Problem Solver (DI03016031) - Problem Solver

Antigravity AI

June 4, 2026

Problem Solver

First Edition: 2026

Author: Antigravity AI
Website: milav.in
YouTube: @milav.dabgar

To my students,
who constantly inspire me to find new analogies,
break down complex theories, and make learning
an engineered science.

Preface

About this Book

Welcome to **Computer Networks**! This textbook is meticulously designed to serve as a comprehensive problem solver and study companion. It provides a robust and intuitive foundation in the core principles of this subject, bridging the gap between theoretical models and practical applications. A unique feature of this book is its focus on decoding complex concepts using clear, step-by-step methodologies and real-world analogies.

Target Audience

This book is intended for students and professionals looking to master the concepts of Computer Networks. Whether you are revising for exams, seeking to solidify your fundamental understanding, or looking for practical examples to clarify abstract concepts, this book has been structured to support your learning journey. It serves as an accessible guide designed to remove the fear of complex topics, replacing it with an appreciation for the underlying mechanics.

Scope and Coverage

The coverage is divided logically to follow standard academic curriculums. Each chapter focuses on key topics, breaking them down into digestible sections:

- **Theoretical Insights** – Core concepts explained intuitively.
- **Method Blocks** – Standardized, step-by-step procedures for solving specific types of problems.
- **Worked Examples** – Fully solved problems that apply the methods in practice.

We hope this resource proves invaluable in your studies and helps you achieve excellence in **Computer Networks**. Happy learning!

Acknowledgments

Writing a comprehensive book that connects the fundamental forces of Chemistry with practical engineering applications requires more than just academic knowledge—it requires a community of support. I would like to express my sincere gratitude to everyone who contributed to the realization of this project.

Specially, I extend my heartfelt gratitude to the **Department of Technical Education (DTE), Government of Gujarat**, and **Government Polytechnic, Palanpur**, for providing an environment that fosters innovation, academic rigor, and continuous professional development.

I am deeply thankful to my family for their unwavering patience and support during the countless hours spent researching, formatting, and refining these chapters.

Finally, my deepest appreciation goes to my students. Your curiosity, challenging questions, and continuous feedback are the true driving force behind the unique analogies and streamlined structure of this book. This textbook was engineered for you.

Antigravity AI

About the Author

Milav Jayeshkumar Dabgar is an engineering educator and R&D professional with over 9 years of experience in electronics hardware development, embedded systems, AI/ML, and full-stack software engineering. He currently serves as a **Lecturer (Class - II) & Innovation Leader** at Government Polytechnic, Palanpur, under the Education Department of the Government of Gujarat.

Milav holds a Master of Engineering (ME) in Communication Systems from L.D. College of Engineering and a Bachelor of Engineering (BE) in Electronics & Communication. He is also currently pursuing a **BS in Data Science and Applications from IIT Madras**, where he has completed both the **Diploma in Programming** and the **Diploma in Data Science** with distinction.

Most recently, he completed the **AICTE QIP PG Certificate Program in Deep Learning from SVNIT Surat** with a **perfect 10/10 CGPA**, topping his batch.

A dedicated mentor, Milav has guided numerous student teams in securing over **INR 45 lakhs** in innovation funding, including INR 25 lakhs from Shark Tank India and INR 20 lakhs through iHub Gujarat, resulting in two student patents. He is recognized as an **NPTEL Evangelist** and **NPTEL Discipline Star** in Computer Science, reflecting his commitment to continuous learning and academic excellence.

His technical expertise spans from embedded systems (8051, STM32, Arduino) to modern web technologies (Next.js, Python, PostgreSQL) and Data Science (TensorFlow, PyTorch). Through this textbook, he aims to simplify complex Engineering Chemistry concepts by integrating relatable, real-world analogies drawn from modern tech and engineering landscapes.

Milav is also an active content creator, running a popular **YouTube channel** (@milav.dabgar) where he shares technical tutorials and academic resources. He maintains a personal blog and resource hub at milav.in and can be reached for professional collaborations on LinkedIn.

Contents

List of Figures

List of Tables

CHAPTER 1

Basic Concepts of Data Structures

1.1 Data Structure Basic Concepts

Data structures provide ways to organize, manage, and store data effectively for efficient access and modification. They are essential for designing efficient algorithms.

Methodology:

Data Structure Operations Data structures typically support the following basic operations. These operations form the foundation of algorithm design.

1. **Traversal**: Visiting every element in the data structure exactly once to access or process the data.
2. **Insertion**: Adding a new element to the data structure at a specified position.
3. **Deletion**: Removing a specific element from the data structure.
4. **Searching**: Finding the location or checking the existence of a specific element.
5. **Sorting**: Arranging elements in a specific logical order (e.g. ascending or descending).
6. **Merging**: Combining elements of two similarly structured data structures into a single one.

1.2 Types of Data Structures

Data structures are broadly classified into primitive and non-primitive types, and further subdivided into linear and non-linear types.

Methodology:

Classification of Data Structures

1. **Primitive Data Structures**: Basic data types provided directly by the programming language (e.g. Integer, Float, Character, Boolean).
2. **Non-Primitive Data Structures**: Complex structures derived from primitive types, used to store a group of values.
   (a) **Linear Data Structures**: Elements are arranged sequentially. Each element is connected to its previous and next elements. Examples include Arrays, Linked Lists, Stacks, and Queues.
   (b) **Non-Linear Data Structures**: Elements are organized hierarchically, without a sequential arrangement. Examples include Trees and Graphs.

1.3 Analysis Terms

Algorithm analysis measures the efficiency of algorithms based on Time Complexity and Space Complexity. Computational problems are often evaluated based on their Big O performance.

Methodology:

Time and Space Complexity Analysis

1. **Time Complexity**: Evaluates the amount of time an algorithm takes to run as a function of the input size n.
2. **Space Complexity**: Evaluates the amount of memory an algorithm requires during its execution as a function of input size n.
3. **Big O Notation** $\mathcal{O}$: Represents the upper bound of the algorithm's complexity. It describes the worst-case scenario.

Common Big O Complexities arranged from most efficient to least efficient:

1. $\mathcal{O}(1)$: Constant Time (Execution time is independent of input size)
2. $\mathcal{O}(\log n)$: Logarithmic Time
3. $\mathcal{O}(n)$: Linear Time
4. $\mathcal{O}(n \log n)$: Linearithmic Time
5. $\mathcal{O}(n^2)$: Quadratic Time
6. $\mathcal{O}(2^n)$: Exponential Time

Worked Example:

Analyzing a Simple Search Algorithm **Problem:** Analyze the time complexity of a simple linear search algorithm that iterates through an array to find a target element in an array of size n.
Solution:

1. **Algorithm Setup**: The algorithm checks each element of the array sequentially starting from the first index (index 0).
2. **Best Case**: The target element is found immediately at the first index. The loop terminates after exactly one iteration. Time Complexity: $\mathcal{O}(1)$.
3. **Worst Case**: The target element is located at the very last index or is not present in the array at all. The loop must iterate through all n elements. Time Complexity: $\mathcal{O}(n)$.
4. **Average Case**: The target is likely located somewhere in the middle. On average, it requires $n/2$ comparisons. In Big O notation, constants are dropped, simplifying to $\mathcal{O}(n)$.
5. **Space Complexity**: The algorithm utilizes a constant amount of extra memory for a loop counter and comparison variable, irrespective of n. Space Complexity: $\mathcal{O}(1)$.

1.4 Python Specific Data Structures

Python natively provides several built-in non-primitive data structures. These built-in types simplify many standard computational tasks.

Methodology:

Built in Python Data Structures

1. **List**: Ordered, mutable sequence. Allows heterogeneous elements and duplicates. Initialized using square brackets `[]`.
2. **Tuple**: Ordered, immutable sequence. Data cannot be modified after creation. Initialized using parentheses `()`.
3. **Set**: Unordered collection of unique elements. Highly optimized for membership testing. Initialized using curly braces `{}`.
4. **Dictionary**: Unordered collection mapping unique keys to values. Keys must be immutable. Initialized using curly braces with key-value pairs `{key: value}`.

Worked Example:

Implementing Python Data Structures **Problem:** Write Python statements to create and manipulate a List, Tuple, Set, and Dictionary.
Solution:

1. **List Operations**:

```
data_list = [10, 20, 30]
data_list.append(40) # Adds 40 to the end
data_list[0] = 15    # Modifies the element at index 0
# State: [15, 20, 30, 40]
```

2. **Tuple Operations**:

```
data_tuple = (10, 20, 30)
# data_tuple[0] = 15  # This would raise a TypeError
val = data_tuple[1]   # Accessing elements is allowed
# State: val is 20
```

3. **Set Operations**:

```
data_set = {10, 20, 30}
data_set.add(40)
data_set.add(10) # Ignored because 10 is already present
# State: {10, 20, 30, 40}
```

4. **Dictionary Operations**:

```
data_dict = {"A": 10, "B": 20}
data_dict["C"] = 30 # Inserts a new key-value pair
data_dict["A"] = 15 # Updates the value for existing key "A"
# State: {"A": 15, "B": 20, "C": 30}
```

1.5 Array in Python

Standard Python lists are highly versatile but incur significant memory overhead. For numerically intensive tasks, arrays that store elements of a uniform data type are required. This can be achieved using the built-in `array` module or the `numpy` library.

Methodology:

Implementing Arrays using the array Module

1. **Importing**: Import the module using `import array`.
2. **Initialization**: Create an array using `arr = array.array(typecode, [elements])`. The `typecode` defines the underlying C-type data structure (e.g. `'i'` for integer, `'f'` for floating point).
3. **Standard Methods**:
   (a) `append(val)`: Appends `val` to the array.
   (b) `insert(idx, val)`: Inserts `val` before index `idx`.
   (c) `pop(idx)`: Removes and returns the item at index `idx`.
   (d) `remove(val)`: Removes the first occurrence of `val`.

Worked Example:

Basic array Module Operations **Problem:** Initialize an integer array containing $1, 2, 3$. Insert 0 at index 0, append 4, and remove the value 2.
Solution:

1. **Import and Initialize**:

```
import array
arr = array.array('i', [1, 2, 3])
```

2. **Insertion**:

```
arr.insert(0, 0)
# Array state: array('i', [0, 1, 2, 3])
```

3. **Appending**:

```
arr.append(4)
# Array state: array('i', [0, 1, 2, 3, 4])
```

4. **Removal**:

```
arr.remove(2)
# Array state: array('i', [0, 1, 3, 4])
```

Methodology:

Implementing Arrays using NumPy NumPy arrays (`ndarray`) are heavily optimized and support multi-dimensional structures.

1. **Importing**: Standard practice is `import numpy as np`.
2. **Initialization**: Create an array via `arr = np.array([elements])`.
3. **Attributes**: `arr.shape` returns a tuple of dimensions; `arr.size` returns the total number of elements.
4. **Vectorization**: NumPy supports operations that implicitly apply element-wise without the need for manual loops, vastly improving execution speed.

Worked Example:

NumPy Vectorized Operations **Problem:** Use NumPy to create an array with elements $10, 20, 30$. Multiply every element by 5 and add 2.
Solution:

1. **Setup**:

```
import numpy as np
np_arr = np.array([10, 20, 30])
```

2. **Apply Vectorized Math**:

```
result = (np_arr * 5) + 2
```

3. **Analysis**: The scalar operations are automatically broadcasted across the array elements. The `result` variable evaluates to `[52, 102, 152]`.

Comparison Between Python Lists and Arrays

Attribute	Python List	Python Array / NumPy Array
Element Type	Heterogeneous (can store different types)	Homogeneous (must store the same type)
Memory Usage	Higher memory footprint due to referencing overhead	Contiguous memory allocation, highly efficient
Performance	Slower for numerical loops	Optimized for mathematical operations and vectorization
Flexibility	Dynamically sized, heavily used for general storage	Strictly typed, built for intensive computation

CHAPTER 2

Basics of Object-Oriented Programming

2.1 OOP Concepts

Object-Oriented Programming (OOP) is a paradigm that models software design around data or objects rather than functions and logic. The foundational computational concepts in OOP involve representing real-world entities through encapsulated properties and behaviors.

Methodology:

Core OOP Principles To design a system using OOP follow these sequential steps:

1. **Identify Objects**: Determine the real-world entities relevant to the problem.
2. **Define Classes**: Create blueprints that define the structure (attributes) and behavior (methods) of the identified objects.
3. **Encapsulate Data**: Restrict direct access to internal object states and expose necessary operations via controlled interfaces.
4. **Apply Inheritance**: Establish hierarchical relationships between classes to promote code reuse.
5. **Implement Polymorphism**: Allow objects of different classes to be treated uniformly through shared interfaces or overridden methods.

2.2 Class and Object

A class defines the blueprint while an object is a concrete instantiation of that blueprint.

Methodology:

Creating Classes and Instantiating Objects

1. Define the class using the `class` keyword followed by the class name.
2. Declare attributes (variables) to store the state.
3. Define methods (functions) to represent behaviors.
4. Instantiate an object by calling the class name as if it were a function.
5. Access object attributes and methods using dot notation (`object.attribute`).

Worked Example:

Implementing a Basic Class and Object **Problem:** Create a `Rectangle` class with length and width attributes. Instantiate an object and calculate its area.
Step 1: Define the class and attributes

```
class Rectangle:
    length = 0
    width = 0

    def calculate_area(self):
        return self.length * self.width
```

Step 2: Instantiate the object

```
rect_obj = Rectangle()
```

Step 3: Assign values and invoke methods

```
rect_obj.length = 10
rect_obj.width = 5
area = rect_obj.calculate_area()
print("Area:", area) # Output: Area: 50
```

2.3 Constructors

Constructors are special methods invoked automatically when a new object is created. They are primarily used for initializing the object's attributes.

Methodology:

Defining and Using Constructors

1. In Python define the constructor using the `__init__` method.
2. Pass `self` as the first parameter to reference the current object.
3. Add required parameters to initialize the object's state dynamically.
4. Assign the parameters to instance variables using `self.attribute_name = parameter_name`.

Worked Example:

Initializing Objects with Constructors **Problem:** Create an `Employee` class that requires a name and salary upon instantiation. Display the employee details.

Step 1: Define the constructor

```
class Employee:
    def __init__(self, emp_name, emp_salary):
        self.name = emp_name
        self.salary = emp_salary

    def display(self):
        print(f"Employee: {self.name} Salary: {self.salary}")
```

Step 2: Instantiate the object with arguments

```
emp1 = Employee("Alice", 50000)
```

Step 3: Call the display method

```
emp1.display() # Output: Employee: Alice Salary: 50000
```

2.4 Types of Methods

Methods in a class can be categorized based on how they interact with the class and its instances.

Methodology:

Implementing Method Types

1. **Instance Methods**: Defined with `self` as the first parameter. They access and modify the object's specific state.
2. **Class Methods**: Defined using the `@classmethod` decorator and take `cls` as the first parameter. They access and modify the class state which applies across all instances.
3. **Static Methods**: Defined using the `@staticmethod` decorator. They take neither `self` nor `cls`. They operate as utility functions independent of class or instance state.

Worked Example:

Using Different Method Types **Problem:** Implement a `MathOperations` class to demonstrate instance methods class methods and static methods.

Step 1: Define the class with all method types

```
class MathOperations:
    counter = 0 # Class attribute

    def __init__(self, value):
        self.value = value # Instance attribute

    def multiply_instance(self, multiplier):
        # Instance method
        return self.value * multiplier

    @classmethod
    def increment_counter(cls):
        # Class method
        cls.counter += 1
        return cls.counter

    @staticmethod
    def add_numbers(a, b):
        # Static method
        return a + b
```

Step 2: Execute the methods

```
# 1. Instance Method
obj = MathOperations(10)
print(obj.multiply_instance(5)) # Output: 50

# 2. Class Method
print(MathOperations.increment_counter()) # Output: 1

# 3. Static Method
print(MathOperations.add_numbers(15, 25)) # Output: 40
```

2.5 Data Encapsulation

Encapsulation restricts direct access to data to prevent unintended modification.

Methodology:

Achieving Data Encapsulation

1. Prefix attribute names with double underscores (e.g. `__balance`) to make them private.
2. Create public `getter` methods to read the private attribute.
3. Create public `setter` methods to modify the private attribute enforcing validation rules if necessary.

Worked Example:

Securing a Bank Account **Problem:** Create a `BankAccount` class with a private balance. Allow deposits and strictly control withdrawals.

Step 1: Define private attributes and public interfaces

```
class BankAccount:
    def __init__(self, initial_balance):
        self.__balance = initial_balance # Private attribute

    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount

    def withdraw(self, amount):
        if 0 < amount <= self.__balance:
            self.__balance -= amount
        else:
            print("Insufficient funds or invalid amount")

    def get_balance(self):
        return self.__balance
```

Step 2: Interact with the encapsulated data

```
account = BankAccount(100)
account.deposit(50)
account.withdraw(30)

print(account.get_balance()) # Output: 120

# Attempting direct access fails:
# print(account.__balance) # Raises AttributeError
```

2.6 Inheritance

Inheritance allows a new class (child) to inherit attributes and methods from an existing class (parent). The five primary types are: Single Multiple Multi-level Hierarchical and Hybrid.

Methodology:

Applying Types of Inheritance

1. **Single**: Class B inherits from Class A.

2. **Multiple**: Class C inherits from Class A and Class B simultaneously (`class C(A, B):`).
3. **Multi-level**: Class C inherits from Class B which inherits from Class A.
4. **Hierarchical**: Multiple child classes (Class B and Class C) inherit from a single parent (Class A).
5. **Hybrid**: A combination of two or more types of inheritance.

Worked Example:

Implementing Multi-level and Multiple Inheritance **Problem:** Demonstrate Multi-level and Multiple inheritance using computational relationships.
Multi-level Inheritance:

```
class BaseNumber:
    def get_base(self): return 10

class Multiplier(BaseNumber):
    def get_multiplier(self): return 5

class Result(Multiplier):
    def calculate(self):
        return self.get_base() * self.get_multiplier()

res = Result()
print("Multi-level Result:", res.calculate()) # Output: 50
```

Multiple Inheritance:

```
class Addition:
    def add(self, a, b): return a + b

class Subtraction:
    def sub(self, a, b): return a - b

class Calculator(Addition, Subtraction):
    pass

calc = Calculator()
print("Multiple Add:", calc.add(10, 5)) # Output: 15
print("Multiple Sub:", calc.sub(10, 5)) # Output: 5
```

2.7 Polymorphism through Inheritance

Polymorphism allows objects of different classes to be treated as objects of a common superclass. Through inheritance child classes can override parent methods to provide specific implementations.

Methodology:

Implementing Polymorphism via Method Overriding

1. Define a base class with a generic method.
2. Define child classes that inherit from the base class.
3. Redefine (override) the generic method in each child class with specific logic.

4. Iterate through a collection of objects calling the same method name allowing the runtime to determine which specific implementation to execute.

Worked Example:

Calculating Area Polymorphically **Problem:** Compute the area for a Circle and a Square using a unified interface.

Step 1: Define parent and child classes with overridden methods

```
class Shape:
    def area(self):
        return 0

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius
    def area(self):
        return 3.14 * (self.radius ** 2)

class Square(Shape):
    def __init__(self, side):
        self.side = side
    def area(self):
        return self.side ** 2
```

Step 2: Process polymorphically

```
shapes = [Circle(5), Square(4)]

for shape in shapes:
    print(f"Area: {shape.area()}")
# Output:
# Area: 78.5
# Area: 16
```

2.8 Abstraction - Abstract Class

Abstraction hides complex implementation details and exposes only the essential features. Abstract classes act as templates dictating which methods must be implemented by child classes.

Methodology:

Creating and Enforcing Abstract Classes

1. Import the `ABC` (Abstract Base Class) module and `abstractmethod` decorator.
2. Create a class inheriting from `ABC`.
3. Define abstract methods using the `@abstractmethod` decorator without providing an implementation (use `pass`).
4. Create concrete child classes that implement all the abstract methods.
5. Instantiate the child classes. (Note: Abstract classes themselves cannot be instantiated).

Worked Example:

Implementing an Abstract Data Processor **Problem:** Create an abstract class `DataProcessor` that mandates a `process_data` method. Implement a child class for text processing.

Step 1: Define the abstract base class

```
from abc import ABC, abstractmethod

class DataProcessor(ABC):
    @abstractmethod
    def process_data(self, data):
        pass
```

Step 2: Implement the concrete subclass

```
class TextProcessor(DataProcessor):
    def process_data(self, data):
        # Convert string to uppercase
        return data.upper()
```

Step 3: Execute the abstraction

```
# processor = DataProcessor() # This would raise TypeError

text_proc = TextProcessor()
result = text_proc.process_data("hello world")
print(result) # Output: HELLO WORLD
```

CHAPTER 3
Stack and Queues

3.1 Overview of Stack

A stack is a linear data structure that follows the Last-In-First-Out (LIFO) principle. The element inserted last is the first one to be removed.

Methodology:

Stack Representation and Pointers

1. A stack can be implemented using an array (list) or a linked list.
2. A pointer called `TOP` keeps track of the topmost element in the stack.
3. Initially when the stack is empty `TOP = -1`.
4. When an element is added `TOP` is incremented.
5. When an element is removed `TOP` is decremented.

3.2 Operations on Stack - Push and Pop

Methodology:

Push Operation Algorithm The `Push` operation adds an item to the top of the stack. Let `S` be the stack array of maximum size `N` and `TOP` be the pointer.

1. **Step 1:** Check for stack overflow. If `TOP == N - 1` then print "Stack Overflow" and exit.
2. **Step 2:** Increment `TOP`. `TOP = TOP + 1`.
3. **Step 3:** Insert the element `X`. `S[TOP] = X`.
4. **Step 4:** End.

Methodology:

Pop Operation Algorithm The `Pop` operation removes the item from the top of the stack.

1. **Step 1:** Check for stack underflow. If `TOP == -1` then print "Stack Underflow" and exit.
2. **Step 2:** Retrieve the top element. `X = S[TOP]`.
3. **Step 3:** Decrement `TOP`. `TOP = TOP - 1`.
4. **Step 4:** Return the element `X`.

3.3 Implementation of Stack using List

Worked Example:

Simulate Push and Pop Operations Given an empty stack `S` of maximum size 4. Perform the following operations and show the stack status and `TOP` value after each operation.

1. Push 10
2. Push 20
3. Push 30
4. Pop
5. Push 40
6. Push 50
7. Push 60

Solution: Initial state: `TOP = -1`, Stack is empty `[]`. Maximum size `N = 4`.

1. **Push 10:** `TOP` is incremented to 0. `S[0] = 10`. Stack: `[10]` (`TOP = 0`)
2. **Push 20:** `TOP` is incremented to 1. `S[1] = 20`. Stack: `[10, 20]` (`TOP = 1`)
3. **Push 30:** `TOP` is incremented to 2. `S[2] = 30`. Stack: `[10, 20, 30]` (`TOP = 2`)
4. **Pop:** Element removed is `S[2]` which is 30. `TOP` is decremented to 1. Stack: `[10, 20]` (`TOP = 1`)
5. **Push 40:** `TOP` is incremented to 2. `S[2] = 40`. Stack: `[10, 20, 40]` (`TOP = 2`)
6. **Push 50:** `TOP` is incremented to 3. `S[3] = 50`. Stack: `[10, 20, 40, 50]` (`TOP = 3`)
7. **Push 60:** `TOP == 3` which is equal to `N - 1`. Condition for Overflow is met. Print "Stack Overflow". Stack remains: `[10, 20, 40, 50]` (`TOP = 3`)

3.4 Application of Stack

3.4.1 Infix Prefix and Postfix Expressions

- **Infix:** Operator is placed between operands (e.g. $A + B$).
- **Prefix:** Operator is placed before operands (e.g. $+AB$).
- **Postfix:** Operator is placed after operands (e.g. $AB+$).

Methodology:

Algorithm for Infix to Postfix Conversion Let `Q` be the infix expression. A stack `S` is used to hold operators. `P` is the output postfix expression.

1. **Step 1:** Push a left parenthesis `(` onto stack `S` and append a right parenthesis `)` to the end of `Q`.
2. **Step 2:** Scan `Q` from left to right and repeat Steps 3 to 6 for each element until stack `S` is empty.
3. **Step 3:** If an operand is encountered add it to `P`.
4. **Step 4:** If a left parenthesis is encountered push it onto `S`.
5. **Step 5:** If an operator $\otimes$ is encountered:

(a) Repeatedly pop from S and add to P each operator on the top of S which has the same or higher precedence than $\otimes$.

(b) Add $\otimes$ to S.

6. **Step 6:** If a right parenthesis is encountered:

   (a) Repeatedly pop from S and add to P each operator on the top of S until a left parenthesis is encountered.

   (b) Remove the left parenthesis (do not add it to P).

Worked Example:

Convert Infix to Postfix Convert the infix expression `A * (B + C) / D` into a postfix expression using a stack.
Solution: Add `)` to the expression: `A * (B + C) / D )` Push `(` onto the stack.

Symbol Scanned	Stack	Postfix Expression (P)
(initial)	(	
A	(	A
*	(*	A
(	(* (	A
B	(* (	A B
+	(* (+	A B
C	(* (+	A B C
)	(*	A B C +
/	(/	A B C + *
D	(/	A B C + * D
)	Empty	A B C + * D /

Final Postfix Expression: `A B C + * D /`

Methodology:

Algorithm for Postfix Evaluation Let P be a postfix expression. A stack S is used to hold operands.

1. **Step 1:** Add a special character (e.g. #) at the end of P to mark the end.
2. **Step 2:** Scan P from left to right and repeat Steps 3 and 4 for each element until # is encountered.
3. **Step 3:** If an operand is encountered push it onto S.
4. **Step 4:** If an operator $\otimes$ is encountered:

   (a) Pop the top two elements from S. Let the top element be A and the second top element be B.

   (b) Evaluate B $\otimes$ A.

   (c) Push the result back onto S.

5. **Step 5:** The result is the top element of S.

Worked Example:

Evaluate Postfix Expression Evaluate the postfix expression `5 6 2 + * 12 4 / -`
Solution:

Symbol Scanned	Stack	Operation
5	5	Push 5
6	5 6	Push 6
2	5 6 2	Push 2
+	5 8	Pop 2 and 6. Compute $6 + 2 = 8$. Push 8.
*	40	Pop 8 and 5. Compute $5 \times 8 = 40$. Push 40.
12	40 12	Push 12
4	40 12 4	Push 4
/	40 3	Pop 4 and 12. Compute $12/4 = 3$. Push 3.
-	37	Pop 3 and 40. Compute $40 - 3 = 37$. Push 37.

Final Result: 37

3.4.2 Recursive Functions

A recursive function is a function that calls itself during its execution. The system internally uses a stack to store the state of the function calls (parameters local variables and return addresses) until the base case is reached and the stack unwinds.

3.5 Overview of Queue

A queue is a linear data structure that follows the First-In-First-Out (FIFO) principle. Elements are added at one end called the `REAR` and removed from the other end called the `FRONT`.

Methodology:

Queue Representation and Pointers

1. A queue can be represented using an array or a linked list.
2. A pointer `FRONT` keeps track of the element to be deleted.
3. A pointer `REAR` keeps track of the last element inserted.
4. Initially when the queue is empty `FRONT = -1` and `REAR = -1`.

3.6 Operations on Queue

Methodology:

Enqueue Operation Algorithm The `Enqueue` operation adds an element to the rear of the queue. Let `Q` be an array of size `N`.

1. **Step 1:** Check for overflow. If `REAR == N - 1` then print "Queue Overflow" and exit.
2. **Step 2:** If `FRONT == -1` and `REAR == -1` (Queue is empty): Set `FRONT = 0` and `REAR = 0`.
3. **Step 3:** Else increment `REAR`. `REAR = REAR + 1`.
4. **Step 4:** Insert element `X`. `Q[REAR] = X`.

Methodology:

Dequeue Operation Algorithm The `Dequeue` operation removes an element from the front of the queue.

1. **Step 1:** Check for underflow. If `FRONT == -1` then print "Queue Underflow" and exit.
2. **Step 2:** Retrieve the element. `X = Q[FRONT]`.
3. **Step 3:** If `FRONT == REAR` (Queue has only one element): Set `FRONT = -1` and `REAR = -1`.
4. **Step 4:** Else increment `FRONT`. `FRONT = FRONT + 1`.
5. **Step 5:** Return element `X`.

3.7 Implementation of Queue using List

Worked Example:

Simulate Enqueue and Dequeue Operations Given an empty simple queue `Q` of size 4. Perform the following operations:

1. Enqueue 10
2. Enqueue 20
3. Dequeue
4. Enqueue 30
5. Enqueue 40
6. Enqueue 50

Solution: Initial state: `FRONT = -1`, `REAR = -1`, Size `N = 4`.

1. **Enqueue 10:** Queue was empty so `FRONT = 0`, `REAR = 0`. `Q[0] = 10`. Queue: `[10]` (`FRONT = 0`, `REAR = 0`)
2. **Enqueue 20:** `REAR = 1`. `Q[1] = 20`. Queue: `[10, 20]` (`FRONT = 0`, `REAR = 1`)
3. **Dequeue:** Remove `Q[FRONT]` which is 10. `FRONT = 1`. Queue: `[-, 20]` (`FRONT = 1`, `REAR = 1`)
4. **Enqueue 30:** `REAR = 2`. `Q[2] = 30`. Queue: `[-, 20, 30]` (`FRONT = 1`, `REAR = 2`)
5. **Enqueue 40:** `REAR = 3`. `Q[3] = 40`. Queue: `[-, 20, 30, 40]` (`FRONT = 1`, `REAR = 3`)
6. **Enqueue 50:** `REAR == 3` which is `N - 1`. Overflow condition met. Print "Queue Overflow". Queue remains: `[-, 20, 30, 40]` (`FRONT = 1`, `REAR = 3`)

3.8 Limitation of Simple Queue

The primary limitation of a simple queue implemented with an array is the inefficient use of space. As elements are dequeued the `FRONT` pointer advances leaving empty slots at the beginning of the array. Even if there are empty slots the queue might report an overflow if `REAR` has reached the maximum size limit.

Worked Example:

Demonstrating the Limitation In the previous example after step 5 the queue was `[-, 20, 30, 40]` with `FRONT = 1` and `REAR = 3`. The slot at index 0 is empty because element 10 was dequeued. When we tried

to Enqueue 50 in step 6 the algorithm checked `REAR == 3` and reported "Queue Overflow". This is an inefficient use of memory since space is available at index 0 but cannot be used in a simple linear queue.

3.9 Circular Queue

A circular queue solves the space inefficiency problem of a simple queue. The array is treated as if it were a circle meaning that the position after the last element is the first element.

Methodology:

Circular Queue Indexing In a circular queue of size `N` the next position is calculated using the modulo operator:

```
Next Position = (Current Position + 1) % N
```

This allows `REAR` to wrap around to the beginning of the array if there is space left.

Methodology:

Enqueue in Circular Queue

1. **Step 1:** Check for overflow. If `(REAR + 1) % N == FRONT` then print "Circular Queue Overflow" and exit.
2. **Step 2:** If `FRONT == -1` and `REAR == -1`: Set `FRONT = 0` and `REAR = 0`.
3. **Step 3:** Else advance `REAR` circularly. `REAR = (REAR + 1) % N`.
4. **Step 4:** Insert element `X`. `Q[REAR] = X`.

Methodology:

Dequeue in Circular Queue

1. **Step 1:** Check for underflow. If `FRONT == -1` then print "Circular Queue Underflow" and exit.
2. **Step 2:** Retrieve the element. `X = Q[FRONT]`.
3. **Step 3:** If `FRONT == REAR` (Only one element left): Set `FRONT = -1` and `REAR = -1`.
4. **Step 4:** Else advance `FRONT` circularly. `FRONT = (FRONT + 1) % N`.
5. **Step 5:** Return element `X`.

Worked Example:

Circular Queue Operations Given a circular queue `CQ` of size `N = 4`. Perform the following operations:

1. Enqueue 10 Enqueue 20 Enqueue 30 Enqueue 40
2. Dequeue
3. Enqueue 50

Solution:

1. **Enqueue 10 20 30 40:** After four enqueues `FRONT = 0`, `REAR = 3`. Queue: `[10, 20, 30, 40]`
2. **Dequeue:** Remove `CQ[FRONT]` (10). `FRONT = (0 + 1) % 4 = 1`. Queue: `[-, 20, 30, 40]` (`FRONT = 1`, `REAR = 3`)

3. **Enqueue 50:** Check Overflow: `(REAR + 1) % 4 = (3 + 1) % 4 = 0`. Is `0 == FRONT` (`FRONT` is 1)? No. So no overflow. `REAR = 0`. `CQ[0] = 50`. Queue: `[50, 20, 30, 40]` (`FRONT = 1`, `REAR = 0`)

Notice how element 50 successfully wrapped around and occupied the empty slot at index 0.

3.10 Application of Queue

Queues are widely used in scenarios where tasks or requests need to be processed in the exact order they arrive:

- **CPU Scheduling:** Operating systems use queues to maintain processes waiting to be executed by the CPU.
- **Print Spooling:** Print jobs are queued and sent to the printer one by one in a FIFO manner.
- **Data Buffering:** Queues are used as buffers in IO operations (e.g. keyboard buffers network packets).

3.11 Differentiate Circular and Simple Queue

Feature	Simple Queue	Circular Queue
Insertion Pattern	Linear (ends at max size).	Circular (wraps around to index 0).
Memory Utilization	Inefficient. Empty spaces at the front cannot be reused.	Efficient. Empty spaces can be reused if rear reaches the end.
Next Position	`REAR + 1`	`(REAR + 1) % N`
Overflow Condition	`REAR == N - 1`	`(REAR + 1) % N == FRONT`
Use Case	Basic task scheduling with unlimited or sufficient bounds.	Traffic light systems and memory buffers where size is strictly fixed.

CHAPTER 4

Linked List

4.1 Overview of Linked List

A linked list is a linear data structure consisting of a sequence of elements called nodes. Unlike arrays, elements are not stored in contiguous memory locations. Instead, each node contains a data element and a reference (pointer) to the next node in the sequence.

Methodology:

Node Representation in Memory Each node essentially contains two parts:

1. **DATA**: The actual value or information stored in the node.
2. **NEXT**: A pointer holding the memory address of the next node.

The first node is accessed via a special pointer called `HEAD`. The `NEXT` pointer of the last node is set to `NULL` to indicate the end of the list.

4.2 Types of Linked Lists

Based on how nodes are connected, linked lists are classified into three primary types:

1. **Singly Linked List**: Each node points only to the next node.
2. **Circular Linked List**: The last node points back to the first node.
3. **Doubly Linked List**: Each node has two pointers pointing to both the previous and next nodes.

4.3 Basic Operations on Singly Linked List

4.3.1 Insertion Operations

A new node can be inserted at the beginning, at the end, or at a specific position.

Methodology:

Algorithm for Insertion at the Beginning Let `HEAD` be the pointer to the first node.

1. Allocate memory for `NEW_NODE`.
2. Set `NEW_NODE.DATA = VALUE`.
3. Set `NEW_NODE.NEXT = HEAD`.
4. Update `HEAD = NEW_NODE`.

Worked Example:

Insert Node at the Beginning Given a singly linked list: $10 \rightarrow 20 \rightarrow 30 \rightarrow$ NULL where `HEAD` points to 10. Insert 5 at the beginning.

1. **Step 1**: Create `NEW_NODE` with data 5.
2. **Step 2**: Set `NEW_NODE.NEXT` to `HEAD` (which points to 10). Thus, `NEW_NODE.NEXT` points to 10.
3. **Step 3**: Update `HEAD` to point to `NEW_NODE` (which is 5).

Final List: $5 \rightarrow 10 \rightarrow 20 \rightarrow 30 \rightarrow$ NULL.

Methodology:

Algorithm for Insertion at the End

1. Allocate memory for `NEW_NODE`.
2. Set `NEW_NODE.DATA = VALUE` and `NEW_NODE.NEXT = NULL`.
3. If `HEAD` is `NULL` (empty list), set `HEAD = NEW_NODE` and stop.
4. Else initialize a temporary pointer `TEMP = HEAD`.
5. Traverse to the last node: While `TEMP.NEXT` $\neq$ `NULL`, set `TEMP = TEMP.NEXT`.
6. Set `TEMP.NEXT = NEW_NODE`.

Worked Example:

Insert Node at the End Given a singly linked list: $10 \rightarrow 20 \rightarrow$ NULL. Insert 30 at the end.

1. **Step 1**: Create `NEW_NODE` with data 30 and `NEXT = NULL`.
2. **Step 2**: List is not empty. Set `TEMP` to point to 10.
3. **Step 3**: `TEMP.NEXT` is not `NULL` (points to 20). Move `TEMP` to 20.
4. **Step 4**: Now `TEMP.NEXT` is `NULL`. Loop terminates.
5. **Step 5**: Set `TEMP.NEXT = NEW_NODE` (Node 20's next points to 30).

Final List: $10 \rightarrow 20 \rightarrow 30 \rightarrow$ NULL.

4.3.2 Deletion Operations

Deleting nodes from a singly linked list involves adjusting pointers to bypass the node and then freeing its memory.

Methodology:

Algorithm for Deletion from the Beginning

1. If `HEAD == NULL`, print "List is Empty" and exit.
2. Set `TEMP = HEAD`.
3. Update `HEAD = HEAD.NEXT`.
4. Free memory of `TEMP`.

Worked Example:

Delete First Node Given list: 5 → 10 → 20 → NULL where `HEAD` points to 5.

1. **Step 1**: Set `TEMP` to point to node 5.
2. **Step 2**: Update `HEAD` to `HEAD.NEXT` (which is node 10). Now `HEAD` points to 10.
3. **Step 3**: Delete `TEMP` (node 5).

Final List: 10 → 20 → NULL.

Methodology:

Algorithm for Deletion of a Specific Node Let `KEY` be the value to delete.

1. If `HEAD == NULL`, print "List is Empty" and exit.
2. If `HEAD.DATA == KEY`, delete from beginning (update `HEAD` to `HEAD.NEXT`) and exit.
3. Set `TEMP = HEAD` and `PREV = NULL`.
4. Traverse to find `KEY`: While `TEMP` $\neq$ `NULL` and `TEMP.DATA` $\neq$ `KEY`, set `PREV = TEMP` and `TEMP = TEMP.NEXT`.
5. If `TEMP == NULL`, `KEY` not found. Exit.
6. Bypass node: Set `PREV.NEXT = TEMP.NEXT`.
7. Free memory of `TEMP`.

Worked Example:

Delete a Specific Node Given list: 10 → 20 → 30 → NULL. Delete node with value 20.

1. **Step 1**: `TEMP` = 10, `PREV` = `NULL`.
2. **Step 2**: `TEMP.DATA` $\neq$ `20`. `PREV` becomes 10, `TEMP` moves to 20.
3. **Step 3**: `TEMP.DATA == 20`. Loop stops.
4. **Step 4**: Set `PREV.NEXT = TEMP.NEXT` (Node 10's `NEXT` becomes Node 30).
5. **Step 5**: Free `TEMP` (Node 20).

Final List: 10 → 30 → NULL.

4.3.3 Counting Nodes

Methodology:

Algorithm to Count Total Nodes

1. Initialize `COUNT = 0`.
2. Set `TEMP = HEAD`.
3. While `TEMP` $\neq$ `NULL`:
   (a) Increment `COUNT = COUNT + 1`.
   (b) Move pointer: `TEMP = TEMP.NEXT`.
4. Return `COUNT`.

Worked Example:

Count Nodes in a List Given list: 15 $\rightarrow$ 25 $\rightarrow$ 35 $\rightarrow$ NULL.

1. **Step 1**: `COUNT = 0`, `TEMP` points to 15.
2. **Step 2**: `TEMP` points to 15. `COUNT = 1`. `TEMP` moves to 25.
3. **Step 3**: `TEMP` points to 25. `COUNT = 2`. `TEMP` moves to 35.
4. **Step 4**: `TEMP` points to 35. `COUNT = 3`. `TEMP` moves to `NULL`.
5. **Step 5**: `TEMP` is `NULL`. Loop stops. Return 3.

4.4 Overview of Circular Linked List

A circular linked list is a variation of a singly linked list where the `NEXT` pointer of the last node is linked back to the first node, forming a continuous circle. There is no `NULL` reference indicating the end of the list.

Methodology:

Algorithm to Traverse a Circular Linked List

1. If `HEAD == NULL`, print "List is Empty" and exit.
2. Set `TEMP = HEAD`.
3. Repeat: Print `TEMP.DATA` and set `TEMP = TEMP.NEXT`.
4. Until `TEMP == HEAD` (we reach the starting node again).

4.5 Difference Between Circular and Singly Linked List

Singly Linked List	Circular Linked List
The `NEXT` pointer of the last node points to `NULL`.	The `NEXT` pointer of the last node points back to the first node.
Traversing stops when a `NULL` pointer is encountered.	Traversing stops when the `NEXT` pointer points to the `HEAD` node.
You cannot reach the first node directly from the last node.	You can easily reach the first node directly from the last node.
Memory allocation and traversal logic are simpler.	Logic requires special condition checks to avoid infinite loops.
Useful for implementing simple stacks and queues.	Useful for implementing round-robin scheduling algorithms.

4.6 Overview of Doubly Linked List

A doubly linked list is a list containing nodes with three fields: two pointer fields and one data field. This allows traversal in both forward and backward directions.

Methodology:

Representation of Doubly Linked List Node Each node is created with three components:

1. **`PREV`**: Pointer to the previous node in the sequence.
2. **`DATA`**: The actual value stored.
3. **`NEXT`**: Pointer to the next node in the sequence.

For the first node, `PREV` is `NULL`. For the last node, `NEXT` is `NULL`.

4.7 Applications of Linked Lists

Linked lists are widely used in computational operations where dynamic memory allocation is essential.

1. **Dynamic Data Structures**: Implementing dynamic stacks and queues that grow and shrink dynamically without size limits.
2. **Graph Representation**: Used to represent adjacency lists in graph algorithms.
3. **Mathematical Operations**: Representing polynomials and performing arithmetic like polynomial addition and multiplication.

Methodology:

Polynomial Representation using Linked List A polynomial expression like $P(x) = c_1x^{e_1} + c_2x^{e_2} + \dots$ is represented using nodes. Each node contains:

1. `COEFF` (coefficient c_i)
2. `EXP` (exponent e_i)
3. `NEXT` (pointer to the next term)

Nodes are sorted in descending order of exponents.

Worked Example:

Represent Polynomial Computationally Represent the mathematical polynomial $5x^3 + 4x^2 - 7x + 2$ as a linked list.

1. **Term 1** ($5x^3$): Create Node 1 with `COEFF = 5`, `EXP = 3`. Link to Node 2.
2. **Term 2** ($4x^2$): Create Node 2 with `COEFF = 4`, `EXP = 2`. Link to Node 3.
3. **Term 3** ($-7x^1$): Create Node 3 with `COEFF = -7`, `EXP = 1`. Link to Node 4.
4. **Term 4** ($2x^0$): Create Node 4 with `COEFF = 2`, `EXP = 0`. Link to `NULL`.

Final Structure: $[5, 3] \to [4, 2] \to [-7, 1] \to [2, 0] \to$ NULL.

CHAPTER 5

Searching and Sorting Methods

5.1 Searching an element into List

5.1.1 Linear Search

Methodology:

Linear Search Algorithm Linear search sequentially checks each element of the list until a match is found or the whole list has been searched.

1. Start from the first element (index 0) of the list.
2. Compare the search key with the current element.
3. If the current element matches the key, return the index.
4. If it does not match, move to the next element.
5. Repeat steps 2-4 until the end of the list is reached.
6. If the end of the list is reached and the key is not found, return -1 (or indicate failure).

Worked Example:

Linear Search to find an element Given the list $A = [12, 45, 67, 89, 23, 56]$ and a search key $K = 23$. Perform a linear search to find the index of K.
Solution:

1. **Step 1:** Compare $K = 23$ with $A[0] = 12$. Since $23 \neq 12$, move to the next element.
2. **Step 2:** Compare $K = 23$ with $A[1] = 45$. Since $23 \neq 45$, move to the next element.
3. **Step 3:** Compare $K = 23$ with $A[2] = 67$. Since $23 \neq 67$, move to the next element.
4. **Step 4:** Compare $K = 23$ with $A[3] = 89$. Since $23 \neq 89$, move to the next element.
5. **Step 5:** Compare $K = 23$ with $A[4] = 23$. Since $23 = 23$, a match is found.

The key 23 is found at index 4.

5.1.2 Binary Search

Methodology:

Binary Search Algorithm Binary search finds the position of a target value within a **sorted** array. It compares the target value to the middle element of the array.

1. Initialize low to 0 and high to $N - 1$ (where N is the number of elements).

2. Repeat while low $\leq$ high:
   (a) Calculate the middle index: mid = low + $\lfloor$(high − low)/2$\rfloor$.
   (b) Compare the search key with A[mid].
   (c) If A[mid] equals the key, return mid.
   (d) If the key is less than A[mid], the key must be in the left half. Set high = mid − 1.
   (e) If the key is greater than A[mid], the key must be in the right half. Set low = mid + 1.
3. If the loop terminates without finding the key, return -1.

Worked Example:

Binary Search on a sorted array Given the sorted array $A = [5, 12, 23, 38, 45, 56, 72, 89]$ and a search key $K = 38$. Trace the binary search algorithm.
Solution: Number of elements $N = 8$. Initially, low = 0, high = 7.
Iteration 1:

- mid = $0 + \lfloor(7 - 0)/2\rfloor = 3$.
- Compare $K = 38$ with $A[3] = 38$.
- Since $38 = 38$, the key is found at index 3.

Let's search for another key, $K = 72$.
Iteration 1 (for $K = 72$):

- low = 0, high = 7.
- mid = 3. $A[3] = 38$.
- Since $72 > 38$, the key is in the right half. Set low = mid + 1 = 4.

Iteration 2:

- low = 4, high = 7.
- mid = $4 + \lfloor(7 - 4)/2\rfloor = 4 + 1 = 5$.
- Compare $K = 72$ with $A[5] = 56$.
- Since $72 > 56$, the key is in the right half. Set low = mid + 1 = 6.

Iteration 3:

- low = 6, high = 7.
- mid = $6 + \lfloor(7 - 6)/2\rfloor = 6$.
- Compare $K = 72$ with $A[6] = 72$.
- Since $72 = 72$, the key is found at index 6.

5.2 Sorting Methods

5.2.1 Bubble Sort

Methodology:

Bubble Sort Algorithm Bubble sort repeatedly steps through the list, compares adjacent elements and swaps them if they are in the wrong order.

1. For i from 0 to $N-1$ (where N is the length of the array):
2. For j from 0 to $N-i-2$:
3. Compare $A[j]$ and $A[j+1]$.
4. If $A[j] > A[j+1]$, swap them.
5. The largest element "bubbles up" to its correct position at the end of the array after each full pass of the inner loop.
6. Continue until the array is completely sorted.

Worked Example:

Sort an array using Bubble Sort Sort the array $A = [25, 15, 30, 10, 5]$ using Bubble Sort.
Solution: Length $N = 5$.
Pass 1 ($i = 0$):

- Compare 25 and 15. Since $25 > 15$, swap. Array: $[15, 25, 30, 10, 5]$
- Compare 25 and 30. Since $25 < 30$, no swap. Array: $[15, 25, 30, 10, 5]$
- Compare 30 and 10. Since $30 > 10$, swap. Array: $[15, 25, 10, 30, 5]$
- Compare 30 and 5. Since $30 > 5$, swap. Array: $[15, 25, 10, 5, \mathbf{30}]$

The largest element (30) is in its correct place.
Pass 2 ($i = 1$):

- Compare 15 and 25. No swap. Array: $[15, 25, 10, 5, 30]$
- Compare 25 and 10. Swap. Array: $[15, 10, 25, 5, 30]$
- Compare 25 and 5. Swap. Array: $[15, 10, 5, \mathbf{25}, \mathbf{30}]$

Pass 3 ($i = 2$):

- Compare 15 and 10. Swap. Array: $[10, 15, 5, 25, 30]$
- Compare 15 and 5. Swap. Array: $[10, 5, \mathbf{15}, \mathbf{25}, \mathbf{30}]$

Pass 4 ($i = 3$):

- Compare 10 and 5. Swap. Array: $[\mathbf{5}, \mathbf{10}, \mathbf{15}, \mathbf{25}, \mathbf{30}]$

The sorted array is $[5, 10, 15, 25, 30]$.

5.2.2 Selection Sort

Methodology:

Selection Sort Algorithm Selection sort works by repeatedly finding the minimum element from the unsorted part and putting it at the beginning.

1. For i from 0 to $N-2$:
2. Assume i is the index of the minimum element (min_idx $= i$).
3. For j from $i+1$ to $N-1$:
4. If $A[j] < A[\text{min\_idx}]$, update min_idx $= j$.
5. Swap the found minimum element with the first element of the unsorted part ($A[i]$ and $A[\text{min\_idx}]$).
6. The sorted part grows from left to right.

Worked Example:

Sort an array using Selection Sort Sort the array $A = [64, 25, 12, 22, 11]$ using Selection Sort.
Solution:
Pass 1 ($i = 0$):

- Unsorted part: $[64, 25, 12, 22, 11]$.
- Find minimum: The minimum is 11 at index 4.
- Swap 64 (at index 0) and 11.
- Array becomes: $[\mathbf{11}, 25, 12, 22, 64]$.

Pass 2 ($i = 1$):

- Unsorted part: $[25, 12, 22, 64]$.
- Find minimum: The minimum is 12 at index 2.
- Swap 25 (at index 1) and 12.
- Array becomes: $[\mathbf{11}, \mathbf{12}, 25, 22, 64]$.

Pass 3 ($i = 2$):

- Unsorted part: $[25, 22, 64]$.
- Find minimum: The minimum is 22 at index 3.
- Swap 25 (at index 2) and 22.
- Array becomes: $[\mathbf{11}, \mathbf{12}, \mathbf{22}, 25, 64]$.

Pass 4 ($i = 3$):

- Unsorted part: $[25, 64]$.
- Find minimum: The minimum is 25 at index 3.
- Swap 25 with itself.
- Array becomes: $[\mathbf{11}, \mathbf{12}, \mathbf{22}, \mathbf{25}, \mathbf{64}]$.

The sorted array is $[11, 12, 22, 25, 64]$.

5.2.3 Insertion Sort

Methodology:

Insertion Sort Algorithm Insertion sort builds the final sorted array one item at a time. It works similarly to the way you sort playing cards in your hands.

1. For i from 1 to $N-1$:
2. Pick the current element key $= A[i]$.
3. Compare the key with elements in the sorted sublist on its left ($A[0 \dots i-1]$).
4. Shift all elements in the sorted sublist that are greater than key one position to the right.
5. Insert the key into its correct position.

Worked Example:

Sort an array using Insertion Sort Sort the array $A = [40, 10, 30, 50, 20]$ using Insertion Sort.
Solution:
Initial State: $[\mathbf{40}, 10, 30, 50, 20]$ (The first element is trivially sorted).
Step 1 ($i = 1$, key $= 10$):

- Compare 10 with 40. Since $40 > 10$, shift 40 to the right.
- Insert 10 at index 0.
- Array: $[\mathbf{10}, \mathbf{40}, 30, 50, 20]$

Step 2 ($i = 2$, key $= 30$):

- Compare 30 with 40. Since $40 > 30$, shift 40 to the right.
- Compare 30 with 10. Since $10 \leq 30$, stop shifting.
- Insert 30 at index 1.
- Array: $[\mathbf{10}, \mathbf{30}, \mathbf{40}, 50, 20]$

Step 3 ($i = 3$, key $= 50$):

- Compare 50 with 40. Since $40 \leq 50$, no shifting needed.
- Insert 50 at its current position.
- Array: $[\mathbf{10}, \mathbf{30}, \mathbf{40}, \mathbf{50}, 20]$

Step 4 ($i = 4$, key $= 20$):

- Compare 20 with 50. Shift 50 to the right.
- Compare 20 with 40. Shift 40 to the right.
- Compare 20 with 30. Shift 30 to the right.
- Compare 20 with 10. Since $10 \leq 20$, stop shifting.
- Insert 20 at index 1.
- Array: $[\mathbf{10}, \mathbf{20}, \mathbf{30}, \mathbf{40}, \mathbf{50}]$

The sorted array is $[10, 20, 30, 40, 50]$.

5.2.4 Quick Sort

Methodology:

Quick Sort Algorithm Quick Sort is a Divide and Conquer algorithm. It picks an element as a pivot and partitions the given array around the picked pivot.

1. **Choose Pivot:** Select an element from the array to act as the pivot (often the first, last, or middle element). Here, we pick the last element.
2. **Partitioning:** Rearrange the array such that all elements smaller than the pivot are placed before it, and all elements greater than the pivot are placed after it. The pivot is now in its final sorted position.
3. **Recursion:** Recursively apply the above steps to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values.
4. The base case for recursion is when the sub-array has length 0 or 1.

Worked Example:

Sort an array using Quick Sort Sort the array $A = [8, 3, 1, 7, 0, 10, 2]$ using Quick Sort (always picking the last element as pivot).
Solution:
Initial Array: $[8, 3, 1, 7, 0, 10, 2]$
Call 1: QuickSort$(0, 6)$

- Pivot is $A[6] = 2$.
- Partitioning the array around 2:
  - Elements less than 2: $1, 0$
  - Elements greater than 2: $8, 3, 7, 10$
- Array after partition: $[1, 0, \mathbf{2}, 8, 3, 7, 10]$. Pivot 2 is at index 2.

Call 2: QuickSort on left sub-array $[1, 0]$ **(indices** 0 **to** 1**)**

- Pivot is 0.
- Partitioning around 0: Elements less than 0: none. Greater: 1.
- Array after partition: $[\mathbf{0}, 1]$. Pivot 0 is at index 0.
- Left of 0 is empty. Right of 0 is [1] (base case, already sorted).

Call 3: QuickSort on right sub-array $[8, 3, 7, 10]$ **(indices** 3 **to** 6**)**

- Pivot is 10.
- Partitioning around 10: All elements $(8, 3, 7)$ are less.
- Array remains $[8, 3, 7, \mathbf{10}]$. Pivot 10 is at index 6.

Call 4: QuickSort on left of 10**:** $[8, 3, 7]$

- Pivot is 7.
- Partitioning around 7: Less: 3. Greater: 8.
- Array after partition: $[3, \mathbf{7}, 8]$. Pivot 7 is at index 4.
- Left of 7 is [3] (sorted), right of 7 is [8] (sorted).

Combining everything, the full array is sorted in place: $[0, 1, 2, 3, 7, 8, 10]$.

5.2.5 Merge Sort

Methodology:

Merge Sort Algorithm Merge Sort is a Divide and Conquer algorithm. It divides the input array into two halves, calls itself for the two halves, and then merges the two sorted halves.

1. **Divide:** If the array has 0 or 1 elements, it is already sorted (base case). Otherwise, find the middle point to divide the array into two halves: $\text{mid} = \text{low} + \lfloor(\text{high} - \text{low})/2\rfloor$.
2. **Conquer:** Recursively sort the first half and the second half.
3. **Combine (Merge):** Merge the two sorted halves into a single sorted array.
   (a) Create temporary arrays for the left and right halves.
   (b) Compare elements from both halves one by one.
   (c) Copy the smaller element to the original array.
   (d) Copy any remaining elements from the left or right half.

Worked Example:

Sort an array using Merge Sort Sort the array $A = [38, 27, 43, 3, 9, 82, 10]$ using Merge Sort.
Solution:
1. Divide Process:

- Divide $[38, 27, 43, 3, 9, 82, 10]$ into $[38, 27, 43, 3]$ and $[9, 82, 10]$.
- Divide $[38, 27, 43, 3]$ into $[38, 27]$ and $[43, 3]$.
- Divide $[38, 27]$ into $[38]$ and $[27]$.
- Divide $[43, 3]$ into $[43]$ and $[3]$.
- Divide $[9, 82, 10]$ into $[9, 82]$ and $[10]$.
- Divide $[9, 82]$ into $[9]$ and $[82]$.

2. Conquer and Merge Process:

- Merge $[38]$ and $[27] \rightarrow$ Compare 38 and 27. $27 < 38$. Result: $[27, 38]$.
- Merge $[43]$ and $[3] \rightarrow$ Result: $[3, 43]$.
- Merge $[9]$ and $[82] \rightarrow$ Result: $[9, 82]$.
- Merge $[27, 38]$ and $[3, 43]$:
  - Compare 27 and 3. Copy 3.
  - Compare 27 and 43. Copy 27.
  - Compare 38 and 43. Copy 38.
  - Copy remaining 43.
  - Result: $[3, 27, 38, 43]$.
- Merge $[9, 82]$ and $[10]$:
  - Compare 9 and 10. Copy 9.
  - Compare 82 and 10. Copy 10.
  - Copy remaining 82.

- Result: $[9, 10, 82]$.

- Merge $[3, 27, 38, 43]$ and $[9, 10, 82]$:
  - $3 < 9 \rightarrow$ Copy 3
  - $27 > 9 \rightarrow$ Copy 9
  - $27 > 10 \rightarrow$ Copy 10
  - $27 < 82 \rightarrow$ Copy 27
  - $38 < 82 \rightarrow$ Copy 38
  - $43 < 82 \rightarrow$ Copy 43
  - Copy remaining 82.
  - Final Result: $[3, 9, 10, 27, 38, 43, 82]$.

The sorted array is $[3, 9, 10, 27, 38, 43, 82]$.

CHAPTER 6

Binary Trees

6.1 Binary Trees and Complete Binary Tree

A binary tree is a hierarchical data structure in which each node has at most two children referred to as the left child and the right child. A strictly binary tree is one where every node has either 0 or 2 children. A complete binary tree is a binary tree in which every level except possibly the last is completely filled and all nodes are as far left as possible.

Methodology:

Array Representation of Complete Binary Tree When a complete binary tree is stored sequentially in a 1-indexed array (where the root is placed at index 1), the following indexing rules apply:

1. The left child of a node at index i is located at index $2i$.
2. The right child of a node at index i is located at index $2i + 1$.
3. The parent of a node at index i (where $i > 1$) is located at index $\lfloor i/2 \rfloor$.

Worked Example:

Finding Node Indices in an Array Representation A complete binary tree is stored in an array. Find the indices of the left child right child and parent of the node located at index 4.
Solution:

1. Given the node index $i = 4$.
2. Left child index: $2i = 2 \times 4 = 8$.
3. Right child index: $2i + 1 = 2 \times 4 + 1 = 9$.
4. Parent index: $\lfloor i/2 \rfloor = \lfloor 4/2 \rfloor = 2$.

6.2 Basic Terms

To analyze binary trees structurally, several foundational terms must be understood:

- **Root**: The topmost node of the tree, which has no parent.
- **Level Number**: The distance of a node from the root. The root is at level 0. The children of a node at level k are at level $k + 1$.
- **Degree**: The number of subtrees (or children) a node directly connects to. In a binary tree, the maximum degree of any node is 2.
- **Leaf Node**: A node with degree 0 (it has no children). Also known as an external node.
- **Internal Node**: A node with a degree of at least 1 (it has at least one child).

Worked Example:

Evaluating Basic Terms of a Binary Tree Consider a binary tree where root A has left child B and right child C. B has left child D and right child E. C has right child F. Identify the leaf nodes the degree of each node and the level of node E.
Solution:

1. **Degrees**: Node A has 2 children (degree 2). Node B has 2 children (degree 2). Node C has 1 child (degree 1). Nodes D E and F have 0 children (degree 0).
2. **Leaf Nodes**: Nodes with degree 0 are D E and F.
3. **Level Number**: Root A is at level 0. Nodes B and C are at level 1. Nodes D E and F are at level 2. Therefore the level of node E is 2.

6.3 Binary Search Tree

A Binary Search Tree (BST) is a specialized binary tree with the following property: for every node, its left subtree contains only nodes with values less than the node's value, and its right subtree contains only nodes with values greater than the node's value.

Methodology:

Insertion in a Binary Search Tree To insert a new value X into a BST:

1. Start at the root. If the tree is empty, create a new node with X as the root.
2. Compare X with the current node's value.
3. If X is less than the current node's value, move to the left child.
4. If X is greater than the current node's value, move to the right child.
5. Repeat steps 2-4 until an empty spot (null child) is found. Insert the new node with value X in that spot.

Worked Example:

Constructing a BST from a Number Sequence Construct a Binary Search Tree by inserting the following sequence of numbers: 50 30 70 20 40 60 80.
Solution:

1. **Insert 50**: Tree is empty. 50 becomes the root.
2. **Insert 30**: $30 < 50$ so 30 becomes the left child of 50.
3. **Insert 70**: $70 > 50$ so 70 becomes the right child of 50.
4. **Insert 20**: $20 < 50$ (go left) then $20 < 30$ (go left). 20 becomes the left child of 30.
5. **Insert 40**: $40 < 50$ (go left) then $40 > 30$ (go right). 40 becomes the right child of 30.
6. **Insert 60**: $60 > 50$ (go right) then $60 < 70$ (go left). 60 becomes the left child of 70.
7. **Insert 80**: $80 > 50$ (go right) then $80 > 70$ (go right). 80 becomes the right child of 70.

Methodology:

Searching in a Binary Search Tree To search for a target value K in a BST:

1. Start at the root node.
2. If the current node is null, K is not in the tree (search fails).
3. If K matches the current node's value, return success.
4. If K is less than the current node's value, recursively search the left subtree.
5. If K is greater than the current node's value, recursively search the right subtree.

Worked Example:

Tracing the Search Operation in a BST Using the BST constructed in the previous example trace the search path for the value 60.
Solution:

1. Start at root: 50. Compare target 60 with 50. Since $60 > 50$ move to the right child.
2. Current node: 70. Compare target 60 with 70. Since $60 < 70$ move to the left child.
3. Current node: 60. Compare target 60 with 60. Match found. Search is successful.

Methodology:

Deletion in a Binary Search Tree To delete a node N from a BST, identify which of the three structural cases applies:

1. **Case 1 (N is a leaf node):** Simply remove the node by setting its parent's pointer to null.
2. **Case 2 (N has exactly one child):** Replace the node with its only child by linking the node's parent directly to the node's child.
3. **Case 3 (N has two children):** Find the inorder successor of N (the smallest value in the right subtree of N). Copy the value of the inorder successor to N. Then recursively delete the original inorder successor (which will always fall into Case 1 or Case 2).

Worked Example:

Deleting Nodes from a Binary Search Tree Consider a BST with root 50. Left subtree nodes are 30 (left child 20 right child 40). Right subtree nodes are 70 (left child 60 right child 80). Demonstrate the deletion of node 20 node 30 and node 50 sequentially.
Solution:

1. **Delete 20 (Leaf Node)**: 20 has no children. It is simply removed. Node 30's left child pointer is set to null.
2. **Delete 30 (Node with One Child)**: Node 30 now only has a right child (40). We replace 30 with 40. Node 50's left child pointer is updated to point directly to 40.
3. **Delete 50 (Node with Two Children)**: 50 has children 40 (left subtree) and 70 (right subtree). Find the inorder successor of 50, which is the smallest value in the right subtree. The right subtree is rooted at 70 and its smallest value is its leftmost child 60. Copy 60 into the root node. Now recursively delete the original node 60 (which is a leaf node). The root is now 60, with left child 40 and right child 70. Node 70's left child pointer becomes null.

6.4 Tree Traversal

Traversal is the systematic process of visiting each node in a tree exactly once. The three standard depth-first traversals recursively visit the Left Subtree (L), Right Subtree (R), and Root node.

Methodology:

Inorder Preorder and Postorder Traversals For a given binary tree:

1. **Inorder Traversal (L - Root - R)**:
   - Traverse the left subtree in inorder.
   - Visit the Root node.
   - Traverse the right subtree in inorder.
2. **Preorder Traversal (Root - L - R)**:
   - Visit the Root node.
   - Traverse the left subtree in preorder.
   - Traverse the right subtree in preorder.
3. **Postorder Traversal (L - R - Root)**:
   - Traverse the left subtree in postorder.
   - Traverse the right subtree in postorder.
   - Visit the Root node.

Worked Example:

Deriving Tree Traversals Consider the following binary tree: Root is A. Left child is B, right child is C. B has left child D and right child E. C has left child F and right child G. E has a left child H. Find the Preorder Inorder and Postorder traversals.

Solution:

1. **Preorder (Root - L - R)**:
   - Visit Root: A
   - Traverse Left Subtree of A: Process B's tree. Root is B, Left is D, Right is E (which has left H). Sequence: B D E H.
   - Traverse Right Subtree of A: Process C's tree. Root is C, Left is F, Right is G. Sequence: C F G.
   - Result: A B D E H C F G
2. **Inorder (L - Root - R)**:
   - Traverse Left Subtree of A: For B's tree, Left is D, Root is B, Right is E (Left H, Root E). Sequence: D B H E.
   - Visit Root: A
   - Traverse Right Subtree of A: For C's tree, Left is F, Root is C, Right is G. Sequence: F C G.
   - Result: D B H E A F C G
3. **Postorder (L - R - Root)**:
   - Traverse Left Subtree of A: For B's tree, Left is D, Right is E (Left H, Root E $\rightarrow$ H E), Root is B. Sequence: D H E B.
   - Traverse Right Subtree of A: For C's tree, Left is F, Right is G, Root is C. Sequence: F G C.
   - Visit Root: A
   - Result: D H E B F G C A

Worked Example:

Inorder Traversal of a Binary Search Tree Using the BST containing the elements 50 30 70 20 40 60 80, perform an inorder traversal and list the visited elements.
Solution:

1. An important property of a Binary Search Tree is that its inorder traversal visits the nodes in an ascending, sorted order.
2. Traversing the left subtree of 50 yields the inorder of the 30-rooted tree: 20 30 40.
3. Visit the root: 50.
4. Traversing the right subtree of 50 yields the inorder of the 70-rooted tree: 60 70 80.
5. Result: 20 30 40 50 60 70 80.

6.5 Applications

Binary trees and their variations are fundamental to many computational algorithms and systemic processes.

1. **Searching and Sorting**: Binary Search Trees provide efficient computational complexity for dynamic data operations, such as searching, inserting, and deleting elements.
2. **Expression Evaluation**: Binary trees are utilized as Expression Trees to represent arithmetic and logical expressions. Traversing these trees helps compilers evaluate and parse instructions (e.g., producing postfix notations from postorder traversal).
3. **Database Indexing**: Balanced variants of binary search trees, particularly B-trees and B+ trees, are heavily employed in relational databases to facilitate extremely fast multi-level indexing and querying.
4. **Data Compression**: Huffman coding trees are binary trees used to assign variable-length binary codes to characters, forming the backbone of many lossless data compression algorithms like ZIP.
5. **Priority Queues and Heaps**: Complete binary trees serve as the underlying data structure for Heaps, which are essential for implementing efficient priority queues and the Heapsort algorithm.

Appendix: Key Formulae

This appendix provides a quick reference to the key formulae and mathematical relationships discussed in this book.

Polynomial Representation (Unit 4)

A polynomial can be represented mathematically as a sum of terms, where each term consists of a coefficient c_i and an exponent e_i:

$$P(x) = c_1 x^{e_1} + c_2 x^{e_2} + \cdots + c_n x^{e_n}$$

Binary Search (Unit 5)

In binary search, the middle index of the current search interval is calculated to divide the search space:

$$\text{mid} = \text{low} + \left\lfloor \frac{\text{high} - \text{low}}{2} \right\rfloor$$

When adjusting the search boundaries based on the comparison with the middle element:

$$\text{high} = \text{mid} - 1 \quad \text{or} \quad \text{low} = \text{mid} + 1$$

Binary Tree Array Representation (Unit 6)

For a complete binary tree (such as a binary heap) represented in an array where the root is at index 1, the relationships for a node at index i are given by:

$$\begin{aligned} \text{Left Child index} &= 2i \\ \text{Right Child index} &= 2i + 1 \\ \text{Parent index} &= \lfloor i/2 \rfloor \end{aligned}$$