRAM

```
class Stack:
    def __init__(self):
        self.stack = []
    def push(self, item):
        """Add item to the top of the stack."""
        self.stack.append(item)
        print(f"Pushed {item} onto the stack.")
    def pop(self):
        """Remove and return the top item from the stack. Raise error if empty."""
        if self.isEmpty():
            raise StackUnderflowError("Stack underflow: Cannot pop from an empty stack.")
        item = self.stack.pop()
        print(f"Popped {item} from the stack.")
        return item
    def peek(self):
        """Return the top item without removing it. Raise error if empty."""
        if self.isEmpty():
```

```
            raise StackUnderflowError("Stack underflow: Cannot peek into an empty stack.")
        print(f"Top item is {self.stack[-1]}")
        return self.stack[-1]
    def isEmpty(self):
        """Return True if stack is empty, else False."""
        return len(self.stack) == 0
    def display(self):
        """Display the stack from top to bottom."""
        if self.isEmpty():
            print("Stack is empty.")
        else:
            print("Stack (top to bottom):", self.stack[::-1])
# Example usage
if __name___== "__main__":
    s = Stack()
       try:
        s.pop()  # Should raise underflow
    except StackUnderflowError as e:
        print(e)
    s.push(10)
    s.push(20)
    s.push(30)
    s.display()
    s.peek()
    s.pop()
    s.display()
    while not s.isEmpty():
```

```
    s.pop()
try:
    s.peek()  # Should raise underflow
except StackUnderflowError as e:
    print€
```

OUTPUT

Stack underflow: Cannot pop from an empty stack.

Pushed 10 onto the stack.

Pushed 20 onto the stack.

Pushed 30 onto the stack.

Stack (top to bottom): [30, 20, 10]

Top item is 30

Popped 30 from the stack.

Stack (top to bottom): [20, 10]

Popped 20 from the stack.

Popped 10 from the stack.

Stack underflow: Cannot peek into an empty stack.

RESULT

To Write a Python program to implement a stack using lists with push(), pop(), peek(), isEmpty() and handle underflow errors is verfied

3. Write a Python program to evaluate a postfix expression using a stack, supporting integers and the operators +, -, *, and /.

AIM

To Write a Python program to evaluate a postfix expression using a stack, supporting integers and the operators +, -, *, and /.

Algorithm to Evaluate a Postfix Expression

1. Initialize an empty stack
2. Read the postfix expression token by token (tokens are usually space-separated)
3. For each token:
    - If the token is a number, push it onto the stack
    - If the token is an operator:
        - Pop the top two elements from the stack (operand2 and operand1)
        - Apply the operator: result = operand1 operator operand2
        - Push the result back onto the stack
4. After processing all tokens, the result will be the only element left in the stack
5. Return the top of the stack as the final result

Python Program

python

CopyEdit

```
def evaluate_postfix(expression):
    stack = []
    operators = {'+', '-', '*', '/'}

    tokens = expression.split()
    for token in tokens:
        if token not in operators:
```

```
            # Push operand to stack
            stack.append(int(token))
        else:
            # Pop two operands
            operand2 = stack.pop()
            operand1 = stack.pop()
            # Apply the operator
            if token == '+':
                result = operand1 + operand2
            elif token == '-':
                result = operand1 - operand2
            elif token == '*':
                result = operand1 * operand2
            elif token == '/':
                # Integer division (float if needed: use float() or operand1 / operand2)
                result = int(operand1 / operand2)

            # Push result back to stack
            stack.append(result)

    return stack[0] if stack else None
# Example usage
expr = "5 1 2 + 4 * + 3 -"  # Equivalent to: 5 + ((1 + 2) * 4) - 3 = 14
print("Result:", evaluate_postfix(expr))
```

OUT PUT

"5 1 2 + 4 * + 3 -":

5 → push

1 → push

2 → push

+ → pop 2 and 1 → 1 + 2 = 3 → push 3

4 → push

* → pop 4 and 3 → 3 * 4 = 12 → push 12

+ → pop 12 and 5 → 5 + 12 = 17 → push 17

3 → push

- → pop 3 and 17 → 17 - 3 = 14

Final Result: 14

RESULT

To Write a Python program to evaluate a postfix expression using a stack, supporting integers and the operators +, -, *, and / is verified .

4. Implement a queue data structure in Python using a custom singly linked list. Include operations such as enqueue(), dequeue(), isEmpty(), and display().

AIM

To Implement a queue data structure in Python using a custom singly linked list. Include operations such as enqueue(), dequeue(), isEmpty(), and display().

Algorithm for Each Operation

1. enqueue(value)

- Create a new node
- If the queue is empty:
  - Set both front and rear to the new node
- Else:
  - Set rear.next to the new node
  - Update rear to the new node

2. dequeue()

- If the queue is empty, return error or None
- Store front.data
- Move front to front.next
- If front becomes None, also set rear = None
- Return stored data

3. isEmpty()

- Return True if front is None, else False

4. display()

- Traverse from front to rear, printing node values

PROGRAM

CopyEdit

```
# Node class for singly linked list
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
# Queue implementation using singly linked list
class Queue:
    def __init__(self):
        self.front = None
        self.rear = None
    # Enqueue operation
    def enqueue(self, value):
        new_node = Node(value)
        if self.rear is None:
            # Queue is empty
            self.front = self.rear = new_node
        else:
            self.rear.next = new_node
            self.rear = new_node

    # Dequeue operation
    def dequeue(self):
        if self.front is None:
            print("Queue is empty. Cannot dequeue.")
```

```
            return None
        result = self.front.data
        self.front = self.front.next
        if self.front is None:
            # Queue became empty after dequeue
            self.rear = None
        return result

    # Check if queue is empty
    def isEmpty(self):
        return self.front is None

    # Display the queue elements
    def display(self):
        if self.isEmpty():
            print("Queue is empty.")
            return
        current = self.front
        while current:
            print(current.data, end=" -> ")
            current = current.next
        print("None")

# Example usage
if __name__ == "__main__":
    q = Queue()
    q.enqueue(10)
```

```
q.enqueue(20)
q.enqueue(30)
q.display()          # Output: 10 -> 20 -> 30 -> None
print("Dequeued:", q.dequeue())  # Output: 10
q.display()          # Output: 20 -> 30 -> None
print("Is Empty?", q.isEmpty())  # Output: False
```

Output

rust

```
10 -> 20 -> 30 -> None
Dequeued: 10
20 -> 30 -> None
Is Empty? False
```

RESULT

To Implement a queue data structure in Python using a custom singly linked list. Include operations such as enqueue(), dequeue(), isEmpty(), and display() is verified.

5. Write a Python program to implement a binary search tree with operations for inserting, deleting, and searching for elements.

AIM

To Write a Python program to implement a binary search tree with operations for inserting, deleting, and searching for elements.

Binary Search Tree (BST) – Overview

A BST is a binary tree where:

- For any node:
  - Values in the left subtree are less than the node's value
  - Values in the right subtree are greater

Algorithms

1. Insert(value)

- If tree is empty, insert at root
- Else:
  - Recursively go left if value < current node
  - Go right if value > current node
  - Insert at the correct position (no duplicates here)

2. Search(value)

- If node is None, return False
- If value == node.data → found
- If value < node.data → search left
- Else → search right

3. Delete(value)

- Three cases:
  1. Node has no children (leaf) → Remove directly
  2. Node has one child → Replace with child

3. Node has two children → Replace with in-order successor (smallest in right subtree)

PROGRAM

python

CopyEdit

```
# Define the node class
class Node:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None


# Define the BST class
class BST:
    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 for a node
def search(self, root, key):
    if root is None or root.key == key:
        return root
    if key < root.key:
        return self.search(root.left, key)
    else:
        return self.search(root.right, key)

# Find minimum value node (used in delete)
def minValueNode(self, node):
    current = node
    while current.left:
        current = current.left
    return current

# Delete a node
def delete(self, root, key):
    if root is None:
        return root
    if key < root.key:
        root.left = self.delete(root.left, key)
    elif key > root.key:
        root.right = self.delete(root.right, key)
    else:
        # Node found
        if root.left is None:
```

```
            return root.right
        elif root.right is None:
            return root.left
        # Node with two children
        temp = self.minValueNode(root.right)
        root.key = temp.key
        root.right = self.delete(root.right, temp.key)
    return root

# In-order traversal (to display the BST)
def inorder(self, root):
    if root:
        self.inorder(root.left)
        print(root.key, end=' ')
        self.inorder(root.right)

# Example usage
if __name___== "__main__":
    tree = BST()
    root = None
    keys = [50, 30, 20, 40, 70, 60, 80]

    # Insert nodes
    for key in keys:
        root = tree.insert(root, key)

    print("In-order traversal of the BST:")
```

```
    tree.inorder(root)  # Output should be sorted: 20 30 40 50 60 70 80

    # Search
    print("\n\nSearch 60:", "Found" if tree.search(root, 60) else "Not found")
    print("Search 90:", "Found" if tree.search(root, 90) else "Not found")

    # Delete a node
    print("\nDeleting 20 (leaf node)...")
    root = tree.delete(root, 20)
    tree.inorder(root)

    print("\n\nDeleting 30 (one child)...")
    root = tree.delete(root, 30)
    tree.inorder(root)

    print("\n\nDeleting 50 (two children)...")
    root = tree.delete(root, 50)
    tree.inorder(root)
```

Output:

sql

CopyEdit

In-order traversal of the BST:

20 30 40 50 60 70 80

Search 60: Found

Search 90: Not found

Deleting 20 (leaf node)...

30 40 50 60 70 80

Deleting 30 (one child)...

40 50 60 70 80

Deleting 50 (two children)...

40 60 70 80

RESULT

To Write a Python program to implement a binary search tree with operations for inserting, deleting, and searching for elements is verified.

6. Implement an AVL tree (a self-balancing binary search tree) in Python, with node insertion and automatic balancing using rotations to maintain the AVL property.

AIM

To Implement an AVL tree (a self-balancing binary search tree) in Python, with node insertion and automatic balancing using rotations to maintain the AVL property.

Algorithm

Step-by-step:

1. Perform a standard BST insertion
2. Update the height of each node during recursion
3. Compute the balance factor:

ini

CopyEdit

balance = height(left subtree) - height(right subtree)

4. If the balance factor is out of range (-1, 0, 1), fix it using rotations:

Four Cases:

- Left Left (LL) → Right Rotate
- Right Right (RR) → Left Rotate
- Left Right (LR) → Left Rotate + Right Rotate
- Right Left (RL) → Right Rotate + Left Rotate

Python Code

python

CopyEdit

```
# Node class
class AVLNode:
    def __init__(self, key):
        self.key = key
```

```
        self.left = None
        self.right = None
        self.height = 1  # New node is initially added at leaf

# AVL Tree class
class AVLTree:

    # Get height of a node
    def get_height(self, node):
        if not node:
            return 0
        return node.height

    # Get balance factor
    def get_balance(self, node):
        if not node:
            return 0
        return self.get_height(node.left) - self.get_height(node.right)

    # Right rotation
    def right_rotate(self, z):
        y = z.left
        T3 = y.right

        # Perform rotation
        y.right = z
        z.left = T3
```

```
    # Update heights
    z.height = 1 + max(self.get_height(z.left), self.get_height(z.right))
    y.height = 1 + max(self.get_height(y.left), self.get_height(y.right))

    return y

# Left rotation
def left_rotate(self, z):
    y = z.right
    T2 = y.left

    # Perform rotation
    y.left = z
    z.right = T2

    # Update heights
    z.height = 1 + max(self.get_height(z.left), self.get_height(z.right))
    y.height = 1 + max(self.get_height(y.left), self.get_height(y.right))

    return y

# Insert node
def insert(self, root, key):
    # 1. Normal BST insertion
    if not root:
        return AVLNode(key)
```

```
elif key < root.key:
    root.left = self.insert(root.left, key)
elif key > root.key:
    root.right = self.insert(root.right, key)
else:
    return root  # Duplicate keys not allowed

# 2. Update height
root.height = 1 + max(self.get_height(root.left), self.get_height(root.right))

# 3. Check balance factor
balance = self.get_balance(root)

# 4. Balance the tree
# Case 1 - Left Left
if balance > 1 and key < root.left.key:
    return self.right_rotate(root)

# Case 2 - Right Right
if balance < -1 and key > root.right.key:
    return self.left_rotate(root)

# Case 3 - Left Right
if balance > 1 and key > root.left.key:
    root.left = self.left_rotate(root.left)
    return self.right_rotate(root)
```

```
        # Case 4 - Right Left
        if balance < -1 and key < root.right.key:
            root.right = self.right_rotate(root.right)
            return self.left_rotate(root)

        return root

    # Inorder traversal (sorted)
    def inorder(self, root):
        if root:
            self.inorder(root.left)
            print(root.key, end=' ')
            self.inorder(root.right)

# ---------------------
# Example usage
if __name___== "__main__":
    tree = AVLTree()
    root = None

    elements = [10, 20, 30, 40, 50, 25]

    for elem in elements:
        root = tree.insert(root, elem)

    print("Inorder traversal of the AVL tree:")
    tree.inorder(root)
```

Output:

objectivec

CopyEdit

Inorder traversal of the AVL tree:

10 20 25 30 40 50

RESULT

To Implement an AVL tree (a self-balancing binary search tree) in Python, with node insertion and automatic balancing using rotations to maintain the AVL property is verifed

7. Write a Python program to represent a graph using an adjacency list or matrix, and implement both Breadth-First Search (BFS) and Depth-First Search (DFS) traversal methods.

AIM

To Write a Python program to represent a graph using an adjacency list or matrix, and implement both Breadth-First Search (BFS) and Depth-First Search (DFS) traversal methods.

Algorithms

- BFS (Breadth-First Search)
  1. Start from a source node
  2. Use a queue to explore nodes level by level
  3. Mark visited nodes to avoid revisiting
  4. Visit all neighbors before going deeper
- DFS (Depth-First Search)
  1. Start from a source node
  2. Use recursion (or a stack) to explore as deep as possible
  3. Backtrack when no unvisited neighbors remain

PROGRAM

python

CopyEdit

```
from collections import deque, defaultdict

class Graph:
    def __init__(self):
        self.adj_list = defaultdict(list)


    # Add edge to the graph (undirected by default)
    def add_edge(self, u, v, directed=False):
```

```
    self.adj_list[u].append(v)
    if not directed:
        self.adj_list[v].append(u)

# Display the graph
def display(self):
    for node in self.adj_list:
        print(f"{node} -> {self.adj_list[node]}")

# Breadth-First Search
def bfs(self, start):
    visited = set()
    queue = deque([start])

    print("\nBFS traversal:")
    while queue:
        node = queue.popleft()
        if node not in visited:
            print(node, end=' ')
            visited.add(node)
            for neighbor in self.adj_list[node]:
                if neighbor not in visited:
                    queue.append(neighbor)

# Depth-First Search (Recursive)
def dfs(self, start):
    visited = set()
```

```
        print("\nDFS traversal:")
        self._dfs_recursive(start, visited)

    def _dfs_recursive(self, node, visited):
        if node not in visited:
            print(node, end=' ')
            visited.add(node)
            for neighbor in self.adj_list[node]:
                if neighbor not in visited:
                    self._dfs_recursive(neighbor,  visited)

# - - - - - - - - - - - - - - - - - - -
# Example usage
if __name___ == "__main__":
    g = Graph()
    g.add_edge(0, 1)
    g.add_edge(0, 2)
    g.add_edge(1, 3)
    g.add_edge(1, 4)
    g.add_edge(2, 5)
    g.add_edge(2, 6)

    print("Graph adjacency list:")
    g.display()

    g.bfs(0)  # Output: 0 1 2 3 4 5 6
    g.dfs(0)  # Output: 0 1 3 4 2 5 6
```

Output Example:

less

CopyEdit

Graph adjacency list:

0 -> [1, 2]

1 -> [0, 3, 4]

2 -> [0, 5, 6]

3 -> [1]

4 -> [1]

5 -> [2]

6 -> [2]

BFS traversal:

0 1 2 3 4 5 6

DFS traversal:

0 1 3 4 2 5 6

RESULT

To Write a Python program to represent a graph using an adjacency list or matrix, and implement both Breadth-First Search (BFS) and Depth-First Search (DFS) traversal methods is verified .

8. Implement Prim's algorithm in Python to find the Minimum Spanning Tree of a weighted, undirected graph.

AIM

To Implement Prim's algorithm in Python to find the Minimum Spanning Tree of a weighted, undirected graph.

Algorithms

Prim's Algorithm – Steps

1. Start with an arbitrary node (say node 0)
2. Use a min-heap (priority queue) to always pick the edge with the smallest weight
3. Maintain a set of visited nodes to avoid cycles
4. For each newly visited node:
    - Add its adjacent edges (that lead to unvisited nodes) into the heap
5. Repeat until all nodes are in the MST

Python Implementation Using Min-Heap

python

CopyEdit

```
import heapq
from collections import defaultdict

class Graph:
    def __init__(self, vertices):
        self.V = vertices
        self.graph = defaultdict(list)  # {u: [(v, weight), ...]}

    def add_edge(self, u, v, weight):
        self.graph[u].append((v, weight))
```

```
    self.graph[v].append((u, weight))  # Because the graph is undirected

def prim_mst(self):
    visited = set()
    min_heap = [(0, 0)]  # (weight, start_node)
    mst_weight = 0
    mst_edges = []

    while len(visited) < self.V and min_heap:
        weight, u = heapq.heappop(min_heap)
        if u in visited:
            continue

        visited.add(u)
        mst_weight += weight

        # Include edge in result if not starting node
        if weight != 0:
            mst_edges.append((prev_node[u], u, weight))

        # Add all unvisited neighbors to heap
        for v, w in self.graph[u]:
            if v not in visited:
                heapq.heappush(min_heap, (w, v))
                prev_node[v] = u  # Store the previous node for edge tracing

    return mst_weight, mst_edges
```

```
# ----------------
# Example usage
if __name__ == "__main__":
    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)
    g.add_edge(1, 4, 5)
    g.add_edge(2, 4, 7)
    g.add_edge(3, 4, 9)

    # Store previous node globally for edge tracking
    prev_node = {}

    total_weight, mst = g.prim_mst()
    print("Minimum Spanning Tree edges and weights:")
    for u, v, w in mst:
        print(f"{u} - {v} (weight: {w})")

    print("Total weight of MST:", total_weight)
```

Output Example:

less

CopyEdit

Minimum Spanning Tree edges and weights:

0 - 1 (weight: 2)

1 - 2 (weight: 3)

1 - 4 (weight: 5)

0 - 3 (weight: 6)

Total weight of MST: 16

RESULT

To Implement Prim's algorithm in Python to find the Minimum Spanning Tree of a weighted, undirected graph is verified

9.Implement Dijkstra's algorithm to find the shortest path from a starting node to all other nodes in a graph with non-negative edge weights.

AIM

To Implement Dijkstra's algorithm to find the shortest path from a starting node to all other nodes in a graph with non-negative edge weights.

Algorithms

1. Initialize distances:
    - Set distance to source node = 0
    - All other nodes = ∞ (infinity)
2. Use a min-heap (priority queue) to select the node with the smallest current distance
3. For each selected node:
    - Visit all unvisited neighbors
    - Update their distances if a shorter path is found
4. Repeat until all nodes are visited (or the heap is empty)

PROGRAM

python

CopyEdit

```
import heapq
from collections import defaultdict


class Graph:
    def __init__(self):
        self.graph = defaultdict(list)  # {node: [(neighbor, weight), ...]}


    def add_edge(self, u, v, weight):
        self.graph[u].append((v, weight))
```

```
        # For undirected graph, add the reverse edge too:
        # self.graph[v].append((u, weight))

    def dijkstra(self, start):
        # Step 1: Initialize distances
        distances = {node: float('inf') for node in self.graph}
        distances[start] = 0

        # Step 2: Min-heap: (distance, node)
        heap = [(0, start)]

        while heap:
            current_dist, current_node = heapq.heappop(heap)

            # Step 3: Visit each neighbor
            for neighbor, weight in self.graph[current_node]:
                distance = current_dist + weight

                # Step 4: Update distance if shorter path is found
                if distance < distances[neighbor]:
                    distances[neighbor] = distance
                    heapq.heappush(heap, (distance, neighbor))

        return distances
```

□ Example Usage

python

CopyEdit

```
if __name___== "__main__":
    g = Graph()
    g.add_edge('A', 'B', 4)
    g.add_edge('A', 'C', 2)
    g.add_edge('B', 'C', 1)
    g.add_edge('B', 'D', 5)
    g.add_edge('C', 'D', 8)
    g.add_edge('C', 'E', 10)
    g.add_edge('D', 'E', 2)
    g.add_edge('D', 'Z', 6)
    g.add_edge('E', 'Z', 3)

    start_node = 'A'
    distances = g.dijkstra(start_node)

    print(f"Shortest distances from node '{start_node}':")
    for node in distances:
        print(f"{node}: {distances[node]}")
```

Output

vbnet

CopyEdit

```
Shortest distances from node 'A':
A: 0
B: 3
C: 2
```

D: 8

E: 10

Z: 13

RESULT

To Implement Dijkstra's algorithm to find the shortest path from a starting node to all other nodes in a graph with non-negative edge weights is verified.

10. Write a Python program to implement Bubble sort, Selection sort, Insertion sort Algorithms.

AIM

To Write a Python program to implement Bubble sort, Selection sort, Insertion sort Algorithms

1. Bubble Sort

Algorithm

1. Repeatedly step through the list
2. Compare adjacent items
3. Swap them if they are in the wrong order
4. Continue until the list is sorted

Time Complexity:

- Worst-case: $O(n^2)$

PROGRAM

python

CopyEdit

```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        # Last i elements are already in place
        for j in range(0, n - i - 1):
            if arr[j] > arr[j + 1]:
                # Swap
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
    return arr
```

○ 2. Selection Sort

- Algorithm:

1. For each index i from 0 to n-1:
    - Find the minimum element from i to end
    - Swap it with element at i

- Time Complexity:

· Worst-case: $O(n^2)$

python

CopyEdit

```
def selection_sort(arr):
    n = len(arr)
    for i in range(n):
        min_index = i
        # Find the smallest element in the remaining unsorted list
        for j in range(i + 1, n):
            if arr[j] < arr[min_index]:
                min_index = j
        # Swap the found minimum with the current element
        arr[i], arr[min_index] = arr[min_index], arr[i]
    return arr
```

○ 3. Insertion Sort

- Algorithm:

1. Start from the second element
2. Compare with all previous elements and insert it into the correct position
3. Shift elements to make space

- Time Complexity:
  - Worst-case: $O(n^2)$
  - Best-case (already sorted): O(n)

python

CopyEdit

```
def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1

        # Shift elements greater than key to the right
        while j >= 0 and arr[j] > key:
            arr[j + 1] = arr[j]
            j -= 1

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

□ Example Usage

python

CopyEdit

```
if __name___== "__main__":
    data = [64, 25, 12, 22, 11]

    print("Original array:", data)

    print("\nBubble Sort:")
```

```
print(bubble_sort(data.copy()))

print("\nSelection Sort:")
print(selection_sort(data.copy()))

print("\nInsertion Sort:")
print(insertion_sort(data.copy()))
```

○ Sample Output

less

CopyEdit

Original array: [64, 25, 12, 22, 11]

Bubble Sort:

[11, 12, 22, 25, 64]

Selection Sort:

[11, 12, 22, 25, 64]

Insertion Sort:

[11, 12, 22, 25, 64]

11.Write a Python program to implement Linear Search and Binary Search, with input validation (e.g., ensuring sorted input for binary search) and returning the index of the searched element if found.

AIM

To Write a Python program to implement Linear Search and Binary Search, with input validation (e.g., ensuring sorted input for binary search) and returning the index of the searched element if found.

Algorithm

Linear Search

5. Start from the first element.
6. Compare each element with the target.
7. If found, return the index.
8. If loop ends without finding, return -1.

o 2. Binary Search

- Prerequisites:
  - Input list must be sorted
- Algorithm:

1. Set low = 0, high = n - 1
2. While low <= high:
   - Compute mid = (low + high) // 2
   - If arr[mid] == target: return mid
   - If arr[mid] < target: search right half (low = mid + 1)
   - Else: search left half (high = mid - 1)
3. If not found, return -1

PROGRAM

python

CopyEdit

```
def linear_search(arr, target):
    """Linear Search algorithm"""
    for index, value in enumerate(arr):
        if value == target:
            return index
    return -1
def binary_search(arr, target):
    """Binary Search algorithm with sorted input validation"""
    if arr != sorted(arr):
        raise ValueError("Input array must be sorted for Binary Search.")

    low = 0
    high = len(arr) - 1

    while low <= high:
        mid = (low + high) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            low = mid + 1
        else:
            high = mid - 1

    return -1
```

□ Example Usage

python

CopyEdit

```
if __name__ == "__main__":
    # Unsorted input for linear search
    unsorted_data = [23, 45, 12, 9, 34, 56]
    target = 34

    print("Linear Search:")
    index = linear_search(unsorted_data, target)
    if index != -1:
        print(f'Element {target} found at index {index}")
    else:
        print(f'Element {target} not found")

    # Sorted input for binary search
    sorted_data = sorted(unsorted_data)
    print("\nBinary Search (on sorted array):")
    try:
        index = binary_search(sorted_data, target)
        if index != -1:
            print(f'Element {target} found at index {index}")
        else:
            print(f'Element {target} not found")
    except ValueError as ve:
        print("Error:", ve)
```

o Sample Output

sql

CopyEdit

Linear Search:

Element 34 found at index 4

Binary Search (on sorted array):

Element 34 found at index 3

RESULT

To Write a Python program to implement Linear Search and Binary Search, with input validation (e.g., ensuring sorted input for binary search) and returning the index of the searched element if found is verified .

12. Write a program to implement Linear Probing and Quadratic Probing for inserting and searching elements.

AIM

To Write a program to implement Linear Probing and Quadratic Probing for inserting and searching elements.

Linear Probing

- Check next index: index = (hash + i) % table_size

◆ Quadratic Probing

- Use a quadratic jump: index = (hash + i^2) % table_size

Algorithm

◆ Insertion

1. Compute the hash index: index = key % table_size
2. If the index is empty, insert the key
3. If there's a collision, probe using:
    - Linear: (index + i) % table_size
    - Quadratic: (index + i*i) % table_size
4. Repeat until an empty slot is found or the table is full

◆ Search

1. Use the same probing method as insertion
2. If key is found, return index
3. If an empty slot is found before the key, it's not present

PROGRAM

python

CopyEdit

```
class HashTable:
    def __init__(self, size):
```

```
    self.size = size
    self.table_linear = [None] * size
    self.table_quadratic = [None] * size

# Linear Probing Insert
def insert_linear(self, key):
    for i in range(self.size):
        index = (key + i) % self.size
        if self.table_linear[index] is None:
            self.table_linear[index] = key
            return index
    raise Exception("Hash table (linear) is full")

# Quadratic Probing Insert
def insert_quadratic(self, key):
    for i in range(self.size):
        index = (key + i*i) % self.size
        if self.table_quadratic[index] is None:
            self.table_quadratic[index] = key
            return index
    raise Exception("Hash table (quadratic) is full")

# Linear Probing Search
def search_linear(self, key):
    for i in range(self.size):
        index = (key + i) % self.size
        if self.table_linear[index] == key:
```

```
            return index
        if self.table_linear[index] is None:
            return -1
    return -1

# Quadratic Probing Search
def search_quadratic(self, key):
    for i in range(self.size):
        index = (key + i*i) % self.size
        if self.table_quadratic[index] == key:
            return index
        if self.table_quadratic[index] is None:
            return -1
    return -1

def display_tables(self):
    print("Linear Probing Table:")
    print(self.table_linear)
    print("Quadratic Probing Table:")
    print(self.table_quadratic)
```

□ Example Usage

python

CopyEdit

```
if __name___== "__main__":
    ht = HashTable(10)
    keys = [27, 18, 29, 28, 39]
```

```
print("Inserting keys using Linear Probing:")
for key in keys:
    ht.insert_linear(key)

print("\nInserting keys using Quadratic Probing:")
for key in keys:
    ht.insert_quadratic(key)

ht.display_tables()

# Search example
key_to_search = 28
print(f"\nSearching for {key_to_search}...")
lin_index = ht.search_linear(key_to_search)
quad_index = ht.search_quadratic(key_to_search)
print(f"Linear Probing Index: {lin_index}")
print(f"Quadratic Probing Index: {quad_index}")
```

Sample Output:

mathematica

CopyEdit

Inserting keys using Linear Probing:

Inserting keys using Quadratic Probing:

Linear Probing Table:

[None, None, None, None, 27, 18, 29, 28, 39, None]

Quadratic Probing Table:

[None, None, None, None, 27, 18, 29, 39, 28, None]

Searching for 28...

Linear Probing Index: 7

Quadratic Probing Index: 8

RESULT

To Write a Python program to implement Linear Search and Binary Search, with input validation (e.g., ensuring sorted input for binary search) and returning the index of the searched element if found.