BCBM301T

DATA STRUCTURES USING C++

DR. C.V. RAMAN UNIVERSITY
Kargi Road, Kota, Bilaspur, (C. G.),
Ph. 07753-253801, 07753-253872
E-mail: info@cvru.ac.in, Website: www.cvru.ac.in

BCBM301T

DATA STRUCTURES USING C++

Credit- 4

Subject Expert Team

Dr. Ayush Kumar Agrawal Assistant Professor, Dr. CV Raman University, Bilaspur	Dr. Abhinav Shukla Associate Professor, Dr. CV Raman University, Bilaspur
Dr. Abhishek Pathak Professor, Dr. CV Raman University, Bilaspur	Dr. Pritendra Malakar Assistant Professor, Dr. CV Raman University, Bilaspur

Course Edited By:

Dr. Vikrant Gupta
Assistant Professor,
Saheed Nand Kumar Patel University, Raigarh,

Unit Written By:

Dr. Ayaz Khan
Associate Professor,
Dr. CV Raman University, Bilaspur

Warning: All rights reserved, No part of this publication may be reproduced or transmitted or utilized or stored in any form or by any means now known or hereinafter invented, electronic, digital or mechanical, including photocopying, scanning, recording or by any information storage or retrieval system, without prior written permission from the publisher.

Edition: 2026

Publish by: DR. C.V. RAMAN UNIVERSITY

Kargi Road, Kota, Bilaspur, (C. G.),

Ph. 07753-253801, 07753-253872

STRUCTURE OF PROGRAM		
Title of the Program	**:**	**BACHELOR OF COMPUTER APPLICATION**
Program Type	**:**	**UNDERGRADUATE**
Program Credits	**:**	**20**
Program Duration	**:**	**3 YEARS**
Eligibility Criteria	**:**	**12th PASS**

PROGRAMME OBJECTIVE

- To produce employable IT workforce, that will have sound knowledge of IT and business fundamentals that can be applied to develop and customize solutions for Small and Medium Enterprises (SME)
- To develop skilled manpower in the various areas of information technology like: Data base management, Software Development, Computer-Languages, Software engineering, Web based applications etc.
- To Work professionally with positive attitude as an individual or in multidisciplinary teams and communicate effectively."

PROGRAMME OUTCOME

- Develop competence in basic technical subjects in computer applications like Programming Languages, Data Structures, Databases, Operating Systems, Software Engineering.
- Identify, analyze, formulate and develop computer applications.
- Map real life scenarios to various theoretical optimal solutions.
- Provide simplest automated solutions to various legacy systems.
- Use modern computing tools and techniques with confidence.

PROGRAMME SPECIFIC OUTCOME

Upon completion of program in Bachelor of Computer application, students will be able:

- Able to implement computer skill in the areas related to multimedia and website design, Data base, Hardware and networking.
- Able to demonstrate basic knowledge in the areas such as, Software Engineering, Data communication and Networking, Data base management, Web Technology and Operating Systems for building IT applications.
- Students will able to understand, analyze and develop computer programs in the areas related to algorithm, system software, web design and networking for efficient design of computer based system.
- Apply standard software engineering practices and strategies in software project development using open source programming environment to deliver a quality of product for business success.
- Student will able to know various issues, latest trends in technology development and thereby innovate new ideas and solutions to existing problems

STRUCTURE OF COURSE		
Course Title	**:**	**DATA STRUCTURE USING C++**
Course Type	**:**	**CORE COURSE**
Course Code	**:**	**BCBM301T**
Course Credits	**:**	**4**
Semester	**:**	**3**

COURSE LEARNING OBJECTIVES:

- Data structures play a central role in modern computer science. Data structures are essential building blocks in obtaining efficient algorithms.
- The objective of the course is to teach students how to design, write, and analyze the performance of programs that handle structured data and perform more complex tasks, typical of larger software projects.
- Students should acquire skills in using generic principles for data representation & manipulation with a view for efficiency, maintainability, and code reuse.
- Another goal of the course is to teach advance data structures concepts, which allow one to store collections of data with fast updates and queries.

COURSE LEARNING OUTCOMES:

After study this student will be able to know about the concepts of Data Structure Using C++ Language, List & Its Operations Concept of Tree, and Algorithm & Graphs Design. Students will also know about the sorting and searching.

Course Content

BLOCK I	**FUNDAMENTALS AND LINEAR STRUCTURES**	**01-76**
Unit 1	Introduction to Data Structures	03
Unit 2	Analysis of Algorithms	21
Unit 3	Arrays, Sparse Matrices, and Memory Management	35
Unit 4	Stacks	53
BLOCK II	**LINKED LISTS AND HIERARCHICAL STRUCTURES**	**77-168**
Unit 5	Recursion	79
Unit 6	Queues	101
Unit 7	Linked List	125
Unit 8	Introduction to Trees	143
BLOCK III	**ADVANCED STRUCTURES AND ALGORITHMIC TECHNIQUES**	**169-286**
Unit 9	Binary Search Tree (BST) and Advanced Trees	171
Unit 10	Introduction to Graphs	195
Unit 11	Graph Algorithms and Traversals	213
Unit 12	Introduction to Sorting and Basic Sorting Techniques	235
Unit 13	Advanced Sorting Techniques and Comparison	257
BLOCK IV	**SEARCHING, HASHING, AND FILE ORGANIZATION**	**287-346**
Unit 14	Searching and Hashing Techniques	287
Unit 15	Introduction to File Structure	309
Unit 16	Types of File Organization and Indexing	327

BLOCK I	FUNDAMENTALS AND LINEAR STRUCTURES	01-76
Unit 1	Introduction to Data Structures	03
Unit 2	Analysis of Algorithms	21
Unit 3	Arrays, Sparse Matrices, and Memory Management	35
Unit 4	Stacks	53

This block introduces the fundamental concepts of data structures and the basic techniques used to organize, store, and process data efficiently. It develops a foundation in data structures, algorithms, and their role in computer programming and problem-solving. The block covers linear structures in which data elements are arranged sequentially and accessed in an organized manner. Key concepts include arrays, strings, stacks, queues, and their basic operations and applications. Students will learn how data can be inserted, deleted, searched, traversed, and updated using appropriate structures. The block also introduces the relationship between data organization and algorithmic efficiency, helping learners select suitable structures for different computational tasks. Through these concepts, students gain a systematic understanding of how linear data structures support efficient program development. The knowledge gained in this block provides the essential groundwork for studying more advanced structures such as linked lists, trees, graphs, and other algorithmic techniques in the subsequent blocks.

UNIT 1
INTRODUCTION TO DATA STRUCTURES

NOTE

UNIT STRUCTURE

1.1 Introduction
1.2 Objective
1.3 Data Structures
1.4 Types of Data Structures
1.5 Primitive Data Structures in C++
1.6 Non -Primitive Data Structures in C++
1.7 Linear Data Structures in C++
1.8 Non-Linear Data Structures in C++
1.9 Types of Non Linear Data Structures in C++
1.10 Summary
1.11 Glossary
1.12 Answer to check your progress
1.13 Suggested readings - OER
1.14 Reference
1.15 Exercise

1.1 INTRODUCTION

Data structures are fundamental concepts in computer science that provide efficient ways to store, organize, and manage data. In C++, data structures allow programmers to handle large amounts of data effectively and perform operations like insertion, deletion, searching, and sorting with minimal effort. They form the backbone of efficient programming, as the choice of an appropriate data structure can greatly influence the performance of an algorithm or software application.

C++ provides both primitive data types such as int, char, and float, and non-primitive data structures like arrays, linked lists, stacks, queues, trees, graphs, and hash tables. Primitive types are used to store single values, while non-primitive structures allow multiple elements to be organized for complex operations. Linear data structures, such as arrays and linked lists, store elements sequentially, whereas non-linear structures, such as trees and graphs, represent hierarchical or networked relationships.

Understanding data structures is crucial not only for academic purposes but also for real-world problem solving. By mastering these

NOTE

concepts, students can design efficient algorithms, optimize memory usage, and develop software that can handle large-scale data effectively.

1.2 OBJECTIVE

After studying this unit, you will be able to:

1. Define the definition, characteristics, and importance of data structures in C++.
2. Differentiate between primitive and non-primitive data structures.
3. Explain linear data structures and their types, including arrays, linked lists, stacks, and queues.
4. Describe non-linear data structures, such as trees, graphs, and hash tables, with examples and applications.
5. Apply knowledge of data structures to choose the appropriate structure for solving programming problems efficiently.

1.3 DATA STRUCTURES

Data structures are one of the most fundamental concepts in computer science and programming. They provide a way to organize, manage, and store data efficiently so that it can be accessed and modified effectively. In C++, a programming language widely used for system programming, application development, and competitive programming, data structures play a crucial role in designing efficient programs.

At its core, a data structure is a particular way of organizing data in memory to perform operations like insertion, deletion, searching, and sorting. Choosing the right data structure can dramatically improve the performance of a program, while a poor choice can make it slow and inefficient. Data structures are not just about storing data—they are about storing it in a way that matches the requirements of the problem being solved.

C++ provides both primitive data types (like int, float, char) and derived data types (like arrays, pointers, structures, and classes) that can be used to build more complex data structures. It also allows programmers to define their own data structures using classes and templates, making C++ a powerful language for implementing abstract data types.

1.3.1 Importance of Data Structures

The importance of data structures lies in their ability to make software more efficient and scalable. For example, consider a situation where we need to store and manage the marks of 1000 students. Using a simple array is one option, but it has limitations, such as fixed size and

NOTE

difficulty in insertion and deletion. In contrast, data structures like linked lists or binary trees allow dynamic storage, faster search, and easier insertion or deletion.

Some of the key benefits of using proper data structures are:

1. Efficient Data Management: Proper data structures help manage large amounts of data efficiently, reducing memory usage and improving execution time.
2. Faster Access and Modification: Using structures like hash tables or trees, data can be searched, inserted, or deleted much faster compared to simple arrays.
3. Code Reusability and Modularity: Data structures encourage modular programming, allowing developers to implement reusable code components.
4. Optimized Problem Solving: Many complex algorithms, like sorting, searching, and graph traversal, rely heavily on appropriate data structures to function efficiently.

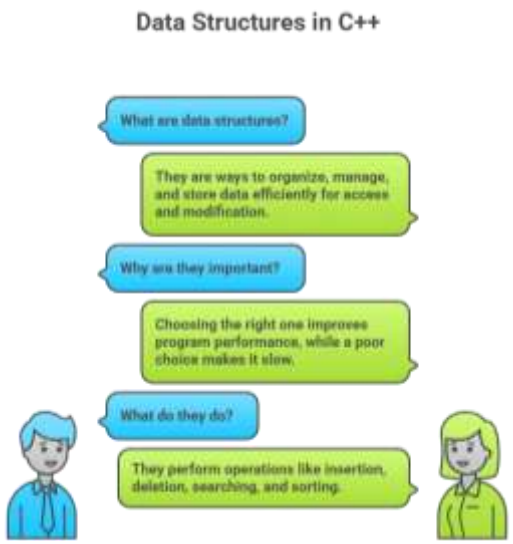


Fig 1.1 - Basics of Data Structures in C++

1.3.2 Types of Data Structures

Data structures in C++ can be broadly categorized into two types:

1. Primitive Data Structures: These are the basic data types provided by C++ itself, such as int, float, double, and char. They store single values and are used as building blocks for more complex structures.
2. Non-Primitive Data Structures: These are more advanced structures that can store collections of data. They are further classified into:

NOTE

- Linear Data Structures: Elements are arranged sequentially. Examples include arrays, linked lists, stacks, and queues. Linear structures are easy to implement and access but may be inefficient for certain operations.
- Non-Linear Data Structures: Elements are arranged hierarchically or in a connected network. Examples include trees and graphs. Non-linear structures are useful when relationships between data elements are complex.

1.3.2.1 Common Data Structures in C++

- Arrays: Arrays are collections of elements of the same type stored in contiguous memory locations. They provide fast access using indices but have a fixed size, limiting flexibility.
- Linked Lists: A linked list consists of nodes connected by pointers. Each node contains data and a reference to the next node. Linked lists allow dynamic memory allocation and efficient insertion/deletion.
- Stacks and Queues: These are abstract data types used to manage data in a specific order. Stacks follow LIFO (Last In, First Out) principle, whereas queues follow FIFO (First In, First Out).
- Trees: Trees represent hierarchical relationships. A binary tree is a common type where each node has at most two children. Trees are used in databases, file systems, and expression parsing.
- Graphs: Graphs represent networks of nodes connected by edges. They are used in social networks, routing algorithms, and optimization problems.
- Hash Tables: Hash tables use hash functions to map keys to values, allowing fast data retrieval. Collision handling techniques are used to manage conflicts.

1.3.3 Data Structures and Algorithms

Data structures and algorithms go hand in hand. While data structures define how data is stored, algorithms define how data is processed. For example, a binary search algorithm requires data to be stored in a sorted array or a binary search tree to function correctly. Similarly, sorting algorithms such as Quick Sort or Merge Sort depend on the structure of the data to optimize performance.

In C++, the Standard Template Library (STL) provides built-in data structures like vector, list, map, and set with ready-to-use functions for insertion, deletion, traversal, and searching. STL helps programmers

save time and implement efficient solutions without manually coding each structure from scratch.

NOTE

1.4 TYPES OF DATA STRUCTURES

Data structures are ways to organize and store data for efficient access and modification. In C++, data structures are broadly classified into Primitive and Non-Primitive types.

1. Primitive Data Structures:
These are the basic data types provided by C++, such as int, float, char, and bool. They store single values and are used as building blocks for more complex structures. For example, an int array can store marks of students, while a char variable can store a grade.

2. Non-Primitive Data Structures:
These structures can store multiple values and are divided into Linear and Non-Linear types.

- Linear Data Structures: Elements are stored sequentially. Examples include:
    - Array: A collection of elements of the same type. Example: int marks[5].
    - Linked List: Nodes connected via pointers. Example: Student record list.
    - Stack: Follows LIFO (Last In, First Out). Example: Undo feature.
    - Queue: Follows FIFO (First In, First Out). Example: Printer queue.
- Non-Linear Data Structures: Elements are connected hierarchically or as a network. Examples include:
    - Tree: Hierarchical data, like a file system.
    - Graph: Represents networks, like social media connections.
    - Hash Table: Stores key-value pairs for fast access, like a dictionary.

1.5 PRIMITIVE DATA STRUCTURES IN C++

Primitive data structures are the most basic building blocks in programming. They store single values and are provided natively by the programming language. In C++, primitive data structures are essential for creating variables, performing calculations, and forming the foundation for more complex data structures. They are simple to use,

NOTE

memory-efficient, and form the basis of algorithms and higher-level structures.

1.5.1 Definition

A primitive data structure is a data type that stores a single value in memory. These structures are atomic, meaning they cannot be broken down into simpler structures within the programming language. They are typically built into the language and optimized for fast access and manipulation.

1.5.2 Types of Primitive Data Structures in C++

1. Integer (int)
    - Description: Stores whole numbers without decimals, either positive or negative.
    - Memory: Usually 4 bytes (may vary by system).
    - Example Uses: Counting items, storing marks of students, maintaining loop counters.
    - Example:
    - int age = 21;
    - int totalMarks = 95;
2. Floating Point (float and double)
    - Description: Stores decimal numbers. float is single precision, and double is double precision.
    - Memory: float – 4 bytes, double – 8 bytes.
    - Example Uses: Storing measurements, prices, percentages.
    - Example:
    - float price = 299.99;
    - double pi = 3.14159265;
3. Character (char)
    - Description: Stores a single character, such as letters, digits, or symbols.
    - Memory: 1 byte.
    - Example Uses: Storing grades, single letters, or symbols.
    - Example:
    - char grade = 'A';
    - char symbol = '';

NOTE

4. Boolean (bool)
    - Description: Stores logical values true or false.
    - Memory: Typically 1 byte.
    - Example Uses: Conditions, flags, or status indicators.
    - Example:
    - bool isPassed = true;
    - bool isEmpty = false;
5. Wide Character (wchar_t)
    - Description: Stores larger character sets, like Unicode characters, allowing internationalization.
    - Memory: Usually 2 or 4 bytes.
    - Example Uses: Handling multilingual text.
    - Example:
    - wchar_t symbol = L'Ω';

1.5.3 Characteristics of Primitive Data Structures

- Atomic: Each variable stores a single value.
- Efficient: Directly managed by the system, making them very fast.
- Fixed Size: Memory allocated depends on the data type.
- Foundation for Complex Structures: Arrays, structures, and classes are built using primitive types.

1.5.4 Importance in Programming

Primitive data structures are crucial because they form the base for all computations in programming. Every program uses primitive types to store and process data. They also provide the foundation for arrays, which are linear collections of primitive elements. Moreover, logical decisions, loops, and arithmetic operations are implemented using primitive data types.

1.6 NON-PRIMITIVE DATA STRUCTURES IN C++

Non-primitive data structures are advanced data structures that store multiple values and are used to organize and manipulate collections of data efficiently. Unlike primitive data structures, which store a single value, non-primitive structures are capable of handling large datasets and complex relationships. They form the backbone of modern programming and are essential for implementing algorithms efficiently.

NOTE

Definition

A non-primitive data structure is a data type that stores a collection of data elements, often of the same type, and allows various operations like insertion, deletion, traversal, and searching. These structures are classified into Linear and Non-Linear types depending on how the data elements are stored and connected.

Check Your Progress 1 .1

Multiple Choice Question (MCQ)

1. What is the main purpose of using data structures?
   a) To increase program length
   b) To organize and manage data efficiently
   c) To avoid using algorithms
   d) To reduce hardware usage

B. True / False

1. Data structures help in efficient data storage and retrieval. (True / False)

C. Short Answer Questions

1. What is a data structure?
2. Name any two types of data structures.

1.7 LINEAR DATA STRUCTURES IN C++

Linear data structures are a category of non-primitive data structures in which data elements are arranged sequentially, one after another. In these structures, each element has a unique predecessor and successor, except for the first and last elements. Linear structures are widely used in programming because of their simplicity, ease of implementation, and efficiency in accessing and processing data. They form the foundation for more complex data structures and algorithms.

1.7.1 Definition

A linear data structure is a structure in which elements are stored in a linear order, and operations like insertion, deletion, traversal, and searching are performed sequentially. In C++, linear structures can be implemented using arrays, linked lists, stacks, and queues.

1.7.2 Types of Linear Data Structures

1.7.2.1 Arrays

NOTE

- Definition: An array is a collection of elements of the same data type stored in contiguous memory locations.
- Characteristics:
    - Fixed size, determined at the time of declaration.
    - Direct access to elements using an index.
    - Efficient for retrieving elements quickly.
- Examples:

1. int marks[5] = {90, 85, 78, 92, 88}; – storing student marks.
2. char vowels[5] = {'a', 'e', 'i', 'o', 'u'}; – storing vowels.

- Applications: Storing data that does not change frequently, such as static tables, grades, or sensor readings.

1.7.2.2 Linked Lists

- Definition: A linked list is a collection of nodes where each node contains data and a pointer to the next node.
- Characteristics:
    - Dynamic size, memory is allocated at runtime.
    - Easy insertion and deletion compared to arrays.
    - Sequential access is required to reach a specific node.
- Examples:

1. A student record list with Name, Roll Number, and pointer to the next node.
2. Music playlist storing Song Name and pointer to the next song.

- Applications: Implementing dynamic memory storage, handling variable-sized datasets, and building complex structures like stacks and queues.

1.7.2.3 Stack

- Definition: A stack is a linear structure that follows LIFO (Last In, First Out) principle.
- Operations: Push (insert), Pop (remove), Peek (retrieve top element).
- Examples:
    1. Undo functionality in a text editor.
    2. Browser back button navigation history.
- Applications: Function call management in programming, expression evaluation, and recursion handling.

NOTE

1.7.2.4 Queue

- Definition: A queue is a linear structure that follows FIFO (First In, First Out) principle.
- Operations: Enqueue (insert at rear), Dequeue (remove from front).
- Examples:
    1. Ticket booking system where the first person in line is served first.
    2. Printer job scheduling in a print queue.
- Applications: Job scheduling, process management in operating systems, and task handling.

Advantages of Linear Data Structures

- Simple and easy to implement.
- Efficient memory utilization for small datasets.
- Useful for sequential data processing.
- Serve as the basis for implementing non-linear structures.

Limitations

- Arrays have a fixed size, making them less flexible.
- Linked lists require extra memory for pointers.
- Accessing an element in a linked list requires traversal from the head.

1.8 NON-LINEAR DATA STRUCTURES IN C++

Non-linear data structures are data structures in which elements are not arranged sequentially. Unlike linear structures, there is no strict predecessor-successor relationship, and data elements can have multiple relationships with other elements. Non-linear structures are essential when the relationship among elements is complex, such as hierarchical relationships or networked connections.

Definition

A non-linear data structure is a data type in which data elements are connected in a hierarchical or networked manner. They allow for efficient representation of complex relationships and are widely used in applications like databases, computer networks, and artificial intelligence. Non-linear data structures include trees, graphs, and hash tables.

NOTE

1.9 TYPES OF NON-LINEAR DATA STRUCTURES IN C++

1.9.1 Trees

- Definition: A tree is a hierarchical structure consisting of nodes connected by edges. Each node contains data and may have child nodes. The topmost node is called the root, and nodes without children are called leaves.
- Characteristics:
    - Efficient for hierarchical data representation.
    - Each child node has exactly one parent.
- Examples:
    1. Binary Tree: Used in expression evaluation and compiler design.
    2. File System Directory: Represents folders and subfolders on a computer.
- Applications: Database indexing, searching algorithms, and file system management.

1.9.2 Graphs

- Definition: A graph consists of a set of vertices (nodes) connected by edges. Graphs can be directed (edges have direction) or undirected (edges are bidirectional).
- Characteristics:
    - Represents complex relationships among data elements.
    - Can model networks, paths, and connections.
- Examples:
    1. Social networks like Facebook or LinkedIn, where nodes are users and edges represent friendships.
    2. Airline flight routes connecting cities.
- Applications: Network routing, pathfinding algorithms, recommendation systems, and circuit design.

1.9.3 Hash Tables

- Definition: A hash table is a data structure that stores key-value pairs. A hash function converts a key into an index to allow fast access to the corresponding value.
- Characteristics:

NOTE

- Provides almost constant-time search on average.
- Collisions (when two keys map to the same index) are handled using chaining or open addressing.

- Examples:
  1. Dictionary where the word is the key and its meaning is the value.
  2. Student records with Roll Number as the key for fast lookup.
- Applications: Fast data retrieval, caching, and implementing associative arrays.

1.9.4 Advantages of Non-Linear Data Structures

- Efficient for representing complex relationships and hierarchies.
- Optimized for searching and sorting operations in large datasets.
- Can model real-world problems such as networks, family trees, and organizational hierarchies.

1.9.5 Limitations

- More complex to implement compared to linear data structures.
- Memory management can be challenging, especially with dynamic structures.
- Traversal often requires advanced algorithms like recursion or graph traversal techniques (DFS, BFS).

NOTE

Activity 1 : Choosing the Right Data Structure for a Real-Life Objective:
To help students understand the importance of selecting appropriate data structures in C++ based on problem requirements.

Activity Description:
Students are given a real-life scenario and must identify suitable **primitive and non-primitive data structures**, justify their choices, and explain the operations involved.

Scenario:
A college wants to develop a **Student Management System** with the following requirements:

1. Store basic student details such as Roll Number, Name, Age, and Grade.
2. Maintain a list of students enrolled in a course.
3. Manage a queue of students waiting for course registration.
4. Keep track of the most recently accessed student record.
5. Quickly search student details using Roll Number.

Tasks for Students:

1. Identify which **primitive data types** will be used to store individual student details and explain why.
2. Select appropriate **linear data structures** (array, linked list, stack, or queue) for requirements 2, 3, and 4, with justification.
3. Choose a **non-linear data structure** for requirement 5 and explain how it improves efficiency.
4. List the basic operations (insert, delete, search, access) performed on each chosen data structure.
5. Write a short explanation (150–200 words) describing how proper data structure selection improves system performance.

Learning Outcome:
By completing this activity, students will be able to relate theoretical concepts of data structures in C++ to real-world applications and understand how the correct choice of data structure enhances efficiency and scalability.

1.10 SUMMARY

Data structures are organized ways of storing and managing data efficiently. They are broadly classified into **primitive** and **non-primitive** types. Primitive data structures, such as int, char, and float, store single values, whereas non-primitive structures handle multiple data elements and include **linear** and **non-linear** types.

Linear data structures, like arrays, linked lists, stacks, and queues, store elements sequentially and are suitable for applications requiring order-based processing. Arrays allow fast indexing, linked lists allow dynamic memory management, stacks follow **LIFO** (Last

NOTE

In, First Out), and queues follow **FIFO** (First In, First Out). Non-linear data structures, such as trees, graphs, and hash tables, represent hierarchical or networked relationships. Trees are used for hierarchical data representation, graphs for modeling networks, and hash tables for efficient key-based data retrieval. Understanding data structures is crucial for implementing efficient algorithms, optimizing memory usage, and solving real-world problems in programming. Mastery of these concepts forms the foundation for advanced topics in software development and computer science.

1.11 GLOSSARY

- **Data Structure:** A way to organize and store data for efficient access and modification.
- **Primitive Data Structure:** Basic data types like int, char, and float storing single values.
- **Non-Primitive Data Structure:** Structures storing multiple data elements, including arrays, lists, trees, and graphs.
- **Linear Data Structure:** Elements arranged sequentially, e.g., arrays, stacks, queues.
- **Non-Linear Data Structure:** Elements connected hierarchically or as a network, e.g., trees, graphs.
- **Array:** A collection of elements of the same type stored in contiguous memory.
- **Linked List:** A dynamic collection of nodes connected via pointers.
- **Stack:** A linear structure following **LIFO** principle.
- **Queue:** A linear structure following **FIFO** principle.
- **Hash Table:** A key-value based data structure allowing fast search using a hash function.

1.12 ANSWER TO CHECK YOUR PROGRESS-POSSIBLE ONE

Check Your Progress 1.1

MCQ

1. Correct answer: **b) To organize and manage data efficiently**

B. True / False

1. **True**

C. Short Answer

1. A data structure is a way of organizing and storing data for efficient use.

NOTE

2. Linear and non-linear data structures.

1.13 SUGGESTED READINGS – OER

Sr. No.	Title	URL	QR Code
1	Data Structure Using C++	http://books.aisectonline.com/User/UnitUrls?unitId=3709	

1.14 REFERENCE

1. Data Structures and Algorithms in C++ – Adam Drozdek – ISBN: 978-8131503098 – 2013
2. Data Structures Using C++ – D. S. Malik – ISBN: 978-8131522266 – 2011
3. Data Structures and Algorithm Analysis in C++ – Mark Allen Weiss – ISBN: 978-8131728262 – 2013
4. Fundamentals of Data Structures in C++ – Ellis Horowitz, Sartaj Sahni – ISBN: 978-8173716058 – 2007
5. Data Structures Through C++ – Yashavant Kanetkar – ISBN: 978-8176560917 – 2010
6. Object-Oriented Programming with C++ – E. Balagurusamy – ISBN: 978-9350947085 – 2014
7. Introduction to Algorithms – Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein – ISBN: 978-0262033848 – 2009
8. The C++ Programming Language – Bjarne Stroustrup – ISBN: 978-0321563842 – 2013

1.15 EXERCISE

Part 1

A. Multiple choice question

1. Which data structure stores elements sequentially?
   a) Tree
   b) Graph
   c) Array
   d) Hash Table
2. A stack follows which principle?
   a) FIFO

NOTE

b) LIFO
c) Random Access
d) None of the above

3. Which of the following is a non-linear data structure?
   a) Array
   b) Queue
   c) Graph
   d) Stack

4. What is the main advantage of linked lists over arrays?
   a) Sequential access
   b) Dynamic memory allocation
   c) Fixed size
   d) Easy indexing

5. Which data structure uses key-value pairs for fast data access?
   a) Array
   b) Stack
   c) Hash Table
   d) Queue

Answers with Explanations:

1 Correct Answer: c) Array

Explanation: An array stores elements of the same data type in contiguous (sequential) memory locations. Each element can be accessed directly using its index, making arrays efficient for sequential storage and fast access. Trees, graphs, and hash tables do not store elements sequentially in memory.

2. Correct Answer: b) LIFO

Explanation: A stack follows the Last In, First out (LIFO) principle, meaning the element that is inserted last is removed first. This behavior is similar to a stack of plates—only the top plate can be removed first.

3. Correct Answer: c) Graph

Explanation: A graph is a non-linear data structure because its elements (nodes) are connected in a network-like structure, not in a single sequence. Arrays, stacks, and queues store data sequentially and are linear data structures.

4. Correct Answer: b) Dynamic memory allocation

Explanation: Linked lists can grow or shrink dynamically during program execution, as memory is allocated at runtime. In contrast, arrays have a fixed size defined at the time of declaration, making them less flexible.

NOTE

5. Correct Answer: c) Hash Table

Explanation: A hash table stores data in key-value pairs and uses a hash function to convert keys into memory locations. This allows very fast data access, usually in constant time, making hash tables ideal for search operations.

B. Small answer type question

1. Define a data structure and explain its importance.
2. Differentiate between primitive and non-primitive data structures with examples.
3. Explain the LIFO and FIFO principles with examples.
4. What is a linked list, and how is it different from an array?
5. Briefly describe the applications of trees and graphs in programming.

Part 2

C. Long answer type question (answer in 200 words)

1. Explain linear data structures in detail with examples of arrays, stacks, queues, and linked lists.
2. Describe non-linear data structures, including trees, graphs, and hash tables, with their applications.
3. Discuss the advantages and limitations of using arrays and linked lists in C++.
4. Explain how stacks and queues are implemented in C++ and provide real-world examples.
5. Describe the importance of data structures in algorithm design and real-world problem-solving.

Part 3

D. Long answer type question (answer in 300 words)

1. Discuss primitive and non-primitive data structures in C++ comprehensively, including types, characteristics, and examples.
2. Explain linear data structures in depth, covering arrays, linked lists, stacks, and queues, including operations, advantages, limitations, and use cases.
3. Discuss non-linear data structures such as trees, graphs, and hash tables in detail, explaining their types, characteristics, applications, and examples.
4. Compare linear and non-linear data structures, highlighting differences, advantages, and scenarios where each is suitable.

NOTE

5. Explain how data structures impact algorithm efficiency and real-world applications, including examples like databases, networking, and operating systems.

UNIT 2
ANALYSIS OF ALGORITHMS

NOTE

UNIT STRUCTURE

2.1 Introduction
2.2 Objective
2.3 Introduction to Algorithms
2.4 Criteria of a Good Algorithm, Time Complexity and Space Complexity
2.5 Summary
2.6 Glossary
2.7 Answer to check your progress
2.8 Suggested readings - OER
2.9 Reference
2.10 Exercise

2.1 INTRODUCTION

Algorithms are the backbone of computer programming and play a critical role in problem-solving. They provide a step-by-step procedure to solve a specific problem efficiently and accurately. While it is important for an algorithm to be correct, understanding its quality and efficiency is equally vital. Not all algorithms that produce correct results are practical, especially when dealing with large datasets or limited resources.

To evaluate an algorithm, we consider certain criteria of a good algorithm, such as correctness, finiteness, definiteness, efficiency, and generality. These characteristics ensure that the algorithm is not only accurate but also reliable, understandable, and applicable to a range of problems.

Apart from these qualitative aspects, algorithms are also analyzed quantitatively using time complexity and space complexity. Time complexity measures the computational time an algorithm requires in relation to the size of input, helping to predict how it scales for large datasets. Space complexity assesses the memory requirements of an algorithm, including storage for input, auxiliary variables, and temporary computations. Understanding these concepts allows programmers to design solutions that are both efficient and resource-conscious, forming the foundation for advanced topics in data structures and programming

2.2 OBJECTIVE

NOTE

After studying this unit, you will be able to:

- Explain the criteria of a good algorithm and understand why each criterion is important.
- Define time complexity and calculate it for simple algorithms.
- Define space complexity and analyze memory requirements for different algorithms.
- Compare algorithms based on their efficiency in terms of time and space.
- Apply these concepts to design and evaluate efficient algorithms for solving computational problems

2.3 INTRODUCTION TO ALGORITHMS

In computer science, an algorithm is the foundation of problem-solving. It serves as a step-by-step procedure or a set of rules that precisely defines how to solve a specific problem. The concept of algorithms is central to all fields of programming and data structures because they provide a systematic way to perform computations, organize data, and solve real-world problems efficiently.

An algorithm can be considered a blueprint for writing programs. No matter which programming language is used, the underlying algorithm defines the logic and the steps to reach the desired solution. Therefore, understanding algorithms is essential before diving into coding.

2.3.1 Definition of an Algorithm

An algorithm is formally defined as:

"A finite set of well-defined instructions or rules that provides a step-by-step solution for a problem or task."

Let's break this down:

1. Finite set of instructions: An algorithm must terminate after executing a certain number of steps. It cannot go on infinitely.
2. Well-defined instructions: Each step must be clear and unambiguous so that it can be executed precisely.
3. Input and Output: An algorithm receives input values, processes them, and produces the desired output.
4. Effectiveness: Every step in the algorithm must be basic enough to be performed, in principle, by a human or a machine.

2.3.2 Characteristics of Algorithms

A good algorithm has certain essential characteristics that make it effective, reliable, and efficient:

NOTE

1. Finiteness: The algorithm must always terminate after a finite number of steps. For instance, an algorithm that searches for a number in a finite list must conclude once the number is found or the list is fully traversed.
2. Definiteness: Each instruction in the algorithm should be clear and unambiguous. For example, instead of saying "sort the numbers", an algorithm should specify whether to use bubble sort, insertion sort, or any other sorting method.
3. Input: An algorithm should accept zero or more inputs. For example, in a program that calculates the sum of two numbers, the inputs are the two numbers themselves.
4. Output: An algorithm must produce at least one output, which is the solution to the problem. In the sum example, the output is the sum of the two numbers.
5. Effectiveness: All steps must be basic enough to be performed easily and efficiently. Complex steps should be broken down into simpler ones.
6. Generality: A good algorithm should be applicable to a set of problems rather than a single specific instance. For example, a sorting algorithm should be able to sort any list of numbers, not just a predefined set.

2.3.3 Role of Algorithms in Problem Solving

Algorithms are crucial because they provide a clear roadmap for solving problems systematically. Without algorithms, solving complex computational problems would be chaotic and inefficient. Some key roles of algorithms in problem-solving are:

1. Efficiency: Algorithms optimize the use of resources like time and memory. For example, sorting a list using quicksort is much faster than using a simple bubble sort for large datasets.
2. Predictability: Algorithms provide a predictable sequence of steps, which ensures consistency in the results. This predictability is essential in software development and system design.
3. Scalability: Well-designed algorithms can handle increasing amounts of data efficiently. This is especially important in fields like data analysis, artificial intelligence, and database management.
4. Automation: By converting an algorithm into a program, repetitive tasks can be automated. For instance, algorithms help in automating billing systems, search engines, and banking operations.

NOTE

5. Problem Decomposition: Algorithms allow complex problems to be broken down into simpler, manageable steps, facilitating better understanding and easier implementation.

2.3.4 Examples of Algorithms

1. Finding the Maximum Number in a List:
    - Input: A list of numbers
    - Steps:
        1. Initialize max as the first number in the list.
        2. Compare max with each element of the list.
        3. If an element is greater than max, update max.
        4. Continue until the end of the list.
    - Output: The largest number in the list
2. Bubble Sort Algorithm:
    - Input: An unsorted list of numbers
    - Steps:
        1. Compare adjacent elements in the list.
        2. Swap them if they are in the wrong order.
        3. Repeat the process for all elements until the list is sorted.
    - Output: Sorted list of numbers

These examples highlight how an algorithm can be simple yet powerful, forming the foundation for programming logic and advanced computational tasks.

Check Your Progress 2.1

Multiple Choice Question (MCQ)

1. What does time complexity measure?
   a) Memory used by an algorithm
   b) Number of instructions in a program
   c) Execution time relative to input size
   d) Programming language used

B. True / False

1. Algorithm analysis helps compare the efficiency of different algorithms. (True / False)

NOTE

C. Short Answer Questions

1. What is meant by time complexity?

.....

.....

2. What is the purpose of analyzing algorithms?

.....

.....

2.4 CRITERIA OF A GOOD ALGORITHM, TIME COMPLEXITY, AND SPACE COMPLEXITY

Algorithms are the backbone of computer programming and problem-solving. While understanding what an algorithm is forms the foundation, it is equally important to know how to evaluate its quality and efficiency. This is done through criteria that define a good algorithm and the concepts of time and space complexity, which measure how efficiently an algorithm performs.

2.4.1 Criteria of a Good Algorithm

A good algorithm is not only correct but also efficient, easy to understand, and reliable. Several criteria are used to evaluate whether an algorithm is of high quality:

1. Correctness
    - An algorithm is correct if it produces the expected output for all valid inputs.
    - For example, a sorting algorithm should correctly sort any list of numbers, not just specific examples.
2. Finiteness
    - The algorithm must terminate after a finite number of steps.
    - Infinite loops or endless recursion violate this criterion.
3. Definiteness
    - Each step of the algorithm must be precisely defined and unambiguous.
    - Ambiguous instructions can lead to different interpretations, making implementation difficult.
4. Input and Output
    - An algorithm should accept well-defined inputs and produce well-defined outputs.

NOTE

- Example: In a program calculating the sum of two numbers, the inputs are the two numbers, and the output is their sum.

5. Efficiency
    - A good algorithm uses the least possible resources (time and memory) to solve the problem.
    - Efficient algorithms are essential for handling large datasets or time-critical applications.
6. Generality
    - The algorithm should be applicable to a class of problems, not just a single instance.
    - Example: A search algorithm should work for any array of numbers or strings, not just a pre-specified array.
7. Understandability
    - The algorithm should be easy to understand and follow.
    - Clear steps improve maintainability and reduce the chance of errors during implementation.



FIG 2.1: Criteria of Good Algorithm

2.4.2 Time Complexity

Time complexity measures how the execution time of an algorithm changes with the size of the input. It gives an estimate of the resources required to solve a problem in term s of time.

- Importance: Knowing the time complexity helps programmers choose the most efficient algorithm for large datasets.

NOTE

- Big-O Notation: Time complexity is often expressed using Big-O notation, which describes the upper bound of the algorithm's growth rate.

2.4.2.1 Examples

1. Linear Search
    - Steps: Check each element in the list sequentially until the target is found.
    - If the list has n elements, in the worst case, all elements are checked.
    - Time Complexity: O(n)
2. Binary Search
    - Steps: Divide the sorted list in half, check the middle element, and repeat on the appropriate half.
    - With n elements, the number of comparisons is roughly $\log_2 n$.
    - Time Complexity: O(log n)
3. Bubble Sort
    - Steps: Repeatedly compare and swap adjacent elements until the list is sorted.
    - For n elements, it requires approximately n^2 comparisons.
    - Time Complexity: $O(n^2)$

Complexity	Example	Characteristics
O(1)	Accessing an array element	Constant time
O(log n)	Binary search	Time grows logarithmically
O(n)	Linear search	Time grows linearly
$O(n^2)$	Bubble sort	Time grows quadratically
O(n!)	Permutations generation	Time grows factorially

Table 2.1: Classification of Time Complexity:

Time complexity provides a quantitative way to compare algorithms and select the most suitable one for a given problem.

2.4.3 Space Complexity

Space complexity measures the amount of memory or storage an algorithm requires during execution. This includes memory for:

1. Input data

NOTE

2. Auxiliary variables
3. Temporary storage used during computation

- Space efficiency is crucial in applications where memory is limited, such as embedded systems or mobile devices.

2.4.3.1 Examples

1. Iterative Fibonacci Calculation
    - Uses two variables to store previous Fibonacci numbers.
    - Space Complexity: O(1)
2. Recursive Fibonacci Calculation
    - Uses recursion stack proportional to n.
    - Space Complexity: O(n)
3. Merge Sort
    - Requires additional space to store temporary arrays during merging.
    - Space Complexity: O(n)

Key Considerations:

- Algorithms with low time complexity may sometimes require high space, and vice versa. This trade-off is known as time-space tradeoff.
- Choosing an algorithm involves balancing execution speed and memory usage based on the problem constraints.

Activity 2.1: Identifying and Analyzing an Algorithm from Daily Life:

Objective:

To help students understand the concept of algorithms, their characteristics, and the idea of time and space complexity by relating them to a real-life problem.

Activity Description:

Ask students to choose one daily-life task (any one of the following or similar):

- Making a cup of tea
- Searching for a book in a cupboard
- Calculating the total bill while shopping
- Finding the maximum mark from a list of exam scores

Task to be Performed:

1. Write the Algorithm

NOTE

- Write a step-by-step algorithm for the chosen task.
  - Clearly mention: Input, Output, Each step in sequence

2. Identify Characteristics of the Algorithm
   From the written steps, identify and explain how the algorithm satisfies:
3. Analyse Efficiency (Conceptual)
   - Discuss how the time complexity would change if:
     - The number of inputs increases (e.g., more items to search or calculate).
   - Discuss the space complexity by identifying:
     - What variables or extra memory are required.
4. Reflection Question (Short Answer)
   - Explain why having a clear algorithm before writing a program is important.

Expected Learning Outcome:
After completing this activity, students will be able to:

- Understand algorithms as step-by-step problem-solving methods
- Relate algorithm characteristics to practical examples
- Gain an introductory idea of time and space complexity
- Appreciate the importance of algorithms in efficient problem solving

2.5 SUMMARY

Algorithms are step-by-step procedures designed to solve specific problems efficiently and accurately. A good algorithm is not only correct but also exhibits key characteristics such as finiteness, definiteness, efficiency, generality, and understandability. Correctness ensures that the algorithm produces the desired output for all valid inputs, while finiteness guarantees that it terminates after a finite number of steps. Definiteness ensures each step is clear and unambiguous, and efficiency relates to optimal use of resources like time and memory. Generality allows an algorithm to handle a range of problem instances, and understandability makes it easier to implement and maintain.

To evaluate an algorithm quantitatively, time complexity and space complexity are used. Time complexity measures the computational time an algorithm takes relative to the input size, helping to assess scalability and performance. Space complexity measures the memory required, including input, auxiliary variables, and temporary storage, which is crucial for memory-constrained systems. Efficient algorithms

NOTE

balance both time and space, though sometimes a trade-off exists. Understanding these concepts is essential for designing optimized programs that are fast, resource-friendly, and capable of handling large datasets. Mastery of algorithm evaluation forms the foundation for advanced programming, data structures, and problem-solving in computer science.

2.6 GLOSSARY

1. Algorithm – A finite set of well-defined instructions for solving a problem.
2. Correctness – The property of an algorithm producing accurate results for all inputs.
3. Finiteness – The property of an algorithm terminating after a finite number of steps.
4. Definiteness – Each step of an algorithm must be unambiguous and clear.
5. Time Complexity – The measure of time an algorithm takes to run in relation to input size.
6. Space Complexity – The measure of memory used by an algorithm during execution.
7. Efficiency – The use of minimal resources (time and memory) to achieve the desired result.
8. Generality – Ability of an algorithm to solve a class of problems, not just a single instance.
9. Big-O Notation – Mathematical notation used to describe the upper bound of an algorithm's growth rate.
10. Auxiliary Space – Temporary or extra memory required by an algorithm apart from input storage.

2.7 ANSWER TO CHECK YOUR PROGRESS-POSSIBLE ONE

Check Your Progress 2.1

MCQ

1. Correct answer: c) Execution time relative to input size

B. True / False

1. True

NOTE

C. Short Answer

1. Time complexity measures how execution time grows with input size.
2. It helps select the most efficient algorithm for a given problem.

2.8 SUGGESTED READINGS – OER

Sr. No.	Title	URL	QR Code
1	Data Structure Using C++	http://books.aisectonline.com/User/UnitUrls?unitId=3710	

2.9 REFERENCE

- Data Structures and Algorithms in C++ – Adam Drozdek – ISBN: 978-8131716229 – 2013
- Introduction to Algorithms – Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest & Clifford Stein – ISBN: 978-0262033848 – 2009
- Fundamentals of Data Structures in C++ – Ellis Horowitz, Sartaj Sahni & Dinesh P. Mehta – ISBN: 978-8173716060 – 2007
- Data Structures Using C and C++ – Yedidyah Langsam, Moshe J. Augenstein & Aaron M. Tenenbaum – ISBN: 978-8131727485 – 2011
- Algorithms – Robert Sedgewick & Kevin Wayne – ISBN: 978-0321573513 – 2011
- Data Structures and Algorithm Analysis in C++ – Mark Allen Weiss – ISBN: 978-8131726228 – 2014
- An Introduction to Data Structures with Applications – Jean-Paul Tremblay & Paul G. Sorenson – ISBN: 978-0070660037 – 2007
- Programming and Problem Solving with C++ – Nell Dale & Chip Weems – ISBN: 978-0763717218 – 2012

2.10 EXERCISE

Part 1

A. Multiple choice question

NOTE

1. Which of the following is a criterion of a good algorithm?
a) Infinite loop
b) Correctness
c) Ambiguity
d) Randomness

2. Time complexity measures:
a) Memory usage
b) Execution time relative to input size
c) Number of inputs
d) Programming language

3. Space complexity includes:
a) Input memory
b) Auxiliary memory
c) Temporary storage
d) All of the above

4. Which algorithm is more efficient for searching in a sorted array?
a) Linear Search
b) Binary Search
c) Bubble Sort
d) Merge Sort

5. Big-O notation is used to:
a) Measure memory only
b) Measure input size
c) Represent the upper bound of algorithm growth
d) Write algorithms in C++

Answers with Explanations:

1 Answer: b) Correctness

Explanation: A good algorithm must produce the correct output for all valid inputs. Correctness ensures that the algorithm solves the intended problem accurately. Options like infinite loops or ambiguity violate the basic characteristics of an algorithm because they prevent the algorithm from terminating or producing reliable results. Randomness is not a desirable criterion in standard algorithms as it can lead to unpredictable outcomes.

2. **Answer: b) Execution time relative to input size**

Explanation: Time complexity evaluates how the running time of an algorithm grows as the input size increases. It is used to analyze and compare algorithms for efficiency, especially with large datasets. Memory usage is related to space complexity, not time complexity. The programming language does not define time complexity, although it can slightly affect execution speed.

NOTE

3. Answer: d) All of the above

Explanation: Space complexity is the total memory an algorithm requires to execute. This includes:

- Memory for storing input data
- Auxiliary memory used for variables and data structures
- Temporary storage like stacks in recursion or temporary arrays in sorting algorithms

4. Answer: b) Binary Search

Explanation: Binary Search is highly efficient for sorted arrays because it divides the array in half repeatedly, reducing the number of comparisons to O(log n). Linear search scans each element one by one, taking O(n) time, which is slower for large datasets. Bubble Sort and Merge Sort are sorting algorithms, not search algorithms, so they are irrelevant for searching.

5. Answer: c) Represent the upper bound of algorithm growth

Explanation: Big-O notation describes the worst-case growth rate of an algorithm as input size increases. It helps compare algorithms and predict their efficiency for large inputs. It is not used just for memory measurement, input size, or coding syntax—it is a mathematical notation for analyzing algorithm performance.

B. Small answer type question

1. Define a good algorithm and explain its importance.
2. What is time complexity and why is it important in evaluating algorithms?
3. What is space complexity? Explain its relevance in programming.
4. List and briefly describe three criteria of a good algorithm.
5. Differentiate between time complexity and space complexity with examples

Part 2

C. Long answer type question (answer in 200 words)

1. Explain the characteristics of a good algorithm in detail.
2. Discuss time complexity and its significance with examples of common algorithms.
3. Describe space complexity and its importance in algorithm design.
4. Explain the trade-off between time and space complexity with suitable examples.

NOTE

5. Compare linear search and binary search in terms of efficiency and applicability.

Part 3

D. Long answer type question (answer in 300 words)

1. Discuss in detail the criteria of a good algorithm with examples.
2. Explain time complexity with examples of different algorithms and their analysis.
3. Discuss space complexity, its calculation, and relevance in practical programming.
4. Explain the concept of time-space trade-off and how it affects algorithm design.
5. Compare and contrast different searching algorithms (linear, binary, hashing) in terms of time and space complexity.

UNIT 3
ARRAYS, SPARSE MATRICES, AND MEMORY MANAGEMENT

NOTE

UNIT STRUCTURE

3.1 Introduction

3.2 Objective

3.3 Array Data Structure

3.4 Sparse Matrices

3.5 Garbage Collection in Data Structures

3.6 Benefits and Disadvantages of Data Structures

3.7 Summary

3.8 Glossary

3.9 Answer to check your progress

3.10 Suggested readings - OER

3.11 Reference

3.12 Exercise

3.1 INTRODUCTION

Data structures are fundamental in computer science for organizing, storing, and managing data efficiently. Among these, arrays form the backbone of most data management systems. An array is a collection of elements of the same type, stored in contiguous memory locations, which allows easy access through indexing. Arrays provide simplicity and speed for tasks like searching, sorting, and traversing. However, they come with limitations such as fixed size and potential wastage of memory.

In contrast, sparse matrices are specialized data structures used for storing matrices that contain a large number of zero elements. Instead of storing all elements, only non-zero elements are stored along with their row and column indices, leading to significant memory savings and faster computations in applications like scientific simulations and engineering problems.

Another critical concept in this unit is memory management, which ensures that memory is allocated and deallocated efficiently during program execution. Techniques like garbage collection automatically

NOTE

reclaim memory occupied by unused objects, reducing memory leaks and improving program stability. Understanding the benefits and disadvantages of these data structures helps developers choose the right structure for specific problems, balancing efficiency, speed, and memory usage. This unit equips students with essential knowledge to manage data effectively in C++ programming.

3.2 OBJECTIVE

After completing this unit, you will be able to:

1. Understand the concept, types, and applications of arrays in programming.
2. Implement and manipulate sparse matrices to optimize memory usage.
3. Explain garbage collection mechanisms and their role in memory management.
4. Analyze the advantages and limitations of different data structures.
5. Apply appropriate data structures for solving real-world problems efficiently.

3.3 ARRAY DATA STRUCTURE

An array can be defined as:

"A collection of similar type of data elements stored at contiguous memory locations and accessible using a single name along with an index or subscript."

In simpler terms, an array allows you to store multiple values under a single variable name, where each value can be accessed using an index number.

Example:

int marks[5] = {85, 90, 75, 60, 95};

Here:

- marks is the array name.
- int is the data type of the array elements.
- 5 is the size of the array (it can store 5 integers).
- {85, 90, 75, 60, 95} are the elements stored in the array.

Accessing elements in the array is done using indices (starting from 0 in C++):

NOTE

```
cout << marks[0]; // Output: 85
cout << marks[3]; // Output: 60
```

Characteristics of Arrays

1. Homogeneous elements – All elements in an array must be of the same data type (int, float, char, etc.).
2. Fixed size – The size of an array must be known at compile time (static arrays) or dynamically allocated in memory (dynamic arrays).
3. Contiguous memory allocation – Elements are stored in consecutive memory addresses.
4. Indexed access – Each element can be directly accessed using its index.
5. Efficient element access – Accessing an element by index is O(1) time complexity.

3.3.1 Types of Arrays

Arrays in C++ can be classified based on dimensions:

3.3.1.1 One-Dimensional Array

A simple list of elements arranged linearly.

Example:

```
int numbers[5] = {1, 2, 3, 4, 5};
```

3.3.1.2 Two-Dimensional Array (Matrix)

An array of arrays, often used to represent tables or matrices.

Example:

```
int matrix[2][3] = {
   {1, 2, 3},
   {4, 5, 6}
};
```

Here, the first index represents the row, and the second index represents the column.

Accessing an element:

```
cout << matrix[1][2]; // Output: 6
```

3.3.1.3 Multi-Dimensional Array

Arrays with more than two dimensions. Used in complex mathematical computations like 3D graphics, simulations, or scientific calculations.

NOTE

Example:

int tensor[2][3][4];

This represents a 3D array with dimensions 2 × 3 × 4.

3.3.2 Array Operations

Arrays support various operations that are essential in programming:

Traversal

Accessing each element of the array one by one.

```
for(int i = 0; i < 5; i++) {
    cout << numbers[i] << " ";
}
```

Insertion

Adding an element at a particular index. Note: For static arrays, insertion may require shifting elements to make space.

Deletion

Removing an element by index and shifting remaining elements to fill the gap.

Searching

Finding the index of a given element. Can be linear search or binary search if the array is sorted.

```
int searchElement = 75;
for(int i = 0; i < 5; i++) {
    if(marks[i] == searchElement) {
        cout << "Element found at index " << i;
    }
}
```

Updating

Modifying an element at a specific index:

```
marks[2] = 80; // Updates the third element to 80
```

3.3.3Applications of Arrays

- Storing marks of students in a class.
- Matrices in mathematics for calculations.

NOTE

- Implementing other data structures like stack, queue, heap, hash table.
- Image processing – Pixel values stored in 2D arrays.
- Databases – Arrays can store small fixed-size records efficiently.

Array Representation in Memory

Arrays are stored in contiguous memory locations. For a one-dimensional array:

int arr[5] = {10, 20, 30, 40, 50};

Memory layout:

Index	0	1	2	3	4
Value	10	20	30	40	50

- Address of arr[i] = base_address + i * size_of_element

For multi-dimensional arrays, row-major or column-major order is used to calculate memory locations

3.4 SPARSE MATRICES

In computer science and mathematics, a matrix is a two-dimensional array of numbers arranged in rows and columns. While matrices are commonly used in mathematics, engineering, and computer programming, certain matrices contain a large number of zero elements compared to non-zero elements. These are called sparse matrices.

A sparse matrix is formally defined as:

"A matrix in which the number of zero elements is significantly greater than the number of non-zero elements."

Conversely, a dense matrix has very few zero elements. Sparse matrices are important because storing all elements including zeros leads to inefficient memory usage, especially for very large matrices in scientific computations, graph algorithms, and data analysis.

Characteristics of Sparse Matrices

1. High zero-element content – More than half (often >70%) of the elements are zeros.
2. Large dimensions – Sparse matrices are usually large in size (e.g., 1000 × 1000) but only a few entries are non-zero.
3. Efficient storage required – Specialized storage techniques are used to avoid wasting memory on zeros.
4. Operations focus on non-zero elements – Algorithms are optimized to process only non-zero entries for efficiency.

NOTE

Here, only 5 elements are non-zero, and the rest are zeros.

Importance of Sparse Matrices

- Memory Efficiency: Storing only non-zero elements saves a lot of memory in large matrices.
- Faster Computations: Many mathematical and scientific operations can ignore zero elements, improving speed.
- Applications:
    - Graph algorithms (adjacency matrices for sparse graphs)
    - Solving systems of linear equations in engineering and physics
    - Image compression
    - Machine learning models with high-dimensional sparse data

3.4.1 Storage Techniques for Sparse Matrices

To save memory, various storage schemes are used for sparse matrices instead of using a regular 2D array.

Coordinate List (COO) Representation

- Store only non-zero elements along with their row and column indices.
- Three arrays are used: row[], col[], and value[].

Example:

Row[] = {0, 1, 2, 3, 1};

Col[] = {2, 0, 3, 1, 2};

Value[] = {3, 22, 7, 5, 8};

- Advantages: Simple to implement, easy to insert new elements.
- Disadvantages: Searching for a specific element is slow (O(n)).

3.4.1.1 Compressed Row Storage (CRS)

- Also called Row Compressed Storage.
- Stores non-zero elements row-wise along with column indices and row pointers.

3.4.1.2 Compressed Column Storage (CCS)

- Similar to CRS but stores elements column-wise.
- Useful when column operations are more frequent than row operations.

NOTE

3.4.1.3 Linked List Representation

- Each non-zero element is stored as a node in a linked list containing:
    - Row index
    - Column index
    - Value
    - Pointer to next non-zero element

Advantages: Dynamic memory usage, insertion and deletion easy. Disadvantages: Random access is slower (O(n) traversal required).

3.4.2 Basic Operations on Sparse Matrices

Even with sparse storage, common matrix operations are needed:

Operation	Methodology	Key Advantage / Format Note
Addition	Only add non-zero elements that share the exact same coordinates (i, j).	Time complexity is reduced significantly compared to standard dense matrix addition.
Multiplication	Multiply only non-zero elements, completely ignoring zero entries.	Efficient algorithms are specifically designed for matrices stored in **CRS** (Compressed Row Storage) or **CCS** (Compressed Column Storage) formats.
Transpose	Swap the row and column indices of the non-zero elements.	This is particularly straightforward to implement when using the **COO** (Coordinate List) representation.
Searching	Search for a specific value by checking *only* the list of non-zero entries.	Drastically reduces search time by avoiding the need to scan empty (zero) cells.

Table 3.1 Optimized Operations on Sparse Matrices

Advantages of Sparse Matrices

NOTE

1. Memory efficient – Only store non-zero elements.
2. Faster computations – Avoid unnecessary operations on zeros.
3. Scalable – Can handle very large matrices.
4. Optimized algorithms – Many linear algebra and graph algorithms are optimized for sparse matrices.

Disadvantages of Sparse Matrices

1. Complexity in implementation – Special storage and operations are required.
2. Slower random access – Unlike dense arrays, accessing a specific element is not always O(1).
3. Overhead of metadata – Row/column indices and pointers require extra memory.
4. Not efficient for dense data – If zeros are few, sparse representation wastes space.

Applications of Sparse Matrices

- Graph Theory: Adjacency matrices for sparse graphs.
- Engineering & Physics: Solving large systems of linear equations.
- Computer Graphics: Image processing and compression.
- Machine Learning: Storing high-dimensional sparse data like text vectors.
- Scientific Computing: Simulations with large, mostly empty datasets.

Example in C++ (COO Representation)

```
#include <iostream>
using namespace std;

struct Element {
    int row;
    int col;
    int value;
};

int main() {
    Element sparse[5] = { {0, 2, 3}, {1, 0, 22}, {2, 3, 7}, {3, 1, 5}, {1, 2, 8} };
```

NOTE

```
    int n = 5;

    cout << "Sparse Matrix Elements (row, col, value):\n";
    for(int i = 0; i < n; i++) {
        cout << "(" << sparse[i].row << ", " << sparse[i].col << ", " << sparse[i].value << ")\n";
    }
    return 0;
}
```

Output:

(0, 2, 3)

(1, 0, 22)

(2, 3, 7)

(3, 1, 5)

(1, 2, 8)

This stores only non-zero elements, reducing memory usage significantly.

Check Your Progress 3.1

A. Multiple Choice Question (MCQ)

1. Why are sparse matrices used?
   a) To increase computation time
   b) To store all elements including zeros
   c) To reduce memory usage
   d) To avoid using arrays

B. True / False

1. Arrays store elements in contiguous memory locations. (True / False)

C. Short Answer Questions

1. What is an array?

.....

.....

2. What is memory management in programming?

.....

.....

NOTE

3.5 GARBAGE COLLECTION IN DATA STRUCTURES

In computer programming, memory management is a crucial aspect of efficient software design. When a program runs, it allocates memory dynamically to store data structures such as linked lists, trees, graphs, and arrays. However, sometimes memory that is no longer needed by the program is not automatically released. This can lead to memory leaks, which reduce system performance.

Garbage Collection (GC) is a process by which a programming environment automatically identifies and frees memory that is no longer in use. This prevents memory wastage and helps programs run efficiently.

Why Garbage Collection is Needed

- Dynamic memory allocation: In languages like C++, memory can be allocated at runtime using new or malloc. Forgetting to free this memory leads to memory leaks.
- Automatic cleanup: Modern languages like Java, Python, and C# use garbage collection to automatically manage memory.
- Avoid program crashes: If memory is exhausted due to unfreed objects, the program may crash or behave unpredictably.

Example in C++ without GC:

```
int* ptr = new int[100];  // Allocates memory
// ptr is no longer used, but memory is not freed
// Memory leak occurs
```

With manual deletion:

```
delete[] ptr;  // Memory freed
```

In languages with garbage collection, the system handles such memory cleanup automatically.

3.5.1 How Garbage Collection Works

Garbage collection identifies unreachable objects—objects that cannot be accessed or referenced by the program. There are multiple techniques:

Reference Counting

- Each object maintains a count of references pointing to it.
- When the reference count drops to zero, the object is no longer in use and is deallocated.

Example:

NOTE

```
class Node {
public:
    int data;
    Node* next;
};


Node* a = new Node();
Node* b = a;  // Reference count = 2
b = nullptr;  // Reference count = 1
a = nullptr;  // Reference count = 0 -> Node deallocated
```

Limitations: Cannot handle cyclic references (e.g., two objects referencing each other).

Mark-and-Sweep Algorithm

- Mark phase: Traverse all objects reachable from program roots (like global variables or stack references) and mark them as "alive."
- Sweep phase: Scan memory, and delete all unmarked objects (garbage).
- Advantages: Can handle cyclic references.
- Disadvantages: May temporarily pause program execution during collection (stop-the-world problem).

Generational Garbage Collection

- Objects are grouped based on age:
    - Young generation: Recently created objects.
    - Old generation: Long-lived objects.
- Most objects die young, so frequent minor collections in the young generation improve efficiency.

Used in languages like Java and C#.

3.6 BENEFITS AND DISADVANTAGES OF DATA STRUCTURES

Benefits of Garbage Collection

1. Automatic memory management – Reduces programmer effort to manually manage memory.

NOTE

2. Prevents memory leaks – Frees unreachable memory automatically.
3. Improves program reliability – Reduces crashes due to memory exhaustion.
4. Supports dynamic data structures – Efficient for linked lists, trees, and graphs that frequently allocate and deallocate memory.

Disadvantages of Garbage Collection

1. Performance overhead – GC consumes CPU cycles, potentially slowing down the program.
2. Non-deterministic – Exact time when memory is freed is unpredictable.
3. Memory fragmentation – GC may not always reclaim memory contiguously.
4. Cannot prevent all memory issues – Logical errors, like retaining unnecessary references, can still cause memory leaks.

Activity 1: The "Space Saver" – Sparse Matrix Compression

Objective: To understand the limitations of standard arrays for storing data with predominantly zero values and to implement a Sparse Matrix representation (specifically the Coordinate List or "Triplet" method) to demonstrate effective memory management and storage optimization.

Materials / Tools / Software Required:

- Computer: With a standard development environment.
- Programming Language: C, C++, or Python (C/C++ is preferred for explicit memory visibility).
- IDE/Text Editor: (e.g., VS Code, CLion, or PyCharm).
- Concept Guide: A brief chart showing Row-Major vs. Column-Major mapping.
- Memory Profiler (Optional): Tools like valgrind (for C++) or sys.getsizeof (for Python) to verify actual memory usage.

Expected Outcome: By the end of this activity, students will be able to:

1. Create a standard 2D array (Dense Matrix) and populate it with "sparse" data (e.g., 90% zeros).
2. Develop an algorithm to scan the dense matrix and convert it into a 3-column structure (Row Index, Column Index, Value).
3. Calculate and Compare the memory usage of the original 2D array versus the new 3-column structure.
4. Visualize how specific memory addresses change when moving from a contiguous block (array) to the compressed format.

3.7 SUMMARY

NOTE

1. This unit introduces arrays, sparse matrices, and memory management, which are crucial for efficient data handling in programming. Arrays are one of the simplest and most widely used data structures, allowing the storage of multiple elements of the same type in a sequential manner. They provide constant-time access using indices, which is useful for searching and sorting operations. However, their fixed size and potential memory wastage are significant drawbacks that developers must consider.
2. Sparse matrices address the inefficiencies of storing large matrices with many zero elements. By storing only non-zero elements along with their positions, sparse matrices save memory and reduce computational overhead. They are particularly useful in fields like scientific computing, engineering simulations, and graph algorithms.
3. Memory management is another critical aspect of data structures. Proper allocation and deallocation of memory ensure that programs run efficiently without leaks or crashes. Garbage collection automates memory reclamation, freeing programmers from manual memory management and reducing errors.
4. Understanding the benefits and disadvantages of data structures allows programmers to choose the most suitable structure for their specific application. Arrays are simple and fast but limited in size, while sparse matrices optimize memory usage but may have slower access times. Efficient memory management ensures optimal performance and stability of applications. By mastering these concepts, students gain the skills to handle data effectively, write efficient programs, and make informed decisions about which data structures to use in various scenarios.

3.8 GLOSSARY

1. Array – A collection of elements of the same type stored in contiguous memory.
2. Index – A number representing the position of an element in an array.
3. Sparse Matrix – A matrix with a majority of zero elements, stored efficiently.
4. Non-zero element – An element in a matrix that is not zero.

NOTE

5. Memory Management – The process of allocating and deallocating memory in a program.
6. Garbage Collection – Automatic process of reclaiming memory occupied by unused objects.
7. Contiguous Memory – A block of memory addresses stored sequentially.
8. Memory Leak – A situation where memory is not properly deallocated.
9. Data Structure – A method of organizing and storing data for efficient access and modification.
10. Efficiency – The ability to perform operations using minimal resources and time.

3.9 ANSWER TO CHECK YOUR PROGRESS-POSSIBLE ONE

Check Your Progress 3.1

MCQ

1. Correct answer: c) To reduce memory usage

B. True / False

1. True

C. Short Answer

1. An array is a collection of same-type elements stored in contiguous memory.
2. Memory management involves allocating and deallocating memory efficiently.

3.10 SUGGESTED READINGS – OER

Sr. No.	Title	URL	QR Code
1	Data Structure Using C++	http://books.aisectonline.com/User/UnitUrls?unitId=3711	

NOTE

3.11 REFERENCE

1. Data Structures and Algorithms in C++ – Michael T. Goodrich, Roberto Tamassia, David M. Mount – ISBN: 9780470383278 – 2011
2. Data Structures and Algorithm Analysis in C++ – Mark Allen Weiss – ISBN: 9780132847377 – 2013
3. Data Structures and Algorithms in C++ – Adam Drozdek – ISBN: 9781133608426 – 2012
4. Data Structures Using C++ – D. S. Malik – ISBN: 9788131518236 – 2012
5. Data Structures Through C++ – Yashavant Kanetkar – ISBN: 9789355515834 – 2024
6. Fundamentals of Data Structures in C++ – Ellis Horowitz, Sartaj Sahni, Dinesh Mehta – ISBN: 9788173716065 – 2008
7. Data Structures: A Pseudocode Approach with C – Richard F. Gilberg, Behrouz A. Forouzan – ISBN: 9780534953965 – 2005
8. Data Structures and Program Design in C++ – Robert L. Kruse, Alexander J. Ryba – ISBN: 9780136159079 – 2010

3.12 EXERCISE

Part 1

A. Multiple choice question

1. Which of the following is true about arrays?
   a) Elements can be of different types
   b) Memory is allocated dynamically only
   c) Elements are stored in contiguous memory locations
   d) Arrays cannot be accessed using indices
2. Sparse matrices are mainly used to:
   a) Increase computation time
   b) Reduce memory usage
   c) Store all zero elements
   d) Avoid arrays
3. Garbage collection is used to:
   a) Allocate new memory
   b) Automatically reclaim unused memory
   c) Store array elements
   d) Sort data
4. Which of the following is a disadvantage of arrays?
   a) Fast access
   b) Fixed size

NOTE

c) Easy traversal
d) Contiguous storage

5. Non-zero elements in a sparse matrix are stored along with:
a) Their value only
b) Their row and column indices
c) Only column index
d) Entire matrix

Answers with Explanations:

1. Answer: C) Elements are stored in contiguous memory locations
Explanation: Arrays store elements in contiguous memory blocks, allowing fast index-based access.

2. Answer: B) Reduce memory usage
Explanation: Sparse matrices contain mostly zeros; storing only non-zero elements saves significant memory.

3. Answer: B) Automatically reclaim unused memory
Explanation: Garbage collection frees memory that is no longer referenced, preventing memory leaks.

4. Answer: B) Fixed size
Explanation: Arrays have a predetermined size, making it difficult to resize dynamically during execution.

5. Answer: B) Their row and column indices
Explanation: To reconstruct the matrix efficiently, sparse matrix storage keeps each non-zero element along with its row and column positions.

B. Small answer type question

1. What is an array, and how is it stored in memory?
2. Explain the difference between sparse and dense matrices.
3. What are the advantages of using sparse matrices?
4. Define garbage collection and its importance in programming.
5. List the disadvantages of arrays

Part 2

C. Long answer type question (answer in 200 words)

1. Explain the structure, advantages, and applications of arrays in C++.

NOTE

2. Describe how a sparse matrix is represented and implemented in programming.
3. Discuss different methods of memory management in data structures.
4. Explain the process and importance of garbage collection in detail.
5. Analyze the benefits and limitations of using arrays and sparse matrices in practical applications

Part 3

D. Long answer type question (answer in 300 words)

1. Explain the difference between one-dimensional and two-dimensional arrays and give real-world examples of their use.
2. Discuss the difference between static and dynamic memory allocation for arrays in C++, highlighting the advantages and disadvantages of each.
3. How can a sparse matrix be represented using the triplet form? Explain with a simple example.
4. What problems can occur if memory is not managed properly in a program, and how does garbage collection help prevent these issues?
5. Compare arrays and linked lists in terms of memory usage, access time, and practical applications. When would you choose one over the other?

UNIT 4
STACKS

NOTE

UNIT STRUCTURE

4.1 Introduction

4.2 Objective

4.3 Introduction to Stacks

4.4 Push and Pop Operations in Stack

4.5 Stack Implementation Using Arrays (Static Implementation)

4.6 Summary

4.7 Glossary

4.8 Answer to check your progress

4.9 Suggested readings - OER

4.10 Reference

4.11 Exercise

4.1 INTRODUCTION

Stacks are one of the most fundamental linear data structures in computer science. They operate on the Last In, First Out (LIFO) principle, meaning that the last element inserted is the first to be removed. Conceptually, a stack can be compared to a pile of plates, where only the top plate is accessible at any given time. Stacks are widely used in programming for managing function calls, expression evaluation, recursion, undo/redo operations, and more.

Stacks can be implemented in two primary ways: using arrays (static implementation) or linked lists (dynamic implementation). Array-based stacks have a fixed size and are simple to implement, but may suffer from overflow. Linked list stacks, on the other hand, are dynamic and allow flexible memory allocation, avoiding the limitations of fixed size.

The Abstract Data Type (ADT) of a stack defines the operations it supports—push, pop, peek, isEmpty, and isFull—without specifying the underlying implementation. Advanced applications include expression conversion and evaluation (infix, postfix, prefix), operator precedence handling, and managing multiple stacks in memory-efficient ways. Understanding stacks provides a foundation for mastering other data structures and algorithms, making them a crucial topic in programming and system design.

NOTE

4.2 OBJECTIVE

After completing this unit, you will be able to:

1. Understand the concept and principles of stacks, including the LIFO property.
2. Implement stacks using arrays and linked lists in C++.
3. Perform basic operations like push, pop, peek, and check for underflow/overflow.
4. Apply stacks in expression conversion, postfix evaluation, and recursion.
5. Understand and implement multiple stacks in a single memory block efficiently

4.3 INTRODUCTION TO STACKS

A stack is a linear data structure that follows a Last In First Out (LIFO) principle. This means that the element inserted last into the stack is the first one to be removed. You can think of it like a stack of plates: the plate placed on the top is the first one you take off.

In computer science, stacks are used to store data temporarily while performing tasks that require reversing operations or tracking execution order. Stacks can be implemented using arrays, linked lists, or dynamic memory allocation.

Key Characteristics

- LIFO order: Last element added is the first to be removed.
- Restricted access: Insertion and deletion can only occur at one end, called the top of the stack.
- Operations: Limited to basic operations like push, pop, peek/top, and isEmpty.

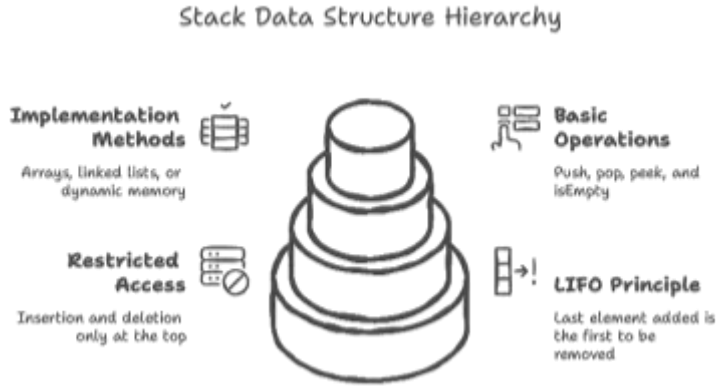


Fig 4.1: Stack Data structure Hierarchy

NOTE

4.3.1 Basic Operations on Stack

Stacks are simple but very powerful due to their operations. The main operations are:

4.3.1.1 Push

The push operation adds an element to the top of the stack.

- Condition: Stack must not be full (in case of array implementation).
- Example: If stack contains [10, 20, 30] and we push 40, it becomes [10, 20, 30, 40].

4.3.1.2 Pop

The pop operation removes the top element from the stack.

- Condition: Stack must not be empty.
- Example: If stack contains [10, 20, 30, 40] and we pop, the stack becomes [10, 20, 30].

4.3.1.3 Peek/Top

The peek operation allows viewing the top element without removing it.

- Example: For stack [10, 20, 30], peek returns 30.

4.3.1.4 isEmpty

Checks whether the stack is empty. Returns true if empty, false otherwise.

4.3.1.5 isFull

Applicable for stack implemented using arrays. Returns true if the stack has reached its maximum capacity.

4.3.2 Types of Stack

Stacks can be classified based on the way they are implemented:

4.3.2.1 Static Stack (Array-based)

- Uses a fixed-size array.
- Easy to implement but size is limited.
- Example in C++:

```
#define MAX 100

int stack[MAX];

int top = -1;


void push(int x) {
```

NOTE

```
    if(top == MAX-1) {
        cout << "Stack Overflow";
    } else {
        stack[++top] = x;
    }
}
```

4.3.2.2 Dynamic Stack (Linked List-based)

- Uses linked nodes and can grow or shrink dynamically.
- No fixed size limit.
- Efficient memory utilization.

4.3.2.3 Specialized Stacks

- Double-ended stack (Deque): Can push/pop at both ends.
- Priority stack: Elements are removed based on priority rather than insertion order.

4.3.3 Applications of Stack

Stacks are widely used in computer science and programming. Some important applications include:

4.3.3.1 Expression Evaluation

Stacks are used in converting infix expressions to postfix/prefix and evaluating them.

4.3.3.2 Function Call Management

Programming languages use call stacks to manage function calls, recursion, and local variables. The last function called is the first to complete execution.

4.3.3.3 Undo Mechanism

Applications like text editors use stacks to implement undo and redo operations. The last action performed is the first one to undo.

4.3.3.4 Browser History

Web browsers use stacks to track page history. The last visited page is the first to go back to when clicking the back button.

4.3.3.5 Backtracking Algorithms

Stacks are used in algorithms that require backtracking, such as solving mazes, puzzles (like Sudoku), and parsing tasks.

4.3.4 Advantages of Stack

1. Simplicity: Easy to implement and understand.

NOTE

2. Efficient memory usage: Dynamic stacks grow only as needed.
3. Supports recursion: Ideal for managing recursive function calls.
4. Useful in algorithm design: Helps in evaluating expressions, parsing, and reversing sequences.

4.3.5 Disadvantages of Stack

1. Limited access: Only the top element is accessible.
2. Fixed size in arrays: Array-based stacks may cause overflow if the size is not sufficient.
3. Not suitable for all problems: For tasks requiring random access, stacks are inefficient.

4.3.6 Stack Implementation in C++ (Simple Example)

```
#include<iostream>
using namespace std;

#define MAX 5
int stack[MAX];
int top = -1;

void push(int x) {
   if(top == MAX-1) cout << "Stack Overflow\n";
   else stack[++top] = x;
}

void pop() {
   if(top == -1) cout << "Stack Underflow\n";
   else cout << "Popped: " << stack[top--] << endl;
}

void display() {
   if(top == -1) cout << "Stack is empty\n";
   else {
      for(int i = top; i >= 0; i--) cout << stack[i] << " ";
      cout << endl;
```

NOTE

```
    }
}

int main() {
    push(10);
    push(20);
    push(30);
    display();
    pop();
    display();
    return 0;
}
```

Output:

30 20 10

Popped: 30

20 10

This program demonstrates a simple array-based stack with push, pop, and display operations.

4.4 PUSH AND POP OPERATIONS IN STACK

Stacks are linear data structures that follow the Last In, First Out (LIFO) principle. The stack allows insertion and deletion of elements only from one end called the top. The two most fundamental operations in a stack are Push and Pop, which handle insertion and deletion, respectively. These operations are the foundation of all stack-based algorithms and applications.

4.4.1 Push Operation

The Push operation is used to insert an element at the top of the stack. When an element is pushed, it becomes the new top of the stack, and any subsequent push operation will place the next element above it.

4.4.1.1 Algorithm for Push Operation

1. Check if the stack is full (this is called stack overflow).
    - If yes, display an error message: *"Stack Overflow"*.
2. If the stack is not full:
    - Increment the top pointer by 1.

NOTE

- Insert the new element at the position indicated by top.

3. End.

Example

Suppose we have an empty stack with a maximum size of 5.

Top Stack Elements

-1 Empty

Operation: Push 10

- Top is incremented from -1 to 0.
- 10 is inserted at position 0.

Top Stack Elements

0 10

Next Operation: Push 20

- Top becomes 1.
- 20 is inserted at the top.

Top Stack Elements

1 20, 10

The Push operation is successful as long as the stack has space.

Overflow Condition

- If the top reaches the maximum size minus 1, any further push will cause stack overflow.
- Example: For a stack of size 5, if the top is 4, pushing another element is not allowed.

4.4.2 Pop Operation

The Pop operation is used to remove an element from the top of the stack. The element removed is always the most recently inserted element due to the LIFO nature.

4.4.2.1. Algorithm for Pop Operation

1. Check if the stack is empty (this is called stack underflow).
    - If yes, display an error message: *"Stack Underflow"*.
2. If the stack is not empty:
    - Retrieve the element at the top of the stack.
    - Decrement the top pointer by 1.

NOTE

- Return the retrieved element.

3. End.

Example

Consider the stack:

Top Stack Elements

1 20, 10

Operation: Pop

- The top element 20 is removed.
- Top is decremented from 1 to 0.

Top Stack Elements

0 10

Next Operation: Pop again

- The top element 10 is removed.
- Top is decremented from 0 to -1 (stack becomes empty).

Top Stack Elements

-1 Empty

Underflow Condition

- If the stack is empty (top = -1), any pop operation will result in stack underflow.

4.4.3 Push and Pop Implementation in C++ (Array-based)

```
#include<iostream>
#define MAX 5
using namespace std;

int stack[MAX];
int top = -1;

void push(int x) {
   if(top == MAX - 1) {
      cout << "Stack Overflow\n";
   } else {
      top++;
```

NOTE

```
        stack[top] = x;
        court << x << " pushed into stack\n";
    }
}

void pop() {
    if(top == -1) {
        cout << "Stack Underflow\n";
    } else {
        cout << stack[top] << " popped from stack\n";
        top--;
    }
}

int main() {
    push(10);
    push(20);
    push(30);
    pop();
    pop();
    pop();
    pop();  // Demonstrates underflow
    return 0;
}
```

Output:

```
10 pushed into stack
20 pushed into stack
30 pushed into stack
30 popped from stack
20 popped from stack
10 popped from stack
Stack Underflow
```

NOTE

This program clearly demonstrates push and pop operations, along with overflow and underflow conditions.

4.4.4 Key Points to Remember

1. Push: Adds an element to the top of the stack.
2. Pop: Removes the top element of the stack.
3. Overflow occurs when pushing into a full stack.
4. Underflow occurs when popping from an empty stack.
5. Push and Pop are O(1) operations, meaning they execute in constant time.

Check Your Progress 4.1

MCQ:

1. Which of the following operations removes the top element from a stack?

a) Push
b) Pop
c) Peek
d) Enqueue

True/False:Stacks follow the First In First Out (FIFO) principle.

Short Questions:

1. Define a stack in computer science.

.....

.....

2. Name two common applications of stacks.

.....

.....

4.5 STACK IMPLEMENTATION USING ARRAYS (STATIC IMPLEMENTATION)

A stack can be implemented in several ways, and one of the simplest is using an array. This method is known as static implementation because the size of the stack is fixed at compile time. Array-based stacks are easy to implement and efficient for small, known-sized stacks.

4.5.1 Concept of Array-based Stack

In an array-based stack, elements are stored in contiguous memory locations. A variable, usually called top, is used to track the index of the topmost element in the stack.

NOTE

- Initially, the stack is empty, so top = -1.
- When an element is pushed, top is incremented, and the element is placed at stack[top].
- When an element is popped, the element at stack[top] is removed, and top is decremented.

4.5.2 Advantages of Array-based Stack

1. Simplicity: Easy to implement using arrays.
2. Direct Access: Array elements are stored in contiguous memory, allowing direct access via indices.
3. Efficiency: Push and Pop operations are O(1) (constant time).

4.5.3 Limitations of Static Stack

1. Fixed Size: Once declared, the maximum size cannot change.
2. Overflow Risk: Pushing more elements than the maximum size causes stack overflow.
3. Memory Wastage: If the stack uses only part of the array, the unused portion is wasted.

4.5.4 Algorithm for Stack Operations Using Array

4.5.4.1 Push Operation

1. Check if the stack is full (top == MAX-1).
    - If yes, display "Stack Overflow" and exit.
2. Increment top by 1.
3. Insert the element at stack[top].

4.5.4.2 Pop Operation

1. Check if the stack is empty (top == -1).
    - If yes, display "Stack Underflow" and exit.
2. Retrieve the element at stack[top].
3. Decrement top by 1.

4.5.4.3 Peek Operation

1. If stack is empty, return "Stack is empty".
2. Otherwise, return stack[top].

4.5.5 C++ Implementation of Static Stack Using Array

```
#include <iostream>

#define MAX 5
```

NOTE

```
using namespace std;

int stack[MAX];
int top = -1;

// Function to push an element
void push(int x) {
    if(top == MAX - 1) {
        cout << "Stack Overflow\n";
        return;
    }
    top++;
    stack[top] = x;
    cout << x << " pushed into stack\n";
}

// Function to pop an element
void pop() {
    if(top == -1) {
        cout << "Stack Underflow\n";
        return;
    }
    cout << stack[top] << " popped from stack\n";
    top--;
}

// Function to display the stack
void display() {
    if(top == -1) {
        cout << "Stack is empty\n";
        return;
    }
```

NOTE

```
    cout << "Stack elements: ";
    for(int i = top; i >= 0; i--) {
        cout << stack[i] << " ";
    }
    cout << endl;
}

int main() {
    push(10);
    push(20);
    push(30);
    display();
    pop();
    display();
    push(40);
    display();
    push(50);
    push(60);  // Overflow demonstration
    display();
    return 0;
}
```

Output:

```
10 pushed into stack
20 pushed into stack
30 pushed into stack
Stack elements: 30 20 10
30 popped from stack
Stack elements: 20 10
40 pushed into stack
Stack elements: 40 20 10
50 pushed into stack
60 pushed into stack
```

NOTE

Stack Overflow

Stack elements: 50 40 20 10

4.5.6 Stack as a Linked List

While stacks can be implemented using arrays (static implementation), using linked lists allows for a dynamic stack that can grow and shrink at runtime. Unlike arrays, linked lists do not require a fixed size and memory is allocated as needed, which avoids stack overflow caused by fixed size limitations.

4.5.6.1 Structure of a Linked List Stack

- Each element in the stack is stored in a node.
- A node contains:
    1. Data: The actual value to be stored.
    2. Pointer (Next): Points to the next node in the stack.
- The top pointer points to the first node in the linked list.

4.5.6.2 Operations

1. Push
    - Create a new node.
    - Insert the node at the beginning of the linked list.
    - Update the top pointer.
2. Pop
    - Remove the node at the beginning of the linked list.
    - Return its value.
    - Update the top pointer.
3. Peek
    - Return the data at the top node without removing it.

4.5.6.3 Advantages of Linked List Stack

- Dynamic size; no overflow until memory is exhausted.
- Efficient memory usage; no preallocation required.

C++ Example

```
#include <iostream>
using namespace std;

struct Node {
   int data;
```

NOTE

```
    Node* next;
};

Node* top = nullptr;

void push(int x) {
    Node* newNode = new Node();
    newNode->data = x;
    newNode->next = top;
    top = newNode;
    cout << x << " pushed into stack\n";
}

void pop() {
    if(top == nullptr) {
        cout << "Stack Underflow\n";
        return;
    }
    Node* temp = top;
    cout << top->data << " popped from stack\n";
    top = top->next;
    delete temp;
}

void display() {
    Node* temp = top;
    cout << "Stack elements: ";
    while(temp) {
        cout << temp->data << " ";
        temp = temp->next;
    }
    court << endl;
```

NOTE

```
}

int main() {
    push(10);
    push(20);
    push(30);
    display();
    pop();
    display();
    return 0;
}
```

4.5.7 Applications of Stacks

Stacks are widely used in computer science, programming, and real-world applications. Some key applications include:

1. Expression Evaluation
    - Converting infix expressions to postfix/prefix.
    - Evaluating arithmetic expressions.
2. Function Call Management
    - Programming languages use a call stack to manage recursive function calls.
3. Undo/Redo Operations
    - Text editors and word processors maintain a stack of operations for undo/redo functionality.
4. Browser History
    - Web browsers maintain a stack of visited pages to implement the back button.
5. Backtracking
    - Used in maze-solving, puzzles (like Sudoku), and parsing problems.

Activity 1 : Implementing Stack Using Array in C++

Objective:
After completing this activity, you will be able to:

1. Understand the concept of stack and its LIFO behavior.
2. Perform stack operations like push, pop, peek, and display.
3. Implement a stack using arrays in C++.
4. Handle stack overflow and underflow conditions effectively.
5. Apply stack concepts to solve simple problems in programming.

Materials / Tools / Software Required:

- Computer or Laptop
- C++ Compiler
- Text editor for writing C++ programs
- Pen and notebook (for manual stack simulation)

Expected Outcome:

1. Students will be able to create a stack using arrays and perform push and pop operations.
2. Overflow and underflow conditions will be correctly identified and handled.
3. Students will be able to simulate stack operations manually and understand the LIFO principle.
4. Students will gain practical experience in implementing a linear data structure and using it in problem-solving scenarios.

4.6 SUMMARY

1. A stack is a linear data structure that follows the Last In, First Out (LIFO) order. It allows insertion and deletion at one end called the top. The fundamental operations of a stack include:

- Push: Adds an element to the top of the stack.
- Pop: Removes the element from the top.
- Peek/Top: Returns the top element without removing it.
- isEmpty and isFull: Check the stack status.

2. Stacks can be implemented in two ways:

1. Array-based stacks (Static implementation): Use a fixed-size array. Push and pop are efficient (O(1)), but overflow occurs if the stack exceeds its size.

NOTE

2. Linked list stacks (Dynamic implementation): Use nodes with data and pointers. These stacks grow and shrink dynamically, avoiding overflow problems.

3. The Stack Abstract Data Type (ADT) provides a conceptual interface independent of implementation, ensuring consistency in operations. Stacks are used extensively in programming, including function call management (recursion), undo/redo operations in software, browser history, and backtracking algorithms.

4. Stacks are also critical in expression handling. Converting infix expressions to postfix or prefix and evaluating them requires a stack. Operator precedence and associativity determine the order of operations in these conversions. Multiple stacks can be implemented in a single array for efficient memory utilization, especially when handling simultaneous operations.

4.7 GLOSSARY

1. Stack – A linear data structure that follows the LIFO principle.
2. LIFO – Last In, First Out; the last element added is the first removed.
3. Push – Operation to insert an element at the top of the stack.
4. Pop – Operation to remove the top element from the stack.
5. Peek/Top – Returns the top element without removing it.
6. Overflow – Condition when pushing an element into a full stack.
7. Underflow – Condition when popping an element from an empty stack.
8. Array-based Stack – Stack implemented using a fixed-size array.
9. Linked List Stack – Stack implemented using dynamic memory allocation and nodes.
10. Postfix Expression – Arithmetic expression where operators come after operands, evaluated using a stack.

4.8 ANSWER TO CHECK YOUR PROGRESS-POSSIBLE ONE

Check Your Progress 4.1

MCQ Answer:
b) Pop – removes the top element from the stack.

NOTE

True/False Answer:
False – stacks follow Last In First Out (LIFO) principle.

Short Answers**:**

1. A stack is a linear data structure where the last inserted element is removed first (LIFO).
2. Applications include expression evaluation and function call management.

4.9 SUGGESTED READINGS – OER

Sr. No.	Title	URL	QR Code
1	Data Structure Using C++	http://books.aisectonline.com/User/UnitUrls?unitId=3712	

4.10 REFERENCE

9. Data Structures and Algorithms in C++ – Michael T. Goodrich, Roberto Tamassia, David M. Mount – ISBN: 9780470383278 – 2011
10. Data Structures and Algorithm Analysis in C++ – Mark Allen Weiss – ISBN: 9780132847377 – 2013
11. Data Structures and Algorithms in C++ – Adam Drozdek – ISBN: 9781133608426 – 2012
12. Data Structures Using C++ – D. S. Malik – ISBN: 9788131518236 – 2012
13. Data Structures Through C++ – Yashavant Kanetkar – ISBN: 9789355515834 – 2024
14. Fundamentals of Data Structures in C++ – Ellis Horowitz, Sartaj Sahni, Dinesh Mehta – ISBN: 9788173716065 – 2008
15. Data Structures: A Pseudocode Approach with C – Richard F. Gilberg, Behrouz A. Forouzan – ISBN: 9780534953965 – 2005
16. Data Structures and Program Design in C++ – Robert L. Kruse, Alexander J. Ryba – ISBN: 9780136159079 – 2010

4.11 EXERCISE

Part 1

A. Multiple choice question

NOTE

1. What is the principle followed by a stack?

a) FIFO

b) LIFO

c) Random Access

d) Priority

2. Which operation adds an element to a stack?

a) Pop

b) Push

c) Peek

d) Enqueue

3. What is the top value of an empty stack?

a) 0

b) -1

c) NULL

d) MAX

4. Which of the following is a dynamic implementation of a stack?

a) Array

b) Linked List

c) Queue

d) Heap

5. What is the postfix equivalent of the infix expression A + B * C?

a) A B + C *

b) A B C + *

c) A B C * +

d) A * B + C

Answers with Explanations:

1. Answer: B) LIFO
Explanation: A stack follows Last-In-First-Out (LIFO), meaning the last element pushed is the first to be popped.

2. Answer: B) Push
Explanation: The push operation inserts (adds) an element onto the top of the stack.

NOTE

3. Answer: B) -1
Explanation: In many implementations, -1 is used to indicate that the stack is empty and has no top element.

4. Answer: B) Linked List
Explanation: A stack can be dynamically implemented using a linked list, allowing growth without a fixed size.

5. Answer: C) A B C * +
Explanation: Postfix (Reverse Polish Notation) places operators after their operands; multiplication has higher precedence, so B * C comes before addition with A.

B. Small answer type question

1. Define a stack and explain its LIFO property.
2. Describe the push and pop operations of a stack.
3. Explain the difference between array-based and linked list stacks.
4. What is stack overflow and underflow? Give examples.
5. Define stack ADT and its significance in programming.

Part 2

C. Long answer type question (answer in 200 words)

1. Explain the static implementation of a stack using arrays with C++ code.
2. Discuss stack operations and their time complexity.
3. Explain the applications of stacks in recursion, undo/redo, and browser history.
4. Describe how operator precedence and associativity are handled during expression evaluation.
5. Explain multiple stacks implementation in a single array

Part 3

D. Long answer type question (answer in 300 words)

1. Describe stack implementation using linked lists, including push, pop, and display operations with C++ examples.
2. Explain the evaluation of postfix expressions using stacks with step-by-step examples.
3. Compare array-based and linked list-based stacks in terms of memory, efficiency, and practical applications.

NOTE

4. Discuss the role of stacks as an ADT in programming and how it abstracts implementation details.
5. Explain real-world applications of stacks, including function calls, expression conversion, backtracking, and multiple stack management.

DR. C.V. RAMAN UNIVERSITY

Chhattisgarh, Bilaspur AN AISECT GROUP UNIVERSITY

Approved by : PCI | AICTE | NCTE | BCI | Member of : AIU | Joint Committee : (UGC | DEB | AICTE) | Recognized by : UGC | A NAAC Accredited University

DR. C. V. RAMAN UNIVERSITY (C.G.)

SLM FEEDBACK FORM

PROGRAMME : _____

COURSE : _____

BLOCK : _____

UNIT : _____

Dear Student,

While studying the units of this block, you may have found certain portions of the text difficult to comprehend. We wish to know your difficulties and suggestions, in order to improve the course. Therefore, we request you to fill up and send us the following questionnaire, which pertains to this block. If you find the space provided insufficient, kindly use a separate sheet.

Please mail to aisectcontent@aisect.org/ for online submission please scan QR code:

Questionnaire

1. **Name** : _____

2. **Enrolment No.** : | | | | | | | | | | | |

3. **How many hours did you need for studying the units?** : | |

4. **Please give your reactions to the following items based on your reading of the block:**

Items	Excellent	Very Good	Good	Poor	Give specific Examples (If poor)
Presentation Quality					
Language and Style					
Illustrations					
Used (diagrams, tables, etc.)					
Conceptual Clarity					
Check Your Progress Questions					
Suggested Readings - OER					

5. **Any other comments:**

_____

_____

_____

_____

***

BLOCK II	LINKED LISTS AND HIERARCHICAL STRUCTURES	77-168
Unit 5	Recursion	79
Unit 6	Queues	101
Unit 7	Linked List	125
Unit 8	Introduction to Trees	143

This block focuses on dynamic and hierarchical data structures that extend the concepts introduced in linear structures. It begins with linked lists, emphasizing their dynamic nature and flexibility compared with arrays. Students will study singly linked lists, doubly linked lists, and circular linked lists, along with operations such as creation, traversal, insertion, deletion, and searching. The block then introduces hierarchical structures, particularly trees, which organize data through parent-child relationships. Concepts related to binary trees, tree traversal techniques, binary search trees, and their applications are explored systematically. Students will develop an understanding of how hierarchical organization can improve data representation and facilitate efficient processing. The block also highlights practical applications of linked lists and trees in computer science and programming. By studying these structures, learners develop the ability to select and implement appropriate data organization techniques for problems involving dynamic storage and hierarchical relationships.

UNIT 5
RECURSION

NOTE

UNIT STRUCTURE

5.1 Introduction

5.2 Objective

5.3 Introduction To Recursion

5.4 Working of Recursion

5.5 Fibonacci Series

5.6 Tower of Hanoi

5.7 Efficiency of Recursion

5.8 Summary

5.9 Glossary

5.10 Answer to check your progress

5.11 Suggested readings - OER

5.12 Reference

5.13 Exercise

5.1 INTRODUCTION

Recursion is a fundamental concept in computer programming where a function calls itself directly or indirectly to solve a problem. It is widely used in algorithms that can be broken down into smaller, similar subproblems, such as factorial calculation, Fibonacci series, and the Tower of Hanoi problem. The core idea behind recursion is divide and conquer: complex problems are divided into smaller, simpler subproblems that are easier to solve, and their solutions are combined to solve the original problem.

Every recursive function requires two main components: a base case and a recursive case. The base case acts as a stopping condition to prevent infinite calls, while the recursive case breaks the problem into smaller instances. Recursion works with the help of a call stack, which keeps track of active function calls. As the function calls itself, new frames are pushed onto the stack, and once the base case is reached, the stack begins to unwind, combining the results of each call to form the final solution.

NOTE

While recursion offers simplicity and readability, it may have efficiency drawbacks such as high memory consumption and slower execution if not implemented carefully. Optimization techniques such as tail recursion, memoization, and iterative alternatives can significantly improve its performance. Understanding recursion is crucial for mastering advanced topics in data structures and algorithms.

5.2 OBJECTIVE

After completing this unit, you will be able to:

1. Explain the concept of recursion and its working with examples.
2. Identify base cases and recursive cases in recursive functions.
3. Write recursive programs for problems such as factorial, Fibonacci series, and Tower of Hanoi.
4. Analyze the efficiency of recursive functions in terms of time and space complexity.
5. Apply optimization techniques like tail recursion and memoization to improve recursive program performance.

5.3 INTRODUCTION TO RECURSION

Recursion is a fundamental concept in computer science and programming where a function calls itself directly or indirectly to solve a problem. It is widely used in algorithms that can be broken down into smaller, similar subproblems, such as in sorting, searching, and mathematical computations like factorials, Fibonacci series, and Tower of Hanoi.

At its core, recursion provides a way to think about problems in terms of “divide and conquer”. Instead of writing complex iterative loops, recursion allows programmers to write cleaner, more readable code by letting the function solve smaller instances of the same problem until a base case is reached.

5.3.1 Characteristics of Recursion

Every recursive function must have two key components:

1. Base Case
   The base case acts as a stopping criterion. Without a base case, the recursion would continue indefinitely, causing a stack overflow. The base case is the simplest scenario, which can be solved directly without further recursion.

   Example: In calculating factorial of a number, 0! or 1! is the base case because the factorial of 0 or 1 is 1.

2. Recursive Case
   The recursive case breaks the problem into smaller instances and calls the same function to solve them. This step continues until the base case is reached.

   Example: Factorial of n can be expressed as n * factorial(n-1).

5.3.2 Working of Recursion

Recursion works using a call stack, which is a special data structure that keeps track of active functions. Each time a recursive function is called, its state is stored on the stack, including the parameters and local variables. Once the base case is reached, the stack begins to unwind, solving each function call in reverse order.

Example: Calculating factorial(3):

1. factorial(3) calls factorial(2)
2. factorial(2) calls factorial(1)
3. factorial(1) is the base case → returns 1
4. factorial(2) returns 2 * 1 = 2
5. factorial(3) returns 3 * 2 = 6

This demonstrates how recursion builds a solution gradually by combining results from smaller subproblems.

5.3.3 Advantages of Recursion

1. Simplifies code: Problems that are inherently repetitive or hierarchical, like tree traversal or combinatorial problems, are easier to solve with recursion than iterative loops.
2. Readable and maintainable: Recursive solutions are often more elegant and closer to the mathematical representation of the problem.
3. Reduces code duplication: Many iterative solutions require auxiliary data structures, which recursion avoids in some cases.

5.3.4 Disadvantages of Recursion

1. Memory consumption: Each recursive call uses stack space, which can lead to a stack overflow if recursion is too deep.
2. Slower execution: Function calls have overhead, making recursion generally slower than equivalent iterative solutions.
3. Complex debugging: Tracing errors in recursive calls can be more difficult, especially in mutually recursive functions.

5.3.5 Types of Recursion

NOTE

1. Direct Recursion: A function calls itself directly.
   Example: Factorial, Fibonacci series.

```
2. int factorial(int n) {
3.     if(n == 0 || n == 1)
4.         return 1;  // Base case
5.     else
6.         return n * factorial(n - 1);  // Recursive case
7. }
```

8. Indirect Recursion: A function calls another function, which eventually calls the first function.
   Example:

```
9. void functionA() { functionB(); }
10. void functionB() { functionA(); }
```

11. Tail Recursion: Recursive call is the last operation in the function. Tail recursion can be optimized by compilers to improve performance.

```
12. int factorial(int n, int result = 1) {
13.     if(n == 0) return result;
14.     return factorial(n - 1, n * result);
15. }
```

16. Non-tail Recursion: Recursive call is followed by other operations.

5.3.6 Common Applications of Recursion

1. Mathematical Problems
   - Factorial of a number
   - Fibonacci series
   - Greatest Common Divisor (GCD)
2. Data Structures
   - Traversal of trees (preorder, inorder, postorder)
   - Traversal of graphs using depth-first search (DFS)
3. Algorithms
   - Divide and conquer algorithms like merge sort, quick sort
   - Backtracking problems like N-Queens, maze solving

5.3.7 Example: Fibonacci Series

The Fibonacci sequence is defined as:

F(n)=F(n−1)+F(n−2),F(0)=0,F(1)=1F(n) = F(n-1) + F(n-2), \quad F(0) = 0, F(1) = 1F(n)=F(n−1)+F(n−2),F(0)=0,F(1)=1

C++ Implementation:

```
#include <iostream>
using namespace std;

int fibonacci(int n) {
    if(n == 0) return 0;  // Base case
    if(n == 1) return 1;  // Base case
    return fibonacci(n - 1) + fibonacci(n - 2);  // Recursive case
}

int main() {
    int n;
    court << "Enter number of terms: ";
    cin >> n;
    for(int i = 0; i < n; i++)
        court << fibonacci(i) << " ";
    return 0;
}
```

Output (for n = 5): 0 1 1 2 3

This example clearly illustrates how recursion solves a problem by breaking it into smaller subproblems and combining their solutions.

5.4 WORKING OF RECURSION

Recursion is a programming technique in which a function calls itself to solve a problem. Understanding how recursion works is crucial for implementing recursive algorithms efficiently. The working of recursion can be understood in terms of function calls, the call stack, base cases, and recursive cases.

5.4.1 Key Concepts in Recursion

1. Base Case
   The base case is the condition that stops recursion. Every

NOTE

recursive function must have a base case to prevent infinite recursion, which can lead to a stack overflow. The base case handles the simplest form of the problem directly.

Example: In calculating factorial, the base case is when n = 0 or n = 1.

2. Recursive Case
   The recursive case is where the function calls itself with a smaller or simpler version of the original problem. This process continues until the base case is reached.

   Example: Factorial of n can be expressed as n * factorial(n-1).

3. Call Stack
   Every function call in C++ is stored in a call stack, a special memory area that keeps track of active function calls. When a recursive function is called:

   - The current state of the function (parameters and local variables) is pushed onto the stack.
   - Execution moves to the next recursive call.
   - Once the base case is reached, the stack unwinds, and results are returned in reverse order of calls.

Understanding the call stack is crucial to visualize how recursion actually solves problems step by step.

5.4.2 Step-by-Step Working of Recursion

Let's take an example to calculate factorial of 4 using recursion.

```
#include <iostream>

using namespace std;


int factorial(int n) {
   if(n == 0 || n == 1)  // Base case
      return 1;
   else
      return n * factorial(n - 1);  // Recursive case
}


int main() {
   int num = 4;
   court << "Factorial of " << num << " is " << factorial(num);
```

```
    return 0;
}
```

Step-by-Step Execution:

1. Call 1: factorial(4) → not base case → calls factorial(3)
2. Call 2: factorial(3) → not base case → calls factorial(2)
3. Call 3: factorial(2) → not base case → calls factorial(1)
4. Call 4: factorial(1) → base case → returns 1

Now the stack unwinds:

- factorial(2) returns 2 * 1 = 2
- factorial(3) returns 3 * 2 = 6
- factorial(4) returns 4 * 6 = 24

Output: Factorial of 4 is 24

This example illustrates how recursion solves smaller subproblems first and combines their results to solve the original problem.

5.4.3 Call Stack Visualization

Function Call	Operation	Stack State
factorial(4)	Calls factorial(3)	[factorial(4)]
factorial(3)	Calls factorial(2)	[factorial(4), factorial(3)]
factorial(2)	Calls factorial(1)	[factorial(4), factorial(3), factorial(2)]
factorial(1)	Base case → returns 1	[factorial(4), factorial(3), factorial(2)]
factorial(2)	Returns 2 * 1 = 2	[factorial(4), factorial(3)]
factorial(3)	Returns 3 * 2 = 6	[factorial(4)]
factorial(4)	Returns 4 * 6 = 24	[] (stack empty)

Table 5.4.1 Call stack visualization

This table shows how each function call waits for its subproblem to be solved before continuing, which is a fundamental characteristic of recursion.

5.4.4 Types of Recursive Function Execution

1. Head Recursion
   The recursive call is the first statement in the function.
   Example: Factorial as shown above.

NOTE

2. Tail Recursion
   The recursive call is the last statement in the function. Tail recursion can be optimized by compilers to reduce stack usage.
3. int factorial(int n, int result = 1) {
4. if(n == 0) return result;
5. return factorial(n - 1, n * result);
6. }
7. Nested Recursion
   A recursive function calls itself multiple times in a nested manner, such as the Fibonacci series.
8. int fibonacci(int n) {
9. if(n == 0) return 0;
10. if(n == 1) return 1;
11. return fibonacci(n - 1) + fibonacci(n - 2);
12. }

5.4.5 Advantages of Understanding Recursion Working

- Helps debug recursive functions effectively.
- Allows understanding of memory usage and potential stack overflow issues.
- Forms a foundation for advanced algorithms such as backtracking, tree traversal, and divide-and-conquer techniques.

5.4.6 Tips for Using Recursion

1. Always define a clear base case.
2. Ensure the problem size reduces with each recursive call.
3. Avoid deep recursion to prevent stack overflow.

Consider tail recursion or iterative solutions for large datasets

5.5 FIBONACCI SERIES

The Fibonacci Series is one of the classic examples of recursion. It is a sequence of numbers in which each number is the sum of the two preceding ones. The sequence starts as:

0,1,1,2,3,5,8,13,21,34,...0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \dots0,1,1,2,3,5,8,13,21,34,...

Mathematically, the Fibonacci numbers are defined as:

NOTE

F(n)={0if n=01if n=1F(n−1)+F(n−2)if n>1F(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F(n-1) + F(n-2) & \text{if } n > 1 \end{cases}F(n)= ⎩⎨⎧ 01F(n−1)+F(n−2)if n=0if n=1if n>1

5.5.1Working of Recursion in Fibonacci Series

1. Base Cases:
    - F(0) = 0
    - F(1) = 1
2. Recursive Case:
    - F(n) = F(n-1) + F(n-2)

When a function calculates F(n), it calls itself twice to calculate F(n-1) and F(n-2) until it reaches the base cases.

5.5.2 C++ Program: Recursive Fibonacci Series

```
#include <iostream>
using namespace std;

int fibonacci(int n) {
   if(n == 0) return 0;       // Base case
   if(n == 1) return 1;       // Base case
   return fibonacci(n - 1) + fibonacci(n - 2); // Recursive case
}

int main() {
   int terms;
   court << "Enter number of terms: ";
   cin >> terms;

   court << "Fibonacci Series: ";
   for(int i = 0; i < terms; i++)
      court << fibonacci(i) << " ";

   return 0;
}
```

NOTE

Sample Output (for n = 7): 0 1 1 2 3 5 8

5.5.3 Call Stack Illustration

To calculate F(4):

1. F(4) = F(3) + F(2)
2. F(3) = F(2) + F(1)
3. F(2) = F(1) + F(0)
4. Base cases return 1 and 0

Stack unwinds and sums the results:

- F(2) = 1 + 0 = 1
- F(3) = 1 + 1 = 2
- F(4) = 2 + 1 = 3

5.6 TOWER OF HANOI

The Tower of Hanoi is a classical puzzle that demonstrates recursion very clearly. It consists of three rods and a number of disks of different sizes. The objective is to move all disks from the source rod to the destination rod following these rules:

1. Only one disk can be moved at a time.
2. Each move consists of taking the top disk from one rod and placing it on another rod.
3. A larger disk cannot be placed on top of a smaller disk.

5.6.1Tower of Hanoi Puzzle Cycle

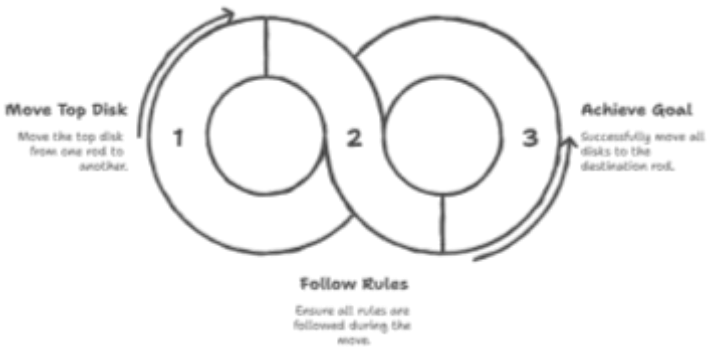


Figure 5.1: Tower of Hanoi Puzzle Cycle

5.6.2 Recursive Solution

Let's denote the rods as Source (S), Auxiliary (A), and Destination (D).

Steps to solve n disks recursively:

1. Move n-1 disks from S to A using D as auxiliary.

2. Move the nth disk (largest) from S to D.
3. Move the n-1 disks from **A** to **D** using **S** as auxiliary.

Mathematical Formula

The minimum number of moves required to solve the Tower of Hanoi with n disks:

T(n)=2n−1T(n) = 2^n - 1T(n)=2n−1

C++ Program: Tower of Hanoi

```
#include <iostream>
using namespace std;


void towerOfHanoi(int n, char source, char auxiliary, char destination)
{
   if(n == 1) { // Base case
      court << "Move disk 1 from " << source << " to " << destination << endl;
      return;
   }
   towerOfHanoi(n - 1, source, destination, auxiliary); // Step 1
   court << "Move disk " << n << " from " << source << " to " << destination << endl; // Step 2
   towerOfHanoi(n - 1, auxiliary, source, destination); // Step 3
}


int main() {
   int n;
   court << "Enter number of disks: ";
   cin >> n;
   towerOfHanoi(n, 'S', 'A', 'D');
   return 0;
}
```

Sample Output (for 3 disks):

Move disk 1 from S to D

Move disk 2 from S to A

Move disk 1 from D to A

NOTE

Move disk 3 from S to D

Move disk 1 from A to S

Move disk 2 from A to D

Move disk 1 from S to D

Explanation of Recursive Working

For 3 disks:

1. Move top 2 disks from S → A
2. Move disk 3 from S → D
3. Move 2 disks from A → D

Each recursive call breaks the problem into smaller subproblems until 1 disk, the base case, is reached. Then the solution builds up as the stack unwinds.

Applications and Importance

- Demonstrates problem-solving using recursion.
- Helps in understanding divide and conquer strategy.
- Forms the basis of learning backtracking algorithms.
- Important for understanding stack memory usage.

Check Your Progress 5.1

MCQ:
What is the base case in a recursive function?
a) The condition that stops recursion
b) The function call itself
c) A loop inside recursion
d) Memory allocation for recursion

True/False:
Every recursive function must have a base case to avoid infinite recursion.

Short Questions:

1. Explain recursion in simple words.

.....

.....

2. Give an example of a problem that can be solved using recursion.

.....

.....

5.7 EFFICIENCY OF RECURSION

Recursion is a powerful programming technique that provides elegant solutions to many problems, especially those that are naturally hierarchical or repetitive. However, efficiency is an important consideration when using recursion, because recursive functions can be memory-intensive and slow if not implemented properly. Understanding the efficiency of recursion involves analyzing time complexity, space complexity, and optimization strategies.

5.7.1. Factors Affecting Efficiency

a) Number of Function Calls

Every recursive call adds a new frame to the call stack, which stores the function's parameters and local variables.

- For problems with single recursion (e.g., factorial), the number of calls is proportional to n.
- For multiple recursion (e.g., Fibonacci series), the number of calls grows exponentially, causing redundant calculations.

Example: Fibonacci(n) using naive recursion:

Fibonacci(5) calls Fibonacci(4) + Fibonacci(3)

Fibonacci(4) calls Fibonacci(3) + Fibonacci(2)

Fibonacci(3) calls Fibonacci(2) + Fibonacci(1)

Here, Fibonacci(3) and Fibonacci(2) are calculated multiple times, resulting in repeated work.

b) Stack Memory Usage

Each recursive call consumes stack memory. Deep recursion may cause stack overflow if the base case is not reached quickly or the recursion is too deep.

- Single recursion has linear stack usage, proportional to n.
- Nested/multiple recursion has exponential stack usage, which is highly inefficient.

c) Base Case and Termination

Efficiency depends on properly defining the base case.

- If the base case is reached late or not correctly defined, the function may run unnecessarily long or never terminate, wasting both time and memory.

5.7.2. Time Complexity Analysis

1. Linear Recursion: Each call reduces the problem size by 1.

NOTE

Example: Factorial(n)

T(n)=T(n−1)+O(1)T(n) = T(n-1) + O(1)T(n)=T(n−1)+O(1)

- Time complexity: O(n)
- Space complexity (stack): O(n)

2. Multiple Recursion: Each call generates multiple sub-calls.

Example: Fibonacci(n)

T(n)=T(n−1)+T(n−2)+O(1)T(n) = T(n-1) + T(n-2) + O(1)T(n)=T(n−1)+T(n−2)+O(1)

- Time complexity: O(2^n) (exponential)
- Space complexity (stack): O(n)

This shows that naive recursive Fibonacci is highly inefficient for large n.

5.7.3. Optimizing Recursive Functions

a) Tail Recursion

Tail recursion occurs when the recursive call is the last operation in the function. Compilers can optimize tail recursion to reuse the same stack frame, reducing stack memory usage.

Example: Factorial (Tail Recursion)

```
int factorial(int n, int result = 1) {
   if(n == 0) return result;
   return factorial(n - 1, n * result); // Tail recursion
}
```

- Time complexity: O(n)
- Space complexity: O(1) if compiler optimizes tail calls

b) Memoization / Dynamic Programming

Memoization stores the results of previous calls to avoid redundant calculations, improving efficiency.

Example: Optimized Fibonacci using Memoization

```
#include <iostream>
using namespace std;

int fibonacci(int n, int memo[]) {
   if(n == 0) return 0;
   if(n == 1) return 1;
```

```
    if(memo[n] != -1) return memo[n]; // Use stored result
    memo[n] = fibonacci(n-1, memo) + fibonacci(n-2, memo);
    return memo[n];
}

int main() {
    int n;
    court << "Enter n: ";
    cin >> n;
    int memo[n+1];
    for(int i=0; i<=n; i++) memo[i] = -1;
    court << "Fibonacci(" << n << ") = " << fibonacci(n, memo);
    return 0;
}
```

- Time complexity: O(n)
- Space complexity: O(n)

This reduces the exponential time complexity of naive recursion to linear time, making it efficient for large inputs.

c) Iterative Approach

Sometimes recursion can be replaced by iteration to improve efficiency.

Example: Iterative Fibonacci

```
int fibonacci(int n) {
    int a = 0, b = 1, c;
    if(n == 0) return 0;
    for(int i=2; i<=n; i++) {
        c = a + b;
        a = b;
        b = c;
    }
    return b;
}
```

- Time complexity: O(n)
- Space complexity: O(1)

NOTE

- Efficient for large n.

further. Recursion works with the help of a call stack, where each function call is stored until it is completed.

Examples such as the factorial of a number, Fibonacci series, and Tower of Hanoi illustrate different types of recursion. Factorial uses single recursion, while Fibonacci demonstrates multiple recursion, which can be inefficient due to repeated calculations. Tower of Hanoi showcases a real-world recursive problem, emphasizing divide and conquer strategies.

Efficiency of recursion depends on the number of function calls, stack memory usage, and proper base case definition. Problems with multiple recursive calls can be slow and memory-intensive. Optimizations such as tail recursion, which allows compilers to reuse stack frames, and memoization, which stores previously computed results, can significantly reduce time and space complexity. Iterative alternatives are also recommended when recursion depth is high.

Understanding recursion is vital for mastering data structures, algorithms, and problem-solving techniques. It provides a clear and elegant approach to breaking down complex problems into manageable parts. By mastering recursion, students can implement efficient solutions for mathematical computations, sorting and searching algorithms, tree and graph traversals, and combinatorial problems

Activity 1: The "Fractal Architect" – Visualizing Recursive Patterns

Objective: To visually demonstrate the concept of recursion—where a function calls itself to solve smaller instances of the same problem—by generating a geometric fractal (specifically the Sierpinski Triangle or a Recursive Tree). This helps students understand base cases and self-similarity.

Materials / Tools / Software Required:

- Computer: Capable of running graphical scripts.
- Programming Language: Python (recommended for its simple turtle graphics library) or JavaScript (HTML5 Canvas).
- Library: Python turtle module or Processing.
- Whiteboard/Paper: For tracing the "call stack" manually before coding.

Expected Outcome: By the end of this activity, students will be able to:

NOTE

5. Identify the Base Case: Determine the stopping condition (e.g., when the length of the line is too small to draw).
6. Identify the Recursive Step: Write the logic where the drawing function calls itself with modified parameters (e.g., shorter length, different angle).
7. Visualize the Stack: Observe how the drawing "pauses" at a branching point, goes deep into one branch, and then "returns" to finish the other branch (backtracking).
8. Produce a complex geometric image using a surprisingly small amount of code (typically under 20 lines)

5.8 SUMMARY

Recursion is a fundamental programming technique in which a function calls itself to solve a problem by breaking it down into smaller, similar subproblems. This unit explains recursion through the divide and conquer approach, where complex problems are simplified into manageable parts until a base case is reached. Every recursive function consists of two essential components: a base case, which stops further recursive calls, and a recursive case, which reduces the problem size and calls the function again. The execution of recursion relies on the call stack, where each function call is stored until it completes, and results are returned as the stack unwinds. The unit discusses the working of recursion in detail using examples such as factorial calculation, Fibonacci series, and the Tower of Hanoi problem. These examples illustrate different forms of recursion, including direct, multiple, tail, and nested recursion. While factorial demonstrates linear recursion, Fibonacci highlights the inefficiency of multiple recursion due to repeated calculations. Tower of Hanoi serves as a classic example of recursion applied to problem-solving using clear logical steps. The advantages of recursion include simpler, more readable code and a closer alignment with mathematical definitions, especially for hierarchical problems like tree traversal and divide-and-conquer algorithms. However, recursion also has drawbacks such as higher memory consumption, slower execution, and the risk of stack overflow. To address these issues, the unit introduces optimization techniques like tail recursion, memoization, and iterative approaches. Overall, this unit emphasizes the importance of understanding recursion for efficient algorithm design. Mastery of recursion builds a strong foundation for advanced topics in data structures, algorithms, backtracking, and problem-solving strategies in computer science.

5.9 GLOSSARY

1. Recursion – A programming technique where a function calls itself to solve a problem.

NOTE

2. Base Case – The condition in a recursive function that stops further recursion.
3. Recursive Case – Part of the function where it calls itself with a smaller problem.
4. Call Stack – Memory structure that stores active function calls in recursion.
5. Stack Overflow – An error caused by excessive recursive calls exceeding stack memory.
6. Tail Recursion – A recursion where the recursive call is the last operation in the function.
7. Memoization – An optimization technique that stores previously computed results to avoid redundancy.
8. Factorial – Product of all positive integers up to a number (n!).
9. Fibonacci Series – A sequence where each number is the sum of the previous two.
10. Tower of Hanoi – A classic recursive puzzle involving moving disks between rods.

5.10 ANSWER TO CHECK YOUR PROGRESS-POSSIBLE ONE

Check Your Progress 5.1

MCQ Answer:
a) The condition that stops recursion.

True/False Answer:
True – without a base case, recursion can run infinitely.

Short Answers**:**

1. Recursion is when a function calls itself to solve smaller subproblems.
2. Examples include factorial calculation, Fibonacci series, or Tower of Hanoi.

5.11 SUGGESTED READINGS – OER

Sr. No.	Title	URL	QR Code
1	Data Structure Using C++	http://books.aisectonline.com/User/UnitUrls?unitId=3713	

5.12 REFERENCE

- Data Structures and Algorithms in C++ – Michael T. Goodrich, Roberto Tamassia, David M. Mount – ISBN: 9780470383278 – 2011
- Data Structures and Algorithm Analysis in C++ – Mark Allen Weiss – ISBN: 9780132847377 – 2013
- Data Structures and Algorithms in C++ – Adam Drozdek – ISBN: 9781133608426 – 2012
- Data Structures Using C++ – D. S. Malik – ISBN: 9788131518236 – 2012
- Data Structures Through C++ – Yashavant Kanetkar – ISBN: 9789355515834 – 2024
- Fundamentals of Data Structures in C++ – Ellis Horowitz, Sartaj Sahni, Dinesh Mehta – ISBN: 9788173716065 – 2008
- Data Structures: A Pseudocode Approach with C – Richard F. Gilberg, Behrouz A. Forouzan – ISBN: 9780534953965 – 2005
- Data Structures and Program Design in C++ – Robert L. Kruse, Alexander J. Ryba – ISBN: 9780136159079 – 2010

5.13 EXERCISE

Part 1

A. Multiple choice question

1. Which of the following is essential for every recursive function?
A) Loop
B) Base case
C) Global variable
D) Array

2. What is the output of factorial (0)?
A) 0
B) 1
C) Undefined
D) -1

3. Which of these problems is an example of multiple recursion?
A) Factorial
B) Fibonacci series
C) Linear search
D) Tower of Hanoi

4. Tail recursion helps in:
A) Reducing memory usage
B) Increasing function calls

NOTE

C) Slowing execution
D) Removing base cases

5. Minimum number of moves required to solve Tower of Hanoi with 3 disks is:
A) 3
B) 5
C) 7
D) 9

Answers with Explanations:

1. Answer: B) Base case
Explanation: A base case stops the recursion; without it, the function would call itself indefinitely, causing a stack overflow.

2. Answer: B) 1
Explanation: By definition, the factorial of 0 is 1.

3. Answer: B) Fibonacci series
Explanation: Fibonacci recursion calls itself twice for each non-base case (fib (n-1) and fib(n-2)), making it a multiple recursion.

4. Answer: A) Reducing memory usage
Explanation: Tail-recursive functions can be optimized by compilers to reuse stack frames, reducing memory usage compared to regular recursion.

5. Answer: C) 7
Explanation: The formula for minimum moves is 2n−12^n - 12n−1. For 3 disks: 23−1=72^3 - 1 = 723−1=7.

B. Small answer type question

1. Explain what recursion is and give an example.
2. What is a base case in recursion and why is it important?
3. Describe the working of a recursive function using factorial as an example.
4. Write a recursive function to generate the Fibonacci series.
5. Explain what a call stack is and its role in recursion.

Part 2

C. Long answer type question (answer in 200 words)

1. Explain the working of recursion with a detailed example of factorial calculation.

2. Describe the Tower of Hanoi problem and explain its recursive solution.
3. Discuss the advantages and disadvantages of using recursion in programming.
4. Write a C++ program for recursive Fibonacci series and explain its working.
5. Explain tail recursion and how it improves the efficiency of recursive functions.

Part 3

D. Long answer type question (answer in 300 words)

1. Analyze the efficiency of recursion in terms of time and space complexity, with examples like factorial and Fibonacci series.
2. Explain multiple recursion with a detailed example, and suggest methods to optimize its efficiency.
3. Compare recursion and iteration for solving problems, highlighting their advantages and disadvantages.
4. Describe memoization in recursion and explain how it optimizes the computation of Fibonacci numbers.
5. Discuss the practical applications of recursion in data structures and algorithms, including tree and graph traversal, sorting algorithms, and combinatorial problems.

UNIT 6
QUEUES

NOTE

UNIT STRUCTURE

6.1 Introduction

6.2 Objective

6.3 Introduction to Queues

6.4 Different Types of Queues

6.5 Queue as an Abstract Data Structure (ADT)

6.6 Queue Implementation Using Linked List

6.7 Applications of Queues

6.8 Summary

6.9 Glossary

6.10 Answer to check your progress

6.11 Suggested readings - OER

6.12Reference

6.13 exercise

6.1 INTRODUCTION

A queue is a fundamental linear data structure in computer science that follows the First-In-First-Out (FIFO) principle. This means that the element inserted first will be the first one removed. Queues are widely used in both real-world scenarios and computer applications to maintain order and manage resources efficiently. Examples of real-life queues include people standing in line at banks, ticket counters, or supermarkets. In computer systems, queues manage tasks such as CPU scheduling, printer spooling, and network packet buffering.

There are several types of queues, each designed for specific needs: linear queue, circular queue, double-ended queue (deque), and priority queue. While a linear queue is the simplest form with insertion at the rear and deletion from the front, a circular queue efficiently utilizes memory by reusing empty spaces. A deque allows insertion and deletion from both ends, providing flexibility, and a priority queue processes elements based on their importance rather than the order of arrival.

Queues can be implemented using arrays or linked lists, with linked lists providing dynamic memory allocation and constant-time

NOTE

operations. The concept of queue as an Abstract Data Type (ADT) emphasizes the interface and behavior, abstracting implementation details. Understanding queues is essential for algorithm design, resource management, and real-world system modeling.

6.2 OBJECTIVE

After completing this unit, you will be able to:

1. Understand the concept of a queue and its FIFO behavior.
2. Explain the different types of queues: linear, circular, deque, and priority queues.
3. Implement queues using arrays and linked lists efficiently.
4. Apply queues in real-world and computer science applications, such as CPU scheduling and networking.
5. Analyze the advantages, limitations, and appropriate use cases of each queue type.

6.3 INTRODUCTION TO QUEUES

6.3.1 Introduction

In computer science, a queue is a linear data structure that follows the First In, First out (FIFO) principle. This means that the element added first to the queue will be the first one to be removed. The queue is analogous to a real-world queue, such as a line at a ticket counter or a bank, where the first person to arrive is the first to be served.

A queue supports two main operations:

1. Enqueue: Adding an element to the rear (or end) of the queue.
2. Dequeue: Removing an element from the front (or start) of the queue.
3. **Example:**

Imagine people standing in a line for a movie ticket:

- Person A arrives first, Person B arrives second, and Person C arrives third.
- The ticket counter serves the person at the front of the line first.
- After serving Person A, the next in line, Person B, is served.

This is exactly how a queue works in data structures.

NOTE

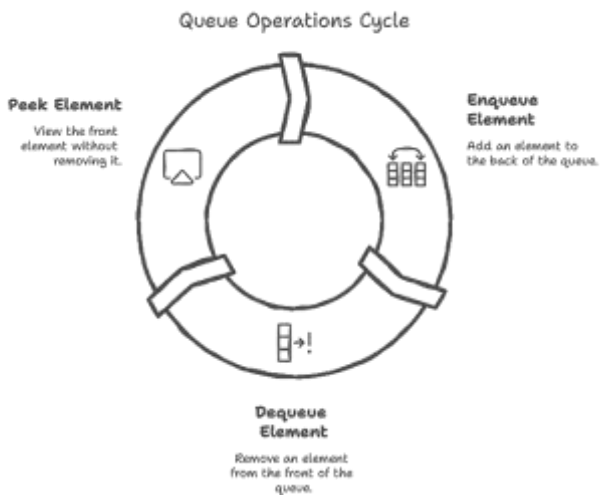


Fig 6.1: Queue Operation Cycle

6.3.2. Characteristics of Queues

A queue has certain important characteristics that differentiate it from other data structures like stacks or arrays:

1. Order of Operations: Queues follow FIFO, unlike stacks that follow LIFO (Last In, First Out).
2. Two Ends: There are two ends in a queue:
    - Front: The end from which elements are removed.
    - Rear (or Back): The end where elements are inserted.
3. Dynamic Size: Depending on implementation, the queue can be fixed-sized (like an array-based queue) or dynamically growable (like a linked-list-based queue).
4. Basic Operations: Besides enqueue and dequeue, other common operations include:
    - Peek/Front: Returns the front element without removing it.
    - isEmpty(): Checks whether the queue is empty.
    - isFull(): Checks whether the queue is full (in case of a fixed-size queue).

6.3.3. Types of Queues

Queues are not limited to just the simple FIFO structure. Several variations are used depending on the application:

a) Simple Queue (Linear Queue)

- Elements are added at the rear and removed from the front.

NOTE

- Limitation: If implemented using arrays, it may lead to wasted space when elements are dequeued, as the front moves forward.

b) Circular Queue

- To overcome the limitation of wasted space in a linear queue, a circular queue wraps around when it reaches the end of the array.
- Both front and rear indices circulate within the fixed array size.
- Efficient memory utilization compared to a simple linear queue.

c) Priority Queue

- Each element is associated with a priority.
- Elements with higher priority are dequeued before elements with lower priority, regardless of their order of insertion.
- Useful in real-time scheduling, CPU task management, and operating systems.

d) Double-Ended Queue (Deque)

- Allows insertion and deletion at both ends, i.e., front and rear.
- Can function both as a queue and a stack, providing great flexibility.

6.3.4. Implementation of Queues

Queues can be implemented using arrays, linked lists, or other dynamic data structures.

a) Array Implementation

- Enqueue: Add element at rear index.
- Dequeue: Remove element from front index.
- Limitation: Linear array queues waste space after repeated dequeues.

b) Linked List Implementation

- A queue can be implemented using a linked list where:
  - Front pointer: Points to the first node.
  - Rear pointer: Points to the last node.
- Advantage: Dynamic size, no need to worry about memory limitations.

6.3.5. Applications of Queues

Queues are widely used in various real-world scenarios and computing systems:

NOTE

1. Operating Systems: Task scheduling, CPU job scheduling, printer spooling.
2. Networking: Handling requests in routers or switches, data packet buffering.
3. Simulation: Customer service systems like banks, airports, and supermarkets.
4. Breadth-First Search (BFS): Queues are used in graph traversal algorithms.
5. Asynchronous Data Transfer: Queues are used in keyboard buffering, I/O buffers, and asynchronous message handling.

6.3.6. Advantages and Disadvantages

6.3.6.1 Advantages

- Simple and easy to understand.
- Efficient in managing resources in a FIFO manner.
- Flexible implementation (array or linked list).

6.3.6.2 Disadvantages

- Fixed-size array queues may lead to wasted memory or overflow.
- Linear queues need shifting of elements in some implementations, causing inefficiency.

6.4 DIFFERENT TYPES OF QUEUES

A queue is a fundamental data structure in computer science that stores data in a First-In-First-Out (FIFO) order. Based on specific requirements and applications, queues can be categorized into several types: Linear Queue, Circular Queue, Double-Ended Queue (Deque), and Priority Queue. Each type has unique characteristics, advantages, and use cases.

6.4.1. Linear Queue

6.4.1.1 Definition

A Linear Queue is the simplest form of a queue. It is a sequential collection of elements in which insertion occurs at the rear and deletion occurs at the front. The first element added is the first one removed, strictly following the FIFO principle.

6.4.1.2 Structure

Front --> [A | B | C | D] <-- Rear

NOTE

6.4.1.3 Operations

1. Enqueue: Insert an element at the rear.
2. Dequeue: Remove an element from the front.
3. isEmpty(): Checks if the queue is empty.
4. isFull(): Checks if the queue is full (in case of fixed size).
5. Peek/Front(): Returns the front element without removing it.

6.4.1.4 Advantages

- Simple to implement.
- Useful in basic queue operations where memory efficiency is not a concern.

6.4.1.5 Limitations

- Wasted Space: Once elements are dequeued from the front, their memory cannot be reused in a fixed-size array.
- Overflow Risk: If the queue reaches maximum size, no further insertions are possible even if space is available due to dequeued elements.

6.4.1.6 Example

Suppose a bank has a single counter:

- Customers arrive in order: John, Alice, Bob.
- John is served first, then Alice, and then Bob.

This simulates a linear queue where the front moves forward as elements are removed.

6.4.2. Circular Queue

6.4.2.1Definition

A Circular Queue is an advanced form of a linear queue designed to overcome the wasted space problem. In a circular queue, the rear wraps around to the front when it reaches the end of the array, forming a circle.

6.4.2.2 Structure

[D | E | F | G]

Rear -> G

Front -> D

- After removing D and E, new elements can be inserted at the beginning of the array without shifting elements.

NOTE

6.4.2.3 Operations

1. Enqueue: Insert an element at the rear. If the rear reaches the last position, it wraps to the first free position.
2. Dequeue: Remove an element from the front.
3. isEmpty(): True if no elements exist.
4. isFull(): True if all positions are filled.

6.4.2.4 Advantages

- Efficient use of memory; no space is wasted.
- Solves the overflow problem of linear queues in array implementation.

6.4.2.5 Applications

- CPU task scheduling.
- Network packet buffers.
- Handling circular lists like a playlist or round-robin scheduling.

6.4.3. Double-Ended Queue (Deque)

6.4.3.1 Definition

A Double-Ended Queue (Deque) is a queue where insertion and deletion can occur at both ends, i.e., front and rear. It combines the functionality of queue and stack.

6.4.3.2 Structure

Front <--> [X | Y | Z] <--> Rear

6.4.3.3 Types of Deque

1. Input-restricted deque: Insertion is allowed at one end only, but deletion can happen from both ends.
2. Output-restricted deque: Deletion is allowed at one end only, but insertion can happen at both ends.

6.4.3.4 Operations

- InsertFront (): Insert an element at the front.
- InsertRear (): Insert an element at the rear.
- DeleteFront (): Delete an element from the front.
- DeleteRear (): Delete an element from the rear.
- IsEmpty (): Checks if deque is empty.

6.4.3.5 Advantages

- Provides flexibility to insert and delete from both ends.

NOTE

- Can act as a stack or queue depending on the requirement.

6.4.3.6 Applications

- Palindrome checking in strings.
- Undo/Redo operations in text editors.
- Sliding window problems in algorithms.

6.4.4. Priority Queue

6.4.4.1 Definition

A Priority Queue is a queue in which each element is associated with a priority level. The element with the highest priority is dequeued before elements with lower priorities, regardless of their order of insertion.

6.4.4.2 Structure

[Element | Priority]

[A | 2]

[B | 5]

[C | 1]

- Element B (priority 5) is removed first, even if it was inserted after A and C.

6.4.4.3 Operations

1. Insert(element, priority): Add element with a priority.
2. Delete(): Remove element with the highest priority.
3. Peek(): Return element with highest priority without removing it.

6.4.4.4 Implementation

- Array or Linked List: Simple implementation by sorting elements based on priority.
- Heap (Min-Heap or Max-Heap): Efficient for insertion and deletion in O(log n) time.

6.4.4.5 Advantages

- Handles tasks where certain operations need higher priority.
- Useful in real-time systems and scheduling.

6.4.4.6 Applications

- CPU task scheduling in operating systems.
- Dijkstra's shortest path algorithm in graphs.

Managing print jobs in a network printer

NOTE

6.5 QUEUE AS AN ABSTRACT DATA STRUCTURE (ADT)

6.5.1 Introduction

In computer science, a queue is not just a data structure, but also an Abstract Data Type (ADT). An Abstract Data Type is a mathematical model for a certain class of data structures that defines what operations are to be performed but not how these operations will be implemented.

A queue as an ADT specifies the behavior and operations of the queue without detailing whether it will be implemented using arrays, linked lists, or other structures. This abstraction allows programmers to focus on the logic and functionality of the queue while implementation details are hidden.

6.5.2. Definition of Queue ADT

A Queue ADT is defined as a collection of elements that follows the First In, First Out (FIFO) principle:

- The element added first is removed first.
- Elements are inserted at the rear and removed from the front.

Queue ADT focuses on what the queue does rather than how it does it. This separation of interface and implementation is a key concept in software design and modular programming.

6.5.3. Basic Operations of Queue ADT

A Queue ADT typically supports the following core operations:

1. Enqueue (Insert):
   Adds an element to the rear of the queue.
   - Input: Element to be added.
   - Output: Updated queue with the new element at the rear.
2. Dequeue (Remove):
   Removes an element from the front of the queue.
   - Input: None.
   - Output: Element removed from the front.
3. Peek / Front:
   Returns the front element without removing it.
   - Useful for checking the next element to be processed.
4. isEmpty():
   Checks whether the queue contains any elements.
   - Returns true if the queue has no elements.

NOTE

5. isFull():
   Checks whether the queue has reached its maximum capacity (for fixed-size queues).
   - Returns true if the queue is full.

These operations define the interface of the Queue ADT, and the underlying implementation may vary.

6.5.4. Properties of Queue ADT

1. FIFO Behavior:
   - Queue ADT always processes elements in first-come, first-served order.
2. Abstract Nature:
   - Users of the ADT do not need to know how the queue is implemented.
   - The focus is on operations and behavior rather than storage details.
3. Encapsulation:
   - Queue ADT encapsulates data and operations.
   - The data representation (array, linked list, etc.) is hidden from the user.

6.5.5. Implementation Options

Queue ADT can be implemented using different data structures, depending on the requirements:

1. Array-Based Queue:
   - Simple, fixed-size implementation using an array.
   - Limitations: May waste space after repeated dequeues; resizing may be required.
2. Circular Queue (Array):
   - Efficient utilization of array space by wrapping around rear and front.
3. Linked List Queue:
   - Dynamic memory allocation; can grow or shrink as needed.
   - No wasted space.
4. Priority Queue (ADT Variation):
   - Queue where elements are dequeued based on priority rather than order of insertion.

NOTE

6.5.6. Advantages of Queue ADT

- Abstraction: Users only focus on queue operations, not implementation details.
- Reusability: The same queue ADT interface can be used with different underlying implementations.
- Modularity: Changes in implementation do not affect the code that uses the ADT.
- Flexibility: Queue ADT can be extended to circular queues, dequeues, and priority queues without changing its interface.

6.5.7. Applications of Queue ADT

Queue ADT is widely used in real-world and computing scenarios, such as:

1. Operating Systems:
    - CPU scheduling, process queues, printer spooling.
2. Networking:
    - Data packets are buffered in routers using queue ADT.
3. Simulation:
    - Models real-world systems such as queues at banks, ticket counters, or supermarkets.
4. Algorithm Design:
    - Breadth-First Search (BFS) in graphs.
    - Task scheduling in parallel computing.
5. I/O Buffers:

Keyboard buffers, printer buffers, or asynchronous data handling.

6.6 QUEUE IMPLEMENTATION USING LINKED LIST

6.6.1 Introduction

A queue is a linear data structure that follows the First In, First Out (FIFO) principle, meaning that the element inserted first is removed first. While queues can be implemented using arrays, using linked lists is often more efficient because a linked list allows dynamic memory allocation and avoids the limitations of fixed-size arrays.

A linked list-based queue consists of nodes, where each node contains:

1. Data: The value stored in the node.
2. Next Pointer: A pointer that points to the next node in the queue.

NOTE

The queue is managed using two pointers:

- Front: Points to the first element of the queue (to be dequeued next).
- Rear: Points to the last element of the queue (where new elements are inserted).

6.6.2. Structure of Queue Using Linked List

A linked list-based queue can be visualized as follows:

Front --> [Data | Next] --> [Data | Next] --> ... --> [Data | NULL] <-- Rear

- The front node is where elements are removed.
- The rear node is where new elements are added.
- The NULL pointer at the rear indicates the end of the queue.

Unlike array-based queues, a linked list-based queue does not require shifting elements when elements are dequeued, making it memory-efficient and suitable for dynamic queues.

6.6.3. Operations on Linked List Queue

a) Enqueue (Insertion at Rear)

Algorithm:

1. Create a new node with the given data.
2. If the queue is empty (front is NULL), set both front and rear to the new node.
3. Otherwise, set rear->next to the new node and update rear to the new node.

Complexity: O(1) – constant time insertion.

b) Dequeue (Deletion from Front)

Algorithm:

1. If the queue is empty (front is NULL), return an error or underflow message.
2. Store the current front node in a temporary pointer.
3. Move front to front->next.
4. Delete the temporary node.
5. If front becomes NULL, set rear to NULL as well.

Complexity: O(1) – constant time deletion.

c) Peek / Front

- Return the data of the front node without removing it.

NOTE

- Useful for checking the next element to be processed.

d) isEmpty()

- Check if front == NULL.
- Returns true if the queue is empty, otherwise false.

6.6.4. Example Implementation in C++

```
#include <iostream>
using namespace std;

// Node structure
struct Node {
    int data;
    Node* next;
};

// Queue class using linked list
class Queue {
    Node* front;
    Node* rear;

public:
    Queue() {
        front = rear = nullptr;
    }

    // Enqueue operation
    void enqueue(int value) {
        Node* temp = new Node;
        temp->data = value;
        temp->next = nullptr;
        if (rear == nullptr) {
            front = rear = temp;
            return;
```

NOTE

```
        }
        rear->next = temp;
        rear = temp;
    }

    // Dequeue operation
    void dequeue() {
        if (front == nullptr) {
            cout << "Queue Underflow\n";
            return;
        }
        Node* temp = front;
        front = front->next;
        if (front == nullptr)
            rear = nullptr;
        delete temp;
    }

    // Display front element
    void peek() {
        if (front != nullptr)
            cout << "Front element: " << front->data << endl;
        else
            cout << "Queue is empty\n";
    }

    // Check if queue is empty
    bool isEmpty() {
        return front == nullptr;
    }
};
```

NOTE

```
int main() {
    Queue q;
    q.enqueue(10);
    q.enqueue(20);
    q.enqueue(30);
    q.peek();    // Front element: 10
    q.dequeue();
    q.peek();    // Front element: 20
    return 0;
}
```

6.6.5. Advantages of Linked List Queue

1. Dynamic Size:
    - The queue can grow or shrink as needed; no fixed size limitation.
2. Efficient Memory Usage:
    - No wasted space, unlike arrays where dequeued spaces remain unused.
3. No Shifting Required:
    - Unlike array-based linear queues, no element shifting is necessary during dequeue.
4. Constant Time Operations:
    - Enqueue and dequeue operations take O(1) time.

6.6.6. Disadvantages

1. Extra Memory:
    - Each node requires extra memory for the next pointer.
2. Pointer Overhead:
    - Operations involve pointer manipulation, which may slightly slow down performance compared to arrays for small queues.

6.6.7. Applications

1. Dynamic Task Scheduling:
    - OS process scheduling where queue size is unpredictable.
2. Networking:

NOTE

- Data packet handling in routers and buffers.

3. Simulation:
   - Real-world queues such as ticket counters or customer service lines.
4. Breadth-First Search (BFS):

Graph traversal algorithms often use linked list queues for dynamic node handling.

Check Your Progress 6.1

MCQ:
Which operation inserts an element at the rear of a queue?
a) Push
b) Pop
c) Enqueue
d) Dequeue

True/False:
Queues follow the Last In First Out (LIFO) principle.

Short Questions:

1. Define a queue in computer science.

.....

.....

2. Name two common types of queues.

.....

.....

6.7 APPLICATIONS OF QUEUES

6.7.1 Introduction

A queue is a linear data structure that follows the First-In-First-Out (FIFO) principle. This means that the element inserted first is removed first. Queues are not just theoretical concepts—they are widely used in computer science, operating systems, networking, and real-life scenarios. The structure of queues allows for organized and fair handling of tasks, requests, or data in sequential order.

Queues can be implemented using arrays, linked lists, or advanced forms like circular queues, deques, and priority queues, depending on the application. Understanding the practical applications of queues helps in designing efficient algorithms and systems.

6.7.2. Operating System Applications

NOTE

Operating systems heavily rely on queues to manage processes, tasks, and resources efficiently. Some key examples include:

a) CPU Scheduling

- The CPU handles multiple processes in a system.
    - Ready Queue**:** Processes waiting to execute are stored in a queue.
- The scheduler selects processes based on FIFO or priority, depending on the scheduling algorithm.
- Example: In a round-robin scheduling, processes are managed in a circular queue.

b) Printer Spooling

- Multiple print jobs sent to a printer are stored in a print queue.
- Jobs are printed in the order they were received (FIFO).
- Ensures fair access to printer resources and avoids conflicts.

c) Disk Scheduling

- Disk I/O requests are queued for efficient read/write operations.
- Queues allow the OS to process requests systematically and minimize seek time.

6.7.3. Networking Applications

Queues are essential in networking for data transfer, buffering, and traffic management**.**

a) Packet Switching

- Routers and switches handle data packets from multiple sources.
- Incoming packets are stored in a queue buffer before being forwarded.
- Queues prevent packet loss during congestion and ensure fair transmission.

b) Bandwidth Management

- Network routers maintain queues for different services or priorities.
- Example: Priority Queue ensures voice or video packets are transmitted before standard data packets to maintain quality of service (QoS).

c) Message Queues

- In distributed systems, processes communicate asynchronously using message queues.

NOTE

- The sender process can enqueue messages without waiting for the receiver to be ready.

6.7.4. Real-Life Applications

Queues are modeled after real-world lines where fairness and order are important:

a) Customer Service

- Banks, post offices, and ticket counters use queues to serve customers in the order they arrive.

b) Call Centers

- Incoming calls are enqueued and answered sequentially.
- Priority queues may be used for VIP customers.

c) Supermarkets and Shops

- Checkout counters use queues to ensure fair service.

These applications demonstrate how queues manage resources effectively and maintain order in real-life systems.

6.7.5. Algorithmic Applications

Queues play a key role in algorithm design and problem-solving:

a) Breadth-First Search (BFS)

- BFS in graphs uses a queue to explore nodes level by level.
- Nodes are enqueued as they are discovered and dequeued when processed.

b) Tree Traversals

- Level-order traversal of a binary tree is implemented using a queue.
- Nodes at each level are enqueued and dequeued sequentially.

c) Sliding Window Problems

- Queues or deques are used in algorithms to efficiently calculate maximum/minimum in a moving window.

d) Job Scheduling

- Queues ensure tasks are executed in order while handling dynamic arrivals of jobs.

6.7.6. Special Queue Applications

a) Circular Queues

- Used in traffic management systems and buffer management where resources are limited and must be reused efficiently.

NOTE

b) Priority Queues

- In operating systems and network scheduling, tasks with higher importance are handled first.
- Example: Emergency response systems use priority queues to serve critical patients first.

c) Deques

- Double-ended queues are used in undo-redo operations in text editors or browser history management.

Activity 1: The "Print Job Manager" – FIFO Simulation

Objective: To implement a First-In-First-Out (FIFO) Queue data structure that simulates a network printer spooler. Students will learn how to manage asynchronous requests (incoming print jobs) and process them in the exact order they arrived, handling limitations like buffer size (Queue Capacity).

Materials / Tools / Software Required:

- Computer: Standard development setup.
- Programming Language: Java, C++, or Python.
- IDE: Eclipse, IntelliJ, or VS Code.
- Data Structure Diagram: A visual reference showing the "Front" (Head) and "Rear" (Tail) pointers.

Expected Outcome: By the end of this activity, students will be able to:

1. Implement Core Operations: Write functions for enqueue() (add a document ID to the rear) and dequeue() (remove/print the document at the front).
2. Manage Pointers: Correctly manipulate the front and rear indices (or pointers) so they wrap around if implementing a Circular Queue.
3. Handle Edge Cases: Write logic to detect and report Queue Overflow (when the printer buffer is full) and Queue Underflow (trying to print when the queue is empty).
4. Simulate Lag: Create a loop where jobs arrive faster than they are printed, forcing the Queue to fill up and demonstrating why buffering is necessary.

NOTE

6.8 SUMMARY

Queues are linear data structures that store elements in a sequential order following the First-In-First-Out (FIFO) principle. They are widely used to manage tasks or data in a fair and organized manner. The linear queue is the simplest type, with insertion at the rear and deletion at the front, but it may waste memory in fixed-size implementations. The circular queue overcomes this limitation by wrapping the rear to the beginning of the array, ensuring efficient memory utilization. A double-ended queue (deque) allows insertion and deletion at both ends, providing flexibility for applications like undo/redo operations. The priority queue handles tasks based on priority rather than order of arrival, which is essential in operating systems, networking, and real-time systems.

Queues can be implemented using arrays or linked lists. Linked list implementation allows dynamic memory allocation and constant-time insertion and deletion without wasted space. Queues are also an important Abstract Data Type (ADT), focusing on operations like enqueue, dequeue, peek, isEmpty, and isFull, while hiding implementation details.

Applications of queues span operating systems (CPU scheduling, printer spooling), networking (packet buffering, message queues), algorithms (BFS, level-order tree traversal), and real-life scenarios (bank lines, call centers, ticket counters). Variants like circular queues, deques, and priority queues enhance their usability in specialized contexts. Understanding queues is essential for efficient resource management, algorithm design, and solving real-world problems

6.9 GLOSSARY

1. Queue: Linear data structure following FIFO.
2. FIFO (First-In-First-Out): Principle where first element inserted is first removed.
3. Linear Queue: Queue with insertion at rear and deletion at front.
4. Circular Queue: Queue where rear wraps around to reuse empty spaces.
5. Deque (Double-Ended Queue): Queue allowing insertion and deletion at both ends.
6. Priority Queue: Queue where elements are removed based on priority.
7. Enqueue: Operation to insert an element into a queue.

NOTE

8. Dequeue**:** Operation to remove an element from a queue.
9. ADT (Abstract Data Type): A model defining operations on data without implementation details.
10. Linked List: Dynamic data structure with nodes containing data and a pointer to the next node.

6.10 ANSWER TO CHECK YOUR PROGRESS-POSSIBLE ONE

Check Your Progress 6.1

MCQ Answer:
c) Enqueue – adds an element at the rear of the queue.

True/False Answer:
False – queues follow First In First Out (FIFO) principle.

Short Answers:

1. A queue is a linear data structure where the first element added is the first removed (FIFO).
2. Common types are Circular Queue and Priority Queue.

6.11 SUGGESTED READINGS – OER

Sr. No.	Title	URL	QR Code
1	Data Structure Using C++	http://books.aisectonline.com/User/UnitUrls?unitId=3714	

6.12 REFERENCE

- Data Structures and Algorithms in C++ – Michael T. Goodrich, Roberto Tamassia, David M. Mount – ISBN: 9780470383278 – 2011
- Data Structures and Algorithm Analysis in C++ – Mark Allen Weiss – ISBN: 9780132847377 – 2013
- Data Structures and Algorithms in C++ – Adam Drozdek – ISBN: 9781133608426 – 2012
- Data Structures Using C++ – D. S. Malik – ISBN: 9788131518236 – 2012

NOTE

- Data Structures Through C++ – Yashavant Kanetkar – ISBN: 9789355515834 – 2024
- Fundamentals of Data Structures in C++ – Ellis Horowitz, Sartaj Sahni, Dinesh Mehta – ISBN: 9788173716065 – 2008
- Data Structures: A Pseudocode Approach with C – Richard F. Gilberg, Behrouz A. Forouzan – ISBN: 9780534953965 – 2005
- Data Structures and Program Design in C++ – Robert L. Kruse, Alexander J. Ryba – ISBN: 9780136159079 – 2010

6.13 EXERCISE

Part 1

A. Multiple choice question

1. Which principle does a queue follow?
a) LIFO
b) FIFO
c) FILO
d) Random

2. Which queue allows insertion and deletion from both ends?
a) Linear Queue
b) Circular Queue
c) Deque
d) Priority Queue

3. Which implementation of a queue uses dynamic memory and avoids wasted space?
a) Array-based Queue
b) Linked List-based Queue
c) Linear Queue
d) Circular Queue

4. In a priority queue, which element is removed first?
a) Last inserted
b) First inserted
c) Highest priority
d) Random

5. Which of the following is not a queue operation?
a) Enqueue
b) Dequeue
c) Push
d) Peek

Answers with Explanations:

NOTE

1. Answer: B) FIFO
Explanation: A queue follows First-In-First-Out (FIFO), meaning the first element inserted is the first to be removed.

2. Answer: C) Deque
Explanation: A Deque (Double-Ended Queue) allows insertion and deletion at both the front and rear ends.

3. Answer: B) Linked List-based Queue
Explanation: Using a linked list, a queue can grow or shrink dynamically, preventing wasted memory unlike fixed-size array implementations.

4. Answer: C) Highest priority
Explanation: Priority queues remove elements based on priority rather than order of insertion.

5. Answer: C) Push
Explanation: Push is used in stacks, not queues. Queues use enqueue to insert and dequeue to remove elements.

B. Small answer type question

1. Define a queue and explain the FIFO principle.
2. What are the basic operations of a queue ADT?
3. Explain the difference between a linear queue and a circular queue.
4. What are the advantages of implementing a queue using a linked list?
5. List three real-world applications of queues.

Part 2

C. Long answer type question (answer in 200 words)

1. Explain the structure, operations, and working of a linear queue with an example.
2. Describe the implementation of a queue using a linked list, including enqueue and dequeue operations.
3. Compare linear queue, circular queue, deque, and priority queue in terms of operations and applications.
4. Explain queue as an Abstract Data Type (ADT), including its properties and importance.

NOTE

5. Describe the role of queues in operating systems and networking applications.

Part 3

D. Long answer type question (answer in 300 words)

1. Discuss in detail the different types of queues, including linear queue, circular queue, deque, and priority queue, with examples.
2. Explain the advantages and limitations of each queue type and how to choose the appropriate queue for an application.
3. Write a detailed explanation of queue implementation using arrays and linked lists, including sample diagrams.
4. Describe the applications of queues in real life, algorithms, and computer systems, highlighting the importance of FIFO behavior.
5. Explain queue operations (enqueue, dequeue, peek, isEmpty, isFull) in the context of ADT, and discuss how ADTs abstract implementation details for practical programming.

UNIT 7
LINKED LIST

NOTE

UNIT STRUCTURE

7.1 Introduction

7.2 Objective

7.3 Concept of Lists, Linked Lists, and Arrays

7.4 Singly (Linear) Linked List

7.5 Circular Singly Linked List (CSLL)

7.6Doubly Linked List (DLL)

7.7 Summary

7.8 Glossary

7.9 Answer to check your progress

7.10 Suggested readings - OER

7.11 Reference

7.12 Exercise

7.1 INTRODUCTION

In computer science, data structures are essential for organizing and storing data efficiently. Among the fundamental data structures, lists and their variations play a crucial role in managing collections of elements. A list is a sequence of elements arranged in a specific order, allowing operations such as insertion, deletion, and traversal. Lists can be static, like arrays, which store elements in contiguous memory locations, or dynamic, like linked lists, which store elements in nodes connected by pointers.

Arrays provide fast, index-based access to elements but have a fixed size, making them less flexible when the number of elements changes. Singly linked lists allow dynamic memory allocation and sequential access, with nodes containing data and a pointer to the next node. Doubly linked lists extend this concept by including pointers to both the next and previous nodes, enabling bi-directional traversal. Circular linked lists link the last node back to the head, allowing continuous traversal, which is useful in operating systems, playlists, and simulations.

NOTE

The concept of a header node is often used to simplify linked list operations by storing metadata like list size or a pointer to the first node. Understanding these structures is critical for designing efficient algorithms, implementing dynamic data structures like stacks and queues, and managing memory effectively in real-world applications

7.2 OBJECTIVE

After completing this unit, you will be able to:

1. Understand the concept of lists and their types, including arrays and linked lists.
2. Implement singly, doubly, and circular linked lists using C++.
3. Explain the role of a header node in managing linked lists efficiently.
4. Compare arrays and linked lists in terms of memory allocation, access, and operations.
5. Identify and apply real-world applications of linked lists in programming, operating systems, and simulations.

7.3 CONCEPT OF LISTS, LINKED LISTS, AND ARRAYS

7.3.1 List

In computer science, a list is a fundamental data structure that represents a collection of elements arranged in a specific order. Each element in a list can be accessed, inserted, or removed based on certain rules. Lists are widely used because they allow us to store multiple items together, making programs more organized and efficient.

There are different types of lists based on their implementation and the type of elements they store:

- Simple Lists: A collection of elements stored sequentially.
- Arrays: A fixed-size collection of similar type elements stored in contiguous memory locations.
- Linked Lists: A dynamic collection of elements where each element points to the next one.

Key Characteristics of Lists:

1. Ordered: Elements have a specific sequence.
2. Indexable (in some types): Elements can be accessed using an index.

3. Dynamic or Static: Some lists like arrays are static (fixed size), while others like linked lists can grow or shrink dynamically.
4. Homogeneous or Heterogeneous: Arrays usually store elements of the same type, while linked lists can store different types of data depending on implementation.

Lists are essential in programming because they help manage data efficiently, and operations such as searching, sorting, and traversal can be performed easily.

7.3.2. Arrays

7.3.2.1 Definition

An array is a collection of elements of the same type stored in contiguous memory locations. Arrays provide efficient random access to elements using an index.

Example in C++:

int marks[5] = {80, 90, 85, 70, 95};

Here, marks is an array of 5 integers. Each element can be accessed using its index (starting from 0).

7.3.2.2 Features of Arrays

- Fixed Size: The number of elements is defined at declaration and cannot be changed.
- Efficient Access: Direct access to elements using an index makes arrays fast.
- Homogeneous Elements: All elements must be of the same data type.

7.3.2.3 Advantages

1. Random Access: Direct access using indices allows efficient retrieval.
2. Memory Efficiency: Stored in contiguous memory, making access faster.
3. Easy to Traverse: Loops can be used to iterate through all elements.

7.3.2.4 Disadvantages

1. Fixed Size: Cannot resize after declaration.
2. Insertion/Deletion Costly: Shifting elements is required when inserting or deleting elements.

7.3.3. Linked Lists

NOTE

7.3.3.1 Definition

A linked list is a linear data structure where elements, called nodes, are connected using pointers. Unlike arrays, linked lists do not require contiguous memory locations, making them flexible for dynamic memory allocation.

7.3.3.2 Structure of a Node

Each node typically contains:

1. Data: Stores the actual value.
2. Pointer/Reference: Points to the next node in the list.

Diagram of a simple linked list:

[Data | Next] -> [Data | Next] -> [Data | Next] -> NULL

7.3.3.3 Types of Linked Lists

1. Singly Linked List: Each node points to the next node. Traversal is in one direction only.
2. Doubly Linked List: Each node contains two pointers – one to the next node and one to the previous node. Allows forward and backward traversal.
3. Circular Linked List: The last node points back to the first node, forming a circle. Can be singly or doubly linked.

7.3.3.4 Advantages of Linked Lists

1. Dynamic Size: Can grow or shrink during runtime.
2. Efficient Insertions/Deletions: Adding or removing nodes does not require shifting elements.
3. Memory Utilization: Memory is allocated as needed.

7.3.3.5 Disadvantages of Linked Lists

1. Sequential Access: Cannot access elements directly; must traverse from the head.
2. Extra Memory: Each node requires additional memory for storing pointers.
3. Complex Implementation: Requires careful handling of pointers to avoid errors.

7.3.3.6 Basic Operations

1. Traversal: Visiting each node to display or process data.
2. Insertion: Adding a node at the beginning, end, or a specific position.

NOTE

3. Deletion: Removing a node from the beginning, end, or a specific position.
4. Searching: Finding a node with specific data.

Example of Singly Linked List Node in C++:

```
struct Node {
   int data;
   Node* next;
};
```

7.4 SINGLY (LINEAR) LINKED LIST

7.4.1 Introduction

A Singly Linked List (SLL), also known as a linear linked list, is one of the simplest and most commonly used types of linked lists in computer science. It is a linear data structure in which elements are not stored in contiguous memory locations (unlike arrays) but are connected using pointers. Each element in a singly linked list is called a node, and each node contains data and a pointer to the next node.

In a singly linked list, traversal is possible in only one direction, from the head node to the last node. The last node contains a pointer pointing to NULL, which indicates the end of the list.

Singly linked lists are widely used in dynamic memory management, implementing stacks, queues, and other complex data structures

7.4.2. Structure of a Node

A node is the basic building block of a linked list. Each node in a singly linked list consists of two parts:

1. Data: Stores the actual value of the node (integer, character, string, or a complex data type).
2. Next (Pointer): Stores the memory address of the next node in the list.

Diagram of a node:

```
+-------+-------+
| Data  | Next  |
+-------+-------+
```

Example in C++:

```
struct Node {
   int data;      // stores value
```

NOTE

```
    Node* next;    // points to the next node
};
```

7.5.3. Structure of a Singly Linked List

A singly linked list consists of a sequence of nodes connected together. The first node is called the head, and the last node points to NULL.

Visual Representation:

Head -> [Data|Next] -> [Data|Next] -> [Data|Next] -> NULL

Here, each node contains data and a pointer to the next node, forming a chain of nodes.

7.4.4. Basic Operations on a Singly Linked List

Singly linked lists support several fundamental operations that are essential for manipulating the data structure.

7.4.4.1 Traversal

Traversal means visiting each node of the linked list starting from the head node to the end. Traversal is used to display all elements or perform operations on each node.

C++ Example:

```
void traverse(Node* head) {
    Node* temp = head;
    while(temp != NULL) {
        cout << temp->data << " -> ";
        temp = temp->next;
    }
    cout << "NULL" << endl;
}
```

Output Example:

10 -> 20 -> 30 -> 40 -> NULL

7.5.4.2 Insertion

Insertion in a singly linked list can be done in three positions:

1. Insertion at the Beginning: New node becomes the head of the list.
2. Insertion at the End: New node is added after the last node.

3. Insertion at a Specific Position: Node is added after a specified position in the list.

C++ Example (Insertion at Beginning):

```
void insertAtBeginning(Node** head, int value) {
    Node* newNode = new Node();
    newNode->data = value;
    newNode->next = *head;
    *head = newNode;
}
```

7.4.4.3 Deletion

Deletion removes a node from a singly linked list. Nodes can be deleted from:

1. Beginning: Remove the head node.
2. End: Remove the last node.
3. Specific Position: Remove a node at a specific position.

C++ Example (Deletion at Beginning):

```
void deleteAtBeginning(Node** head) {
    if(*head == NULL) return;
    Node* temp = *head;
    *head = (*head)->next;
    delete temp;
}
```

7.4.4.4 Searching

Searching involves finding whether a node with a specific value exists in the list. Traversal is used for searching since nodes are connected sequentially.

C++ Example:

```
bool search(Node* head, int key) {
    Node* temp = head;
    while(temp != NULL) {
        if(temp->data == key)
            return true;
        temp = temp->next;
    }
```

NOTE

```
    return false;
}
```

7.54.5. Applications of Singly Linked Lists

Singly linked lists are widely used in scenarios where dynamic memory allocation and efficient insertion/deletion are required:

- Implementation of stacks and queues.
- Polynomial arithmetic and expression evaluation.
- Dynamic memory management in operating systems.
- Graph adjacency lists representation.
- Navigation systems, such as browser history (previous links).

7.5 CIRCULAR SINGLY LINKED LIST (CSLL)

7.5.1 Introduction

A Circular Singly Linked List (CSLL) is a variation of the singly linked list in which the last node points back to the first node, forming a circle. Unlike a linear singly linked list, a circular singly linked list does not have a NULL at the end**,** and traversal can continue indefinitely if not stopped explicitly.

CSLL is a dynamic data structure, meaning it can grow or shrink during runtime, and is particularly useful in situations where all nodes need to be accessed repeatedly in a circular manner**.**

Applications of CSLL include:

- Operating system process scheduling (round-robin scheduling).
- Music or video playlist looping**.**
- Circular queues implementation**.**
- Simulation of real-time systems.

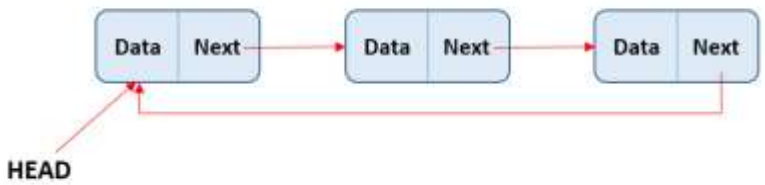


Fig 7.1: CIRCULAR SINGLY LINKED LIST (CSLL)

7.5.2. Structure of a Node

Diagram of a node:

NOTE

```
+-------+-------+
| Data  | Next  |
+-------+-------+
```

C++ Definition of a Node:

```
struct Node {
    int data;      // value of the node
    Node* next;    // pointer to the next node
};
```

7.5.3. Structure of Circular Singly Linked List

Visual Representation:

```
Head -> [Data|Next] -> [Data|Next] -> [Data|Next] -+
      ^-----------------------------------------+
```

Key Points:

- Any node can be considered the starting point because the list is circular.
- Traversal must stop when the starting node is reached again.

7.5.4. Operations on Circular Singly Linked List

7.5.4.1 Traversal

C++ Example:

```
void traverse(Node* head) {
    if(head == NULL) return;
    Node* temp = head;
    do {
        cout << temp->data << " -> ";
        temp = temp->next;
    } while(temp != head);
    cout << "(back to head)" << endl;
}
```

Output Example:

```
10 -> 20 -> 30 -> 40 -> (back to head)
```

NOTE

7.5.4.2 Insertion

C++ Example (Insertion at End):

```
void insertAtEnd(Node** head, int value) {
    Node* newNode = new Node();
    newNode->data = value;
    if(*head == NULL) {
        *head = newNode;
        newNode->next = newNode; // points to itself
        return;
    }
    Node* temp = *head;
    while(temp->next != *head)
        temp = temp->next;
    temp->next = newNode;
    newNode->next = *head;
}
```

7.5.4.3 Deletion

C++ Example (Deletion at Beginning):

```
void deleteAtBeginning(Node** head) {
    if(*head == NULL) return;
    Node* temp = *head;
    if(temp->next == *head) { // only one node
        delete temp;
        *head = NULL;
        return;
    }
    Node* last = *head;
    while(last->next != *head)
        last = last->next;
    Node* toDelete = *head;
    *head = (*head)->next;
    last->next = *head;
    delete toDelete;
```

NOTE

```
}
```

7.5.4.4 Searching

C++ Example:

```
bool search(Node* head, int key) {
    if(head == NULL) return false;
    Node* temp = head;
    do {
        if(temp->data == key)
            return true;
        temp = temp->next;
    } while(temp != head);
    return false;
}
```

7.5.5. Applications of Circular Singly Linked List

1. Round-Robin Scheduling in Operating Systems – processes are executed in a circular order.
2. Music/Video Playlist Loops – automatically repeats the playlist.
3. Circular Queues Implementation – improves memory utilization.
4. Real-time Simulations – for repeating cyclic events.
5. Network Applications – for token ring network simulation.

Check Your Progress 7.1

MCQ:
Which of the following is true about a linked list?
a) Elements are stored in contiguous memory locations
b) Each node contains data and a pointer to the next node
c) Linked lists cannot grow dynamically
d) It does not use pointers

True/False:
In a singly linked list, each node can point to multiple next nodes.

Short Questions:

1. Define a linked list.

.....

.....

2. Name two advantages of using linked lists over arrays.

.....

.....

NOTE

7.6 DOUBLY LINKED LIST (DLL)

7.6.1 Introduction

A Doubly Linked List (DLL) is a linear data structure in which each node contains two pointers in addition to the data: one pointer points to the next node**,** and the other points to the previous node. This bidirectional linking allows traversal in both forward and backward directions, unlike singly linked lists which allow traversal only in one direction.

Doubly linked lists are widely used in computer science for applications where bi-directional traversal, efficient insertion, and deletion are required. They serve as the foundation for complex data structures like deque, stacks, navigation systems, and certain types of graphs**.**

7.6.2. Structure of a Node

Each node in a DLL contains three components:

1. Data: Stores the actual value of the node (e.g., integer, character, string).
2. Next Pointer: Points to the next node in the sequence.
3. Previous Pointer: Points to the previous node in the sequence.

Diagram of a node:

+-------+-------+-------+

| Prev | Data | Next |

+-------+-------+-------+

C++ Structure Example:

```
struct Node {
   int data;       // stores the value
   Node* next;     // pointer to next node
   Node* prev;     // pointer to previous node
};
```

7.66.3. Structure of a Doubly Linked List

A DLL is formed by connecting nodes using next and previous pointers. The first node is called the head, and its previous pointer is NULL. The last node is called the tail, and its next pointer is NULL.

Visual Representation:

NULL <- [Prev|Data|Next] <-> [Prev|Data|Next] <-> [Prev|Data|Next] -> NULL

Key Points:

NOTE

- Traversal can occur forward using the next pointer or backward using the prev pointer.
- Insertion and deletion are more flexible compared to singly linked lists.

7.6.4. Operations on Doubly Linked List

Doubly linked lists support all fundamental operations, including traversal, insertion, deletion, and searching.

7.6.4.1 Traversal

Traversal in a DLL can be done forward (head to tail) or backward (tail to head).

Forward Traversal Example in C++:

```
void traverseForward(Node* head) {
    Node* temp = head;
    while(temp != NULL) {
        cout << temp->data << " <-> ";
        temp = temp->next;
    }
    cout << "NULL" << endl;
}
```

Backward Traversal Example in C++:

```
void traverseBackward(Node* tail) {
    Node* temp = tail;
    while(temp != NULL) {
        cout << temp->data << " <-> ";
        temp = temp->prev;
    }
    cout << "NULL" << endl;
}
```

7.6.4.2 Insertion

C++ Example (Insertion at Beginning):

```
void insertAtBeginning(Node** head, int value) {
    Node* newNode = new Node();
    newNode->data = value;
    newNode->prev = NULL;
    newNode->next = *head;
```

NOTE

```
    if(*head != NULL)
        (*head)->prev = newNode;
    *head = newNode;
}
```

7.6.4.3 Deletion

C++ Example (Deletion at Beginning):

```
void deleteAtBeginning(Node** head) {
    if(*head == NULL) return;
    Node* temp = *head;
    *head = (*head)->next;
    if(*head != NULL)
        (*head)->prev = NULL;
    delete temp;
}
```

Activity 1: The "Dynamic Playlist" – Music Player Simulation

Objective: To understand the structure of a Singly Linked List by modeling a music playlist. Students will learn how to create non-contiguous data structures where every element (Node) holds data (Song Title) and a reference (Pointer) to the next element, allowing for efficient insertion and deletion without shifting memory.

Materials / Tools / Software Required:

- Computer: Standard development setup.
- Programming Language: C++ or Java (preferred for clear reference/pointer handling), or Python.
- IDE/Text Editor: VS Code, IntelliJ, or similar.
- Visual Aid: Paper cards and string (for a physical walkthrough of "linking" nodes before coding).

Expected Outcome: By the end of this activity, students will be able to:

1. Define the Node Class: Create a class/struct containing a string for the song title and a pointer/reference named next.
2. Manage the Head: Implement logic to track the start of the list.
3. Perform Insertion: Write a function to add a "Song" between two existing songs by strictly changing pointer references (not moving data in memory).
4. Perform Deletion: Remove a song from the playlist and correctly "patch" the list by linking the previous node directly

7.7 SUMMARY

Lists are a core data structure in computer science, providing a way to store and manage collections of elements efficiently. Arrays are fixed-size lists that store elements in contiguous memory locations, offering fast random access but limited flexibility for insertion and deletion. In contrast, linked lists are dynamic structures composed of nodes connected via pointers, allowing efficient memory allocation and flexible insertion/deletion.

A singly linked list (SLL) stores data in nodes, each containing a pointer to the next node, allowing sequential traversal from the head to the last node. Doubly linked lists (DLL) enhance traversal by including a pointer to the previous node, enabling bi-directional movement. Circular singly linked lists (CSLL) link the last node back to the head, allowing cyclic traversal useful in operating system scheduling, music playlists, and simulations.

The header node is a special node at the beginning of a list, simplifying list management by storing metadata, such as the number of nodes or pointers to the first node. Linked lists find applications in implementing stacks, queues, deques, polynomial arithmetic, graph adjacency lists, undo-redo systems, and dynamic memory management.

Understanding lists and linked lists equips students to design efficient algorithms, manage dynamic memory, and implement real-world data structures. They form the foundation for advanced structures such as trees, graphs, and hash tables, and are essential for problem-solving in programming, simulations, and software development.

7.8 GLOSSARY

1. List – A collection of elements arranged in a specific order.
2. Array – A fixed-size collection of elements stored in contiguous memory.
3. Singly Linked List – A linked list where each node points to the next node.
4. Doubly Linked List – A linked list where each node has pointers to both next and previous nodes.
5. Circular Linked List – A linked list where the last node points back to the first node.
6. Node – The basic element of a linked list, containing data and pointer(s).
7. Header Node – A special node at the beginning of a linked list containing metadata.
8. Traversal – Visiting each element in a data structure to process or display data.

NOTE

9. Insertion – Adding a new node or element to a data structure.
10. Deletion – Removing a node or element from a data structure.

7.9 ANSWER TO CHECK YOUR PROGRESS-POSSIBLE ONE

Check Your Progress 7.1

MCQ Answer:
b) Each node contains data and a pointer to the next node.

True/False Answer:
False – each node points to only one next node in a singly linked list.

Short Answers:

1. A linked list is a linear data structure where elements (nodes) are connected using pointers.
2. Advantages: Dynamic memory allocation and easy insertion/deletion.

7.10 SUGGESTED READINGS – OER

Sr. No.	Title	URL	QR Code
1	Data Structure Using C++	http://books.aisectonline.com/User/UnitUrls?unitId=3715	

7.11 REFERENCE

1. Data Structures and Algorithms in C++ – Michael T. Goodrich, Roberto Tamassia, David M. Mount – ISBN: 9780470383278 – 2011
2. Data Structures and Algorithm Analysis in C++ – Mark Allen Weiss – ISBN: 9780132847377 – 2013
3. Data Structures and Algorithms in C++ – Adam Drozdek – ISBN: 9781133608426 – 2012
4. Data Structures Using C++ – D. S. Malik – ISBN: 9788131518236 – 2012
5. Data Structures Through C++ – Yashavant Kanetkar – ISBN: 9789355515834 – 2024
6. Fundamentals of Data Structures in C++ – Ellis Horowitz, Sartaj Sahni, Dinesh Mehta – ISBN: 9788173716065 – 2008

7. Data Structures: A Pseudocode Approach with C – Richard F. Gilberg, Behrouz A. Forouzan – ISBN: 9780534953965 – 2005
8. Data Structures and Program Design in C++ – Robert L. Kruse, Alexander J. Ryba – ISBN: 9780136159079 – 2010

7.12 EXERCISE

Part 1

A. Multiple choice question

1. Which of the following is a dynamic data structure?
a) Array
b) Singly Linked List
c) Static Variable
d) Integer

2. In a singly linked list, each node contains:
a) Data only
b) Data and pointer to previous node
c) Data and pointer to next node
d) Only pointer

3. The last node in a circular linked list points to:
a) NULL
b) Head node
c) Previous node
d) Middle node

4. Which data structure allows bi-directional traversal?
a) Array
b) Singly Linked List
c) Doubly Linked List
d) Circular Singly Linked List

5. A header node in a linked list is used to:
a) Store first node data
b) Simplify list operations
c) Store last node
d) Delete nodes

Answers with Explanations:

1. Answer: B) Singly Linked List
Explanation: Linked lists can grow or shrink at runtime, unlike arrays which have fixed size, making them dynamic data structures.

2. Answer: C) Data and pointer to next node
Explanation: Each node stores the actual data and a pointer to the next node in the sequence.

NOTE

3. Answer: B) Head node
Explanation: In a circular linked list, the last node links back to the first (head) node, forming a circular structure.

4. Answer: C) Doubly Linked List
Explanation: Doubly linked lists have pointers to both the previous and next nodes, enabling traversal in both directions.

5. Answer: B) Simplify list operations
Explanation: A header node acts as a placeholder at the start of a list to make insertion, deletion, and traversal easier.

B. Small answer type question

1. What is a singly linked list, and how does it differ from an array?
2. Explain the structure of a doubly linked list node.
3. Define a circular singly linked list and its advantages.
4. What is a header node, and why is it used in linked lists?
5. List at least five real-world applications of linked lists.

Part 2

C. Long answer type question (answer in 200 words)

1. Describe the structure, advantages, and disadvantages of singly linked lists.
2. Compare arrays and linked lists in terms of memory, insertion, deletion, and access.
3. Explain the types of doubly linked lists and their applications.
4. How is traversal, insertion, and deletion performed in a circular singly linked list?
5. Discuss the role of linked lists in operating systems and dynamic memory management.

Part 3

D. Long answer type question (answer in 300 words)

1. Explain in detail the concept of singly, doubly, and circular linked lists with diagrams and examples.
2. Describe the structure and operations of a header node and its significance in managing linked lists.
3. Discuss various applications of linked lists in real-world software and system-level programming.
4. Compare arrays, singly linked lists, doubly linked lists, and circular linked lists in terms of memory efficiency, traversal, insertion, and deletion.
5. Explain how linked lists are used to implement stacks, queues, and deques with suitable examples.

UNIT 8
INTRODUCTION TO TREES

NOTE

UNIT STRUCTURE

8.1 Introduction

8.2 Objective

8.3 Tree Data Structure: Concepts and Terminology

8.4 Representation of Trees in Memory

8.5 The Binary Tree and Its Representation

8.6 Types of Binary Trees & Basic Operations

8.7 Summary

8.8 Glossary

8.9 Answer to check your progress

8.10 Suggested readings - OER

8.11 Reference

8.12 Exercise

8.1 INTRODUCTION

Data structures are the backbone of efficient software design. While our previous units focused on linear data structures—such as Arrays, Linked Lists, Stacks, and Queues—where data is arranged sequentially, real-world data often demands a more complex organization. In this unit, we transition from linear to hierarchical data structures with the Tree.

Imagine the file system on your computer, the HTML structure of a webpage, or the organizational chart of a corporation. None of these can be efficiently represented by a straight line of data. They require a parent-child relationship, a structure where one element branches out into many. This is exactly what a Tree data structure provides.

In this unit, we will explore the fundamental concepts of Trees, specifically focusing on the Binary Tree, which is a cornerstone of Computer Science. We will examine how to define trees, the terminology used to describe them (such as Root, Leaf, Height, and Depth), and how they are stored in computer memory using both Arrays (Sequential) and Linked Lists (Dynamic) representations. Finally, we will learn how to navigate these structures using various Traversal algorithms (Pre-order, In-order, and Post-order), which are essential for processing

NOTE

hierarchical data. This unit lays the necessary groundwork for advanced topics like Binary Search Trees (BST) and Heaps.

8.2 OBJECTIVE

After completing this unit, you will be able to:

1. Define a Tree data structure and identify its key components, including the Root, Nodes, Edges, Leaves, and Subtrees.
2. Differentiate between various types of Binary Trees, such as Full, Complete, Perfect, and Skewed Binary Trees.
3. Implement a Binary Tree in C++ using both Sequential (Array-based) and Linked (Pointer-based) representations.
4. Analyze the advantages and disadvantages of Array vs. Linked representations in terms of memory usage and access speed.
5. Apply recursive algorithms to perform Pre-order, In-order, and Post-order traversals on a Binary Tree

8.3 TREE DATA STRUCTURE: CONCEPTS AND TERMINOLOGY

8.3.1 Introduction

In our previous studies of data structures, we focused on linear data structures like Arrays, Linked Lists, Stacks, and Queues. In those structures, data is arranged sequentially—one element after another. However, many real-world data relationships are not sequential; they are hierarchical.

Think of the file system on your computer: you have a main drive (C:), inside which are folders, and inside those folders are files or sub-folders. This cannot be efficiently represented by a straight line of data. This is where the Tree data structure comes in.

A Tree is a non-linear, hierarchical data structure consisting of a collection of nodes such that:

1. There is a distinguished node called the Root.
2. The remaining nodes are partitioned into disjoint sets, each of which is itself a tree (called a subtree).

Unlike arrays where access is O(1), or linked lists where access is linear O(n), trees provide a middle ground, often allowing for searching and sorting in O(\log n) time, making them incredibly efficient for managing large datasets.

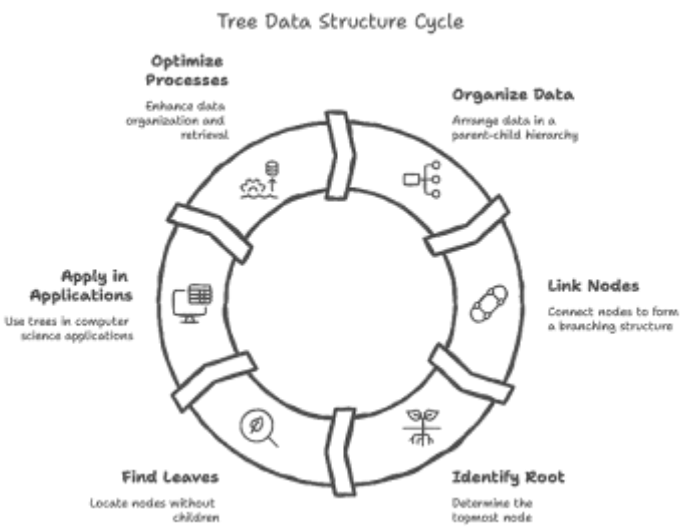


Fig 8.1: The Tree Data Structure Cycle

8.3.2. Key Terminology (The Vocabulary of Trees)

To work with trees in C++, you must understand the specific vocabulary used to describe them. In an exam or interview, these definitions are crucial.

- Node: The fundamental component of a tree. A node usually contains two things: the data (value) and pointers (references) to other nodes.
- Root: The topmost node of the tree. It is the only node in the tree that has no parent. If the root is NULL, the tree is empty.
- Edge: The connecting link between two nodes. If a tree has N nodes, it will have exactly N-1 edges.
- Parent: A node that has an outgoing edge to another node.
- Child: A node that has an incoming edge from a parent.
- Siblings: Nodes that share the same parent.
- Leaf Node (External Node): A node that does not have any children. These are the "bottom" elements of the tree.
- Internal Node: A node that has at least one child (i.e., non-leaf nodes).
- Degree:
    - Degree of a Node: The total number of children a node has.
    - Degree of a Tree: The maximum degree of any node in the entire tree.

NOTE

- Level: The generation of the node. The Root is at Level 0. Its children are at Level 1, their children at Level 2, and so on.
- Height: The length of the longest path from a specific node to a leaf node.
    - *Height of the Tree* = Height of the Root node.
- Depth: The length of the path from the Root to a specific node.
- Subtree: Any node in the tree, along with all its descendants, forms a subtree.

8.3.3. General Trees vs. Binary Trees

While a "General Tree" can have any number of children (Degree N), in Computer Science, we focus heavily on the Binary Tree.

8.3.3.1 The Binary Tree

A Binary Tree is a specialized tree data structure where each node can have at most two children. These children are distinctively referred to as the:

1. Left Child
2. Right Child

Because the max degree is 2, Binary Trees are much easier to implement and manipulate in C++ compared to general trees.

8.3.3.2 Types of Binary Trees

1. Full Binary Tree: Every node has either 0 or 2 children. No node has just 1 child.
2. Complete Binary Tree: All levels are completely filled except possibly the last level, and the last level has all keys as left as possible. (This is essential for Heap data structures).
3. Perfect Binary Tree: All internal nodes have two children, and all leaf nodes are at the same level.
4. Skewed Binary Tree: A tree dominated by either left children or right children (effectively resembling a Linked List).

8.3.4. Representing Trees in C++

In C++, a tree is not a built-in data type; we must construct it using struct or class. Since trees are dynamic (nodes can be added or removed), we typically use pointers.

8.3.4.1 The Structure Definition

Here is the standard boilerplate code for a Tree Node in C++:

C++

NOTE

```
#include <iostream>
using namespace std;

// Definition of the Node
struct Node {
    int data;           // The data stored in the node
    Node* left;         // Pointer to the left child
    Node* right;        // Pointer to the right child

    // Constructor for easy node creation
    Node(int val) {
        data = val;
        left = nullptr;
        right = nullptr;
    }
};

int main() {
    // Creating the Root
    Node* root = new Node(10);

    // Creating Children
    root->left = new Node(20);
    root->right = new Node(30);

    // Linking a grandchild
    root->left->left = new Node(40);

    /* Visual Representation of the above code:
          10
         /  \
        20   30
```

NOTE

```
      /
     40
   */


   return 0;
}
```

Explanation of Code:

- We use a struct named Node.
- int data holds the integer value.
- Node* left and Node* right are self-referential pointers. They point to other objects of the same Node type.
- In main(), we use new to dynamically allocate memory in the Heap for these nodes.

8.3.5. Why do we need Trees? (Advantages)

Students often ask: *"Why not just use an Array?"*

1. Hierarchical Data: As mentioned, arrays cannot naturally represent hierarchies (like HTML DOM structures or Organization charts). Trees do this natively.
2. Efficient Search: In a sorted array, search is O(\log n) (Binary Search), but insertion is slow (O(n)) because you have to shift elements. In a Binary Search Tree (BST), both search and insertion can be optimized to O(\log n).
3. Flexible Size: Like Linked Lists, Trees can grow dynamically. You do not need to declare a fixed size in advance.
4. Specialized Operations: Trees enable operations like finding the "lowest common ancestor" or "shortest path" in networks, which are difficult in linear structures.

8.3.6. Applications of Trees

- File Systems: The folder structure in Windows/Linux/macOS is a tree.
- Compilers: Code is parsed into a "Syntax Tree" to understand grammar (e.g., verifying that parentheses match).
- Databases: Most databases use B-Trees or B+ Trees to index data for ultra-fast retrieval.

Networking: Routing algorithms use trees to find the most efficient path for data packets

8.4 REPRESENTATION OF TREES IN MEMORY

NOTE

8.4.1 Introduction

In the previous section, we visualized trees as circles (nodes) connected by lines (edges). However, computer memory is linear—it is essentially a long tape of byte addresses. It does not natively understand "left child" or "right child" or "hierarchy."

To bridge this gap, we need specific methods to map the hierarchical structure of a tree onto the linear memory of a computer. There are two primary ways to represent trees in C++:

1. Sequential Representation (using Arrays)
2. Linked Representation (using Pointers/Linked Lists)

8.4.2. Sequential Representation (Array-Based)

This method uses a standard one-dimensional array to store the tree nodes. It relies on a mathematical formula to determine the parent-child relationships based on the index of the array.

8.4.2.1 How it Works

Imagine we have a Binary Tree. We number the nodes level-by-level, starting from the Root (Level 0), going left to right.

- The Root node is stored at index 0.
- If a node is stored at index i:
    - Its Left Child is stored at index (2 \times i) + 1.
    - Its Right Child is stored at index (2 \times i) + 2.
    - Its Parent is stored at index \lfloor (i - 1) / 2 \rfloor.

8.4.2.2 Example

Consider a tree: Root A, Left Child B, Right Child C.

- Array Index 0: A (Root)
- Array Index 1: B (Left child of A \rightarrow 2(0)+1 = 1)
- Array Index 2: C (Right child of A \rightarrow 2(0)+2 = 2)

If B has a left child D, it would go to index 2(1)+1 = 3.

8.4.2.3 Advantages

1. Fast Access: Accessing a parent or a child is a simple mathematical operation (O(1)). No pointers are needed to traverse.
2. Efficiency for Complete Binary Trees: If the tree is a "Complete Binary Tree" (where all levels are filled), this method is

NOTE

extremely memory efficient because there are no empty gaps in the array. This is strictly used in Heap data structures (Heap Sort, Priority Queues).

8.4.2.4 Disadvantages

1. Wasted Space (The Sparsity Problem): This is the major drawback. If you have a "skewed" tree (e.g., a tree that only goes right: A -> B -> C -> D), you still have to reserve space for all the non-existent left children to maintain the index logic.
    - *Example:* A right-skewed tree of depth 10 might require an array size of 1024 (2^{10}), but only store 10 actual values.
2. Fixed Size: Arrays in C++ generally have a fixed size (unless using std::vector). If the tree grows beyond the array limit, you must resize and copy everything, which is expensive.

8.4.3. Linked Representation (Pointer-Based)

This is the most common method used in General Programming and BCA curriculums because it handles the dynamic nature of trees much better than arrays.

8.4.3.1 How it Works

Instead of calculating indices, we use the memory address to connect nodes. Each node is a distinct block of memory (allocated in the Heap) containing:

1. Data: The actual value (int, char, etc.).
2. Pointer to Left Child: Stores the address of the left node.
3. Pointer to Right Child: Stores the address of the right node.

If a node does not have a child, the corresponding pointer is set to NULL (or nullptr in modern C++).

8.4.3.2 C++ Implementation

This implementation uses a struct or class.

C++

```
#include <iostream>

using namespace std;


struct TreeNode {

   int data;

   TreeNode* left;  // Pointer to left child
```

NOTE

```
    TreeNode* right; // Pointer to right child

    // Constructor
    TreeNode(int value) {
        data = value;
        left = nullptr;
        right = nullptr;
    }
};

int main() {
    // 1. Create the Root
    TreeNode* root = new TreeNode(10);

    // Memory: [ Left: NULL | Data: 10 | Right: NULL ]

    // 2. Add Children
    root->left = new TreeNode(20);
    root->right = new TreeNode(30);

    // Memory: Root's 'left' now holds the address of the node 20.
    // Memory: Root's 'right' now holds the address of the node 30.

    return 0;
}
```

8.4.3.3 Advantages

1. Dynamic Size: Memory is allocated only when a new node is created. There is zero wasted space for empty slots.
2. Flexibility: It easily handles arbitrary shapes of trees (skewed, sparse, dense) without any modification to the logic.
3. Insertion/Deletion: Adding a node simply involves changing a pointer. You do not need to shift elements like in an array.

NOTE

8.4.3.4 Disadvantages

1. Pointer Overhead: Every node requires extra memory to store the two pointers (usually 4 or 8 bytes per pointer). For a tree of integers, the pointers might take up more space than the actual data!
2. No Direct Access: To reach a node, you must start from the Root and traverse pointers. You cannot jump directly to "Node #5".

8.4.4. Comparison Summary

This table is crucial for your exams to quickly differentiate between the two.

Feature	Sequential (Array)	Linked (Pointer)
Memory Allocation	Static (mostly)	Dynamic
Space Efficiency	Good for Complete Trees; Bad for Sparse Trees	Efficient for all tree types
Overhead	None (only data is stored)	High (pointers take extra space)
Access Speed	Random Access possible via index	Sequential Access (must traverse)
Implementation	Complex index calculations	Simple pointer manipulation
Best Use Case	Heaps, Priority Queues	General Binary Trees, BSTs, AVL Trees

8.4.5. Representation of General Trees (N-ary Trees)

Note: This is an advanced concept but important for a "detailed" understanding.

In a standard Binary Tree, we know we have exactly two pointers (Left/Right). But what if a node can have 5 or 10 children (a General Tree)? We cannot pre-define child1, child2, ... child10 inside the struct because we don't know how many children a node will have.

The "Left-Child, Right-Sibling" Representation

To represent a general tree using binary tree logic, we use a clever transformation:

1. Left Pointer: Points to the *first child* of the node.
2. Right Pointer: Points to the *next sibling* of the node.

NOTE

This allows us to represent a tree of any degree using a structure that looks exactly like a Binary Tree.

8.5 THE BINARY TREE AND ITS REPRESENTATION

8.5.1. The Binary Tree: Concept & Definition

In the previous unit, we discussed trees in general. Now, we narrow our focus to the Binary Tree, the workhorse of data structures. It is the foundation for Binary Search Trees (BST), Heaps, Expression Trees, and even modern compression algorithms like Huffman Coding.

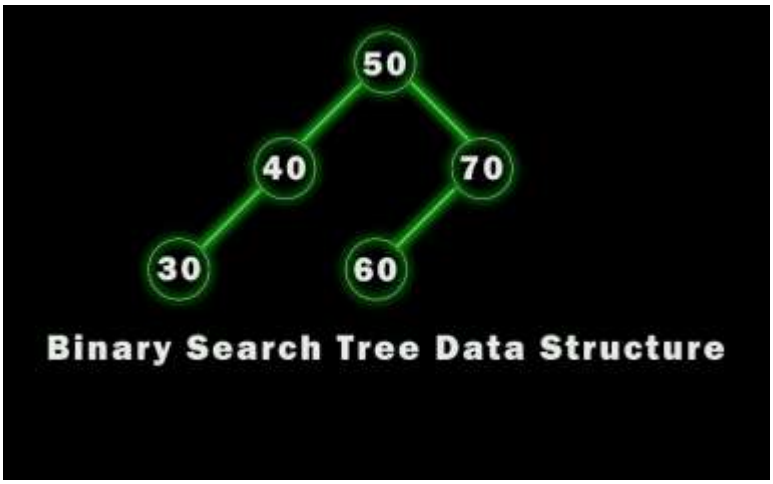
8.5.1.1 Definition

A Binary Tree is a hierarchical data structure in which each node has at most two children. This is a strict constraint. Unlike a general tree where a node might have 10 or 100 children, in a binary tree, the degree of any node can only be 0, 1, or 2.

These children are specifically distinguished as:

1. The Left Child
2. The Right Child

This distinction is crucial. A binary tree where the root has a *left* child A is structurally different from a binary tree where the root has a *right* child A, even if the data is identical.



8.5.1.2 Mathematical Properties

To understand binary trees deeply, you must understand their mathematical limits. These are often asked in BCA exams:

- Maximum Nodes at Level l: 2^l
    - Level 0 (Root): 2^0 = 1 node.
    - Level 1: 2^1 = 2 nodes.
    - Level 2: 2^2 = 4 nodes.

NOTE

- Maximum Nodes in a Tree of Height h: 2^{h+1} - 1
    - If a tree has a height of 3 (levels 0, 1, 2, 3), it can have at most 2^4 - 1 = 15 nodes.
- Minimum Height for N Nodes: \lfloor \log_2 N \rfloor
    - This property is why binary trees are efficient. We can pack a huge amount of data (N) into a very short height (\log N), allowing for fast search.

8.5.2. Array Representation of Binary Tree

(Also known as Sequential Representation or Linear Representation)

When we store a tree in an Array, we are essentially flattening the hierarchy. We rely on the indices of the array to remember the parent-child relationships implicitly, rather than using pointers.

8.5.2.2 The Storage Logic

We store the tree nodes level-by-level (Level Order), moving from left to right.

- Root: Stored at Index 0.
- Left Child: If a node is at index i, its left child is at index 2i + 1.
- Right Child: If a node is at index i, its right child is at index 2i + 2.
- Parent: If a node is at index i, its parent is at index (i - 1) / 2 (using integer division).

8.5.2.3 Worked Example

Imagine a Binary Tree:

- Root: A
- A has Left Child B and Right Child C.
- B has Right Child D (No left child).
- C has Left Child E.

Mapping to Array:

1. Index 0: A (Root)
2. Index 1: B (Left of A: 2*0 + 1)
3. Index 2: C (Right of A: 2*0 + 2)
4. Index 3: _ (Empty! B has no left child. 2*1 + 1 = 3. We leave this space blank.)
5. Index 4: D (Right of B: 2*1 + 2 = 4)
6. Index 5: E (Left of C: 2*2 + 1 = 5)

NOTE

Final Array: [A, B, C, NULL, D, E]

8.5.2.4 C++ Code Concept

C++

```
char tree[10] = {'A', 'B', 'C', '\0', 'D', 'E'};
// Function to print left child of node at index i
void printLeftChild(int i) {
    int leftIndex = 2 * i + 1;
    if (tree[leftIndex] != '\0')
        cout << tree[leftIndex];
    else
        cout << "No Left Child";
}
```

8.5.2.5 Pros and Cons

- Pros: Fast access to data. Excellent for Complete Binary Trees (Heaps). No space wasted on pointer variables.
- Cons: The Space Penalty. If the tree is "skewed" (e.g., a straight line of right children), the array representation is disastrous. To store a right-skewed tree of height 10 (just 11 nodes), you would need an array of size 2^{11} \approx 2048, leaving 2037 spaces empty.

8.5.3. Linked List Representation of Binary Tree

(Also known as Dynamic Node Representation)

This is the standard, most flexible way to represent trees in C++. Instead of mathematical indices, we use memory addresses. We model each element as a distinct object (Node) that points to its children.

8.5.3.1 The Node Structure

In a Doubly Linked List, a node has Prev and Next. Similarly, in a Binary Tree, a node has Left and Right.

Visualizing the Node:

Plaintext

```
+----------------------+
|       DATA           |  <--- The Value (e.g., 50)
+----------------------+
| Left Ptr | Right Ptr |  <--- The Addresses
+----------------------+
```

NOTE

- If a node is a Leaf Node, both Left Ptr and Right Ptr will be NULL.
- If a node has only a left child, Right Ptr is NULL.

8.5.3.2 C++ Implementation (The Standard Approach)

This is the code you will likely use in your practical exams.

C++

```
#include <iostream>
using namespace std;

// 1. Define the Blueprint
struct Node {
    int data;
    Node* left;
    Node* right;

    // Constructor creates a new node with data and null pointers
    Node(int val) {
        data = val;
        left = NULL;
        right = NULL;
    }
};
int main() {
    /* Constructing the Tree:
            1
          /   \
         2     3
    */
    // Step 1: Create the nodes dynamically in Heap memory
    Node* root = new Node(1);
    Node* p1 = new Node(2);
    Node* p2 = new Node(3);
    // Step 2: Link them together
```

NOTE

```
    root->left = p1;   // Connect 2 to the left of 1
    root->right = p2;  // Connect 3 to the right of 1
    return 0;
}
```

8.5.3.3 Pros and Cons

Pros:

True Dynamism: Memory is allocated exactly when needed. No wasted space for "empty" nodes.

Flexibility: It handles skewed trees, sparse trees, and dense trees with equal efficiency.

Easy Restructuring: Moving a subtree is as simple as changing one pointer address; in an array, you'd have to shift huge blocks of data.

Cons:

Pointer Overhead: We pay a "tax" for this flexibility. Every node needs extra memory for the two pointers (usually 8 bytes on a 32-bit system). For a tree storing small integers, the pointers might take up more RAM than the data itself!

No Random Access: You cannot calculate the address of Node #5. You must traverse from the Root to find it.

Check Your Progress 8.1

MCQ:
In a binary tree, each node can have at most how many children?
a) 1
b) 2
c) 3
d) Unlimited

True/False:
A tree with n nodes always has n-1 edges.

Short Questions:

1. Define a tree in data structures.

.....

.....

2. Name two types of trees commonly used in computer science.

.....

.....

NOTE

8.6 TYPES OF BINARY TREES & BASIC OPERATIONS

8.6.1. Types of Binary Trees

Not all binary trees are created equal. Depending on how the nodes are arranged and the restrictions placed on them, Binary Trees are classified into several specific types. Understanding these differences is crucial for analyzing complexity (Time and Space).

A. Full Binary Tree (Strictly Binary Tree)

A Binary Tree is called a Full Binary Tree if every node has either 0 or 2 children. No node in a Full Binary Tree can have exactly one child.

- Nodes: If a Full Binary Tree has I internal nodes, it will effectively have I + 1 leaf nodes.
- Structure: It looks "triangular" but might not be symmetric.

B. Complete Binary Tree

This is one of the most important types, used in Heaps. A Binary Tree is Complete if:

1. All levels are completely filled, except possibly the last level.
2. The last level is filled from left to right without any gaps.

- Significance: Because there are no gaps in the sequence, Complete Binary Trees can be perfectly represented using an Array, saving memory on pointers.

C. Perfect Binary Tree

This is the "ideal" tree. A Binary Tree is Perfect if:

1. All internal nodes have exactly 2 children.
2. All leaf nodes are at the same level (depth).

- Mathematical Property: A perfect binary tree of height h has exactly 2^{h+1} - 1 nodes. This is the maximum number of nodes possible for that height.

D. Degenerate (or Skewed) Binary Tree

A tree where every parent node has exactly one child. This effectively creates a linked list structure.

- Left-Skewed: Every node only has a left child.
- Right-Skewed: Every node only has a right child.
- Performance: This is the "worst-case scenario" for trees. Searching becomes O(n) (linear) instead of O(\log n).

8.6.2. Basic Operations on Binary Trees

NOTE

Just like with integers or arrays, we need to perform operations on our data structure to make it useful.

1. Insertion: Adding a new node to the tree. The location depends on the type of tree (e.g., in a Binary Search Tree, smaller values go left, larger go right).
2. Deletion: Removing a node. This is complex because we must reconnect the disjointed subtrees.
3. Searching: Finding if a specific value exists in the tree.
4. Height Calculation: Finding the number of edges from the root to the deepest leaf.
5. Traversal: Visiting every node in the tree exactly once.

8.6.3. Tree Traversals (Recursive Algorithms)

Traversal is the process of visiting every single node in the tree exactly once to perform an operation (like printing the data). Unlike arrays, which are linear (visit index 0, then 1, then 2), trees are hierarchical, so there is no single "correct" path.

We use Recursive Algorithms because trees are recursive structures (every subtree is itself a tree).

The three standard Depth-First Search (DFS) traversals are:

1. Pre-order (Root \rightarrow Left \rightarrow Right)
2. In-order (Left \rightarrow Root \rightarrow Right)
3. Post-order (Left \rightarrow Right \rightarrow Root)

A. Pre-order Traversal

Strategy: Root First.

1. Visit the Root.
2. Traverse the Left Subtree (recursively).
3. Traverse the Right Subtree (recursively).

Use Case: Used to create a copy of the tree. It preserves the structural hierarchy.

C++ Implementation:

C++

```
void printPreorder(Node* node) {
   if (node == NULL)
      return; // Base case: If tree is empty, stop.
```

NOTE

```
    // 1. Visit the Root
    cout << node->data << " ";

    // 2. Go Left
    printPreorder(node->left);

    // 3. Go Right
    printPreorder(node->right);
}
```

B. In-order Traversal

Strategy: Root in the Middle.

1. Traverse the Left Subtree (recursively).
2. Visit the Root.
3. Traverse the Right Subtree (recursively).

Use Case: This is the most famous traversal. If you perform In-order traversal on a Binary Search Tree (BST), it prints the values in sorted (ascending) order.

C++ Implementation:

C++

```
void printInorder(Node* node) {
    if (node == NULL)
        return;
    // 1. Go Left
    printInorder(node->left);
    // 2. Visit the Root
    cout << node->data << " ";
    // 3. Go Right
    printInorder(node->right);
}
```

C. Post-order Traversal

Strategy: Root Last.

1. Traverse the Left Subtree (recursively).
2. Traverse the Right Subtree (recursively).

NOTE

3. Visit the Root.

Use Case: Used for Deleting a tree. You must delete the children before you delete the parent (otherwise you lose the pointers to the children). Also used in evaluating mathematical expressions (Postfix notation).

C++ Implementation:

C++

```
void printPostorder(Node* node) {
    if (node == NULL)
        return;
    // 1. Go Left
    printPostorder(node->left);
    // 2. Go Right
    printPostorder(node->right);
    // 3. Visit the Root
    cout << node->data << " ";
}
```

Activity 1: The "Corporate Hierarchy" – Building a Tree Structure

Objective: To introduce the Tree data structure by modeling a real-world hierarchical system: a company's organizational chart. Students will move beyond linear data structures (arrays, lists) to understand parent-child relationships, root nodes, and leaf nodes.

Materials / Tools / Software Required:

- Computer: Standard development setup.
- Programming Language: Java, C++, or Python.
- IDE: Eclipse, VS Code, or PyCharm.
- Diagramming Tool (Optional): Draw.io or Lucidchart for the initial design phase.

Expected Outcome: By the end of this activity, students will be able to:

1. Define Tree Terminology: Identify the Root (CEO), Internal Nodes (Managers), and Leaves (Employees with no reports).
2. Implement the Node Class: Create a class structure where each node contains data (Employee Name/Role) and a list/array of references to children nodes.
3. Construct the Tree: Manually instantiate nodes and link them to build a hierarchy (e.g., assign two "Team Leads" to the "CTO").
4. Traverse the Tree: Write a recursive function to print the entire hierarchy in a structured format (using indentation to represent depth/levels).

NOTE

8.7 SUMMARY

1. In this unit, we introduced the concept of the Tree, a non-linear hierarchical data structure. Unlike arrays or linked lists where data is linear, trees organize data in a parent-child relationship. We established that the top node is called the Root, and nodes with no children are called Leaves. The connection between two nodes is an Edge.

2. We focused heavily on the Binary Tree, a specialized tree where every node has at most two children: a Left Child and a Right Child. We categorized Binary Trees into specific types:

- Full Binary Tree: Every node has 0 or 2 children.
- Complete Binary Tree: All levels are filled, used efficiently in Heaps.
- Skewed Binary Tree: Resembles a linked list and offers poor performance.

3. We explored two primary ways to represent trees in memory:

1. Sequential Representation (Arrays): Uses mathematical indices (2i+1, 2i+2) to locate children. It is fast but wastes huge amounts of memory for sparse or skewed trees.
2. Linked Representation (Pointers): Uses a struct with pointers to left and right children. This is the standard approach in C++ as it is dynamic and flexible, despite the overhead of storing pointers.

4. Finally, we covered Tree Traversals, the algorithms used to visit every node in the tree. We learned that since trees are non-linear, we cannot simply loop through them. Instead, we use recursion.

- Pre-order (Root-Left-Right): Useful for copying trees.
- In-order (Left-Root-Right): Produces sorted output in Binary Search Trees.
- Post-order (Left-Right-Root): Useful for deleting trees or evaluating postfix expressions.

8.8 GLOSSARY

1. Root: The topmost node in a tree hierarchy; the only node without a parent.
2. Leaf Node: A node that has no children (degree 0); also called an external node.

NOTE

3. Binary Tree: A tree data structure where each node has at most two children (left and right).
4. Height of Tree: The number of edges on the longest path from the root to a leaf node.
5. Complete Binary Tree: A binary tree where all levels are completely filled, except possibly the last, which is filled from left to right.
6. Sequential Representation: Storing a tree in an array where relationships are determined by index formulas.
7. Linked Representation: Storing a tree using dynamic memory allocation where nodes point to their children.
8. Traversal: The process of visiting every node in a tree data structure exactly once.
9. In-order Traversal: A traversal method following the sequence: Left Subtree \rightarrow Root \rightarrow Right Subtree.
10. Recursion: A programming technique where a function calls itself; essential for tree operations

8.9 ANSWER TO CHECK YOUR PROGRESS-POSSIBLE ONE

Check Your Progress 8.1

MCQ Answer:
b) 2 – each node in a binary tree can have at most two children.

True/False Answer:
True – a tree with n nodes has exactly n-1 edges.

Short Answers:

1. A tree is a hierarchical data structure with nodes connected by edges, having one root node and zero or more child nodes.
2. Common types: Binary Tree, Binary Search Tree (BST).

8.10 SUGGESTED READINGS – OER

Sr. No.	Title	URL	QR Code
1	Data Structure Using C++	http://books.aisectonline.com/User/UnitUrls?unitId=3716	

NOTE

8.11 REFERENCE

1. Data Structures and Algorithms in C++ – Michael T. Goodrich, Roberto Tamassia, David M. Mount – ISBN: 9780470383278 – 2011
2. Data Structures and Algorithm Analysis in C++ – Mark Allen Weiss – ISBN: 9780132847377 – 2013
3. Data Structures and Algorithms in C++ – Adam Drozdek – ISBN: 9781133608426 – 2012
4. Data Structures Using C++ – D. S. Malik – ISBN: 9788131518236 – 2012
5. Data Structures Through C++ – Yashavant Kanetkar – ISBN: 9789355515834 – 2024
6. Introduction to Algorithms – Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein – ISBN: 9780262046305 – 2022 (4th Edition)
7. Data Structures: A Pseudocode Approach with C – Richard F. Gilberg, Behrouz A. Forouzan – ISBN: 9780534953965 – 2005
8. Fundamentals of Data Structures in C++ – Ellis Horowitz, Sartaj Sahni, Dinesh Mehta – ISBN: 9788173716065 – 2008

8.12 EXERCISE

Part 1

B. Multiple choice question

Q1. What is the maximum number of children a node can have in a Binary Tree?

A) 0

B) 1

C) 2

D) Any number

Q2. In an Array representation of a Binary Tree, if a parent node is at index i, where is its Left Child stored?

A) 2i

B) 2i + 1

C) 2i + 2

D) i / 2

Q3. Which traversal technique visits the Root node last?

A) In-order

B) Pre-order

NOTE

C) Post-order

D) Level-order

Q4. A Binary Tree where every node has either 0 or 2 children is called a:

A) Complete Binary Tree

B) Full Binary Tree

C) Perfect Binary Tree

D) Skewed Binary Tree

Q5. What is the main disadvantage of using Array representation for a Skewed Binary Tree?

A) It is difficult to implement.

B) It is slower than Linked representation.

C) It wastes a significant amount of memory space.

D) It cannot handle integer data.

Answers with Explanations:

1. Answer: C) 2
Explanation: By definition, each node in a binary tree can have at most two children — a left and a right child.

2. Answer: B) 2i + 1
Explanation: In array representation, left child of node at index i is stored at 2i + 1 and right child at 2i + 2.

3. Answer: C) Post-order
Explanation: Post-order traversal visits the left subtree, then right subtree, and finally the root node.

4. Answer: B) Full Binary Tree
Explanation: A full binary tree is defined as one in which every node has either zero or two children.

5. Answer: C) It wastes a significant amount of memory space.
Explanation: In a skewed binary tree, array representation leads to many unused indices, resulting in wasted memory.

B. Small answer type question

1. Explain the difference between a General Tree and a Binary Tree with a suitable diagram.
2. Define Height and Depth of a tree. How are they calculated?
3. Why the Linked Representation of a binary tree is generally preferred over the Sequential Representation in C++?

NOTE

4. Write the recursive algorithm (steps) for In-order Traversal.
5. What is a Skewed Binary Tree? Why is it considered inefficient for searching operations?

Part 2

C. Long answer type question (answer in 200 words)

1. Discuss the Sequential (Array) Representation of a Binary Tree. Explain the formulas used to find children and parents. What are the limitations of this method when the tree is not complete?
2. Differentiate between Full Binary Tree, Complete Binary Tree, and Perfect Binary Tree. Provide a small visual example for each.
3. Write a C++ program (using a struct) to create a simple Binary Tree with a Root and two children. Explain the role of the new keyword in your code.
4. Compare and contrast Linear Data Structures (like Arrays/Linked Lists) with Non-Linear Data Structures (like Trees). Why do we need Trees?
5. Explain the concept of Tree Traversal. Why do we need three different types of Depth-First traversals (Pre, In, Post)? Give one practical application for each.

Part 3

D. Long answer type question (answer in 300 words)

1. Write a complete C++ program to insert nodes into a binary tree and perform an In-order Traversal. Provide a "Dry Run" of your code using the example values: 10, 20, 30, 40, 50.
2. "A Binary Tree is a recursive data structure." Justify this statement. Explain how a Binary Tree is stored in the computer's memory using a Linked List, detailing the structure of the Node and how pointers connect the hierarchy.
3. Consider the following algebraic expression: (A + B) * (C - D). Draw the Expression Tree for this equation. Then, perform Pre-order, In-order, and Post-order traversals on it. What do the resulting sequences represent in terms of Polish Notation?
4. Create a comparative chart between Array Representation and Linked Representation of Binary Trees. Discuss factors such as Memory Allocation, Access Time, Insertion complexity, and suitability for different types of trees (Complete vs. Sparse).
5. Explain the recursive logic behind Pre-order Traversal. Write the C++ function for it. Then, trace the execution stack (step-by-step) for a tree with 3 levels to show how the computer handles the recursion internally.

DR. C.V. RAMAN UNIVERSITY

Chhattisgarh, Bilaspur AN AISECT GROUP UNIVERSITY

Approved by : PCI | AICTE | NCTE | BCI | Member of : AIU | Joint Committee : (UGC | DEB | AICTE) | Recognized by : UGC | A NAAC Accredited University

DR. C. V. RAMAN UNIVERSITY (C.G.)

SLM FEEDBACK FORM

PROGRAMME : _____

COURSE : _____

BLOCK : _____

UNIT : _____

Dear Student,

While studying the units of this block, you may have found certain portions of the text difficult to comprehend. We wish to know your difficulties and suggestions, in order to improve the course. Therefore, we request you to fill up and send us the following questionnaire, which pertains to this block. If you find the space provided insufficient, kindly use a separate sheet.

Please mail to aisectcontent@aisect.org/ for online submission please scan QR code:

Questionnaire

1. **Name** : _____

2. **Enrolment No.** : | | | | | | | | | | | |

3. **How many hours did you need for studying the units?** : | |

4. **Please give your reactions to the following items based on your reading of the block:**

Items	Excellent	Very Good	Good	Poor	Give specific Examples (If poor)
Presentation Quality					
Language and Style					
Illustrations					
Used (diagrams, tables, etc.)					
Conceptual Clarity					
Check Your Progress Questions					
Suggested Readings - OER					

5. **Any other comments:**

_____

_____

_____

_____

***

BLOCK III	**ADVANCED STRUCTURES AND ALGORITHMIC TECHNIQUES**	**169-286**
Unit 9	Binary Search Tree (BST) and Advanced Trees	171
Unit 10	Introduction to Graphs	195
Unit 11	Graph Algorithms and Traversals	213
Unit 12	Introduction to Sorting and Basic Sorting Techniques	235
Unit 13	Advanced Sorting Techniques and Comparison	257

This block develops students' knowledge of advanced data structures and algorithmic techniques used for solving complex computational problems. It builds upon linear, linked, and hierarchical structures by introducing structures such as graphs and other advanced organizational methods. Students will explore graph terminology, representations, traversal techniques, and important applications. The block also emphasizes fundamental algorithmic strategies, including sorting, recursion, and techniques for improving the efficiency of data processing. Different approaches to organizing and manipulating data are examined with attention to their computational requirements and practical applications. Students will learn how algorithms interact with data structures and how appropriate combinations can improve program performance. The concepts presented in this block encourage systematic problem-solving and analytical thinking by requiring learners to evaluate alternative methods for a given computational task. Overall, the block prepares students to apply advanced data structures and algorithmic techniques in programming, software development, and real-world computational applications.

UNIT 9
BINARY SEARCH TREE (BST) AND ADVANCED TREES

NOTE

UNIT STRUCTURE

9.1 Introduction

9.2 Objective

9.3 The Binary Search Tree (BST)

9.4 Creation of a Binary Search Tree (BST)

9.5 Operations on Binary Search Trees (BST)

9.6 Threaded Binary Trees & Height Balanced Trees

9.7 B-Trees & Applications

9.8 Summary

9.9 Glossary

9.10 Answer to check your progress

9.11 Suggested readings - OER

9.12 Reference

9.13 Exercise

9.1 INTRODUCTION

Tree data structures represent a significant leap in computer science, moving beyond linear storage methods like arrays and linked lists into hierarchical organizations of data. At the core of this unit is the Binary Search Tree (BST), a fundamental structure designed to optimize data retrieval. Unlike standard binary trees, a BST enforces a specific ordering property where every node's left child is strictly smaller and the right child is strictly larger than the parent. This "sorted invariant" allows for powerful efficiency, enabling search, insertion, and deletion operations to ideally run in O(\log n) time—cutting the search space in half with every step.

However, the unit also acknowledges the limitations of a standard BST. In worst-case scenarios, such as inserting sorted data, a BST can degenerate into a "skewed" tree, performing no better than a linked list with O(n) complexity. To address this, we explore advanced variations. We introduce Height Balanced Trees (AVL Trees), which use rotations to self-correct and guarantee logarithmic speed. We examine Threaded

NOTE

Binary Trees, which ingeniously utilize wasted null pointers to eliminate the need for recursion stacks during traversal. Finally, we discuss B-Trees, the industry standard for database and file system storage, optimized to minimize slow disk accesses by maintaining a wide, shallow structure

9.2 OBJECTIVE

After Completing this unit you will be able to:

1. Define and Construct a Binary Search Tree (BST) from a given dataset, ensuring the correct logical ordering of nodes (Left < Parent < Right).
2. Perform and Analyze core BST operations including Searching, Insertion, and the complex three-case Deletion logic.
3. Explain and Differentiate between various tree traversals (In-Order, Pre-Order, Post-Order) and their specific utility in processing data.
4. Implement and Evaluate advanced tree optimizations, specifically how AVL Trees maintain balance via rotations and how Threaded Trees optimize traversal memory.
5. Apply tree concepts to real-world scenarios, selecting the appropriate structure (BST, AVL, or B-Tree) for specific problems like database indexing or memory-constrained environments.

9.3 THE BINARY SEARCH TREE (BST)

A Fundamental Data Structure for Efficient Lookup

A Binary Search Tree (BST) is a node-based binary tree data structure which has the following properties:

1. The left subtree of a node contains only nodes with keys less than the node's key.
2. The right subtree of a node contains only nodes with keys greater than the node's key.
3. The left and right subtrees must also be binary search trees.

This structure allows for fast lookup, addition, and removal of items, and can be used to implement dynamic sets and lookup tables.

9.3.1 Anatomy of a BST

To understand a BST, we must first understand its building blocks.

- Node: The fundamental unit containing data (the Key) and pointers to its children.

NOTE

- Root: The topmost node of the tree (it has no parent).
- Edge: The link connecting two nodes.
- Leaf: A node with no children (both left and right pointers are null).
- Height: The number of edges on the longest path from the node to a leaf.
- Depth: The number of edges from the root to the node.

9.3.1.1 The Ordering Property

The most crucial aspect of a BST is its sorted invariant. For any given node N:

\text{All Keys in } N.left < N.key < \text{All Keys in } N.right

Note: Handling duplicates varies by implementation. Some BSTs strictly forbid duplicates, while others allow them (usually placing them consistently in either the left or right subtree).

9.3.2 Core Operations

The efficiency of a BST comes from its operations, which generally run in O(h) time, where h is the height of the tree.

A. Searching

Searching is similar to a binary search on a sorted array. To search for a value k:

1. Start at the Root.
2. If the tree is null, the value is not found.
3. If the root's value equals k, you have found the target.
4. If k is less than the root's value, move to the Left Child.
5. If k is greater than the root's value, move to the Right Child.
6. Repeat recursively.

B. Insertion

Insertion is essentially a "Search" that stops when it hits a dead end (NULL), and plants a new node there.

1. Compare the value to be inserted with the current node.
2. If the value is smaller, traverse left; if larger, traverse right.
3. Repeat until you reach a spot where the child is NULL.
4. Insert the new node in that empty spot.

Example: To insert 45 into a tree with root 50:

- 45 < 50, go Left.
- (Assume left child is 30) 45 > 30, go Right.
- (Assume right child is 40) 45 > 40, go Right.

NOTE

- (Assume right child is NULL) Insert 45 here.

C. Deletion

Deletion is the most complex operation because we must maintain the BST property after removing a node. There are three scenarios:

Case 1: Node is a Leaf

Simply remove the node. The parent's pointer to this node becomes NULL.

Case 2: Node has One Child

Copy the child to the node and delete the child. Effectively, the child "moves up" to take the parent's place.

Case 3: Node has Two Children

This is the tricky part. You cannot simply pull the node out.

1. Find the In-Order Successor of the node you want to delete.
    - The In-Order Successor is the smallest value in the right subtree.
2. Replace the value of the node to be deleted with the successor's value.
3. Delete the successor node (which will be a Case 1 or Case 2 deletion, as the successor cannot have a left child).

9.3.3 Complexity Analysis

The performance of a BST relies heavily on its Height (h).

- Average Case (O(\log n)): Occurs when the tree is "balanced" (roughly equal nodes on left and right).
- Worst Case (O(n)): Occurs when the tree becomes "skewed" (essentially a linked list). This happens if you insert sorted data (e.g., 1, 2, 3, 4, 5) into a naive BST.

Space Complexity

- O(n) to store n nodes.
- O(h) for the recursion stack during traversal/search.

9.4 CREATION OF A BINARY SEARCH TREE (BST)

Creating a BST typically involves taking a collection of data (like an array or list) and inserting elements one by one into the tree structure. The position of every new element is determined by the BST Property:

Left Child < Parent < Right Child

There are two primary ways to create a BST:

1. Naive Creation: Inserting elements from an unsorted list one by one.

2. Balanced Creation: Creating a tree from a sorted list to ensure minimal height.

NOTE

9.4.1. Naive Creation (From Unsorted Data)

This is the standard method. You start with an empty tree and iterate through your data set, calling the insert() function for each item.

9.4.1.1 The Algorithm

1. First Element: The first item in your list becomes the Root.
2. Next Element: Compare the new item with the Root.
    - If smaller \rightarrow Go Left.
    - If larger \rightarrow Go Right.
3. Repeat Downwards: Continue comparing with subsequent child nodes until you hit a NULL (empty) spot.
4. Place Node: Insert the new item in that empty spot.
5. Repeat: Do this for every item in the list.

9.4.1.2 Step-by-Step Walkthrough

Let's build a BST from this array: [45, 10, 7, 90, 12, 50]

Step 1: Insert 45

- Tree is empty.
- 45 becomes the Root.

Step 2: Insert 10

- Compare 10 with 45.
- 10 < 45. Move Left.
- Left is empty. Insert 10 as left child of 45.

Step 3: Insert 7

- Compare 7 with 45 (7 < 45). Move Left.
- Compare 7 with 10 (7 < 10). Move Left.
- Left is empty. Insert 7 as left child of 10.

Step 4: Insert 90

- Compare 90 with 45 (90 > 45). Move Right.
- Right is empty. Insert 90 as right child of 45.

Step 5: Insert 12

- Compare 12 with 45 (12 < 45). Move Left.
- Compare 12 with 10 (12 > 10). Move Right.

NOTE

- Right is empty. Insert 12 as right child of 10.

Step 6: Insert 50

- Compare 50 with 45 (50 > 45). Move Right.
- Compare 50 with 90 (50 < 90). Move Left.
- Left is empty. Insert 50 as left child of 90.

9.4.1.3 Resulting Structure

After inserting all elements, the logical structure looks like this:

Plaintext

```
     45
    /  \
   10    90
  /  \   /
 7   12 50
```

9.4.2. Complexity of Creation

The time it takes to build a tree depends heavily on the initial order of the data.

9.4.2.1 Average Case: O(N \log N)

If the data is random, the tree tends to be reasonably balanced.

- We insert N items.
- Each insertion takes roughly O(\log N) time (traversing the height).
- Total: N \times \log N.

9.4.2.2 Worst Case: O(N^2) (The "Skewed" Problem)

If the input array is already sorted (e.g., [1, 2, 3, 4, 5]), the tree will not branch.

- 1 becomes Root.
- 2 goes to Right of 1.
- 3 goes to Right of 2.
- 4 goes to Right of 3.

This creates a "Skewed Tree" (effectively a Linked List). Searching this tree becomes as slow as searching a list (O(N)).

9.4.3. Creating a Balanced BST from a Sorted Array

NOTE

To prevent the "Skewed Tree" problem when you have sorted data, you must choose the insertion order carefully. You cannot simply iterate from start to finish.

The Strategy:

To get a balanced tree, the Root must be the Median (middle element) of the data.

9.4.3.1 The Algorithm (Divide and Conquer)

1. Find the Middle element of the array. Make this the Root.
2. Take the Left Sub-array (elements before middle). Find its middle and make it the Left Child.
3. Take the Right Sub-array (elements after middle). Find its middle and make it the Right Child.
4. Recursively repeat until no elements remain.

Example: [10, 20, 30, 40, 50, 60, 70]

1. Mid is 40. It becomes Root.
2. Left slice is [10, 20, 30]. Mid is 20. Becomes Left Child of 40.
3. Right slice is [50, 60, 70]. Mid is 60. Becomes Right Child of 40.
4. Recursively handle [10], [30], [50], [70]...

This ensures the tree height remains minimized (O(\log N)).

9.4.4. Code Implementation (Python)

Here is a script that demonstrates both methods: creating a tree one-by-one, and creating a perfect tree from a sorted list.

Python

```
class Node:
    def __init__(self, key):
        self.val = key
        self.left = None
        self.right = None


# --- Method 1: Naive Insertion ---
def insert(root, key):
    if root is None:
        return Node(key)
    if key < root.val:
```

NOTE

```
        root.left = insert(root.left, key)
    else:
        root.right = insert(root.right, key)
    return root

def create_bst_naive(arr):
    if not arr: return None
    root = Node(arr[0])
    for i in range(1, len(arr)):
        insert(root, arr[i])
    return root

# --- Method 2: Sorted Array to Balanced BST ---
def sorted_array_to_bst(arr):
    if not arr:
        return None

    # Find middle index
    mid = len(arr) // 2

    # Make middle element the root
    root = Node(arr[mid])

    # Recursively build left and right subtrees
    root.left = sorted_array_to_bst(arr[:mid])
    root.right = sorted_array_to_bst(arr[mid+1:])

    return root

# --- Helper to print tree ---
def in_order_print(root):
    if root:
```

NOTE

```
    in_order_print(root.left)
    print(root.val, end=' ')
    in_order_print(root.right)

# Test
data = [45, 10, 7, 90, 12, 50]
root = create_bst_naive(data)
print("Tree created (In-Order):")
in_order_print(root)
# Output will be sorted: 7 10 12 45 50 90
```

9.5 OPERATIONS ON BINARY SEARCH TREES (BST)

The true power of a Binary Search Tree lies in its operations. Because the tree is sorted, we can perform operations much faster than in an unsorted array or linked list—typically skipping half the remaining data at every step.

9.5.1. Searching

Searching is the process of finding a specific node with a given key in the BST. It is very similar to the binary search algorithm used on sorted arrays.

9.5.1.1 The Logic

1. Start at the Root.
2. Compare the target value (k) with the current node's value (val).
    - Case A: If root is NULL, the value does not exist in the tree. Return False or NULL.
    - Case B: If k == val, you have found the node. Return the node.
    - Case C: If k < val, the target must be in the left subtree. Move Left.
    - Case D: If k > val, the target must be in the right subtree. Move Right.
3. Repeat recursively (or iteratively) until found or the subtree is empty.

9.5.1.2 Algorithm (Recursive)

Python

```
def search(root, key):
```

NOTE

```
# Base Cases: root is null or key is present at root
if root is None or root.val == key:
    return root

# Key is greater than root's key -> Search Right
if root.val < key:
    return search(root.right, key)

# Key is smaller than root's key -> Search Left
return search(root.left, key)
```

9.5.2. Insertion

Insertion is essentially a "failed search." You search for the value you want to add. When you hit a NULL (a dead end) where the node should be, you create a new node and place it there.

9.5.2.1 The Logic

1. Start at the Root.
2. Compare the new key (k) with the current node (val).
    - If k < val: Go Left.
        - If the left child is NULL, insert the new node here.
    - If k > val: Go Right.
        - If the right child is NULL, insert the new node here.
3. Note: If k == val, typically nothing happens (duplicates are usually ignored or handled by a counter).

9.5.2.2 Step-by-Step Example

Insert 40 into the tree below:

Plaintext

```
  50
 /  \
30   70
```

1. Start at 50. 40 < 50, so go Left.
2. Arrive at 30. 40 > 30, so go Right.
3. Right child of 30 is NULL.
4. Insert 40 as the right child of 30.

NOTE

9.5.3. Deletion

Deletion is the most complex operation because removing a node might break the tree's structure (leaving "holes" or detached branches). We must reconnect the tree while maintaining the BST sorting property.

There are three distinct cases depending on how many children the target node has.

Case 1: Node is a Leaf (No Children)

This is the simplest case.

- Action: Simply remove the node.
- Implementation: Set the parent's pointer to this node to NULL.

Case 2: Node has One Child

The node acts as a single link in a chain.

- Action: Bypass the node. Link the parent of the deleted node directly to the single child.
- Analogy: If you remove a middle link in a chain, you connect the previous link to the next one.

Case 3: Node has Two Children

This is the tricky case. You cannot just delete the node because it has two sub-branches attached to it.

- Action:
    1. Find the In-Order Successor of the node.
        - Successor = The smallest value in the Right Subtree (go Right once, then go Left as far as possible).
    2. Copy the Successor's value into the node you want to delete.
    3. Delete the Successor node (which is now a duplicate).
        - Note: The successor is guaranteed to have either 0 or 1 child (never 2), so deleting it is easy (Case 1 or 2).

Step-by-Step Example (Case 3)

Delete 50 (Root) from this tree:

Plaintext

```
    50
   /  \
  30   70
```

NOTE

```
 / \
60  80
```

1. Target is 50. It has two children (30 and 70).
2. Find Successor (smallest in right subtree):
    - Go Right to 70.
    - Go Left to 60. (60 is the successor).
3. Replace 50 with 60.

   Plaintext

```
    60
   / \
  30  70
     / \
    60  80 <-- Duplicate 60 here
```

4. Delete the original successor node (the 60 at the bottom).
    - The 60 at the bottom is a leaf (Case 1). Just remove it.

9.5.4. Python Implementation (Full Operations)

Python

```
class Node:
    def __init__(self, key):
        self.val = key
        self.left = None
        self.right = None

# --- SEARCH ---
def search(root, key):
    if root is None or root.val == key:
        return root
    if root.val < key:
        return search(root.right, key)
    return search(root.left, key)

# --- INSERT ---
```

NOTE

```
def insert(root, key):
    if root is None:
        return Node(key)

    if key < root.val:
        root.left = insert(root.left, key)
    else:
        root.right = insert(root.right, key)
    return root

# --- DELETE ---
def deleteNode(root, key):
    if root is None:
        return root

    # 1. Find the node
    if key < root.val:
        root.left = deleteNode(root.left, key)
    elif key > root.val:
        root.right = deleteNode(root.right, key)
    else:
        # 2. Node found. Handle the 3 cases:

        # Case 1 & 2: No child or One child
        if root.left is None:
            return root.right
        elif root.right is None:
            return root.left

        # Case 3: Two children
        # Find inorder successor (smallest in right subtree)
        temp = minValueNode(root.right)
```

NOTE

```
        # Copy content
        root.val = temp.val

        # Delete the successor
        root.right = deleteNode(root.right, temp.val)

    return root

def minValueNode(node):
    current = node
    while current.left is not None:
        current = current.left
    return current
```

Check Your Progress 9.1

MCQ:
In a Binary Search Tree (BST), for any node, all values in the left subtree are _____ the node, and all values in the right subtree are _____ the node.
a) Greater, Smaller
b) Smaller, Greater
c) Equal, Greater
d) Smaller, Equal

True/False:
An AVL tree is a self-balancing binary search tree.

Short Questions:

1. Define a Binary Search Tree (BST).

.....

.....

2. Name two types of advanced trees and their special property.

.....

.....

9.6 THREADED BINARY TREES & HEIGHT BALANCED TREES

NOTE

9.6.1. Threaded Binary Trees

In a standard Binary Tree, nearly half of the pointers are NULL (specifically, leaf nodes have null left and right pointers). This is a waste of memory and requires a stack or recursion to traverse the tree. Threaded Binary Trees solve this by utilizing those NULL pointers to store references to other nodes, facilitating efficient traversal without a stack.

9.6.1.1 The "Thread" Concept

Instead of being NULL, a pointer is made to point to an ancestor node in the traversal order.

- Left NULL pointer: Points to the In-Order Predecessor (the node visited immediately before this one).
- Right NULL pointer: Points to the In-Order Successor (the node visited immediately after this one).

9.6.1.2 Types of Threaded Trees

1. Single Threaded: Only the Right null pointers are used to point to the in-order successor.
2. Double Threaded: Both Left and Right null pointers are utilized (Left \rightarrow Predecessor, Right \rightarrow Successor).

9.6.1.3 Node Structure

To distinguish between a "real" child pointer and a "thread," we add boolean flags (tags) to the node:

- LTag: 0 if left points to a child; 1 if left is a thread.
- RTag: 0 if right points to a child; 1 if right is a thread.

Advantages:

- Traversal: Linear traversal is faster; no recursion or auxiliary stack needed.
- Memory: effectively utilizes wasted space.

9.6.2. Height Balanced Trees (AVL Trees)

A Height Balanced Tree, most commonly implemented as an AVL Tree (Adelson-Velsky and Landis), is a self-balancing Binary Search Tree. It solves the problem of "skewing," where a standard BST can degenerate into a linked list (O(n) search time) if data is inserted in sorted order.

NOTE

9.6.2.1 The Balance Property

In an AVL tree, for every node, the difference between the height of the left subtree and the right subtree must be at most 1.

| \text{Height}(Left) - \text{Height}(Right) | \le 1

This difference is called the Balance Factor (BF).

- BF = -1, 0, or 1: The node is balanced.
- BF < -1 or > 1: The node is unbalanced, and the tree must be restructured.

9.6.2.2 Rebalancing: Rotations

When a node becomes unbalanced after insertion or deletion, we perform Rotations to restore balance. There are four cases:

1. Left-Left (LL) Case: Insertion into the left subtree of the left child.
    - Fix: Right Rotation.
2. Right-Right (RR) Case: Insertion into the right subtree of the right child.
    - Fix: Left Rotation.
3. Left-Right (LR) Case: Insertion into the right subtree of the left child.
    - Fix: Left Rotation on the child, then Right Rotation on the current node.
4. Right-Left (RL) Case: Insertion into the left subtree of the right child.
    - Fix: Right Rotation on the child, then Left Rotation on the current node.

Complexity:

Because the height is strictly controlled to \approx 1.44 \log_2 n, all operations (Search, Insert, Delete) are guaranteed to be O(\log n).

9.7 B-TREES & APPLICATIONS

9.7.1 B-Trees

While AVL trees are great for memory, B-Trees are optimized for disk storage (databases and filesystems). Accessing data on a hard drive is slow; B-Trees minimize the number of disk accesses by making the tree "fat" and short rather than tall and thin.

NOTE

9.7.1.1 Definition

A B-Tree of order m is a self-balancing tree with the following properties:

1. Multi-way: Each node can have more than two children (up to m).
2. Multiple Keys: Each node contains multiple keys (up to m-1), stored in sorted order.
3. Leaf Depth: All leaf nodes must be at the same depth (perfectly balanced).
4. Occupancy: Every node (except root) must be at least half full (\lceil m/2 \rceil - 1 keys).

9.7.1.2 How it Works

- Searching: Similar to a BST, but within each node, we perform a linear or binary search on the keys to decide which child pointer to follow.
- Insertion (Splitting): New keys are added to leaf nodes. If a node becomes full (has m keys), it splits into two, and the middle key moves up to the parent. This helps the tree grow upwards, keeping it balanced.
- Deletion (Merging): If a node becomes too empty, it can borrow keys from siblings or merge with a sibling.

Why use B-Trees?

A B-Tree node usually matches the size of a disk block (e.g., 4KB). This means one disk read fetches hundreds of keys at once. A tree with millions of items might only be 3 or 4 levels deep, requiring only 3-4 disk seeks to find any data.

9.7.2 Applications of Binary Trees

Binary trees are not just theoretical concepts; they power many critical systems in computing.

A. Expression Evaluation (Syntax Trees)

Compilers use Expression Trees to parse mathematical formulas and code logic.

- Leaves: Operands (variables like a, b or constants like 5).
- Internal Nodes: Operators (+, -, *).
- Example: Traversing the tree in Post-Order allows the computer to evaluate complex equations like (a + b) * (c - d).

B. Data Compression (Huffman Coding)

NOTE

Huffman Trees are used in compression formats like ZIP, JPEG, and MP3.

- Frequent characters (like 'e' or 'a') are placed near the root (short binary codes).
- Rare characters (like 'z' or 'q') are placed deep in the tree (longer binary codes).
- This minimizes the total number of bits required to store a file.

C. Database Indexing

While B-Trees are standard, variations of Binary Search Trees (like Red-Black Trees) are used in:

- System Libraries: std::map in C++ and TreeMap in Java.
- OS Schedulers: The Linux kernel uses a Red-Black tree to manage process scheduling (CFS), allowing it to quickly pick the next task to run.

D. Routing Tables (Tries / Binary Tries)

Routers use specialized tree structures to store IP addresses. A binary tree allows the router to quickly match the prefix of an incoming packet's IP address to determine the correct destination network (Longest Prefix Match).

Activity 1: The "Living Dictionary" – Search & Balance Simulation

Objective:

To implement a Binary Search Tree (BST) that functions as a dictionary (storing words and definitions). The goal is to demonstrate the O(\log n) search efficiency compared to a linear list, and crucially, to create a "degenerate" (skewed) tree to visually prove why Advanced Trees (like AVL or Red-Black trees) are necessary for self-balancing.

Materials / Tools / Software Required:

- Computer: Standard development setup.
- Programming Language: C++, Java, or Python.
- Dataset: A small text file containing 50–100 random words, and another list of 50 words sorted alphabetically.
- Visualization Tool (Optional): An online BST visualizer (like VisualAlgo) to compare against their code logic.

Expected Outcome:

By the end of this activity, students will be able to:

1. Enforce BST Properties: Implement the insertion logic where Left Child < Parent < Right Child.

NOTE

5. Perform Fast Lookups: Write a search function that navigates the tree, proving that looking up a word takes far fewer steps than scanning an array.
6. Create a "Worst Case": Insert the list of *sorted* words (A, B, C...) into the tree and observe how the tree transforms into a vertical line (Linked List), degrading performance to O(n).
7. Identify the Fix: Recognize exactly where the tree becomes unbalanced and theoretically apply a "Rotation" (the core mechanic of Advanced Trees) to restore balance.

9.8 SUMMARY

This unit provided a comprehensive examination of hierarchical data structures, beginning with the Binary Search Tree (BST). We established that the defining characteristic of a BST is its ordering property: for any node, all keys in the left subtree are smaller, and all keys in the right subtree are larger. This property enables efficient algorithms for Searching, Insertion, and Deletion, which typically operate in O(h) time, where h is the height of the tree. We detailed the specific logic for these operations, paying special attention to the deletion process, which requires handling three distinct scenarios: deleting a leaf, deleting a node with one child, and the complex case of deleting a node with two children by replacing it with its In-Order Successor.

We also covered Tree Traversals, the methods used to visit every node in the tree. We learned that In-Order traversal yields sorted data, Pre-Order is useful for copying trees, and Post-Order is ideal for deleting trees.

Crucially, we addressed the performance pitfalls of naive BSTs. We learned that inserting pre-sorted data leads to a Skewed Tree with O(n) complexity. To solve this, we introduced Height Balanced Trees, specifically AVL Trees. These trees automatically maintain a Balance Factor (difference in subtree heights) of -1, 0, or 1 using four types of Rotations (LL, RR, LR, RL), guaranteeing O(\log n) performance.

Beyond balancing, we explored memory optimization via Threaded Binary Trees, which replace null pointers with references to in-order predecessors and successors, facilitating stack-free traversal. Finally, we examined B-Trees, a multi-way tree structure designed for disk storage. Unlike binary trees, B-Trees are short and fat, storing multiple keys per node to minimize expensive disk I/O operations, making them essential for modern databases.

NOTE

9.9 GLOSSARY

1. Binary Search Tree (BST): A tree where the left child is strictly less than the parent, and the right child is strictly greater.
2. Root: The topmost node of a tree that has no parent.
3. Leaf: A node with no children (both left and right pointers are null).
4. In-Order Successor: The node with the smallest value in the right subtree; used during the deletion of a node with two children.
5. Skewed Tree: A BST that has degenerated into a linked list structure (worst-case height), usually due to sorted input.
6. AVL Tree: A self-balancing BST where the height difference between left and right subtrees is at most 1.
7. Balance Factor: The value calculated as Height(Left Subtree) - Height(Right Subtree) in an AVL tree.
8. Threaded Binary Tree: A binary tree that uses null pointers to store references to the predecessor or successor nodes for efficient traversal.
9. B-Tree: A self-balancing, multi-way search tree optimized for systems that read and write large blocks of data (like disks).
10. Rotation: An operation used in AVL trees (Left or Right) to change the structure of the tree to restore balance while maintaining the BST property.

9.10 ANSWER TO CHECK YOUR PROGRESS-POSSIBLE ONE

Check Your Progress 9.1

MCQ Answer:
b) Smaller, Greater – left subtree < node < right subtree.

True/False Answer:
True – AVL trees maintain balance by ensuring height difference ≤ 1.

Short Answers:

1. A BST is a binary tree where the left child is smaller and the right child is greater than the parent node.
2. Advanced trees: AVL Tree (self-balancing), Red-Black Tree (balanced with color properties).

9.11 SUGGESTED READINGS – OER

NOTE

Sr. No.	Title	URL	QR Code
1	Data Structure Using C++	http://books.aisectonline.com/User/UnitUrls?unitId=3717	

9.12 REFERENCE

- Data Structures and Algorithms in C++ – Michael T. Goodrich, Roberto Tamassia, David M. Mount – ISBN: 9780470383278 – 2011
- Data Structures and Algorithm Analysis in C++ – Mark Allen Weiss – ISBN: 9780132847377 – 2013
- Data Structures and Algorithms in C++ – Adam Drozdek – ISBN: 9781133608426 – 2012
- Data Structures Using C++ – D. S. Malik – ISBN: 9788131518236 – 2012
- Data Structures Through C++ – Yashavant Kanetkar – ISBN: 9789355515834 – 2024
- Introduction to Algorithms – Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein – ISBN: 9780262046305 – 2022 (4th Edition)
- Data Structures: A Pseudocode Approach with C – Richard F. Gilberg, Behrouz A. Forouzan – ISBN: 9780534953965 – 2005
- Fundamentals of Data Structures in C++ – Ellis Horowitz, Sartaj Sahni, Dinesh Mehta – ISBN: 9788173716065 – 2008

9.13 EXERCISE

Part 1

A. Multiple choice question

1. What is the worst-case time complexity for searching in a standard (non-balanced) Binary Search Tree?

A) O(1)

B) O(\log n)

C) O(n)

D) O(n \log n)

NOTE

2. Which traversal method results in the values of a BST being printed in sorted ascending order?

A) Pre-Order

B) Post-Order

C) In-Order

D) Level-Order

3. In a Threaded Binary Tree, what does a null "Right" pointer typically point to?

A) The Root node

B) The In-Order Successor

C) The In-Order Predecessor

D) It remains null

4. Which condition indicates that a node in an AVL Tree is unbalanced and requires rotation?

A) Balance Factor is 0

B) Balance Factor is 1

C) Balance Factor is -1

D) Balance Factor is 2

5. Why are B-Trees preferred over AVL trees for database storage?

A) They are easier to implement.

B) They minimize disk I/O by having a large branching factor (short height).

C) They use less memory than AVL trees.

D) They do not require balancing.

Answers with Explanations:

1. Answer: C) O(n)
Explanation: In a non-balanced BST, all nodes could be skewed to one side, forming a linear chain, making search time linear in the number of nodes.

2. Answer: C) In-Order
Explanation: In-order traversal visits the left subtree, root, and right subtree recursively, producing nodes in ascending order for a BST.

3. Answer: B) The In-Order Successor
Explanation: Threaded Binary Trees use null right pointers to point to the in-order successor, allowing traversal without a stack or recursion.

NOTE

4. Answer: D) Balance Factor is 2
Explanation: An AVL tree node with a balance factor of +2 or -2 is unbalanced and requires rotation to restore height balance.

5. Answer: B) They minimize disk I/O by having a large branching factor (short height).
Explanation: B-Trees are optimized for disk storage; high branching factor reduces tree height and the number of disk accesses needed for search, insert, or delete.

B. Small answer type question

1. Explain the "Ordering Property" of a Binary Search Tree and how it facilitates efficient searching compared to an unsorted array.
2. Define a "Skewed Binary Search Tree." How is it created, and what impact does it have on the time complexity of operations?
3. What is an In-Order Successor? Describe clearly how to find it within a Binary Search Tree structure.
4. Briefly explain the concept of a "Thread" in a Threaded Binary Tree and the specific memory problem it aims to solve.
5. What is the Balance Factor in an AVL Tree? What specific range of values is considered "balanced"?

Part 2

C. Long answer type question (answer in 200 words)

1. Detailed the logic required to delete a node from a Binary Search Tree when that node has two children. Explain why we cannot simply remove the node and how the In-Order Successor is used to maintain the BST property.
2. Describe the algorithm for Insertion into a Binary Search Tree. Walk through the recursive steps taken when inserting a value that is smaller than the root but larger than the root's left child.
3. Explain the four types of Rotations used in AVL Trees (LL, RR, LR, RL). Describe the specific scenarios (in terms of where a new node was inserted) that trigger each of these rotation types.
4. Compare and contrast Single Threaded and Double Threaded Binary Trees. How does the utilization of left and right null pointers differ between the two, and how does this affect traversal capabilities?
5. Discuss the difference between Naive Creation of a BST (inserting items one by one) and Balanced Creation (creating from a sorted array). Why does the naive approach often fail to produce an optimal tree?

NOTE

Part 3

D. Long answer type question (answer in 300 words)

1. Provide a comprehensive comparison between a standard BST and a B-Tree. Discuss their structural differences (branching factor, node contents), their primary use cases (In-memory vs. Disk storage), and how their height grows relative to the number of nodes (N). Explain why the B-Tree's structure is critical for database performance.
2. Write a detailed narrative explaining why we need self-balancing trees. Start with the limitations of the standard BST, specifically the worst-case O(n) scenario. Then, describe how the AVL Tree solves this using the Balance Factor and Rotations. Finally, explain the trade-offs involved—why might one choose a Red-Black tree or B-Tree over an AVL tree in certain system libraries?
3. detailed the three depth-first traversal methods (In-Order, Pre-Order, Post-Order). For each method, describe the order of visitation (Root, Left, Right) and provide a concrete application or scenario where that specific traversal is uniquely useful (e.g., Expression evaluation, copying trees, sorting data).
4. Suppose you are given a sorted array of integers. detailed the "Divide and Conquer" algorithm used to convert this array into a perfectly height-balanced Binary Search Tree. Explain why choosing the middle element as the root at every step is mathematically necessary to minimize the tree's height.
5. detailed four real-world applications of the tree structures discussed in this unit. Specifically, cover how Expression Trees are used in compilers, how Huffman Trees are used in data compression, how B-Trees are used in databases, and how Tries/Binary Tries are used in network routing.

UNIT 10
INTRODUCTION TO GRAPHS

NOTE

UNIT STRUCTURE

10.1 Introduction

10.2 Objective

10.3 Introduction to Graphs in Data Structures

10.4 Types of Graphs: Undirected and Directed

10.5 Graph Representation: Adjacency Matrix

10.6 Graph Representation: Adjacency List

10.7 Summary

10.8 Glossary

10.9 Answer to check your progress

10.10 Suggested readings - OER

10.11 Reference

10.12 Exercise

10.1 INTRODUCTION

In the vast landscape of data structures, the Graph stands out as a powerful and versatile tool for modeling complex, non-linear relationships. While linear structures like arrays and linked lists store data sequentially, graphs mirror the real world, where connections are often meshed, cyclic, or hierarchical without a strict root. At its core, a graph is a mathematical structure consisting of vertices (nodes) representing entities, and edges (links) representing the relationships between them.

Whether we are navigating a city using GPS, analyzing social networks to suggest friends, or ranking web pages on a search engine, we are relying on graph theory. The efficiency of these applications depends heavily on how we classify these graphs—such as distinguishing between Directed (one-way) and Undirected (two-way) relationships—and how we store them in memory. The choice between using an Adjacency Matrix for dense graphs or an Adjacency List for sparse graphs is a critical decision that impacts computational performance. This unit explores these foundational concepts, equipping you with the knowledge to understand how complex systems are modeled and traversed in computer science.

NOTE

10.2 OBJECTIVE

After completing this unit, you will be able to:

1. Define the fundamental components of a graph, including vertices, edges, paths, and cycles.
2. Distinguish between different types of graphs, specifically Directed vs. Undirected and Weighted vs. Unweighted.
3. Analyze the properties of graph connectivity, including indegree, outdegree, and the implications of cycles (cyclic vs. acyclic).
4. Evaluate the two primary graph representations—Adjacency Matrix and Adjacency List—in terms of space and time complexity.
5. Apply the correct graph representation strategy (Matrix or List) based on whether a given graph is dense or sparse.

10.3 INTRODUCTION TO GRAPHS IN DATA STRUCTURES

In the realm of computer science, data structures are divided into two main categories: linear (like arrays, linked lists, and stacks) and non-linear (like trees and graphs). While linear structures store data sequentially, non-linear structures are designed to represent hierarchical or interconnected relationships.

Among these, the Graph is perhaps the most versatile and powerful. It is a fundamental data structure used to model pairwise relations between objects. From the complex web of social media connections to the routing algorithms that power GPS navigation, graphs are the backbone of many modern technologies.

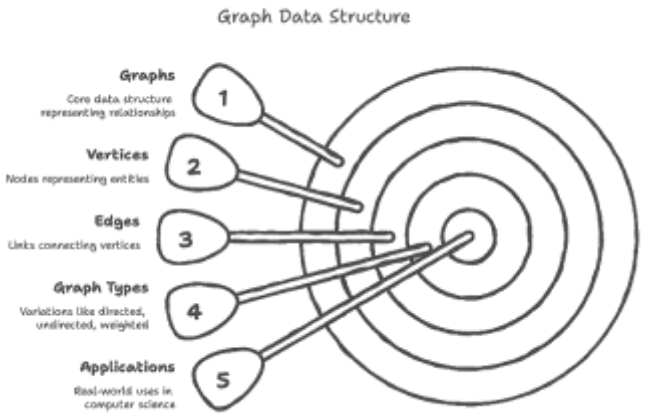


Fig 10.1: Graph Data Structure

NOTE

10.3.1. What is a Graph?

Formally, a graph G is defined as an ordered pair G = (V, E), where:

- V is a set of Vertices (also called nodes). These represent the entities or objects.
- E is a set of Edges (also called links or arcs). These represent the connections or relationships between the vertices.

Unlike a tree, which has a strict hierarchy (root to leaf), a graph has no distinct root. Any node can be connected to any other node, allowing for complex, cyclic, and mesh-like structures.

10.3.2. Key Terminology

To work with graphs effectively, one must understand the specific vocabulary used to describe them:

- Vertex (Node): The fundamental unit of the graph.
- Edge: The line connecting two vertices. An edge connecting vertex u and vertex v is denoted as (u, v).
- Adjacent: Two vertices are "adjacent" if there is an edge directly connecting them.
- Degree: The number of edges connected to a vertex. In directed graphs, this is split into indegree (incoming edges) and outdegree (outgoing edges).
- Path: A sequence of vertices where each adjacent pair is connected by an edge.
- Cycle: A path that starts and ends at the same vertex.

10.3.3. Types of Graphs

Graphs are classified based on the characteristics of their edges. Understanding these distinctions is crucial for selecting the right algorithms.

A. Directed vs. Undirected

- Undirected Graph: The edges have no direction. The relationship is two-way. If an edge exists between A and B, you can go from A to B and B to A.
    - *Example:* A Facebook friendship (if I am your friend, you are mine).
- Directed Graph (Digraph): The edges have a direction, indicated by an arrow. An edge (A, B) allows travel only from A to B, not vice-versa.

NOTE

- *Example:* An Instagram follow (I can follow you without you following me).

B. Weighted vs. Unweighted

- Unweighted Graph: All edges are considered equal; they simply indicate a connection exists.
- Weighted Graph: Each edge is assigned a numerical value (weight or cost). This weight might represent distance, time, resistance, or cost.
  - *Example:* A map where nodes are cities and edges are roads; the "weight" is the distance in kilometers.

C. Cyclic vs. Acyclic

- Cyclic Graph: Contains at least one cycle (a path that loops back to the start).
- Acyclic Graph: Contains no cycles. A famous type is the DAG (Directed Acyclic Graph), often used in scheduling and dependency resolution.

10.3.4. Graph Representation

How do we store a graph in a computer's memory? The choice of representation significantly affects the performance (time and space complexity) of graph algorithms. The two most common methods are:

A. Adjacency Matrix

This is a 2D array (matrix) of size V \times V.

- If there is an edge between vertex i and vertex j, the cell at row i, column j is marked with a 1 (or the weight).
- If there is no edge, it is marked with 0 (or infinity).

Pros & Cons:

- Advantage: Checking if an edge exists between two specific nodes is very fast: O(1).
- Disadvantage: It consumes O(V^2) space, which is inefficient for sparse graphs (graphs with many vertices but few edges).

B. Adjacency List

This involves an array of lists. Each index of the array represents a vertex, and the linked list at that index contains all the vertices adjacent to it.

Pros & Cons:

NOTE

- Advantage: Space efficient: O(V + E). It only stores edges that actually exist. This is the preferred method for most real-world applications where graphs are sparse.
- Disadvantage: Checking if an edge exists is slower: O(degree), as you must traverse the list.

10.3.5. Basic Graph Traversal Algorithms

Just as we loop through arrays, we "traverse" graphs to visit nodes. The two primary methods are:

Breadth-First Search (BFS)

BFS explores the graph layer by layer. It starts at a source node and visits all immediate neighbors before moving to the next level neighbors.

- Data Structure Used: Queue (FIFO).
- Application: Finding the shortest path in an unweighted graph, peer-to-peer networks.

Depth-First Search (DFS)

DFS explores as far as possible along each branch before backtracking. It goes deep into a path until it hits a dead end, then steps back to try a different path.

- Data Structure Used: Stack (LIFO) or Recursion.
- Application: Solving mazes, detecting cycles, topological sorting.

10.3.6. Real-World Applications

Graphs are not just theoretical constructs; they model many real-world systems:

1. Social Networks: Users are vertices; friendships are edges. Algorithms analyze these to suggest "People you may know."
2. Google Maps / GPS: Intersections are vertices; roads are edges. Weighted graph algorithms (like Dijkstra's Algorithm) calculate the shortest or fastest route.
3. World Wide Web: Web pages are vertices; hyperlinks are directed edges. Google's PageRank algorithm uses this graph to rank search results.

Network Routing: Computer networks use graphs to determine the most efficient path for data packets to travel across the internet

10.4 TYPES OF GRAPHS: UNDIRECTED AND DIRECTED

In the study of graph theory and data structures, the most fundamental classification of a graph is based on the nature of its edges. Specifically,

NOTE

does the relationship between two nodes have a direction? This distinction splits graphs into two primary categories: Undirected Graphs and Directed Graphs (Digraphs).

While they share the same basic building blocks (vertices and edges), the algorithms used to traverse, analyze, and optimize them differ significantly based on this property.

10.4.1. Undirected Graphs

An Undirected Graph is the simplest form of a graph. It represents a system where relationships are symmetric and two-way. If a connection exists between two entities, it applies equally in both directions.

10.4.1.1 Formal Definition

Mathematically, an undirected graph G is defined as a pair (V, E), where:

- V is a set of vertices.
- E is a set of edges, where each edge is an unordered pair of vertices.

If there is an edge connecting vertex u and vertex v, it is denoted as \{u, v\}.

Because the pair is unordered:

\{u, v\} = \{v, u\}

This means navigating from u to v is identical to navigating from v to u.

10.4.1.2 Key Properties

1. Symmetry:

The defining characteristic of an undirected graph is symmetry. There are no "arrows" on the lines connecting the nodes.

2. Degree of a Vertex:

In an undirected graph, the degree of a vertex v, denoted as \deg(v), is simply the number of edges incident to (connected to) it.

- The Handshaking Lemma: A fundamental theorem in graph theory states that the sum of degrees of all vertices in an undirected graph is exactly twice the number of edges:

  \sum_{v \in V} \deg(v) = 2|E|

  Reasoning: Every edge connects two vertices, contributing +1 to the degree of each endpoint.

3. Connectivity:

- Connected Graph: An undirected graph is "connected" if there is a path between every pair of vertices.

NOTE

- Disconnected Graph: If you cannot reach some vertices from others, the graph is disconnected. The isolated clusters of vertices are called Connected Components.

10.4.1.3 Representation in Memory

When storing an undirected graph in an Adjacency Matrix (a 2D array A):

- The matrix is Symmetric.
- If A[i][j] = 1, then A[j][i] must also be 1.
- This redundancy means you effectively only need to store the upper or lower triangular half of the matrix to capture all information.

10.4.1.4 Real-World Analogy

Social Networks (Facebook Friendships):

Facebook uses an undirected graph to model friendships.

- Vertex: A User.
- Edge: A Friendship.
- Logic: If Alice is friends with Bob, Bob is automatically friends with Alice. It is impossible to have a one-sided friendship in this system.

10.4.2. Directed Graphs (Digraphs)

A Directed Graph, or Digraph, represents relationships that are asymmetric. The edges have a defined direction, essentially acting as "one-way streets."

10.4.2.1 Formal Definition

A directed graph G is defined as a pair (V, E), where:

- V is a set of vertices.
- E is a set of edges, where each edge is an ordered pair of vertices.

An edge is denoted as (u, v).

- u is the tail (start).
- v is the head (destination).
- (u, v) \neq (v, u). The existence of a path from u to v implies nothing about the path from v to u.

10.4.2.2 Key Properties

1. Indegree and Outdegree:

NOTE

Unlike undirected graphs, we cannot simply count "connections." We must split the degree into two metrics:

- Indegree (\deg^-(v)): The number of edges coming *into* vertex v.
- Outdegree (\deg^+(v)): The number of edges going *out* of vertex v.
- The Sum of Degrees:

  \sum_{v \in V} \deg^-(v) = \sum_{v \in V} \deg^+(v) = |E|

2. Reachability & Connectivity:

Connectivity is more complex in digraphs:

- Strongly Connected: A graph is strongly connected if every vertex is reachable from every other vertex (i.e., you can get from A \to B and B \to A, though likely via different paths).
- Weakly Connected: A graph is weakly connected if replacing all directed edges with undirected edges would make the graph connected (i.e., the nodes are linked, but the one-way signs might prevent full traversal).

3. Sources and Sinks:

- Source: A vertex with an Indegree of 0 (only outgoing edges).
- Sink: A vertex with an Outdegree of 0 (only incoming edges).

10.4.2.3 Representation in Memory

In an Adjacency Matrix:

- The matrix is NOT Symmetric.
- A[i][j] = 1 means there is an edge from i to j. This does not tell us anything about A[j][i].

10.4.2.4 Real-World Analogy

Social Media (Twitter/X or Instagram):

These platforms use directed graphs to model "following."

- Vertex: A User.
- Edge: A Follow.
- Logic: If Alice follows Bob (Edge Alice \to Bob), Bob does not necessarily follow Alice. The relationship is directional.

Web Page Ranking:

The World Wide Web is a massive digraph.

- Vertex: Web page.
- Edge: Hyperlink.

NOTE

- Logic: Page A links to Page B. This counts as a "vote" for Page B, but Page B might not link back to A.

10.4.3. Special Structures: Cycles and Acyclic Graphs

The distinction between directed and undirected graphs dramatically changes how we handle cycles (loops).

10.4.3.1 Cyclic Graphs

- Undirected Cycle: A path that starts and ends at the same vertex using a sequence of distinct edges (e.g., A - B - C - A).
- Directed Cycle: A path that follows the arrows to return to the start (e.g., A \to B \to C \to A). If the arrows point A \to B and A \to C, no cycle can be formed between them unless there is a return path.

10.4.3.2 Directed Acyclic Graph (DAG)

This is a specific and highly important type of Directed Graph that contains no cycles.

- Importance: DAGs are used to model processes where there is a clear "before" and "after," such as prerequisites in a university course list or dependencies in software compilation (Makefiles).

Topological Sort: Only DAGs can undergo "Topological Sorting," a linear ordering of vertices such that for every directed edge u \to v, vertex u comes before v in the ordering. This is impossible in cyclic graphs or undirected graphs

10.5 GRAPH REPRESENTATION: ADJACENCY MATRIX

To process a graph using a computer, we must translate the abstract mathematical concept of vertices (V) and edges (E) into concrete data structures stored in memory. The first fundamental method is the Adjacency Matrix.

10.5.1. Concept and Definition

An Adjacency Matrix is a 2D array (or matrix) of size V \times V, where V is the number of vertices in the graph. It represents connections between nodes using a grid format.

Let A be the adjacency matrix. The value of the cell at row i and column j, denoted as A[i][j], indicates the relationship between vertex i and vertex j.

- For Unweighted Graphs:
    - A[i][j] = 1 if there is an edge from i to j.

NOTE

- A[i][j] = 0 if there is no edge.

- For Weighted Graphs:
  - A[i][j] = w (where w is the weight) if an edge exists.
  - A[i][j] = \infty (or 0, depending on context) if no edge exists.

10.5.2. Characteristics

- Symmetry (Undirected Graphs): If the graph is undirected, the matrix is symmetric across the main diagonal. If vertex A is connected to vertex B, then A[A][B] = 1 and A[B][A] = 1.
- Space Complexity: The matrix always requires V^2 space, regardless of how many edges exist. For a graph with 10,000 nodes, you need an array of size 100,000,000, even if there are only 2 edges.
- Index Access: Vertices must be mapped to integers (indices 0 to V-1) to address the matrix rows and columns.

10.5.3. Pros and Cons

Advantages

- Constant Time Lookups (O(1)): Checking if an edge exists between two specific vertices (e.g., "Is A connected to B?") is instantaneous. You simply check the value at matrix[A][B].
- Simplicity: It is easy to implement and debug.
- Dense Graphs: It is efficient for dense graphs (where the number of edges |E| is close to |V|^2).

Disadvantages

- Wasted Space (O(V^2)): For sparse graphs (where |E| \ll |V|^2), most of the matrix is filled with zeros. This wastes significant memory.
- Slow Iteration: To find all neighbors of a specific vertex, you must scan the entire row, taking O(V) time, even if the vertex has only one neighbor.

10.5.4. Example

Consider a graph with vertices 0, 1, 2.

- 0 is connected to 1.
- 1 is connected to 2.

The matrix would look like this:

\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}

NOTE

Check Your Progress 10.1

MCQ:
Which of the following represents a graph in computer science?
a) A collection of nodes connected by edges
b) A linear sequence of elements
c) A stack of data items
d) A sorted array

True/False:
In an undirected graph, edges have a direction.

Short Questions:

1. Define a graph in data structures.

.....

.....

2. Name two common types of graphs.

.....

.....

10.6 GRAPH REPRESENTATION: ADJACENCY LIST

The Adjacency List is the alternative representation designed to solve the space inefficiency of the adjacency matrix. It is the industry standard for most real-world graph applications, such as social networks or web crawling, where data is sparse.

10.6.1. Concept and Definition

An Adjacency List represents a graph as an array of lists.

- We create an array of size V.
- Each index i in the array represents vertex i.
- The element at index i is a list (often a Linked List or a Dynamic Array/Vector) containing all vertices adjacent to i.

If vertex 0 is connected to vertices 2 and 5, the list at index 0 will simply contain [2, 5].

10.6.2. Characteristics

- Dynamic Size: The memory used is proportional to the actual data. If a vertex has no neighbors, its list is empty.

NOTE

- Space Complexity: The space required is O(V + E) for directed graphs and O(V + 2E) for undirected graphs. This is significantly smaller than V^2 for sparse graphs.
- Weighted Graphs: To handle weights, the list nodes store pairs: (Neighbor, Weight).

10.6.3. Pros and Cons

Advantages

- Space Efficient: It only consumes memory for edges that actually exist. For a graph with 10,000 nodes and 2 edges, it uses minimal memory.
- Fast Iteration: To traverse the neighbors of a vertex, you only iterate through the specific list of connected nodes. The time taken is proportional to the degree of that vertex, not the total number of vertices V.

Disadvantages

- Slower Lookups (O(degree)): Checking if an edge exists between A and B is not instantaneous. You must go to list A and traverse it to see if B is present.
- Implementation Complexity: Slightly more complex to implement than a simple 2D array, as it involves pointers or dynamic data structures.

10.6.4. Example

Consider the same graph as before: Vertices 0, 1, 2.

- 0 is connected to 1.
- 1 is connected to 2 and 0.
- 2 is connected to 1.

The List Representation:

- Array[0]: \rightarrow [1]
- Array[1]: \rightarrow [0, 2]

Array[2]: \rightarrow [1]

Activity 1: "The Social Web" – Modeling Connections

Objective:

To understand the fundamental components of a Graph (Vertices and Edges) by modeling a small social network. Students will learn the difference between two primary storage methods: Adjacency Matrix (good for dense connections) and Adjacency List (good for sparse connections).

NOTE

Materials / Tools / Software Required:

- Computer: Standard development setup.
- Programming Language: Python, Java, or C++.
- Whiteboard/Paper: To draw the network before coding.
- Visual Guide: A chart comparing Matrix (V \times V grid) vs. List (array of linked lists) representations.

Expected Outcome:

By the end of this activity, students will be able to:

1. Define Graph Elements: Map "People" to Vertices (Nodes) and "Friendships" to Edges.
2. Implement Storage: Write code to store these relationships using an Adjacency List (memory efficient for social networks where one person knows few others relative to the total population).
3. Create Connections: Write a function addConnection(Person A, Person B) that creates an Undirected Edge (since friendship is mutual).
4. Query Relationships: Write a function to print "All friends of Person A" by traversing the specific list associated with that vertex.

10.7 SUMMARY

The study of graphs provides a framework for solving problems related to pairwise connections. A graph G = (V, E) is defined by its set of vertices V and edges E. Unlike trees, graphs do not necessarily have a root node and can contain cycles, making them suitable for modeling complex networks like the internet or transportation grids.

A primary distinction in graph theory is the directionality of edges. Undirected graphs represent symmetric relationships (like a two-way road or a mutual friendship), where an edge \{u, v\} implies movement is possible in both directions. In contrast, Directed graphs (Digraphs) model asymmetric relationships (like a one-way street or a social media "follow"), where an edge (u, v) distinctively travels from a tail to a head. Understanding this affects how we calculate the "degree" of a node, splitting it into in degree and out degree for digraphs.

Crucially, the way we represent these graphs in memory dictates algorithm efficiency. The Adjacency Matrix uses a 2D array (V \times V), offering O(1) edge lookups but consuming high memory (O(V^2)), making it ideal for dense graphs. Conversely, the Adjacency List uses an array of linked lists, consuming memory proportional to the edges (O(V+E)). This makes it the standard for real-world, sparse graphs, despite slightly slower edge lookup times (O(degree)). Mastering these trade-offs is essential for designing efficient algorithms.

NOTE

10.8 GLOSSARY

1. Vertex (Node): The fundamental unit in a graph representing an entity or object.
2. Edge: The connection or link between two vertices.
3. Directed Graph (Digraph): A graph where edges have a specific direction (one-way).
4. Undirected Graph: A graph where edges have no direction (two-way/symmetric).
5. Weighted Graph: A graph where edges are assigned a numerical value (cost/distance).
6. Adjacency Matrix: A 2D array representation of a graph, efficient for dense graphs.
7. Adjacency List: A representation using an array of lists, efficient for sparse graphs.
8. Degree: The total number of edges connected to a vertex.
9. Cyclic Graph: A graph that contains at least one path that starts and ends at the same vertex.
10. DAG (Directed Acyclic Graph): A directed graph with no cycles, used for topological sorting.

10.9 ANSWER TO CHECK YOUR PROGRESS-POSSIBLE ONE

Check Your Progress 10.1

MCQ Answer:
a) A collection of nodes connected by edges.

True/False Answer:
False – undirected graph edges do not have a direction.

Short Answers:

1. A graph is a set of vertices (nodes) connected by edges.
2. Common types: Directed and Undirected graphs.

10.10 SUGGESTED READINGS – OER

Sr. No.	Title	URL	QR Code
1	Data Structure Using C++	http://books.aisectonline.com/User/UnitUrls?unitId=3718	

NOTE

10.11 REFERENCE

1. Introduction to Algorithms – Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein – ISBN: 9780262046305 – 2022 (4th Edition)
2. Algorithms in C, Part 5: Graph Algorithms, 3rd Edition – Robert Sedgewick – ISBN: 9780201316636 – 2001
3. Algorithms Unlocked – Thomas H. Cormen – ISBN: 9780262518802 – 2013
4. Data Structures and Algorithms in C++ – Michael T. Goodrich, Roberto Tamassia, David M. Mount – ISBN: 9780470383278 – 2011
5. Algorithms – Robert Sedgewick, Kevin Wayne – ISBN: 9780321573513 – 2011
6. Guide to Graph Algorithms: Sequential, Parallel and Distributed – Kayhan Erciyes – ISBN: 9783319732350 – 2018/2019
7. Graphs and Graph Algorithms – Mark Needham, Amy E. Hodler – ISBN: 9781492047674 – 2020/2021
8. Graph Theory with Applications to Engineering and Computer Science – Narsingh Deo – ISBN: 9788120308299 (varies by edition) – 2016

10.12 EXERCISE

Part 1

A.Multiple choice question

1. In an undirected graph, the sum of the degrees of all vertices is equal to:

A) The number of edges

B) Twice the number of edges

C) The number of vertices

D) Squared number of edges

2. Which data structure is most space-efficient for storing a sparse graph?

A) Adjacency Matrix

B) Adjacency List

C) 2D Array

D) Stack

3. What is the space complexity of an Adjacency Matrix for a graph with V vertices?A) O(V)

NOTE

B) O(E)

C) O(V^2)

D) O(V + E)

4. In a Directed Graph, a vertex with an Indegree of 0 is called a:

A) Sink

B) Source

C) Root

D) Leaf

5. Which of the following allows constant time O(1) lookup to check if an edge exists between two nodes?

A) Adjacency List

B) Linked List

C) Adjacency Matrix

D) Binary Tree

Answers with Explanations:

1. Answer: B) Twice the number of edges
Explanation: Each edge in an undirected graph contributes to the degree of two vertices, so the total sum of degrees = 2 × number of edges.

2. Answer: B) Adjacency List
Explanation: Adjacency lists store only existing edges, so for sparse graphs (few edges), they use much less memory than adjacency matrices.

3. Answer: C) O (V^2)
Explanation: An adjacency matrix requires a V × V matrix to store all possible edges, resulting in O(V^2) space regardless of the number of edges.

4. Answer: B) Source
Explanation: A source vertex has no incoming edges (indegree 0) and can be a starting point for graph traversal.

5. Answer: C) Adjacency Matrix
Explanation: Adjacency matrices allow direct access to any edge using matrix indices, giving O(1) lookup time.

B. Small answer type question

NOTE

1. Define a "Cycle" in a graph and explain the difference between a cyclic and an acyclic graph.
2. Explain the concept of "Degree" in an undirected graph versus a directed graph.
3. What is a Weighted Graph? Provide two real-world examples where edge weights are necessary.
4. Why is an Adjacency Matrix considered inefficient for sparse graphs?
5. Describe the difference between a "Source" and a "Sink" in a Directed Graph

Part 2

C. Long answer type question (answer in 200 words)

1. Compare and contrast Directed and Undirected graphs. Focus on their edge definitions, symmetry, and real-world applications.
2. Discuss the structure of an Adjacency List. How does it manage memory dynamically, and why is this preferred for social network analysis?
3. Explain the Handshaking Lemma in the context of Undirected Graphs. Why is the sum of degrees always even?
4. Analyze the time complexity differences between Adjacency Matrices and Adjacency Lists for the operations of "Adding a Vertex" and "Checking if an Edge Exists."
5. What is a Directed Acyclic Graph (DAG)? Explain its significance in computer science, specifically regarding dependencies and scheduling.

Part 3

D. Long answer type question (answer in 300 words)

1. You are designing a navigation system for a city (like Google Maps). Would you model the city using a Directed or Undirected graph? Would you use an Adjacency Matrix or List? Justify your choices based on memory constraints and the nature of traffic rules.
2. Provide a detailed comparative analysis of Graph Representations. Create a theoretical scenario where an Adjacency Matrix is the superior choice over an Adjacency List, despite its high space complexity.
3. Explore the concept of Graph Traversal. Although BFS and DFS were briefly mentioned, explain how the choice of data

NOTE

structure (Matrix vs. List) impacts the implementation and efficiency of these traversal algorithms.

4. Mathematical representation of graphs is crucial for algorithms. Detail how a graph G=(V,E) is mapped into computer memory. Discuss how indices are assigned to vertices and how this mapping supports the logic of an Adjacency Matrix.
5. Investigate the limitations of linear data structures (arrays, linked lists) that necessitate the use of Graphs. Why can't a simple list effectively model a peer-to-peer network or the World Wide Web?

UNIT 11
GRAPH ALGORITHMS AND TRAVERSALS

UNIT STRUCTURE

11.1 Introduction

11.2 Objective

11.3 Graph Traversals: A Comprehensive Guide

11.4 Searching in Graphs: Core Concepts & Algorithms

11.5 Minimum Spanning Tree (MST): Algorithms & Theory

11.6 Shortest Path Algorithms: A Comprehensive Guide

11.7 Summary

11.8 Glossary

11.9 Answer to check your progress

11.10 Suggested readings - OER

11.11 Reference

11.12 Exercise

11.1 INTRODUCTION

Graphs are among the most versatile and powerful data structures in computer science, capable of modeling complex relationships ranging from social networks and transportation systems to biological neural maps. However, the static structure of a graph is useless without efficient algorithms to navigate and analyze it. This unit explores the fundamental algorithmic techniques required to solve critical graph-based problems.

We begin with Graph Traversals, the bedrock of graph theory, where we examine how to systematically visit every node in a structure using Breadth-First Search (BFS) and Depth-First Search (DFS). Building on this, we explore Searching Strategies, distinguishing between blind exploration and informed searches like A* that use heuristics to guide decision-making.

Moving beyond simple navigation, we tackle optimization problems. We analyze Minimum Spanning Trees (MST), specifically Kruskal's and Prim's algorithms, which are essential for designing efficient networks with minimal cost. Finally, we delve into Shortest Path Algorithms,

NOTE

studying how to calculate optimal routes using Dijkstra's algorithm for positive weights, Bellman-Ford for graphs with negative edges, and Floyd-Warshall for all-pairs analysis. Mastery of these algorithms provides the toolkit necessary for solving complex connectivity and optimization challenges in the real world.

11.2 OBJECTIVE

After Completing this unit you will be able to:

1. Differentiate between Breadth-First Search (BFS) and Depth-First Search (DFS) in terms of data structures used (Queue vs. Stack), traversal patterns, and appropriate use cases.
2. Apply Minimum Spanning Tree algorithms, specifically Kruskal's and Prim's, to solve network connectivity problems by identifying the minimum edge weight subset required to connect all vertices.
3. Analyze and implement Shortest Path algorithms, selecting the correct approach (Dijkstra, Bellman-Ford, or Floyd-Warshall) based on graph constraints such as edge weights and cycle types.
4. Evaluate search strategies by comparing Uninformed searches with Informed (Heuristic) searches like A* to understand trade-offs in optimality and memory usage.
5. Calculate the time and space complexity of various graph algorithms to determine their efficiency and feasibility for large-scale datasets.

11.3 GRAPH TRAVERSALS: A COMPREHENSIVE GUIDE

Graph traversal is the process of visiting (checking and/or updating) every node in a graph tree structure exactly once. Such traversals are classified by the order in which the nodes are visited. The two most fundamental algorithms for graph traversal are Breadth-First Search (BFS) and Depth-First Search (DFS).

These algorithms form the backbone of many complex operations in computer science, including pathfinding, cycle detection, topological sorting, and artificial intelligence decision-making.

11.3.1. Breadth-First Search (BFS)

11.3.1.1 Concept and Intuition

Breadth-First Search is an algorithm for traversing or searching tree or graph data structures. It starts at a selected node (often called the

'source' or 'root') and explores all of the neighbor nodes at the present depth prior to moving on to the nodes at the next depth level.

NOTE

Think of BFS as a wildfire spreading evenly in all directions, or ripples in a pond moving outward from where a stone was dropped. It explores the graph layer by layer:

1. Layer 0: The source node.
2. Layer 1: Immediate neighbors of the source.
3. Layer 2: Neighbors of the Layer 1 nodes (that haven't been visited).

11.3.1.2 The Data Structure: Queue

BFS utilizes a Queue (First-In-First-Out, or FIFO) data structure. The queue ensures that nodes are processed in the exact order they were discovered, maintaining the "level-order" nature of the traversal.

11.3.1.3 Algorithm Steps

1. Initialize: Create a boolean array visited[] to keep track of visited nodes (initially all false). Create a Queue Q.
2. Start: Mark the starting node as visited and enqueue it into Q.
3. Loop: While Q is not empty:
    - Dequeue a vertex v from Q.
    - Process v (e.g., print it).
    - For every adjacent vertex u of v:
        - If u has not been visited:
            - Mark u as visited.
            - Enqueue u.

11.3.1.4 Pseudocode

Plaintext

```
BFS(Graph G, Node start_node):
    Let Q be a Queue
    Let visited be a Set or Array

    Q.enqueue(start_node)
    visited.add(start_node)

    while Q is not empty:
```

NOTE

```
current_node = Q.dequeue()
PRINT current_node


for neighbor in G.adjacent(current_node):
   if neighbor is not in visited:
      visited.add(neighbor)
      Q.enqueue(neighbor)
```

11.3.1.5 Detailed Example Trace

Consider a graph:

- A is connected to B and C.
- B is connected to D and E.
- C is connected to F.

Goal: Run BFS starting from A.

1. Init: Queue = [A], Visited = {A}.
2. Step 1: Dequeue A. Process A. Neighbors are B, C.
    - Mark B visited, Enqueue B.
    - Mark C visited, Enqueue C.
    - Queue = [B, C].
3. Step 2: Dequeue B. Process B. Neighbors are D, E.
    - Mark D visited, Enqueue D.
    - Mark E visited, Enqueue E.
    - Queue = [C, D, E].
4. Step 3: Dequeue C. Process C. Neighbor is F.
    - Mark F visited, Enqueue F.
    - Queue = [D, E, F].
5. Step 4, 5, 6: Dequeue D, E, and F respectively. They have no unvisited neighbors.
    - Queue becomes empty. Stop.

Output: A -> B -> C -> D -> E -> F

11.3.1.6 Complexity Analysis

- Time Complexity: O(V + E)
    - Where V is the number of vertices and E is the number of edges.

NOTE

- Every vertex is enqueued and dequeued exactly once (O(V)).
- Every edge is scanned once (in directed graphs) or twice (in undirected graphs) when checking adjacency (O(E)).

- Space Complexity: O(V)
  - In the worst case, the queue might contain all the vertices at a specific level. For a "star" graph, this could be V-1 nodes. We also need O(V) space for the visited array.

11.3.1.7 Key Applications of BFS

1. Shortest Path in Unweighted Graphs: BFS guarantees finding the shortest path (minimum number of edges) between a start node and any other reachable node.
2. Peer-to-Peer Networks: Finding the nearest neighbor or broadcasting packets.
3. Social Networking: Finding people within a certain "degree of separation" (e.g., "Friends of Friends").
4. Web Crawlers: Google uses BFS variants to discover new web pages by following links from known pages.

11.3.2. Depth-First Search (DFS)

11.3.2.1 Concept and Intuition

Depth-First Search explores as far as possible along each branch before backtracking. It dives deep into the graph structure, hitting a dead end, and then steps back to find a new path to explore.

Think of DFS as solving a maze. You walk down a path until you hit a wall, then you turn around (backtrack) to the last junction and try a different path.

11.3.2.2 The Data Structure: Stack

DFS utilizes a Stack (Last-In-First-Out, or LIFO) data structure. This can be implemented explicitly using a stack data structure or implicitly using the Call Stack via recursion.

11.3.2.3 Algorithm Steps (Recursive Approach)

1. Initialize: Create a visited[] array.
2. Start: Call the recursive DFS function on the starting node.
3. Recursive Function DFS(node):
   - Mark node as visited.
   - Process node (e.g., print it).

NOTE

- For every adjacent vertex neighbor of node:
  - If neighbor is not visited, recursively call DFS(neighbor).

11.3.2.4 Pseudocode (Recursive)

Plaintext

```
DFS(Graph G, Node current_node, Set visited):
   visited.add(current_node)
   PRINT current_node

   for neighbor in G.adjacent(current_node):
      if neighbor is not in visited:
         DFS(G, neighbor, visited)
```

11.3.2.5 Detailed Example Trace

Using the same graph as BFS:

- A connects to B, C.
- B connects to D, E.
- C connects to F.

Goal: Run DFS starting from A.

1. Start at A. Mark A visited. Print A.
2. Check neighbors of A (B, C). Let's pick B.
3. Recurse to B. Mark B visited. Print B.
4. Check neighbors of B (D, E). Let's pick D.
5. Recurse to D. Mark D visited. Print D.
6. Check neighbors of D. None unvisited. Backtrack to B.
7. Check remaining neighbors of B (E). Pick E.
8. Recurse to E. Mark E visited. Print E.
9. Check neighbors of E. None unvisited. Backtrack to B.
10. All neighbors of B done. Backtrack to A.
11. Check remaining neighbors of A (C). Pick C.
12. Recurse to C. Mark C visited. Print C.
13. Check neighbors of C (F). Pick F.
14. Recurse to F. Mark F visited. Print F.
15. Backtrack to C -> Backtrack to A. Done.

NOTE

Output: A -> B -> D -> E -> C -> F

(Note: Output order may vary depending on which neighbor is chosen first).

11.3.2.6 Complexity Analysis

- Time Complexity: O(V + E)
    - Similar to BFS, every node is visited once, and every edge is checked once (or twice for undirected).
- Space Complexity: O(V)
    - In the worst case (a long, straight line graph like a linked list), the recursion stack will grow to depth V.
    - We also need O(V) for the visited array.

11.3.2.7 Key Applications of DFS

1. Cycle Detection: DFS is excellent for detecting cycles in both directed and undirected graphs (by detecting "back edges" pointing to ancestors).
2. Topological Sorting: Used for scheduling tasks with dependencies (e.g., compiling code modules).
3. Solving Puzzles: Mazes, Sudoku solvers, and pathfinding where the solution is far from the start (requiring depth).
4. Finding Strongly Connected Components: Algorithms like Kosaraju's or Tarjan's use DFS.

11.4 SEARCHING IN GRAPHS: CORE CONCEPTS & ALGORITHMS

Graph Searching differs slightly from Graph Traversal. While traversal implies visiting every node in the graph (e.g., for printing or updating), searching focuses on finding a specific target node or the optimal path to it from a start node.

Broadly, graph search algorithms are categorized into Uninformed (Blind) Search, which has no information about the goal's location, and Informed (Heuristic) Search, which uses hints to find the goal faster.

11.4.1. Uninformed Search Strategies

These algorithms explore the graph without any clue as to which direction is "correct." They systematically examine nodes until the goal is found.

NOTE

11.4.1.1 Breadth-First Search (BFS)

BFS explores the graph layer-by-layer. It is defined by its "radiating" nature—checking all immediate neighbors before moving deeper.

- The Mechanism: Uses a Queue (FIFO).
- Search Strategy: It checks nodes at distance d=1 from the start, then all nodes at d=2, and so on.
- Best Use Case: Finding the Shortest Path in unweighted graphs (e.g., fewest hops to a destination).

Algorithm for Search:

1. Add Start_Node to Queue.
2. While Queue is not empty:
    - Dequeue Current.
    - Check: Is Current the Target? If yes, Return Success/Path.
    - Else, enqueue all unvisited neighbors of Current and mark them visited.

11.4.1.2 Depth-First Search (DFS)

DFS explores as deep as possible along each branch before backtracking. It is defined by its "aggressive" nature—diving down a path until it hits a dead end.

- The Mechanism: Uses a Stack (LIFO) or Recursion.
- Search Strategy: It picks a neighbor, then that neighbor's neighbor, repeating until it hits a leaf or visited node, then backtracks.
- Best Use Case: Puzzles (like Mazes) where solutions are deep, or when memory is limited (DFS uses less memory than BFS on average).

Algorithm for Search:

1. Push Start_Node to Stack.
2. While Stack is not empty:
    - Pop Current.
    - Check: Is Current the Target? If yes, Return Success.
    - Else, push unvisited neighbors to Stack.

Critical Note: DFS does not guarantee the shortest path. It might find a convoluted path to the neighbor next door by traversing the entire graph first.

NOTE

11.4.2. Advanced Search: Informed & Weighted Strategies

In many real-world scenarios (like Google Maps), edges have "weights" (distance, time, cost). Standard BFS/DFS cannot handle these efficiently.

11.4.2.1 Dijkstra's Algorithm (Uniform Cost Search)

Dijkstra's is a variation of BFS that prioritizes nodes based on the accumulated cost from the start node, rather than just the "hop count."

- Mechanism: Uses a Priority Queue.
- Strategy: Always expands the "cheapest" node next.
- Result: Guarantees the shortest path in weighted graphs (with non-negative weights).

11.4.2.2 A* Search (A-Star)

This is the gold standard for pathfinding in AI and games. It combines Dijkstra's cost mechanism with a Heuristic (h).

- Formula: f(n) = g(n) + h(n)
    - g(n): Exact cost from start to node n.
    - h(n): Estimated cost from node n to the goal (e.g., straight-line distance).
- Strategy: It explores the most promising paths first, "guided" by the heuristic.

11.4.3. Real-World Applications

1. Social Networks (BFS): "People you may know" features search for 2nd or 3rd connections.
2. Web Crawlers (BFS/DFS): Search engines use traversals to index the internet.
3. GPS Navigation (A):* Finding the fastest route from A to B uses A* with road distances as weights and physical distance as the heuristic.

Garbage Collection (DFS): Programming languages (like Java) use Mark-and-Sweep algorithms (a form of DFS) to find reachable objects in memory.

11.5 MINIMUM SPANNING TREE (MST): ALGORITHMS & THEORY

A Minimum Spanning Tree (MST) is a subset of the edges of a connected, edge-weighted undirected graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight.

NOTE

Key Properties of an MST

1. Connectivity: It must connect all vertices in the graph. If a graph has V vertices, the MST will have exactly V-1 edges.
2. Acyclicity: It must not contain any cycles (loops).
3. Minimality: The sum of the weights of the edges in the MST is the lowest possible sum among all possible spanning trees for that graph.
4. Uniqueness: If all edge weights are distinct, the MST is unique. If some edges have the same weight, there may be multiple valid MSTs, but they will all have the same total weight.

Real-World Applications:

- Network Design: Laying cables to connect cities with minimal wire cost.
- Circuit Design: Connecting pins on an electrical component using the least amount of wire.
- Clustering: Used in machine learning for hierarchical clustering.

11.5.1. Kruskal's Algorithm

11.5.1.1 Concept: The "Edge-Centric" Approach

Kruskal's algorithm is a greedy algorithm that focuses entirely on the edges. It treats the graph as a forest (a collection of individual trees) and merges them together.

The Strategy:

"Always pick the cheapest edge available in the entire graph, as long as it doesn't create a loop."

11.5.1.2 Algorithm Steps

1. Sort: List all edges in the graph and sort them in non-decreasing order of their weight.
2. Initialize: Create a set for each vertex (initially, every vertex is its own separate tree/component).
3. Iterate: Loop through the sorted edges:
    - Let the current edge connect vertex u and vertex v.
    - Check if u and v are already in the same set (connected component).
    - If they are different: Add the edge to the MST and merge the two sets (Union operation).

NOTE

- If they are the same: Discard the edge (adding it would create a cycle).

4. Terminate: Stop when V-1 edges have been added.

11.5.1.3 Data Structure: Union-Find (Disjoint Set)

To efficiently check if adding an edge creates a cycle, Kruskal's uses the Union-Find data structure.

- Find(x): Identifies which set vertex x belongs to.
- Union(x, y): Merges the set containing x and the set containing y.

11.5.1.4 Complexity Analysis

- Sorting Edges: O(E \log E) or O(E \log V).
- Union-Find Operations: O(E \alpha(V)), where \alpha is the Inverse Ackermann function (nearly constant).
- Total Time Complexity: O(E \log E) (dominated by the sort).
- Space Complexity: O(V + E) to store the graph and disjoint sets.

11.5.2. Prim’s Algorithm

11.5.2.1 Concept: The "Vertex-Centric" Approach

Prim’s algorithm is also a greedy algorithm, but it grows a single tree from a starting point. It acts like a "mold" or "oil spill" spreading from a source node to the nearest unvisited node.

The Strategy:

"Start at one node. At every step, add the cheapest edge that connects a node inside our growing tree to a node outside the tree."

11.5.2.2 Algorithm Steps

1. Initialize:
    - Choose an arbitrary starting vertex (e.g., node A).
    - Maintain two sets of vertices: Visited (in the MST) and Unvisited.
    - Initialize a Priority Queue (Min-Heap) with all edges connected to the start node.
2. Grow the Tree:
    - While the Visited set does not contain all vertices:
        - Extract the edge with the minimum weight from the Priority Queue.
        - Let this edge connect u (visited) and v (unvisited).

NOTE

- If v is already visited, discard the edge.
- If v is unvisited:
  - Mark v as visited.
  - Add the edge to the MST.

Add all valid edges connected to v into the Priority Queue.

Check Your Progress 11.1

MCQ:
Which graph traversal algorithm uses a queue to explore nodes level by level?
a) Depth-First Search (DFS)
b) Breadth-First Search (BFS)
c) Dijkstra's Algorithm
d) Prim's Algorithm

True/False:
Depth-First Search (DFS) explores all vertices of a graph by going as deep as possible before backtracking.

Short Questions:

1. Define Breadth-First Search (BFS).

.....

.....

2. Name two applications of graph traversal algorithms.

.....

.....

11.6 MINIMUM SPANNING TREE (MST): ALGORITHMS & THEORY

The Shortest Path problem is one of the most fundamental challenges in graph theory. It involves finding a path between two vertices (nodes) in a graph such that the sum of the weights of its constituent edges is minimized.

Types of Shortest Path Problems:

1. Single-Source Shortest Path: Find the shortest path from one source node to all other nodes in the graph (e.g., Dijkstra, Bellman-Ford).

NOTE

2. All-Pairs Shortest Path: Find the shortest path between every pair of nodes in the graph (e.g., Floyd-Warshall).
3. Unweighted vs. Weighted: Does distance matter?
    - Unweighted: The shortest path is simply the fewest number of edges (Hops).
    - Weighted: The shortest path is the minimum total sum of edge weights (Cost/Distance).

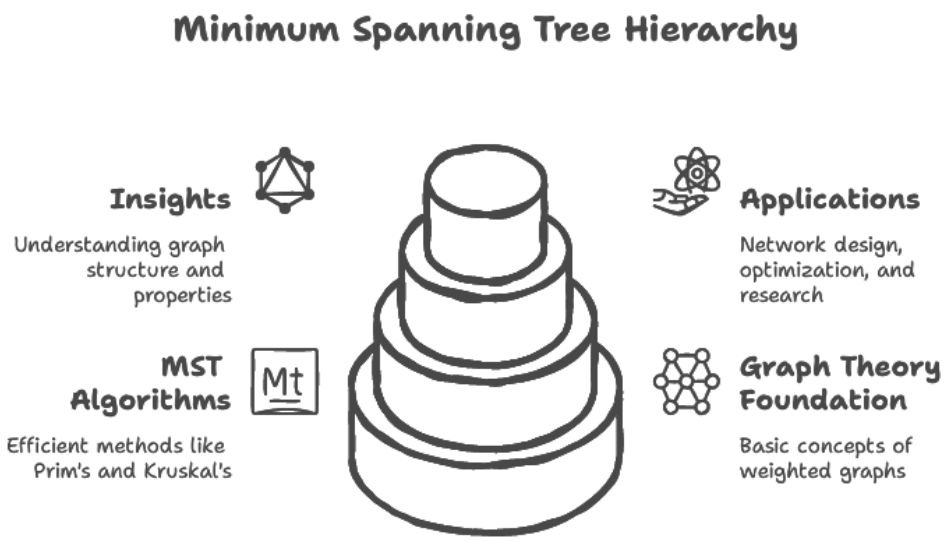


Fig 11.1: Minimum Spanning Tree Hierarchy

11.6.1. Unweighted Graphs: Breadth-First Search (BFS)

For graphs where all edges are equal (or weight = 1), the shortest path is found using BFS. Because BFS explores layer-by-layer, the first time it reaches a node, it is guaranteed to be via the shortest path (minimum hops).

- Complexity: O(V + E)
- Limitation: Fails immediately if edges have different weights (e.g., a road of 5km vs a road of 100km).

11.6.2. Weighted Graphs (Positive Weights): Dijkstra's Algorithm

Dijkstra's Algorithm is the standard for finding the shortest path from a single source when edge weights are non-negative. It is a Greedy Algorithm.

11.6.2.1 The Concept

Dijkstra's algorithm maintains a set of "finalized" vertices and a set of "tentative" vertices. At every step, it selects the tentative vertex with the smallest known distance from the source, finalizes it, and "relaxes" its neighbors.

Relaxation:

NOTE

Relaxation is the process of trying to lower the cost of reaching a vertex v by using a vertex u.

If \ d[u] + cost(u, v) < d[v], \ then \ d[v] = d[u] + cost(u, v)

11.6.2.2 Algorithm Steps

1. Init: Set distance to Source = 0, and all others = \infty. Add all pairs (0, Source) to a Min-Priority Queue.
2. Loop: While the Priority Queue is not empty:
    - Extract the node u with the minimum distance.
    - If u is already finalized, skip it.
    - Mark u as finalized.
    - For each neighbor v of u:
        - Perform Relaxation: If distance to u + weight(u,v) < current distance to v, update v's distance and push new distance to the Priority Queue.

11.6.2.3 Complexity

- Time: O(E \log V) using a Binary Heap (Priority Queue).
- Space: O(V) to store distances and the queue.

11.6.3. Weighted Graphs (Negative Weights): Bellman-Ford Algorithm

Dijkstra's algorithm fails if the graph has negative edge weights (it might finalize a node too early). The Bellman-Ford Algorithm handles negative weights and can also detect negative weight cycles.

11.6.3.1 The Concept

Bellman-Ford is a Dynamic Programming approach. It works on the principle that in a graph with V vertices, the shortest simple path can have at most V-1 edges.

11.6.3.2 Algorithm Steps

1. Init: Set distance to Source = 0, all others = \infty.
2. Relaxation Loop: Repeat V-1 times:
    - Iterate through all edges (u, v) in the graph.
    - Relax the edge: If d[u] + weight(u, v) < d[v], update d[v].
3. Negative Cycle Check:
    - Iterate through all edges one more time.
    - If any edge can still be relaxed, it means there is a Negative Weight Cycle (a loop where the total sum is negative, allowing infinite cost reduction).

NOTE

11.6.3.3 Complexity

- Time: O(V \times E). (Much slower than Dijkstra).
- Space: O(V).

11.6.4. All-Pairs Shortest Path: Floyd-Warshall Algorithm

If you need the shortest distance between every pair of nodes (Node A to B, A to C, B to C, etc.), running Dijkstra V times is inefficient for dense graphs. Floyd-Warshall solves this using Dynamic Programming.

11.6.4.1 The Concept

It builds a matrix dist[][] where dist[i][j] is the shortest distance from i to j.

It iterates through every vertex k and asks: "Is it faster to go from i to j directly, or by taking a detour through k?"

11.6.4.2 The Core Logic

dist[i][j] = \min(dist[i][j], \ dist[i][k] + dist[k][j])

11.6.4.3 Complexity

- Time: O(V^3).
- Space: O(V^2) (Adjacency Matrix).
- Use Case: Small graphs (V < 500) where you need a complete distance map.

Activity 1: The "Maze Runner" – Solving Path finding

Objective: To apply Graph Traversal Algorithms—specifically Breadth-First Search (BFS) and Depth-First Search (DFS)—to solve a maze. This visually demonstrates the difference in how these algorithms explore data: BFS expanding in waves (finding the shortest path) versus DFS diving deep into one path (exploring exhaustively).

Materials / Tools / Software Required:

- Computer: Standard development setup.
- Programming Language: Python (highly recommended for grid visualization) or C++/Java.
- Input Data: A simple 2D Grid text file representing a maze (where 0 is a path, 1 is a wall, S is Start, E is End).
- Graph Paper: For students to manually trace the algorithms before coding.

Expected Outcome: By the end of this activity, students will be able to:

1. Model the Grid as a Graph: Treat each cell in the maze as a Vertex and adjacent walkable cells as connected by Edges.

NOTE

5. Implement DFS (Stack-based): Write a recursive or stack-based algorithm to solve the maze. They will observe that DFS often finds *a* path quickly but it is rarely the *shortest* one (it tends to produce a "snake-like" solution).
6. Implement BFS (Queue-based): Write a queue-based algorithm. They will observe that BFS expands equally in all directions and guarantees finding the shortest path (minimum number of steps) in an unweighted grid.
7. Visualize the "Frontier": Print or animate the grid step-by-step to see the "wavefront" of BFS vs. the "tentacle" of DFS.

11.7 SUMMARY

Graph algorithms form the core of computational problem-solving, categorized primarily into traversal, searching, connectivity, and pathfinding.

Graph Traversal is the process of visiting all nodes in a graph. Breadth-First Search (BFS) utilizes a Queue to explore the graph layer-by-layer, making it ideal for finding the shortest path in unweighted graphs. In contrast, Depth-First Search (DFS) uses a Stack (or recursion) to dive deep into a branch before backtracking, making it superior for puzzle solving, topological sorting, and cycle detection. While BFS guarantees the shortest path in terms of "hops," DFS is more memory-efficient in exploring deep solution spaces.

Searching extends traversal by hunting for specific targets. Uninformed search (like BFS/DFS) operates blindly, while Informed search (like A*) uses heuristics to estimate the cost to the goal, significantly speeding up the process in complex environments like game maps.

For optimization, Minimum Spanning Trees (MST) allow us to connect a set of vertices with the minimum total edge weight. Kruskal's Algorithm takes an edge-centric greedy approach, sorting all edges and adding them if they don't form a cycle (managed via Union-Find). Prim's Algorithm is vertex-centric, growing a single tree from a source node using a Priority Queue. Kruskal's is better for sparse graphs, while Prim's excels in dense graphs.

Finally, Shortest Path Algorithms solve the routing problem. Dijkstra's Algorithm is the standard for non-negative weighted graphs, using relaxation to find the optimal path. However, if negative weights exist, Bellman-Ford is required, as it can handle negative edges and detect negative cycles. for comprehensive network analysis, Floyd-Warshall computes distances between every pair of nodes using dynamic

programming. Understanding these distinctions ensures the selection of the right tool for specific computational constraints.

NOTE

11.8 GLOSSARY

1. Adjacency: The state of two vertices being connected directly by an edge.
2. Backtracking: An algorithmic technique used in DFS where the search returns to the most recent junction after hitting a dead end.
3. Relaxation: The process in Shortest Path algorithms of updating the shortest known distance to a vertex if a cheaper path is found via a neighbor.
4. Priority Queue: A data structure used in Dijkstra's and Prim's algorithms that allows for efficient retrieval of the element with the minimum priority (weight/cost).
5. Heuristic: An estimated "guess" used in Informed Search (like A*) to guide the algorithm toward the goal more efficiently.
6. Disjoint Set (Union-Find): A data structure used in Kruskal's algorithm to track which set a vertex belongs to and to detect cycles.
7. Negative Cycle: A cycle in a graph where the sum of edge weights is negative, allowing for infinite cost reduction in pathfinding.
8. Spanning Tree: A subgraph that connects all vertices in a graph with the minimum number of edges (V-1) and no cycles.
9. Greedy Algorithm: An algorithmic paradigm (used in Prim's, Kruskal's, Dijkstra's) that makes the locally optimal choice at each step.
10. Time Complexity: A measure of the amount of time an algorithm takes to run as a function of the length of the input (e.g., O(V+E)).

11.9 ANSWER TO CHECK YOUR PROGRESS-POSSIBLE ONE

Check Your Progress 11.1

MCQ Answer:
b) Breadth-First Search (BFS) – uses a queue to explore nodes level by level.

NOTE

True/False Answer:
True – DFS explores as far along a branch before backtracking.

Short Answers:

1. BFS is a traversal algorithm that visits all vertices level by level using a queue.
2. Applications: Shortest path finding, network routing, cycle detection.

11.10 SUGGESTED READINGS – OER

Sr. No.	Title	URL	QR Code
1	Data Structure Using C++	http://books.aisectonline.com/User/UnitUrls?unitId=3719	

11.11 REFERENCE

1. Introduction to Algorithms – Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein – ISBN: 9780262046305 – 2022 (4th Edition)
2. Algorithms in C, Part 5: Graph Algorithms, 3rd Edition – Robert Sedgewick – ISBN: 9780201316636 – 2001
3. Algorithms Unlocked – Thomas H. Cormen – ISBN: 9780262518802 – 2013
4. Data Structures and Algorithms in C++ – Michael T. Goodrich, Roberto Tamassia, David M. Mount – ISBN: 9780470383278 – 2011
5. Algorithms – Robert Sedgewick, Kevin Wayne – ISBN: 9780321573513 – 2011
6. Guide to Graph Algorithms: Sequential, Parallel and Distributed – Kayhan Erciyes – ISBN: 9783319732350 – 2018/2019
7. Graphs and Graph Algorithms – Mark Needham, Amy E. Hodler – ISBN: 9781492047674 – 2020/2021
8. Graph Theory with Applications to Engineering and Computer Science – Narsingh Deo – ISBN: 9788120308299 (varies by edition) – 2016

NOTE

11.12 EXERCISE

Part 1

A. Multiple choice question

1. Which data structure is primarily used to implement Breadth-First Search (BFS)?

A) Stack

B) Priority Queue

C) Queue (FIFO)

D) Hash Map

2. What is the primary limitation of Dijkstra's Algorithm?

A) It cannot handle directed graphs.

B) It fails if the graph contains negative edge weights.

C) It is slower than Bellman-Ford.

D) It cannot find the shortest path in unweighted graphs.

3. Which algorithm is best suited for detecting a negative weight cycle in a graph?

A) Prim's Algorithm

B) Floyd-Warshall

C) Dijkstra's Algorithm

D) Bellman-Ford Algorithm

4. In Kruskal's algorithm, which operation is used to prevent the formation of cycles?

A) Backtracking

B) Relaxation

C) Union-Find (Disjoint Set checks)

D) Depth-First Search

5. What is the space complexity of Depth-First Search (DFS) in the worst-case scenario (excluding the graph storage)?

A) O(1)

B) O(V)

C) O(V^2)

D) O(E)

NOTE

Answers with Explanations:

1. Answer: C) Queue (FIFO)
Explanation: BFS explores nodes level by level, so a queue is used to process nodes in the order they are discovered (FIFO order).

2. Answer: B) It fails if the graph contains negative edge weights.
Explanation: Dijkstra's algorithm assumes all edge weights are non-negative; negative edges can cause it to produce incorrect shortest paths.

3. Answer: D) Bellman-Ford Algorithm
Explanation: Bellman-Ford can detect negative weight cycles because it relaxes edges repeatedly and checks for further reductions after V-1 iterations.

4. Answer: C) Union-Find (Disjoint Set checks)
Explanation: Union-Find tracks connected components and ensures that adding an edge does not connect two nodes already in the same set, preventing cycles.

5. Answer: B) O(V)
Explanation: DFS uses a recursion stack (or explicit stack) that can grow up to the number of vertices (V) in the worst case.

B. Small answer type question

1. Explain the core difference in traversal strategy between BFS and DFS.
2. Define a Minimum Spanning Tree (MST) and list two of its key properties.
3. Why does Dijkstra's algorithm fail when applied to graphs with negative edge weights?
4. Briefly describe the "Relaxation" step used in shortest path algorithms.
5. Differentiate between Uninformed Search and Informed Search strategies.

Part 2

C. Long answer type question (answer in 200 words)

1. Compare and contrast Prim's Algorithm and Kruskal's Algorithm. Discuss their methodologies and identify which graph types (sparse vs. dense) suit each algorithm best.

NOTE

2. Describe the Bellman-Ford algorithm. How does it handle negative edge weights, and how does it detect the presence of negative weight cycles?
3. Explain the role of the Priority Queue in Dijkstra's algorithm. How does it contribute to the "Greedy" nature of the algorithm and its overall efficiency?
4. Discuss the concept of Backtracking in Depth-First Search (DFS). How does this mechanism allow DFS to solve problems like maze navigation or cycle detection?
5. Analyze the complexity (Time and Space) of the Floyd-Warshall algorithm. Why is it considered less efficient than Dijkstra's algorithm for Single-Source Shortest Path problems?

Part 3

D. Long answer type question (answer in 300 words)

1. You are designing a navigation system for a city map. Detailed roads have positive lengths, but some theoretical "wormholes" might have negative travel time. Provide a detailed comparative analysis of BFS, Dijkstra, and Bellman-Ford, justifying which algorithm you would choose for finding the shortest path and why the others might fail or be inefficient.
2. Trace the execution of Kruskal's Algorithm on a hypothetical graph with 5 vertices and 7 weighted edges. Describe the sorting of edges, the use of the Union-Find data structure at each step, and how the algorithm ensures the final result is both a connected tree and minimal in weight.
3. DFS and BFS are fundamentally different tools. Elaborate on three distinct real-world applications for each algorithm (6 applications total). For each application, explain why that specific traversal method is the required or optimal choice over the other.
4. Explain the A (A-Star) Search algorithm in detail. Discuss the significance of the heuristic function h(n) and the cost function g(n). How does A balance the optimality of Dijkstra's algorithm with the speed of a greedy best-first search?**
5. Consider a scenario where you need to verify if a complex software project with various dependent modules can be compiled (Topological Sort) and if there are any circular dependencies (Cycles). Explain how Depth-First Search (DFS) can be utilized to solve both problems, detailing the use of "visited" arrays and recursion stacks.

NOTE

UNIT 12
INTRODUCTION TO SORTING AND BASIC SORTING TECHNIQUES

UNIT STRUCTURE

12.1 Introduction

12.2 Objective

12.3 Sorting

12.4 Bubble Sort

12.5 Selection Sort

12.6 Insertion Sort

12.7 Summary

12.8 Glossary

12.9 Answer to check your progress

12.10 Suggested readings – OER

12.11 Reference

12.12 Exercise

12.1 INTRODUCTION

Sorting is a foundational concept in computer science and data processing, referring to the systematic arrangement of data elements into a specific sequence, typically ascending or descending order. At its core, sorting transforms a disorganized collection of items—whether they are numbers, strings, or complex objects—into a structured permutation that facilitates efficient retrieval and analysis. The importance of sorting extends far beyond simple organization; it is a critical optimization step for other algorithms. For instance, search algorithms like Binary Search require sorted input to function, reducing search operations from linear to logarithmic time. Additionally, sorted data simplifies tasks such as finding duplicates, calculating medians, and merging datasets.

Sorting algorithms are generally categorized into comparison-based methods, which order elements by comparing them pairwise, and non-comparison-based methods, which exploit data properties like integer

NOTE

ranges. In this unit, we focus on the "Simple Sorts"—Bubble Sort, Selection Sort, and Insertion Sort. While these comparison-based algorithms share a worst-case time complexity of O(n^2), making them less suitable for massive datasets, they provide essential insights into algorithmic logic, memory management, and stability. Understanding these mechanisms is the first step toward mastering more complex "Divide and Conquer" algorithms and understanding computational efficiency.

12.2 OBJECTIVE

After Completing this unit you will be able to:

1. Define and classifications of sorting: Articulate the definition of sorting, distinguish between stable and unstable sorts, and explain the difference between in-place and out-of-place algorithms.
2. Analyze Time and Space Complexity: Evaluate the performance of Bubble, Selection, and Insertion sorts in Best, Average, and Worst-case scenarios using Big O notation.
3. Execute Bubble Sort: Implement and trace the Bubble Sort algorithm, understanding how the largest elements "bubble" to the top through repeated swapping.
4. Execute Selection Sort: Demonstrate the logic of Selection Sort by identifying minimum values and effectively minimizing memory writes compared to other simple sorts.
5. Execute Insertion Sort: Apply the adaptive logic of Insertion Sort to place elements into their correct positions relative to a sorted sublist, mimicking the way cards are sorted in hand.

12.3 SORTING

Sorting is the operation of rearranging a sequence of elements so that they adhere to a specific order, such as ascending (low to high) or descending (high to low).

In formal terms, given a list of items A = (a_1, a_2, ..., a_n), sorting transforms it into a permutation A' = (a'_1, a'_2, ..., a'_n) such that a'_1 \leq a'_2 \leq ... \leq a'_n (for ascending order).

12.3.1 Why is Sorting Important?

- Search Efficiency: Sorting is often a precursor to searching. Algorithms like Binary Search require data to be sorted first, reducing search time from linear O(n) to logarithmic O(\log n).

NOTE

- Data Analysis: It is easier to find the median, mode, or range of a dataset when it is sorted.
- Canonicalization: Comparing two lists to see if they contain the same elements is trivial if both lists are sorted.
- Optimization: Many greedy algorithms (e.g., Kruskal's algorithm for Minimum Spanning Trees) rely on sorting input data to find optimal solutions.

12.3.2. Key Terminology in Sorting

To evaluate a sorting algorithm, we look at several characteristics beyond just speed.

12.3.2.1 Time Complexity

This measures how the algorithm's running time grows as the input size (n) increases.

- Best Case: The time taken when the input is already sorted.
- Average Case: The expected time on random input.
- Worst Case: The time taken on the most "difficult" input (e.g., reverse sorted).

12.3.2.2 Space Complexity (In-place vs. Out-of-place)

- In-place: The algorithm requires only a constant amount of extra memory space (O(1)) regardless of input size. It rearranges elements within the original array. (e.g., Quick Sort, Bubble Sort).
- Out-of-place: The algorithm requires extra memory proportional to the input size (O(n)) to hold temporary data. (e.g., Merge Sort).

12.3.2.3 Stability

A sorting algorithm is stable if it preserves the relative order of equal elements. If item A and item B have the same value, and A appears before B in the original list, a stable sort ensures A appears before B in the sorted list. This is crucial when sorting complex objects (e.g., sorting Excel rows by "Last Name" then "First Name").

12.3.3. Classifications of Sorting Algorithms

Sorting algorithms generally fall into two broad categories based on how they process data.

12.3.3.1 Comparison-Based Sorting

These algorithms sort data by comparing elements pairwise (a > b). Mathematically, they cannot perform better than O(n \log n) in the average case.

NOTE

1. Bubble Sort

The simplest sorting algorithm. It repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. The pass through the list is repeated until the list is sorted.

- Mechanism: "Bubbles" the largest element to the end in each pass.
- Best for: Educational purposes or extremely small datasets.
- Complexity: O(n^2).

2. Selection Sort

This algorithm divides the input list into two parts: the sublist of items already sorted and the sublist of items remaining to be sorted. It finds the smallest element in the unsorted sublist and swaps it with the leftmost unsorted element.

- Mechanism: Selects the minimum and moves it to the front.
- Best for: Situations where memory writes are expensive (it minimizes swaps).
- Complexity: O(n^2).

3. Insertion Sort

Builds the final sorted array one item at a time. It takes an element from the unsorted portion and inserts it into its correct position within the sorted portion.

- Mechanism: Like sorting playing cards in your hand.
- Best for: Small datasets or "almost sorted" data (it is very fast, O(n), in these cases).
- Complexity: O(n^2).

4. Merge Sort

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.

- Mechanism: Recursively splits the list until elements are singletons, then merges them in order.
- Best for: Linked Lists and large datasets where stability is required.
- Complexity: O(n \log n).

NOTE

5. Quick Sort

Another Divide and Conquer algorithm. It picks an element as a pivot and partitions the given array around the picked pivot (putting smaller elements to the left, larger to the right).

- Mechanism: Partitioning is the key logic.
- Best for: General purpose in-memory sorting (often faster than Merge Sort in practice due to cache locality).
- Complexity: Average O(n \log n), Worst O(n^2).

12.3.3.2 Non-Comparison-Based Sorting

These algorithms do not compare elements. Instead, they exploit the properties of the data (e.g., knowing data are integers within a specific range). They can achieve linear time complexity O(n).

1. Counting Sort

Counts the number of objects having distinct key values. Then, it uses arithmetic to calculate the position of each object in the output sequence.

- Constraint: Works only when the range of input values (k) is not significantly larger than the number of objects (n).
- Complexity: O(n + k).

2. Radix Sort

Sorts numbers digit by digit, from the least significant digit to the most significant digit, usually using Counting Sort as a subroutine.

- Constraint: Works on integers or strings of fixed length.
- Complexity: O(nk) where k is the number of digits.

12.4 BUBBLE SORT

Bubble Sort is a comparison-based sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. This process repeats until the list is fully sorted.

It is named "Bubble Sort" because with each full pass through the list, the largest remaining element "bubbles up" to its correct position at the end of the array, much like air bubbles rising to the surface of water.

12.4.1Key Characteristics

- Type: Comparison Sort
- Method: Exchanging (Swapping)

NOTE

- Stability: Stable (Does not change the relative order of elements with equal values)
- In-place: Yes (Requires only O(1) extra memory)

12.4.2. How It Works (The Logic)

The algorithm uses a nested loop structure:

1. Outer Loop: Controls the number of passes. For a list of n elements, we need n-1 passes to ensure every element is in place.
2. Inner Loop: Iterates through the list, comparing adjacent items (index and index+1).

12.4.2.1 The Rule

For every element at index i:

- Compare element A[i] with element A[i+1].
- If A[i] > A[i+1] (the left is larger than the right), Swap them.
- If they are in the correct order, do nothing and move to the next pair.

12.4.3. Step-by-Step Example

Let's sort the array: [5, 1, 4, 2, 8]

12.4.3.1 Pass 1

- (5, 1), 4, 2, 8 \rightarrow Swap (5 > 1) \rightarrow [1, 5, 4, 2, 8]
- 1, (5, 4), 2, 8 \rightarrow Swap (5 > 4) \rightarrow [1, 4, 5, 2, 8]
- 1, 4, (5, 2), 8 \rightarrow Swap (5 > 2) \rightarrow [1, 4, 2, 5, 8]
- 1, 4, 2, (5, 8) \rightarrow No Swap (5 < 8) \rightarrow [1, 4, 2, 5, 8]
- *Result:* The largest element 8 has "bubbled" to the correct position at the end.

12.4.3.2 Pass 2

- (1, 4), 2, 5, 8 \rightarrow No Swap \rightarrow [1, 4, 2, 5, 8]
- 1, (4, 2), 5, 8 \rightarrow Swap (4 > 2) \rightarrow [1, 2, 4, 5, 8]
- 1, 2, (4, 5), 8 \rightarrow No Swap \rightarrow [1, 2, 4, 5, 8]
- *(Note: We don't check 8 again, it's already sorted.)*
- *Result:* The second largest element 5 is now in place.

12.4.3.3 Pass 3

- (1, 2), 4, 5, 8 \rightarrow No Swap \rightarrow [1, 2, 4, 5, 8]

NOTE

- 1, (2, 4), 5, 8 \rightarrow No Swap \rightarrow [1, 2, 4, 5, 8]
- *Result:* The list is actually sorted now, but the standard algorithm might continue unless optimized.

12.4.4. Pseudocode & Implementation

12.4.4.1 Standard Pseudocode

Plaintext

```
procedure bubbleSort(list)
  n = length(list)
  for i from 0 to n-1
    for j from 0 to n-i-1
      if list[j] > list[j+1]
        swap(list[j], list[j+1])
      end if
    end for
  end for
end procedure
```

12.4.5. Complexity Analysis

12.4.5.1 Time Complexity

- Worst Case (O(n^2)): Occurs when the array is reverse sorted (e.g., [5, 4, 3, 2, 1]). The algorithm must perform the maximum number of comparisons and swaps.
- Average Case (O(n^2)): For random data, elements are generally out of order and require significant swapping.
- Best Case (O(n)): Occurs when the array is already sorted (using the optimized version). The algorithm makes one pass (n comparisons), makes no swaps, and the swapped flag terminates the loop.

12.4.5.2 Space Complexity

- O(1): It is an in-place sorting algorithm. It only requires a single additional memory space for a temporary variable used during swapping.

12.4.6. Advantages and Disadvantages

12.4.6.1 Advantages

1. Simplicity: The logic is easy to understand and implement.

NOTE

2. No Extra Memory: Great for systems with very limited memory.
3. Stable Sort: Useful if you are sorting data with multiple properties (e.g., sorting a list of people by age, but keeping people of the same age in their original alphabetical order).
4. Detects Sortedness: The optimized version can detect if a list is already sorted very quickly (O(n)).

12.4.6.2 Disadvantages

1. Performance: It is incredibly slow for large datasets compared to algorithms like Quick Sort or Merge Sort.
2. Redundant comparisons: Even in the optimized version, it performs many comparisons that advanced algorithms avoid.

12.5 SELECTION SORT

Selection Sort divides the input list into two parts:

1. Sorted Sublist: Built up from left to right at the front (left) of the list.
2. Unsorted Sublist: The remaining items that occupy the rest of the list.

Initially, the sorted sublist is empty and the unsorted sublist is the entire input list. The algorithm proceeds by finding the smallest element in the unsorted sublist, exchanging (swapping) it with the leftmost unsorted element, and moving the sublist boundaries one element to the right.

Key Characteristics

- Type: Comparison Sort
- Method: Selection
- Stability: Unstable (Swapping elements over long distances can change the relative order of equal elements).
- In-place: Yes (Requires O(1) extra memory).

2. How It Works (The Logic)

The algorithm uses a nested loop structure:

1. Outer Loop: Iterates from the starting index i to n-1. This index i represents the first element of the *unsorted* partition.
2. Inner Loop: Scans the *unsorted* part of the array (from i+1 to n) to find the index of the minimum value.

The Rule

For every position i in the array:

NOTE

- Assume the element at i is the minimum (min_index = i).
- Scan all subsequent elements (j from i+1 to end).
- If you find a value smaller than the current minimum (arr[j] < arr[min_index]), update min_index to j.
- After scanning the rest of the list, if min_index is not i, Swap arr[i] with arr[min_index].

3. Step-by-Step Example

Let's sort the array: [64, 25, 12, 22, 11]

Pass 1 (i = 0):

- Current unsorted list: [64, 25, 12, 22, 11]
- Assume min is 64 (index 0).
- Compare with 25: New min is 25 (index 1).
- Compare with 12: New min is 12 (index 2).
- Compare with 22: Min is still 12.
- Compare with 11: New min is 11 (index 4).
- Action: Swap index 0 (64) with index 4 (11).
- Array: [11, 25, 12, 22, 64] (11 is now sorted).

Pass 2 (i = 1):

- Current unsorted list: [25, 12, 22, 64]
- Assume min is 25 (index 1).
- Compare with 12: New min is 12 (index 2).
- Compare with ... rest are larger.
- Action: Swap index 1 (25) with index 2 (12).
- Array: [11, 12, 25, 22, 64] (11, 12 are sorted).

Pass 3 (i = 2):

- Current unsorted list: [25, 22, 64]
- Assume min is 25.
- Compare with 22: New min is 22.
- Compare with 64: Min is still 22.
- Action: Swap index 2 (25) with index 3 (22).
- Array: [11, 12, 22, 25, 64] (Sorted part grows).

Pass 4 (i = 3):

- Current unsorted list: [25, 64]
- Assume min is 25.
- Compare with 64: Min is still 25.
- Action: min_index equals i, so no swap is needed (or swap with self).
- Array: [11, 12, 22, 25, 64].

NOTE

Pass 5: The last element (64) is automatically in the correct place.

4. Pseudocode & Implementation

Pseudocode

Plaintext

```
procedure selectionSort(list)
  n = length(list)
  for i from 0 to n - 2
    min_index = i
    for j from i + 1 to n - 1
      if list[j] < list[min_index]
        min_index = j
      end if
    end for
    if min_index != i
      swap(list[i], list[min_index])
    end if
  end for
end procedure
```

5. Complexity Analysis

Time Complexity

Selection Sort is unique among simple sorts because its best, average, and worst cases are usually identical.

- Worst Case (O(n^2)): It requires two nested loops. Even if the array is already sorted, the algorithm doesn't "know" it; it must still scan the entire remaining list to confirm that the current element is indeed the minimum.[12]
- Best Case ([34]O(n^2)): Same reason. There is no mechanism to break early (unlike Bubble Sort's flag).[56]
- Average Case ([78]O(n^2)): It always performs [910]\frac{n(n-1)}{2} comparison[11]s.[12]

Space Complexity[13]

- O(1): It is s[14]trictly an in-place algorithm, requiring only one temporary variable for swapping.

6. Advantages and Disadvantages

NOTE

Advantages

1. Minimizes Swaps: This is the biggest advantage. Selection Sort makes a maximum of n swaps. Bubble Sort can make up to O(n^2) swaps. If "writing" to memory is significantly more expensive than "reading" (e.g., in certain embedded systems or Flash memory), Selection Sort is preferable to Bubble/Insertion Sort.
2. Simple Logic: Easy to implement.

Disadvantages

1. Slow Speed: O(n^2) makes it inefficient for large lists.
2. Unstable: This is a major drawback compared to Bubble and Insertion Sort.
    - *Example of Instability:* Consider the list [2a, 2b, 1].
    - In the first pass, 1 is the minimum. It swaps with 2a.
    - Result: [1, 2b, 2a]. The relative order of the 2s has changed.
3. No Adaptive Behavior: It takes the same amount of time to sort an already sorted list as it does a random list.

Check Your Progress 12.1

MCQ:
Which of the following is a basic comparison-based sorting technique?
a) Merge Sort
b) Quick Sort
c) Bubble Sort
d) Radix Sort

True/False:
Selection sort repeatedly selects the smallest (or largest) element and places it at the correct position.

Short Questions:

1. Define sorting in computer science.

.....

.....

2. Name two basic sorting techniques and their key idea.

.....

.....

NOTE

12.6 INSERTION SORT

Insertion Sort is a comparison-based algorithm that builds a sorted array by repeatedly taking the next element from the "unsorted" portion and inserting it into its correct position within the "sorted" portion.

Imagine you are holding a hand of cards. You pick up a new card from the table, scan your current hand from right to left, and slide the new card into the correct spot. That is Insertion Sort.

12.6.1 Key Characteristics

- Type: Comparison Sort
- Method: Insertion
- Stability: Stable (Preserves the relative order of equal elements).
- In-place: Yes (Requires O(1) extra memory).
- Adaptive: Yes (It is significantly faster—O(n)—if the list is already or nearly sorted).

12.6.2. How It Works (The Logic)

The algorithm virtually splits the array into two parts:

1. Sorted Sublist: Initially contains only the first element (a single element is always sorted).
2. Unsorted Sublist: Contains the remaining elements.

The algorithm iterates through the unsorted sublist:

1. Pick: Take the first element (the "key") from the unsorted sublist.
2. Compare & Shift: Compare the "key" with elements in the sorted sublist (moving right to left). If an element is larger than the key, shift that element one position to the right.
3. Insert: Once you find a value smaller than the key (or reach the start of the list), insert the key into that empty gap.

12.6.3. Step-by-Step Example

Let's sort the array: [5, 2, 4, 6, 1, 3]

Start: [5] is the sorted part. [2, 4, 6, 1, 3] is unsorted.

Pass 1 (Key = 2):

- Compare 2 with 5.
- 5 > 2, so shift 5 to the right.
- Reached start. Insert 2 at index 0.
- Array: [2, 5, 4, 6, 1, 3]

NOTE

Pass 2 (Key = 4):

- Compare 4 with 5.
- 5 > 4, shift 5 to the right.
- Compare 4 with 2.
- 2 < 4, stop. Insert 4 after 2.
- Array: [2, 4, 5, 6, 1, 3]

Pass 3 (Key = 6):

- Compare 6 with 5.
- 5 < 6, stop immediately. No shifts needed.
- Array: [2, 4, 5, 6, 1, 3]

Pass 4 (Key = 1):

- Compare 1 with 6 \rightarrow shift 6.
- Compare 1 with 5 \rightarrow shift 5.
- Compare 1 with 4 \rightarrow shift 4.
- Compare 1 with 2 \rightarrow shift 2.
- Reached start. Insert 1 at index 0.
- Array: [1, 2, 4, 5, 6, 3]

Pass 5 (Key = 3):

- Shift 6, shift 5, shift 4.
- Stop at 2 (since 2 < 3). Insert 3.
- Array: [1, 2, 3, 4, 5, 6]

12.6.4. Pseudocode & Implementation

12.6.4.1 Pseudocode

Plaintext

```
procedure insertionSort(list)
  n = length(list)
  for i from 1 to n-1
    key = list[i]
    j = i - 1

    // Move elements of list[0..i-1] that are greater than key
    // to one position ahead of their current position
```

NOTE

```
    while j >= 0 and list[j] > key
      list[j + 1] = list[j]
      j = j - 1
    end while

    list[j + 1] = key
  end for
end procedure
```

12.6.5. Complexity Analysis

12.6.5.1 Time Complexity

- Best Case (O(n)): This occurs when the array is already sorted. The inner while loop condition (key < arr[j]) fails immediately for every element, resulting in constant time comparisons per pass.
- Worst Case (O(n^2)): This occurs when the array is reverse sorted. Every element must be compared with and shifted past every element in the sorted portion.
- Average Case (O(n^2)): On random data, you typically shift half of the sorted subarray.

12.6.5.2 Space Complexity

- O(1): Strictly in-place. Only one variable (key) is needed for storage.

12.6.6. Advantages and Disadvantages

12.6.6.1 Advantages

1. Adaptive: It is extremely efficient for small datasets or data that is already "mostly sorted" (e.g., adding a few new entries to a sorted database).
2. Stable: It preserves the order of duplicates.
3. Online: It can sort a list as it receives it. You don't need all the data upfront; you can insert elements one by one as they arrive.
4. Low Overhead: For very small arrays (e.g., n < 50), Insertion Sort is often faster than Quick Sort or Merge Sort because it has no recursion overhead or complex partition logic.

12.6.6.2 Disadvantages

1. Scalability: It is very slow for large datasets (O(n^2)).
2. High Write Volume: It performs many writes (shifts) to memory, though usually fewer than Bubble Sort's swaps.

NOTE

Activity 1: The "Algorithm Face-Off" – Bubble vs. Selection Sort

Objective:

To implement and compare two fundamental O(n^2) sorting algorithms: Bubble Sort and Selection Sort. The goal is not just to sort data, but to instrument the code to "keep score" (count comparisons and swaps) to understand the subtle but important efficiency differences between them.

Materials / Tools / Software Required:

- Computer: Standard development setup.
- Programming Language: Python (easy to read) or C++ (good for performance timing).
- Dataset: An array of 500 random integers (generated via code).
- Stopwatch/Timer: (Optional) to measure execution time for larger arrays (e.g., 10,000 elements).

Expected Outcome:

By the end of this activity, students will be able to:

1. Implement Bubble Sort: Write the nested loop structure that "bubbles" the largest element to the end by repeatedly swapping adjacent elements.
2. Implement Selection Sort: Write the logic that scans the list to find the minimum value and performs a single swap per pass.
3. Instrument the Code: Add integer counters (swap count, comparison count) inside the loops of both algorithms.
4. Analyze the Data: Run both algorithms on the exact same dataset. They will discover that while both perform roughly the same number of *comparisons* (O(n^2)), Selection Sort performs significantly fewer *swaps* (max O(n)) compared to Bubble Sort, making it faster in practice for writing to memory.

12.7 SUMMARY

1. In this unit, we explored the mechanics and applications of three fundamental sorting algorithms: Bubble Sort, Selection Sort, and Insertion Sort. We began by establishing that sorting is the process of rearranging a sequence of elements, such as A = (a_1, a_2, ..., a_n), into a specific order. We defined key criteria for evaluating these algorithms, specifically Time Complexity (how speed correlates with input size), Space Complexity (memory

NOTE

usage), and Stability (preservation of the relative order of equal elements).

2. We first examined Bubble Sort, the simplest algorithm, which works by repeatedly stepping through the list, comparing adjacent elements, and swapping them if they are in the wrong order. We learned that while it is stable and easy to implement, it is inefficient for large lists with a time complexity of O(n^2). However, an optimized version can detect an already sorted list in O(n) time.
3. Next, we covered Selection Sort. Unlike Bubble Sort, Selection Sort reduces the number of swaps by scanning the unsorted portion of the list to find the absolute minimum element and placing it at the beginning. While this effectively minimizes memory writes—making it useful for systems where writing to memory is expensive—it is generally an unstable algorithm and retains an O(n^2) complexity even in the best case.
4. Finally, we discussed Insertion Sort, an adaptive algorithm that builds the final sorted array one item at a time. By taking an element and shifting sorted items to the right to make space, Insertion Sort performs exceptionally well on small or partially sorted datasets, achieving a best-case time of O(n). It combines the stability of Bubble Sort with superior practical performance, often making it the preferred choice among the O(n^2) algorithms for real-world scenarios involving small data chunks.

12.8 GLOSSARY

1. Sorting: The algorithmic process of arranging data in a specific order, such as numerical or lexicographical.
2. Stability: A property of a sorting algorithm where elements with equal values retain their original relative order in the sorted output.
3. In-place: An algorithm that transforms input using no auxiliary data structures, requiring only a small, constant amount of extra memory space (O(1)).
4. Time Complexity: A computational concept that quantifies the amount of time an algorithm takes to run as a function of the length of the input.
5. Bubble Sort: A sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order.

NOTE

6. Selection Sort: A sorting algorithm that divides the input into sorted and unsorted regions, repeatedly selecting the smallest element from the unsorted region to add to the sorted region.
7. Insertion Sort: A sorting algorithm that builds the final sorted array one item at a time by inserting elements into their correct position within a sorted sublist.
8. Adaptive: A property of a sorting algorithm (like Insertion Sort) where the efficiency improves significantly if the input data is already substantially sorted.
9. Swap: The operation of exchanging the positions of two data elements in memory.
10. Pass: A complete iteration through the dataset or a portion of the dataset during the execution of a sorting algorithm.

12.9 ANSWER TO CHECK YOUR PROGRESS-POSSIBLE ONE

Check Your Progress 12.1

MCQ Answer:
c) Bubble Sort – it is a basic comparison-based sorting technique.

True/False Answer:
True – Selection sort selects the minimum/maximum and places it correctly.

Short Answers:

1. Sorting is the process of arranging data in a specific order (ascending or descending).
2. Two basic techniques: Bubble Sort (compare and swap adjacent elements), Selection Sort (select min/max each pass).

12.10 SUGGESTED READINGS – OER

Sr. No.	Title	URL	QR Code
1	Data Structure Using C++	http://books.aisectonline.com/User/UnitUrls?unitId=3720	

NOTE

12.11 REFERENCE

- Introduction to Algorithms – Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein – ISBN: 9780262046305 – 2022 (4th Edition)
- Data Structures and Algorithms in C++ – Adam Drozdek – ISBN: 9781133608426 – 2012
- Algorithms – Robert Sedgewick, Kevin Wayne – ISBN: 9780321573513 – 2011
- Data Structures and Algorithm Analysis in C++ – Mark Allen Weiss – ISBN: 9780132847377 – 2013
- Data Structures and Algorithms 1: Sorting and Searching – Kurt Mehlhorn – ISBN: 9783642696725 – 2013
- Sorting Algorithms and Techniques: Definitive Reference for Developers and Engineers – Richard Johnson – ISBN: (likely B0FFJTHGQW e-book) – 2025
- Algorithms Unlocked – Thomas H. Cormen – ISBN: 9780262518802 – 2013
- Efficient Algorithm Design – (Various Authors) – ISBN: 9781835886823 – 2021

12.12 EXERCISE

Part 1

A. Multiple choice question

1. Which of the following sorting algorithms is NOT stable?

A) Bubble Sort

B) Insertion Sort

C) Selection Sort

D) Merge Sort

2. What is the Best Case time complexity of the optimized Bubble Sort?

A) O(1)

B) O(n)

C) O(n \log n)

D) O(n^2)

Answer: B) O(n)

3. Which algorithm is generally preferred when memory writes (swaps) are very expensive?

A) Selection Sort

NOTE

B) Insertion Sort

C) Bubble Sort

D) Quick Sort

4. In Insertion Sort, what happens in the Worst Case scenario (reverse sorted data)?

A) No swaps are performed.

B) Each element is compared only once.

C) Every element must be compared and shifted past every element in the sorted portion.

D) The algorithm finishes in O (n \log n) time.

5. Which of the following best describes the "In-place" property?

A) The algorithm requires extra memory proportional to the input size (O (n)).

B) The algorithm preserves the relative order of equal elements.

C) The algorithm requires only a constant amount of extra memory (O (1)).

D) The algorithm sorts the list in descending order.

Answers with Explanations:

1. Answer: C) Selection Sort
Explanation: Selection Sort may swap non-adjacent elements, which can change the relative order of equal elements, making it unstable.

2. Answer: B) O(n)
Explanation: If the array is already sorted, the optimized Bubble Sort can detect no swaps are needed and complete in linear time.

3. Answer: A) Selection Sort
Explanation: Selection Sort minimizes the number of swaps compared to Bubble or Insertion Sort, making it suitable when swaps are costly.

4. Answer: C) Every element must be compared and shifted past every element in the sorted portion.
Explanation: In reverse order, each new element has to move past all already sorted elements, resulting in $O(n^2)$ comparisons and shifts.

5. Answer: C) The algorithm requires only a constant amount of extra memory (O(1)).
Explanation: An in-place algorithm sorts the data using only a fixed, small amount of additional memory, without allocating arrays proportional to input size.

NOTE

B. Small answer type question

1. Define Stability in sorting. Explain why it might be important when sorting a database of students first by grade, then by name.
2. Explain the core logic of Selection Sort. How does it differ from Bubble Sort in terms of how it handles element movement?
3. Describe the Best Case scenario for Insertion Sort. Why does the algorithm achieve O(n) time complexity in this specific situation?
4. Compare the Space Complexity of Merge Sort and Bubble Sort. Why is Bubble Sort considered an "in-place" algorithm?
5. What is the primary disadvantage of Bubble Sort? despite its simplicity, why is it rarely used for large, real-world datasets?

Part 2

C. Long answer type question (answer in 200 words)

1. Trace the execution of Bubble Sort. Given the array [5, 1, 4, 2, 8], describe step-by-step how the first pass is completed. Explain the concept of "bubbling" based on this example.
2. Analyze the "Adaptive" nature of Insertion Sort. Explain how the algorithm behaves when a new element is added to an already sorted list. Why makes this specific property useful for "Online" sorting where data arrives in streams?
3. Contrast Selection Sort and Insertion Sort. While both have O(n^2) worst-case complexity, they excel in different scenarios. Discuss one scenario where Selection Sort is better and one where Insertion Sort is better.
4. Discuss the importance of Sorting in Computer Science. Beyond just organizing a list, explain how sorting optimizes other algorithms (like Search) and aids in data analysis (finding modes/medians).
5. Explain the concept of Time Complexity in the context of Simple Sorts. Why do Bubble, Selection, and Insertion sorts all fall into the O(n^2) category for average cases? What does this quadratic growth mean for processing time as data size doubles?

Part 3

D. Long answer type question (answer in 300 words)

1. Detailed Algorithm Walkthrough: Selection Sort. Provide a complete, pass-by-pass explanation of sorting the array [64, 25, 12, 22, 11] using Selection Sort. For each pass, identify the current minimum, the swap that occurs, and the state of the sorted vs. unsorted sublists.

NOTE

2. Comprehensive Comparison of Simple Sorting Algorithms. Create a structured detailed comparison of Bubble Sort, Selection Sort, and Insertion Sort. Your answer must cover: Time Complexity (Best/Avg/Worst), Space Complexity, Stability, and the specific mechanism (Swapping vs. Selection vs. Insertion) used by each.
3. Pseudocode Analysis. Write the pseudocode for Insertion Sort and explain it line-by-line. specifically focus on the while loop that performs the shifting operation. Explain how this loop contributes to both the stability of the algorithm and its worst-case time complexity.
4. Optimizing Bubble Sort. Standard Bubble Sort is inefficient even on sorted lists. Explain the logic behind the "Optimized Bubble Sort" using a swapped flag. Provide a logic flow or pseudocode for this optimization and explain mathematically how it reduces the Best Case complexity to Linear Time O(n).
5. Real-world Application Scenarios. You are a software engineer choosing a sorting algorithm for three different modules. Justify which algorithm (Bubble, Selection, Insertion, or Merge) you would choose for:
    - A small list of 10 items in a microcontroller with very limited write-cycles.
    - A live feed of sports scores where new scores arrive one by one and need to be displayed instantly.
    - A massive database of 1 million user records that needs to be sorted nightly

UNIT 13
ADVANCED SORTING TECHNIQUES AND COMPARISON

NOTE

UNIT STRUCTURE

13.1 Introduction

13.2 Objective

13.3 Merge Sort

13.4 Quick Sort

13.5 Shell Sort

13.6 Address Calculation Sort

13.7 Radix Sort

13.8 Summary

13.9 Glossary

13.10 Answer to check your progress

13.11 Suggested readings - OER

13.12 Reference

13.13 Exercise

13.1 INTRODUCTION

Sorting is a fundamental operation in computer science that involves arranging data elements in a specific order, usually ascending or descending. Efficient sorting is essential for improving the performance of searching, data processing, and decision-making tasks in computer applications. As datasets grow larger and more complex, basic sorting techniques become inefficient, creating the need for advanced sorting algorithms.

This unit focuses on advanced sorting techniques such as Merge Sort, Quick Sort, Shell Sort, Address Calculation Sort (Bucket/Hash-based Sort), and Radix Sort. These algorithms are designed to handle large datasets efficiently by using strategies like divide and conquer, partitioning, gap sequences, hashing, and digit-wise processing. Each algorithm has unique characteristics in terms of time complexity, space usage, stability, and suitability for different types of data.

NOTE

In addition to understanding individual algorithms, this unit emphasizes the comparison of sorting techniques to help select the most appropriate method based on application requirements such as speed, memory constraints, data size, and data type. Mastery of these advanced sorting techniques is crucial for designing efficient software systems, databases, and real-time applications.

13.2 OBJECTIVE

After completing this unit, you will be able to:

1 Understand the need for advanced sorting techniques over basic sorting methods.
2 Explain the working principles of Merge Sort, Quick Sort, and Shell Sort.
3 Analyze non-comparison sorting methods such as Address Calculation Sort and Radix Sort.
4 Compare sorting algorithms based on time complexity, space complexity, stability, and performance.
5 Select an appropriate sorting technique for different real-world applications and data conditions.

13.3 MERGE SORT

Merge Sort is one of the most efficient and popular sorting algorithms used in computer science. It is a Divide and Conquer algorithm that sorts an array by breaking it down into smaller, manageable subarrays, sorting those subarrays, and then merging them back together.

It was invented by John von Neumann in 1945 and is highly respected for its predictable performance, as it guarantees O(n \log n) time complexity in the worst-case scenario, unlike Quicksort which can degrade to O(n^2).

13.3.1 Core Principle: Divide and Conquer

Merge Sort operates on three main steps:

1. Divide: The algorithm recursively divides the input array into two halves until it reaches the base case—subarrays containing a single element (which are inherently sorted).
2. Conquer: The sort happens implicitly. Once the array is broken down to single elements, the "merging" process begins.
3. Combine (Merge): The algorithm repeatedly merges the sorted subarrays back together to produce new, larger sorted subarrays until a single sorted array remains.

13.3.2 How Merge Sort Works: Step-by-Step

NOTE

To understand the mechanics, imagine you have a deck of cards. Instead of trying to sort the whole deck at once, you split the deck in half, then split those halves, and so on, until you have piles of just one card. Then, you pick up two piles, compare the cards, and place them in order. You repeat this, merging larger and larger piles until the deck is whole again—and sorted.

13.3.2.1 The Merge Procedure (The "Heart" of the Algorithm)

The magic of Merge Sort lies in the merge() function. This function takes two sorted subarrays and combines them into a single sorted array.

Logic:

1. Create pointers for the beginning of both subarrays (Left and Right).
2. Compare the elements at the current pointers.
3. Place the smaller element into a new "result" array.
4. Move the pointer of the chosen element forward.
5. Repeat until one subarray is empty.
6. Append any remaining elements from the non-empty subarray to the result.

13.3.3 Visual Walkthrough

Let's sort the following array:

[38, 27, 43, 3, 9, 82, 10]

13.3.3.1 Phase 1: Splitting (Divide)

We repeatedly find the midpoint and split the array.

1. [38, 27, 43, 3] and [9, 82, 10]
2. [38, 27], [43, 3] ... [9, 82], [10]
3. Eventually, we get single elements: [38], [27], [43], [3], [9], [82], [10]

13.3.3.2 Phase 2: Merging (Conquer & Combine)

Now we merge pairs of subarrays in sorted order.

1. Merge 1-element arrays:
    - Compare 38 & 27 \rightarrow [27, 38]
    - Compare 43 & 3 \rightarrow [3, 43]
    - Compare 9 & 82 \rightarrow [9, 82]
    - [10] waits.
2. Merge 2-element arrays:

NOTE

- Merge [27, 38] and [3, 43]:
  - Compare 27 vs 3 (Pick 3)
  - Compare 27 vs 43 (Pick 27)
  - Compare 38 vs 43 (Pick 38)
  - Remaining: 43
  - Result: [3, 27, 38, 43]
- Merge [9, 82] and [10]:
  - Result: [9, 10, 82]

3. Final Merge:
   - Merge [3, 27, 38, 43] and [9, 10, 82]
   - Final Result: [3, 9, 10, 27, 38, 43, 82]

13.3.4 Pseudocode Implementation

Here is the high-level logic often used in implementation.

Plaintext

```
FUNCTION MergeSort(array)
   IF length of array <= 1
      RETURN array

   mid = length of array / 2
   leftHalf = MergeSort(array[0...mid])
   rightHalf = MergeSort(array[mid...end])

   RETURN Merge(leftHalf, rightHalf)

FUNCTION Merge(left, right)
   result = []
   WHILE left is not empty AND right is not empty
      IF left[0] <= right[0]
         APPEND left[0] TO result
         REMOVE left[0] FROM left
      ELSE
         APPEND right[0] TO result
```

REMOVE right[0] FROM right

APPEND remaining elements of left TO result

APPEND remaining elements of right TO result

RETURN result

13.3.5 Complexity Analysis

13.3.5.1 Time Complexity

Merge Sort is unique because its time complexity is consistent across all cases.

- Best Case: O(n \log n)
- Average Case: O(n \log n)
- Worst Case: O(n \log n)

Why?

The list is always divided into two halves, which takes logarithmic time (\log n levels of division). At each level, we iterate through the 1n elements to merge them. Therefore, the total work is 2n \times \log n.3

13.3.5.2 Space Complexity[4]

- Space: 5O(n)

  Merge sort is not an in-place sorting algorithm (for arrays). It requires additional memory to store the temporary suba6rrays during the merge process. This is its primary disadvantage compared to QuickSort or HeapSort.

13.3.6 Real-World Applications

1. Sorting Linked Lists: Merge Sort is the preferred algorithm for sorting linked lists. Unlike arrays, linked lists do not support random access, making QuickSort inefficient. Merge Sort accesses data sequentially, which matches how linked lists work.
2. External Sorting: When data is too large to fit in memory (RAM)—e.g., sorting a 100GB file with only 8GB RAM—Merge Sort is used. The data is broken into "chunks," sorted, and then merged back together from the disk.
3. Inversion Count: It is used in problems where you need to count "inversions" in an array (how far an array is from being sorted).

13.4 QUICK SORT

NOTE

Quick Sort is widely regarded as one of the most efficient sorting algorithms for general-purpose use. Like Merge Sort, it utilizes the Divide and Conquer strategy, but it approaches the problem differently. Instead of splitting the array down the middle, Quick Sort splits the array based on values.

Invented by Sir Tony Hoare in 1959, it is the standard sorting algorithm in many system libraries (like C++'s std::sort often uses a variation of it) because it is typically faster in practice than other O(n \log n) algorithms due to efficient memory usage and cache locality.

13.4.1 Core Principle: Partitioning

The engine that drives Quick Sort is the process of Partitioning.

1. Pick a Pivot: Select one element from the array to act as a reference point (the "pivot").
2. Partition: Reorder the array so that all elements smaller than the pivot come before it, and all elements greater than the pivot come after it. The pivot is now in its final, correct sorted position.
3. Recourse: Apply the above steps recursively to the sub-array of smaller elements and the sub-array of larger elements.

13.4.2 How Quick Sort Works: Step-by-Step

The crucial decision in Quick Sort is how to choose the pivot. Common strategies include:

- Always picking the first element.
- Always picking the last element (used in the example below).
- Picking a random element (helps avoid worst-case scenarios).
- Median-of-three: Picking the median of the first, middle, and last elements.

13.4.2.1 The Partition Procedure

This is where the sorting happens.

1. We choose the last element as the Pivot.
2. We set a pointer (let's call it I) to track the boundary of the "smaller than pivot" section.
3. We scan the array with another pointer (j).
4. If array[j] is smaller than the Pivot, we swap array[I] and array[j] and move the i boundary forward.
5. Finally, we swap the Pivot into the i position (right between the small and large numbers).

NOTE

13.4.3 Visual Walkthrough

Let's sort the array: [10, 80, 30, 90, 40, 50, 70]

Step 1: Partitioning the full array

- Pivot: 70 (Last element)
- Compare 10 vs 70: Smaller. (Keep it left).
- Compare 80 vs 70: Larger. (Leave it).
- Compare 30 vs 70: Smaller. (Swap 30 with the first "large" element 80).
    - Array is now: [10, 30, 80, 90, 40, 50, 70]
- Compare 90 vs 70: Larger. (Leave it).
- Compare 40 vs 70: Smaller. (Swap 40 with 80).
    - Array is now: [10, 30, 40, 90, 80, 50, 70]
- Compare 50 vs 70: Smaller. (Swap 50 with 90).
    - Array is now: [10, 30, 40, 50, 80, 90, 70]
- Final Step: Place Pivot 70 in its correct spot (between smaller 50 and larger 80).
    - Result: [10, 30, 40, 50, **70**, 90, 80]

Step 2: Recursive Calls

Now, 70 is locked in place. We recursively apply the same logic to the left sub-array [10, 30, 40, 50] and the right sub-array [90, 80].

- Left Side: Depending on the pivot, it will quickly see these are already sorted relative to each other.
- Right Side ([90, 80]): Pivot is 80. 90 is larger. Pivot moves to front.
    - Result: [80, 90]

Final Sorted Array: [10, 30, 40, 50, 70, 80, 90]

13.4.4 Pseudocode Implementation

Here is the standard logic using the "Lomuto partition scheme" (pivot is the last element).

Plaintext

```
FUNCTION QuickSort(array, low, high)
  IF low < high
    // partitionIndex is the index where the pivot is now placed
    partitionIndex = Partition(array, low, high)
```

NOTE

```
        // Recursively sort elements before and after partition
        QuickSort(array, low, partitionIndex - 1)
        QuickSort(array, partitionIndex + 1, high)


FUNCTION Partition(array, low, high)
    pivot = array[high]
    i = (low - 1)  // Index of smaller element


    FOR j = low TO high - 1
        IF array[j] < pivot
            i = i + 1
            SWAP array[i] WITH array[j]


    SWAP array[i + 1] WITH array[high]
    RETURN (i + 1)
```

13.4.5 Complexity Analysis

13.4.5.1 Time Complexity

Quick Sort is sensitive to the choice of pivot.

- Best Case: O(n \log n)
    - Occurs when the pivot always divides the array into two nearly equal halves.
- Average Case: O(n \log n)
    - In most real-world random data, this is the performance.
- Worst Case: O(n^2)
    - Occurs when the pivot is always the smallest or largest element (e.g., trying to sort an already sorted array using the last element as a pivot). This creates a "skewed" tree where one side is empty.

13.4.5.2 Space Complexity

- Space: O(\log n)
    - Quick Sort is an in-place sort (it doesn't need a second array like Merge Sort).

NOTE

- However, it uses stack space for recursion. In the worst case, this stack can grow to O(n), but with optimization, it is O(\log n).

13.5 SHELL SORT

Shell Sort is a highly efficient, in-place comparison sort. It is essentially an optimized version of Insertion Sort.

While Insertion Sort is efficient for small datasets or nearly sorted arrays, it becomes very slow for large, unsorted lists because it only moves elements one position at a time. If a small element is at the far right end of the array, it takes many steps to move it to its correct place on the left.

Shell Sort solves this by allowing the exchange of items that are far apart. It was invented by Donald Shell in 1959, making it one of the oldest sorting algorithms still in use.

Core Principle: Gap Sequences

The unique feature of Shell Sort is the Gap.

1. Instead of comparing immediate neighbors (like Insertion Sort), Shell Sort compares elements separated by a specific "gap" distance.
2. It sorts these "sub-lists" of elements.
3. The gap is then reduced (usually divided by 2), and the process repeats.
4. The final pass always uses a gap of 1, which is equivalent to a standard Insertion Sort. However, by this time, the array is guaranteed to be "nearly sorted," ensuring the final pass is extremely fast.

13.5.1 How Shell Sort Works: Step-by-Step

Imagine arranging a messy bookshelf. Insertion sort works by taking a book and sliding it slowly past every other book. Shell sort works by looking at the 1st, 5th, and 10th books and sorting just those three. Then it looks at the 2nd, 6th, and 11th. By doing this coarse sorting first, books move huge distances quickly to get *roughly* to the right section.

The Mechanism

1. Initialize Gap: Start with a large gap (often n/2).
2. Compare & Swap: Compare elements at index and index + gap. If they are in the wrong order, swap them.

NOTE

3. Slide: Continue this for all elements in the array for the current gap.
4. Shrink Gap: Reduce the gap and repeat.
5. Finish: When the gap reaches 1, perform the final polish.

13.5.2 Visual Walkthrough

Let's sort the array: [23, 12, 1, 8, 34, 54, 2, 3]

Length (n) = 8

Pass 1: Gap = 4 (n/2)

We compare elements that are 4 spaces apart:

- Index 0 (23) vs Index 4 (34): Order is correct (23 < 34). No swap.
- Index 1 (12) vs Index 5 (54): Order is correct.
- Index 2 (1) vs Index 6 (2): Order is correct.
- Index 3 (8) vs Index 7 (3): WRONG. Swap 8 and 3.

Array after Pass 1: [23, 12, 1, 3, 34, 54, 2, 8]

(Notice how the small number 3 jumped halfway across the array in a single move).

Pass 2: Gap = 2 (Previous gap / 2)

We compare elements 2 spaces apart (0 vs 2, 1 vs 3, etc.):

- 23 vs 1: Swap. \rightarrow [1, 12, 23, 3, 34, 54, 2, 8]
- 12 vs 3: Swap. \rightarrow [1, 3, 23, 12, 34, 54, 2, 8]
- 23 vs 34: Correct.
- 12 vs 54: Correct.
- 34 vs 2: Swap. \rightarrow [1, 3, 23, 12, 2, 54, 34, 8]
    - *Check backward:* Now we check the new 2 against 23 (previous in gap). Swap.
- 54 vs 8: Swap.
    - *Check backward:* Check new 8 against 12. Swap.

Array after Pass 2: [1, 3, 2, 8, 23, 12, 34, 54]

(The array is now "nearly sorted".)

Pass 3: Gap = 1 (Previous gap / 2)

This is a standard Insertion Sort.

- We iterate through. 1, 3 are fine.
- 3 vs 2: Swap.

NOTE

- 23 vs 12: Swap.
- Everything else is in order.

Final Sorted Array: [1, 2, 3, 8, 12, 23, 34, 54]

Pseudocode Implementation

Here is the logic using Shell's original sequence (halving the gap).

Plaintext

```
FUNCTION ShellSort(array)
   n = length of array
   gap = n / 2

   // Start with a big gap, then reduce the gap
   WHILE gap > 0

      // Do a gapped insertion sort for this gap size.
      // The first gap elements a[0..gap-1] are already in gapped order
      // keep adding one more element until the entire array is gap sorted
      FOR i = gap TO n - 1

         temp = array[i]
         j = i

         // Shift earlier gap-sorted elements up until the correct
         // location for array[i] is found
         WHILE j >= gap AND array[j - gap] > temp
            array[j] = array[j - gap]
            j = j - gap

         // Put temp (the original a[i]) in its correct location
         array[j] = temp

      gap = gap / 2 // Reduce gap
```

NOTE

RETURN array

13.5.3 Complexity Analysis

13.5.3.1 Time Complexity

The complexity of Shell Sort is heavily dependent on the Gap Sequence used. It is notoriously difficult to analyze mathematically.

- Worst Case (Shell's Sequence - N/2): O(n^2)
    - This happens if the "evens" and "odds" don't mix until the very last step.
- Worst Case (Hibbard's Sequence - 2^k - 1): O(n^{3/2})
- Best Case: O(n \log n)
    - Occurs when the array is already sorted.
- Average Case: Generally considered around O(n^{1.25}) or O(n^{1.5}).

13.5.3.2 Space Complexity

- Space: O(1)
    - It is strictly an in-place algorithm. No recursion stack (unlike Quick Sort) and no auxiliary arrays (unlike Merge Sort).

Gap Sequences (The Secret Sauce)

The efficiency of Shell Sort is entirely dictated by how you shrink the gap.

1. Shell's Original Sequence: N/2, N/4, \dots, 1
    - Simple to write, but not the most efficient.
2. Knuth's Sequence: h = 3h + 1 (\dots, 121, 40, 13, 4, 1)
    - Generally performs better than Shell's original sequence.
3. Hibbard's Sequence: 2^k - 1 (1, 3, 7, 15, \dots)
    - Mathematical guarantees that worst-case is better than O(n^2).

Real-World Applications

1. Embedded Systems: Because the code is tiny and it doesn't use the "Call Stack" (recursion) or dynamic memory allocation (heap), it is very safe for critical embedded devices (e.g., the code running your car's ECU).
2. uClibc Library: The C library used in many Linux embedded systems uses Shell Sort for its qsort implementation when memory is tight.

NOTE

3. Compression Compressors: Some compression algorithms use Shell Sort as a pre-processing step because it is "good enough" and very light on resources.

13.6 ADDRESS CALCULATION SORT

Address Calculation Sort (often referred to as Hashing Sort or a variation of Bucket Sort) is a non-comparison-based sorting algorithm. Unlike Merge Sort or Quick Sort, which rely on comparing items to each other (A > B), this algorithm uses the value of the item itself to calculate its position (address) in the final sorted list.

It works similarly to how a mail carrier sorts mail: looking at the zip code (address) and tossing the letter into the corresponding bin.

Core Principle: Hashing

The algorithm relies on a Hash Function (or Address Function).

1. Map: A function f(x) maps every input element to a specific "address" (index) in an auxiliary array of pointers (often called buckets or sub-lists).
2. Distribute: Elements are placed into these buckets. If a collision occurs (two items map to the same bucket), they are usually inserted in sorted order (like a mini-Insertion Sort within the bucket).
3. Collect: Finally, the algorithm walks through the buckets in order and concatenates the elements to form the sorted list.

13.6.1 How Address Calculation Sort Works: Step-by-Step

The efficiency of this sort depends heavily on the "Address Function." A good function spreads data uniformly across the buckets. A common function for numbers is:

f(x) = \lfloor \frac{x - \min}{\max - \min} \times (n - 1) \rfloor

(Where n is the number of buckets).

The Mechanism

1. Create Buckets: Initialize an array of empty sub-lists (buckets).
2. Calculate Address: For each element in the input array, calculate its bucket index.
3. Insert & Sort: Insert the element into the calculated bucket. To keep things efficient, most implementations keep the buckets sorted *as* items are added (using Insertion Sort logic).
4. Concatenate: Iterate through all buckets and join them into a single array.

NOTE

13.6.2 Visual Walkthrough

Let's sort the array: [29, 12, 45, 18, 32, 21, 48]

- Max value: 48
- Min value: 12 (approx, let's assume range 0-50 for simplicity)
- Buckets: 5 buckets (indices 0 to 4)
- Address Function: f(x) = \lfloor x / 10 \rfloor (Simplest version: dividing by 10)

13.6.2.1 Phase 1: Distribution (Hashing)

1. Item 29: 29/10 = 2. Insert into Bucket 2.
    - Bucket 2: [29]
2. Item 12: 12/10 = 1. Insert into Bucket 1.
    - Bucket 1: [12]
3. Item 45: 45/10 = 4. Insert into Bucket 4.
    - Bucket 4: [45]
4. Item 18: 18/10 = 1. Insert into Bucket 1.
    - Collision! Insert 18 into [12]. 18 > 12, so it goes after.
    - Bucket 1: [12, 18]
5. Item 32: 32/10 = 3. Insert into Bucket 3.
    - Bucket 3: [32]
6. Item 21: 21/10 = 2. Insert into Bucket 2.
    - Collision! Insert 21 into [29]. 21 < 29, so it goes before.
    - Bucket 2: [21, 29]
7. Item 48: 48/10 = 4. Insert into Bucket 4.
    - Collision! Insert 48 into [45]. 48 > 45, so it goes after.
    - Bucket 4: [45, 48]

State of Buckets:

- Bucket 0: []
- Bucket 1: [12, 18]
- Bucket 2: [21, 29]
- Bucket 3: [32]
- Bucket 4: [45, 48]

13.6.2.2 Phase 2: Concatenation (Collection)

We simply visit the buckets in order and list the contents.

[] + [12, 18] + [21, 29] + [32] + [45, 48]

Final Sorted Array: [12, 18, 21, 29, 32, 45, 48]

Pseudocode Implementation

Plaintext

```
FUNCTION AddressCalculationSort(array)
    n = length of array
    Create buckets[0...n-1] (array of empty lists)

    // Step 1: Find Range (to normalize addresses)
    maxVal = Maximum value in array
    minVal = Minimum value in array
    range = maxVal - minVal

    // Step 2: Distribute
    FOR each element 'x' in array:
        // Calculate normalized address (0 to n-1)
        address = Floor( ((x - minVal) / range) * (n - 1) )

        // Insert 'x' into buckets[address] in sorted order
        InsertSorted(buckets[address], x)

    // Step 3: Collect
    result = []
    FOR each bucket in buckets:
        Append all elements of bucket to result

    RETURN result

FUNCTION InsertSorted(bucket, value)
    // Standard insertion sort into a small list
    Iterate through bucket, find correct spot, insert value.
```

13.6.3 Complexity Analysis

NOTE

NOTE

13.6.3.1 Time Complexity

The performance depends entirely on the data distribution.

- Best/Average Case: O(n)
    - If the data is Uniformly Distributed, elements land in different buckets. Each insertion takes constant time O(1). We do this n times.
- Worst Case: O(n^2)
    - If the address function is poor (or data is skewed), all elements might land in one single bucket. Since we use Insertion Sort inside the bucket, the complexity degrades to that of Insertion Sort (O(n^2)).

13.6.3.2 Space Complexity

- Space: O(n)
    - We need memory to store the buckets (linked lists or dynamic arrays) which inevitably hold all n items.

Check Your Progress 13.1

MCQ:
Which sorting algorithm divides the array into halves, sorts them, and then merges the sorted halves?
a) Quick Sort
b) Merge Sort
c) Bubble Sort
d) Selection Sort

True/False:
Quick Sort always guarantees O(n log n) time complexity in all cases.

Short Questions:

1. Define Merge Sort and its main idea.

.....

.....

2. Name two differences between Merge Sort and Quick Sort.

.....

.....

13.7 RADIX SORT

Radix Sort is a unique, non-comparative sorting algorithm. While algorithms like Quick Sort or Merge Sort work by comparing two items

NOTE

to see which is bigger (A > B), Radix Sort works by processing the individual digits (or characters) of the numbers.

It dates back to 1887 (Herman Hollerith) for use on tabulating machines (punch cards). It is exceptionally fast for integers and fixed-length strings because it avoids the mathematical overhead of direct comparison.

Core Principle: Digit-by-Digit Processing

Radix sort exploits the fact that information is often represented in "positional notation" (like units, tens, hundreds).

1. Bucketing: It places numbers into buckets (0-9) based on a specific digit.
2. Stable Sub-routine: It typically uses a stable sort (like Counting Sort) to sort digits from the Least Significant Digit (LSD) to the Most Significant Digit (MSD).
3. Layering: By the time you reach the most significant digit, the stability of the previous steps ensures the entire number is sorted.

13.7.1 How Radix Sort Works: Step-by-Step

The most common implementation is LSD (Least Significant Digit) Radix Sort.

13.7.1.1 The Mechanism

1. Find the Max: Identify the largest number to know how many digits (passes) are needed.
2. Pass 1 (Units): Sort the entire array based only on the units digit (0-9).
3. Pass 2 (Tens): Re-sort the array based only on the tens digit. *Crucially, this must be a stable sort so we don't undo the work of Pass 1.*
4. Pass N: Continue until the most significant digit is handled.

13.7.2 Visual Walkthrough

Let's sort: [170, 45, 75, 90, 802, 24, 2, 66]*Max number is 802 (3 digits), so we need 3 passes.*

13.7.2.1 Pass 1: Sort by Units Place (1s)

- 170, 90 (ends in 0)
- 802, 2 (ends in 2)
- 24 (ends in 4)
- 45, 75 (ends in 5)

NOTE

- 66 (ends in 6)
- Result: [170, 90, 802, 2, 24, 45, 75, 66]

13.7.2.2 Pass 2: Sort by Tens Place (10s)

- 802, 2 (0 in tens place - treated as 02)
- 24 (2 in tens place)
- 45 (4 in tens place)
- 66 (6 in tens place)
- 170, 75 (7 in tens place)
- 90 (9 in tens place)
- Result: [802, 2, 24, 45, 66, 170, 75, 90]*(Notice* 170 *came before* 75 *because in the units pass,* 0 *was before* 5*. Stability preserved the order.)*

13.7.2.3 Pass 3: Sort by Hundreds Place (100s)

- 2, 24, 45, 66, 75, 90 (0 in hundreds place)
- 170 (1 in hundreds place)
- 802 (8 in hundreds place)
- Final Result: [2, 24, 45, 66, 75, 90, 170, 802]

13.7.3 **Pseudocode Implementation**

Plaintext

```
FUNCTION RadixSort(array)
   maxVal = GetMax(array)

   // Do counting sort for every digit.
   // exp is 10^i where i is current digit number
   exp = 1
   WHILE (maxVal / exp) > 0
      CountingSortByDigit(array, exp)
      exp = exp * 10

FUNCTION CountingSortByDigit(array, exp)
   n = length of array
   output = array of size n
   count = array of size 10 (initialized to 0)
```

NOTE

```
// Store count of occurrences in count[]
FOR i = 0 TO n-1
    index = (array[i] / exp) % 10
    count[index]++

// Change count[i] so it contains actual position in output[]
FOR i = 1 TO 9
    count[i] += count[i - 1]

// Build the output array (Go backwards to maintain stability)
FOR i = n-1 DOWNTO 0
    index = (array[i] / exp) % 10
    output[count[index] - 1] = array[i]
    count[index]--

COPY output TO array
```

13.7.4 **Complexity Analysis**

13.7.4.1 Time Complexity

- Complexity: O(d \times (n + b))
    - n: Number of elements.
    - b: Base (usually 10 for decimal numbers).
    - d: Number of digits in the largest number.
- Since d and b are often small constants compared to n, Radix Sort effectively runs in Linear Time O(n), making it faster than Quick Sort for huge lists of integers.

13.7.4.2 Space Complexity

- Space: O(n + b)
    - It requires an output array of size n and a count array of size b. It is not in-place.

NOTE

Activity 1: The "Need for Speed" – Divide & Conquer Showdown

Objective:

To implement Merge Sort and Quick Sort to process large datasets. Students will transition from the slow 1O(n^2) algorithms to efficient 2O(n \log n) "Divide and Conquer" algorithms.3 The goal is to observe the massive performance jump and understand the trade-offs: Merge Sort's stability/memory usage versus Quick Sort's in-place speed/pivot vulnerability.

Materials / Tools / Software Required:

- Computer: Standard development setup.
- Programming Language: C++, Java, or Python (C++ is ideal for demonstrating Quick Sort's cache efficiency).
- Dataset:
  - One array of 100,000 random integers.
  - One array of 100,000 *already sorted* integers.
- Timing Library: (e.g., <chrono> in C++, time module in Python) to measure execution speed in milliseconds.

Expected Outcome:

By the end of this activity, students will be able to:

1. Implement Divide and Conquer: Write recursive functions that break the problem down (Splitting/Partitioning) and combine results.
2. Code the Mechanics:
   - Implement the merge() helper to combine two sorted subarrays (requires auxiliary memory).
   - Implement the partition() helper (Lomuto or Hoare scheme) to sort elements around a pivot *in-place*.
3. Benchmark Performance: Run these algorithms against the Bubble Sort from the previous unit on the 100,000-item dataset. They will likely see Bubble Sort take minutes (or timeout), while Merge/Quick sort take milliseconds.
4. Crash Quick Sort: Run the Quick Sort algorithm (using the first element as a pivot) on the *already sorted* dataset to trigger its [4]O(n^2) worst-case behavior, causing a stack overflow or massive slowdown, demonstrating the importance of pivot selection (e.g., using Randomized Pivot or Median-of-Three)

13.8 SUMMARY

NOTE

Sorting algorithms are fundamental tools in computer science, used to organize data efficiently. This unit explores several advanced techniques, categorized broadly into comparison-based algorithms (Merge Sort, Quick Sort, Shell Sort) and non-comparison-based algorithms (Address Calculation Sort, Radix Sort).

Merge Sort and Quick Sort both utilize the "Divide and Conquer" strategy. Merge Sort recursively splits an array until single elements remain and then merges them back together in sorted order. It is highly predictable, guaranteeing a time complexity of O(n \log n) in all cases, and is a "stable" sort, meaning it preserves the order of duplicate elements. However, it is not "in-place" as it requires O(n) auxiliary space, making it less memory-efficient for arrays but excellent for Linked Lists. Conversely, Quick Sort partitions an array around a chosen "pivot." While it typically outperforms Merge Sort due to cache efficiency and requires less memory (O(\log n) stack space), it is an unstable sort and can degrade to O(n^2) time complexity if the pivot selection is poor.

Shell Sort serves as an optimized version of Insertion Sort. It overcomes the limitation of moving elements one position at a time by comparing elements separated by a "gap." As the algorithm progresses, the gap shrinks, eventually becoming a standard Insertion Sort on a nearly sorted array. It is efficient for medium-sized datasets and embedded systems due to its low memory overhead (O(1)).

Finally, the unit covers linear-time algorithms like Address Calculation Sort and Radix Sort. Address Calculation (or Bucket Sort) uses a hash function to map elements to buckets, while Radix Sort processes numbers digit-by-digit (usually from least significant to most significant). These methods can achieve O(n) time complexity, significantly faster than comparison sorts for large datasets, but they often require more memory and are limited to specific data types like integers or fixed-length strings.

13.9 GLOSSARY

1. Divide and Conquer: A strategy where a problem is broken down into smaller sub-problems, solved recursively, and then combined to form the final solution (used in Merge and Quick Sort).
2. Stable Sort: A sorting attribute where the relative order of elements with equal values is preserved after sorting.

NOTE

3. In-Place Sort: An algorithm that sorts the data within the original structure using only a small, constant amount of extra memory space.
4. Pivot: A reference element selected in Quick Sort used to partition the array into values smaller than and larger than itself.
5. Gap Sequence: The specific intervals used in Shell Sort to compare elements far apart; the efficiency of the sort depends heavily on the sequence chosen (e.g., Shell's or Knuth's).
6. Partitioning: The process in Quick Sort of reordering the array so elements less than the pivot come before it and elements greater come after it.
7. Merge Function: The core routine in Merge Sort that takes two sorted subarrays and combines them into a single sorted array.
8. Hash Function: A mathematical function used in Address Calculation Sort to map an input value to a specific index (address) or bucket.
9. LSD (Least Significant Digit): A method in Radix Sort where processing begins with the rightmost digit (units place) and moves to the left.
10. Recursion: A programming technique where a function calls itself to solve smaller instances of the problem, heavily used in Merge Sort and Quick Sort.

13.10 ANSWER TO CHECK YOUR PROGRESS-POSSIBLE ONE

Check Your Progress 13.1

MCQ Answer:
b) Merge Sort – it divides the array and merges sorted halves.

True/False Answer:
False – Quick Sort has worst-case $O(n^2)$ time complexity in some cases.

Short Answers:

1. Merge Sort is a divide-and-conquer algorithm that splits, sorts, and merges arrays.
2. Differences: Merge Sort is stable; Quick Sort is generally faster but can be unstable; Merge Sort uses extra memory, Quick Sort is in-place.

NOTE

13.11 SUGGESTED READINGS – OER

Sr. No.	Title	URL	QR Code
1	Data Structure Using C++	http://books.aisectonline.com/User/UnitUrls?unitId=3721	

13.12 REFERENCE

- Introduction to Algorithms – Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein – ISBN: 9780262046305 – 2022 (4th Edition)
- Data Structures and Algorithms in C++ – Adam Drozdek – ISBN: 9781133608426 – 2012
- Algorithms – Robert Sedgewick, Kevin Wayne – ISBN: 9780321573513 – 2011
- Data Structures and Algorithm Analysis in C++ – Mark Allen Weiss – ISBN: 9780132847377 – 2013
- Data Structures and Algorithms 1: Sorting and Searching – Kurt Mehlhorn – ISBN: 9783642696725 – 2013
- Sorting Algorithms and Techniques: Definitive Reference for Developers and Engineers – Richard Johnson – ISBN: (likely B0FFJTHGQW e-book) – 2025
- Algorithms Unlocked – Thomas H. Cormen – ISBN: 9780262518802 – 2013
- Efficient Algorithm Design – (Various Authors) – ISBN: 9781835886823 – 2021

13.13 EXERCISE

Part 1

A. Multiple choice question

1. Which of the following sorting algorithms guarantees a time complexity of O (n \log n) in the worst-case scenario?

A) Quick Sort

B) Shell Sort

C) Merge Sort

D) Insertion Sort

NOTE

2. What is the primary disadvantage of Quick Sort compared to Merge Sort?

A) It requires O (n) auxiliary space.

B) Its worst-case time complexity is O (n^2).

C) It cannot sort integers.

D) It is slower on average.

3. Shell Sort is essentially an optimization of which elementary sorting algorithm?

A) Selection Sort

B) Bubble Sort

C) Insertion Sort

D) Counting Sort

4. In Radix Sort, if we are sorting the numbers [170, 45, 75], which digit is processed first in the LSD approach?

A) The hundreds place (1, 0, 0)

B) The tens place (7, 4, 7)

C) The units place (0, 5, 5)

D) The largest number (170)

5. Address Calculation Sort relies on a specific component to distribute elements into buckets. What is it?

A) A Pivot

B) A Gap Sequence

C) A Hash Function

D) A Recursive Call

Answers with Explanations:

1. Answer: C) Merge Sort
Explanation: Merge Sort always divides the array and merges sorted halves, ensuring O (n log n) performance in all cases, unlike Quick Sort which can degrade to O (n^2).

2. Answer: B) Its worst-case time complexity is O (n^2).
Explanation: Quick Sort is efficient on average but can hit O (n^2) if pivot selection is poor, whereas Merge Sort guarantees O (n log n) consistently.

3. Answer: C) Insertion Sort
Explanation: Shell Sort improves Insertion Sort by comparing

NOTE

and swapping elements far apart (using gaps) to reduce total movements.

4. Answer: C) The units place (0, 5, 5)
Explanation: LSD (Least Significant Digit) Radix Sort starts sorting from the least significant digit (units), then moves to tens and hundreds.

5. Answer: C) A Hash Function
Explanation: Address Calculation Sort uses a hash function to compute the storage location (bucket) for each element for direct placement.

B. Small answer type question

1. Explain the concept of "Stability" in sorting algorithms and provide one example where it is crucial.
2. Why is Merge Sort preferred over Quick Sort for sorting Linked Lists?
3. Describe the "Best Case" scenario for Quick Sort and what conditions must be met for it to occur.
4. What is the purpose of the "Gap" in Shell Sort, and how does it change during the execution of the algorithm?
5. What is the primary limitation of non-comparative sorting algorithms like Radix Sort or Address Calculation Sort?

Part 2

C. Long answer type question (answer in 200 words)

1. Compare and contrast the Space Complexity of Merge Sort and Quick Sort. Why is one considered "in-place" while the other is not?
2. Explain the "Partitioning" process in Quick Sort. How does the choice of the pivot element affect the efficiency of this process?
3. Describe the logic behind the Merge() function in Merge Sort. How does it combine two sorted subarrays into a single result?
4. Discuss the impact of different Gap Sequences (Shell's vs. Hibbard's) on the performance of Shell Sort. Why is the sequence choice important?
5. Explain how Address Calculation Sort achieves O(n) time complexity in the best case, and describe a scenario where it degrades to O(n^2).

Part 3

D. Long answer type question (answer in 300 words)

NOTE

1. Provide a detailed step-by-step visual walkthrough of sorting the array [38, 27, 43, 3, 9, 82, 10] using Merge Sort. Include the splitting phase and the merging phase.
2. Analyze the worst-case time complexity of Quick Sort. Describe the specific data arrangement that causes this, the mathematical result, and strategies (like random pivots) used to avoid it.
3. Write the pseudocode for Radix Sort and explain how the algorithm processes an array of integers using the Least Significant Digit (LSD) method.
4. Evaluate the suitability of Shell Sort for Embedded Systems compared to Merge Sort and Quick Sort. Consider factors like code size, recursion, and memory usage.
5. Create a comprehensive comparison table and discussion for Merge Sort, Quick Sort, and Radix Sort based on Time Complexity, Space Complexity, Stability, and ideal Use Cases.

DR. C.V. RAMAN UNIVERSITY

Chhattisgarh, Bilaspur / AN AISECT GROUP UNIVERSITY

Approved by : PCI | AICTE | NCTE | BCI | Member of : AIU | Joint Committee : (UGC | DEB | AICTE) | Recognized by : UGC | A NAAC Accredited University

DR. C. V. RAMAN UNIVERSITY (C.G.)

SLM FEEDBACK FORM

PROGRAMME : _____

COURSE : _____

BLOCK : _____

UNIT : _____

Dear Student,

While studying the units of this block, you may have found certain portions of the text difficult to comprehend. We wish to know your difficulties and suggestions, in order to improve the course. Therefore, we request you to fill up and send us the following questionnaire, which pertains to this block. If you find the space provided insufficient, kindly use a separate sheet.

Please mail to aisectcontent@aisect.org/ for online submission please scan QR code:

Questionnaire

1. **Name** : _____

2. **Enrolment No.** : | | | | | | | | | | | |

3. **How many hours did you need for studying the units?** : | |

4. **Please give your reactions to the following items based on your reading of the block:**

Items	Excellent	Very Good	Good	Poor	Give specific Examples (If poor)
Presentation Quality					
Language and Style					
Illustrations					
Used (diagrams, tables, etc.)					
Conceptual Clarity					
Check Your Progress Questions					
Suggested Readings - OER					

5. **Any other comments:**

_____

_____

_____

_____

***

BLOCK IV	SEARCHING, HASHING, AND FILE ORGANIZATION	287-346
Unit 14	Searching and Hashing Techniques	287
Unit 15	Introduction to File Structure	309
Unit 16	Types of File Organization and Indexing	327

This block focuses on techniques used to locate, retrieve, organize, and manage data efficiently. Searching methods form an important part of the block, with emphasis on sequential and binary searching and their suitability for different types of data. Students will examine the principles, operations, and efficiency of searching techniques and learn how proper data organization can improve retrieval performance. The block also introduces hashing as an efficient technique for storing and accessing data through key-based addressing. Concepts such as hash functions, hash tables, collisions, and collision-resolution methods are explored to develop a practical understanding of hashing. The block further discusses file organization and the methods used to store and retrieve large volumes of data efficiently. Students will examine different approaches to file organization and understand their relevance to data management systems. Together, these concepts provide a comprehensive foundation for designing efficient data retrieval and storage solutions in computer applications.

UNIT 14
SEARCHING AND HASHING TECHNIQUES

NOTE

UNIT STRUCTURE

14.1 Introduction

14.2 Objective

14.3 Linear Search (Sequential Search)

14.4 Binary Search

14.5 Searching in an Ordered Table

14.6 Indexed Sequential Search

14.7 Interpolation Search

14.8 Hash Tables (Hashing)

14.9 Collision Resolution Techniques

14.10 Summary

14.11 Glossary

14.12 Answer to check your progress

14.13 Suggested readings - OER

14.14 Reference

14.15 Exercise

14.1 INTRODUCTION

Searching is one of the most fundamental operations performed in computer science. It involves locating a specific element, called a key, within a collection of data. As the size of data grows, the efficiency of searching techniques becomes critically important for overall system performance. Searching techniques form the backbone of many real-world applications such as databases, file systems, search engines, and information retrieval systems.

This unit introduces various searching algorithms, starting from simple techniques like Linear (Sequential) Search, which checks elements one by one, to more efficient methods such as Binary Search, which uses a divide-and-conquer strategy on sorted data. It also covers advanced searching methods used with ordered data, including Indexed

NOTE

Sequential Search and Interpolation Search, which further optimize search performance.

In addition, the unit explores Hashing techniques, which provide near-constant time searching by using hash functions to directly compute storage locations. Hash tables and collision resolution methods are discussed in detail to understand how fast data access is achieved in practice. Overall, this unit focuses on understanding the working principles, efficiency, advantages, and applications of different searching and hashing techniques.

14.2 OBJECTIVE

After completing this unit, you will be able to:

1. Understand the concept and importance of searching in data structures.
2. Explain the working of Linear Search and Binary Search algorithms.
3. Analyze and compare searching techniques based on time and space complexity.
4. Describe searching methods used in ordered tables such as indexed and interpolation search.
5. Understand hashing concepts, hash tables, and collision resolution techniques.

14.3 LINEAR SEARCH (SEQUENTIAL SEARCH)

14.3.1 Introduction

Linear Search, often called Sequential Search, is the simplest and most intuitive searching algorithm in computer science. It is the method most people use instinctively in their daily lives.

Imagine you are looking for a specific playing card in a deck that has been shuffled. You would likely pick up the top card, look at it, and if it’s not the one you want, move to the next. You repeat this process until you find the card or run out of cards. This "check one by one" approach is exactly how Linear Search operates.

Unlike more complex algorithms (like Binary Search) that require data to be sorted, Linear Search works on unsorted collections, making it highly flexible.

NOTE

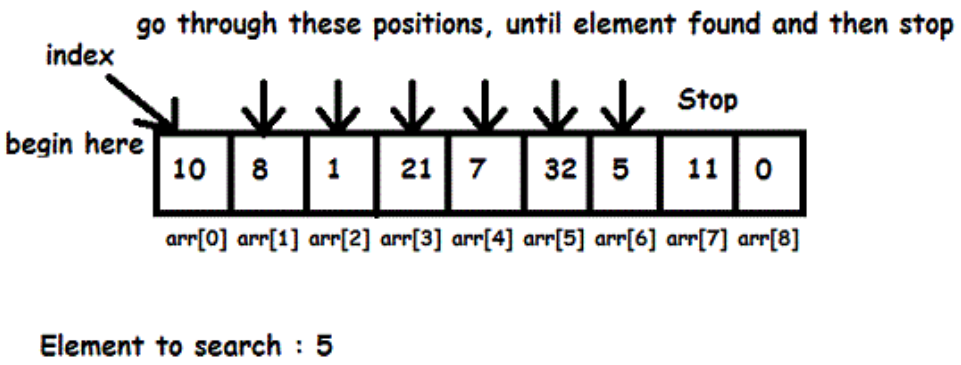


Fig 14.1: LINEAR SEARCH

14.3.2. How It Works

The algorithm traverses a list (or array) element by element from the start to the end.

1. Start: Begin at the first element (index 0) of the list.
2. Compare: Compare the current element with the Target Value (the key you are looking for).
3. Match Found: If the current element matches the target, the search is successful. The algorithm returns the index (position) of the element.
4. No Match: If the current element does not match, move to the next element.
5. End of List: If the algorithm reaches the end of the list without finding a match, the search terminates, indicating the target is not present (often returning -1 or null).

14.3.2.1 Step-by-Step Example

Consider the following list of numbers: [10, 50, 30, 70, 80, 20]

Target: We want to find the number 30.

- Step 1: Check index 0 (Value: 10). Is 10 = 30? No. Move next.
- Step 2: Check index 1 (Value: 50). Is 50 = 30? No. Move next.
- Step 3: Check index 2 (Value: 30). Is 30 = 30? Yes.
- Result: Return Index 2.

14.3.3. Pseudocode and Implementation

Here is the logic represented in generic Pseudocode, followed by a Python implementation.

NOTE

14.3.3.1 Pseudocode

Plaintext

```
FUNCTION LinearSearch(list, target):
    FOR index FROM 0 TO length(list) - 1:
        IF list[index] EQUALS target:
            RETURN index
    END FOR
    RETURN -1 (indicating not found)
END FUNCTION
```

14.3.4. Complexity Analysis

To understand the efficiency of an algorithm, we analyze its Time and Space complexity. In the analysis below, n represents the number of elements in the list.

14.3.4.1 Time Complexity

- Best Case: O(1)

  The target element is located at the very first index (index 0). The algorithm makes only one comparison and terminates immediately.

- Worst Case: O(n)

  The target element is at the very last index, or the element is not present in the list at all. The algorithm must visit every single element.

- Average Case: O(n)

  On average, the target element might be somewhere in the middle. Mathematically, the expected number of comparisons is roughly n/2. In Big O notation, we drop constants, so it remains O(n).

14.3.4.2 Space Complexity

- Space Complexity: O(1)

  Linear Search is an "in-place" algorithm. It does not require any additional storage or data structures (like a hash map or a recursive stack) proportional to the input size. It only requires a few variables for pointers/indices.

14.3.5. Comparison: Linear vs. Binary Search

A common question is how Linear Search compares to Binary Search.

Feature	Linear Search	Binary Search

NOTE

Data Requirement	Works on Unsorted or Sorted data.	strictly requires Sorted data.
Approach	Sequential (one by one).	Divide and Conquer (splits list in half).
Time Complexity	O(n) (Slower for large data).	O(\log n) (Much faster for large data).
Implementation	Very Simple.	Slightly more complex.
Efficiency	Efficient for small datasets.	Efficient for very large datasets.

14.3.6. Advantages and Disadvantages

14.3.6.1 Advantages

1. Simplicity: It is very easy to understand and implement.
2. No Sorting Required: Unlike Binary Search, you do not need to sort the data first. Sorting itself takes time (usually O(n \log n)), so for a single search on an unsorted list, Linear Search is actually faster than "Sort + Binary Search."
3. Memory Efficient: It requires constant space O(1).
4. Streaming Data: It works well when data is being received in a stream and the total size is unknown.

14.3.6.2 Disadvantages

1. Slow on Large Datasets: As the size of the input (n) grows, the time taken grows linearly. For a list of 10 million items, the worst case requires 10 million comparisons.
2. Inefficient for Repeated Searches: If you need to search a static list thousands of times, Linear Search is a poor choice. It is better to sort the data once and use Binary Search or use a Hash Map.

14.3.7. When to Use Linear Search?

Despite its simplicity, Linear Search is still widely used in production code in specific scenarios:

- Small Lists: When n is small (e.g., < 100 items), the overhead of complex algorithms is unnecessary. Linear search is often faster purely due to CPU caching and low overhead.
- Unsorted Data: When you cannot afford the time to sort the data (e.g., a one-time search).
- Searching Linked Lists: Linked Lists do not support random access (jumping to the middle), so Binary Search is generally

NOTE

not applicable. Linear traversal is the standard way to search a Linked List.

- Debugging/Sanity Checks: Because the code is simple, it is often used to verify results of more complex algorithms during testing.

14.4 BINARY SEARCH

14.4.1 Introduction

Binary Search is a highly efficient searching algorithm used to find an element's position in a sorted list. Unlike Linear Search, which checks every single element, Binary Search uses a Divide and Conquer strategy to drastically reduce the number of comparisons needed.

Think of Binary Search like looking up a word in a physical dictionary. You don't start at the first page and read every word. Instead, you open the book to the middle. If the word you want is alphabetically after the page you opened, you ignore the first half of the book and only look at the second half. You repeat this process—splitting the remaining pages in half—until you find the word.

Crucial Requirement: Binary Search works only on lists that are already sorted.

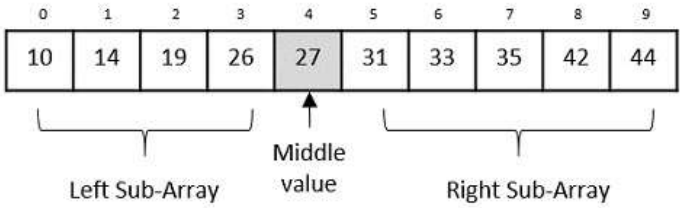


FIG 14.2: BIANRY SEARCH

14.4.2. How It Works

The algorithm repeatedly divides the search interval in half.

1. Find Middle: Calculate the middle index of the current list (or sub-list).
2. Compare: Compare the Middle Element with the Target Value.
3. Match: If Middle == Target, the search is done. Return the middle index.
4. Target is Smaller: If Target < Middle, the target *must* be in the left half. Discard the right half.
5. Target is Larger: If Target > Middle, the target *must* be in the right half. Discard the left half.

NOTE

6. Repeat: Repeat steps 1–5 on the new, smaller half until the target is found or the search interval becomes empty.

14.4.2.1 Step-by-Step Example

Consider this sorted list: [10, 20, 30, 40, 50, 60, 70]

Target: We want to find 60.

- Step 1:
  - List: [10, 20, 30, 40, 50, 60, 70] (Indices 0 to 6)
  - Middle Index: 3 (Value: 40)
  - Compare: Is 60 > 40? Yes.
  - *Action:* Ignore left half. Keep indices 4 to 6 ([50, 60, 70]).
- Step 2:
  - New List Segment: [50, 60, 70]
  - Middle Index: 5 (Value: 60)
  - Compare: Is 60 = 60? Yes.
  - *Action:* Match Found. Return Index 5.

Note: Linear search would have taken 6 steps to find this; Binary search took only 2.

14.4.3. Pseudocode and Implementation

Binary search can be implemented Iteratively (using loops) or Recursively. The Iterative approach is generally preferred in production to avoid memory overhead.

14.4.3.1 Pseudocode

Plaintext

```
FUNCTION BinarySearch(list, target):
   SET low = 0
   SET high = length(list) - 1


   WHILE low <= high:
      mid = (low + high) / 2
      IF list[mid] EQUALS target:
         RETURN mid
      ELSE IF list[mid] < target:
         low = mid + 1  (Target is in right half)
      ELSE:
```

NOTE

high = mid - 1 (Target is in left half)

END WHILE

RETURN -1 (Not found)

END FUNCTION

14.4.4. Complexity Analysis

Binary Search is famous for its logarithmic speed. In the analysis below, n is the number of elements.

14.4.4.1 Time Complexity

- Best Case: O(1)

  The target is exactly at the middle index on the very first check.

- Worst Case: O(\log n)

  The target is at the ends of the list or not present.

  - *Why Log n?* Every step cuts the data in half. If you have n items, you can divide by 2 roughly \log_2(n) times before you reach a single element.
  - *Example:* For 1,000,000 items, Linear Search takes up to 1,000,000 steps. Binary Search takes only 20 steps (2^{20} \approx 1,000,000).

- Average Case: O(\log n)

14.4.4.2 Space Complexity

- Iterative Implementation: O(1)

  It uses constant extra space (variables for low, high, mid).

- Recursive Implementation: O(\log n)

  The recursion stack takes up memory proportional to the depth of the search tree. This is why iterative is often safer.

14.4.5. Comparison: Linear vs. Binary Search

Table 14.1: Linear vs. Binary Search

Feature	Linear Search	Binary Search
Prerequisite	None (works on any list).	Data must be sorted.
Algorithm Type	Sequential.	Divide and Conquer.
Time (Worst)	O(n) (Linear).	O(\log n) (Logarithmic).

NOTE

Usage	Small or unsorted data.	Large, static, sorted libraries.
Complexity	Simple logic.	Tricky logic (off-by-one errors common).

14.4.6. Advantages and Disadvantages

14.4.6.1 Advantages

1. Speed: Extremely fast for large datasets. It scales beautifully; doubling the input size only adds one extra step to the search.
2. Efficiency: It minimizes the number of comparisons, which is crucial if the comparison operation itself is expensive (e.g., comparing long strings or complex objects).

14.4.6.2 Disadvantages

1. Sorting Requirement: The list must be sorted first. Sorting takes O(n \log n) time. If you only search once, sorting + binary search is slower than just doing a linear search.
2. Complex Memory: It relies on "Random Access" (jumping to the middle index). This means it works great on Arrays but is poor for Linked Lists, where you cannot jump to the middle easily.
3. Implementation Errors: It is notoriously easy to introduce bugs (like infinite loops or missing the edge elements) when writing the low <= high condition or updating low = mid + 1.

14.4.7. When to Use Binary Search?

- Large Databases: Searching in databases (like SQL indexes) where data is stored in sorted order (B-Trees).
- Repeated Searches: If you have a static dataset (e.g., a dictionary of English words) that you will search thousands of times, sort it once and use Binary Search forever.
- Library Functions: Most built-in "find" or "contains" functions for sorted collections (like std::binary_search in C++ or bisect in Python) use this algorithm.
- Numerical Solutions: Finding the square root of a number or solving equations where the answer lies within a sorted range (monotonic functions).

14.5 SEARCHING IN AN ORDERED TABLE

Searching in an ordered (sorted) table is a fundamental operation in computer science. When data is pre-sorted, we can utilize algorithms

NOTE

that are significantly faster than scanning every single item (Linear Search).

Why Order Matters

If a table (or array) is ordered, the value of an element gives you information about the location of *other* elements.

- If current_value < target, you know the target must be to the right.
- If current_value > target, you know the target must be to the left.

This property allows us to skip vast sections of data, reducing the time complexity from O(n) to O(\log n) or even O(\log (\log n)) in some cases.

Common Approaches:

1. Binary Search: The standard approach (splitting the list in half).
2. Indexed Sequential Search: A hybrid approach using an auxiliary index.

Interpolation Search: An "intelligent" variation of binary search for uniform data.

14.6 INDEXED SEQUENTIAL SEARCH

Indexed Sequential Search is a technique that combines the simplicity of Linear Search with the speed of indexed access. It is widely used in database systems and file storage where loading the entire dataset into memory is impossible.

14.6.1 How It Works

Instead of searching the massive main table, we create a smaller Index Table.

- The Index Table: Contains "guide" entries that represent ranges of the main table (e.g., the starting value of every 10th block).
- The Process:
    1. Search the Index: Scan the small Index Table to find the specific "block" or "page" where the target value *should* be.
    2. Search the Block: Go to the main table at the position indicated by the index and perform a Linear Search only within that specific, small block.

14.6.2 Step-by-Step Example

Imagine a library with 1,000 books sorted alphabetically, but you don't want to scan all 1,000.

NOTE

- Index: You have a small sheet of paper listing which letter starts each shelf:
    - Shelf 1: Starts with 'A'
    - Shelf 2: Starts with 'C'
    - Shelf 3: Starts with 'F'
- Target: Book "Darwin".
- Step 1: Check Index. 'D' is greater than 'C' (Shelf 2) but less than 'F' (Shelf 3). So, the book must be on Shelf 2.
- Step 2: Go to Shelf 2 and linearly scan *only* the books on that shelf to find "Darwin".

14.6.3 Complexity Analysis

- Time Complexity: O(\sqrt{n})
    - If we choose the block size to be \sqrt{n}, we search an index of size \sqrt{n} and then a block of size \sqrt{n}.
- Space Complexity: O(n)
    - Requires storage for the Index Table (though usually much smaller than the main data).

14.6.4 Pros and Cons

- Pros: Faster than Linear Search; efficient for data stored on disk (minimizes expensive disk reads).

Cons: Slower than Binary Search; requires extra memory for the index; the index must be updated whenever the main data changes

14.7 INTERPOLATION SEARCH

Interpolation Search is an improved variant of Binary Search that mimics how humans search for a name in a phone book or a word in a dictionary.

14.7.1 How It Works

Binary Search always checks the middle. Interpolation Search tries to predict exactly where the item is, based on the value of the key.

- If the Target is closer to the *end* of the range (e.g., "Zebra"), the algorithm looks near the *end*.
- If the Target is closer to the *start* (e.g., "Apple"), it looks near the *start*.

It uses a mathematical formula to calculate the "Probe Position" (pos) rather than just (low + high) / 2.

NOTE

14.7.2 The Formula

The position is calculated using linear interpolation:

pos = low + \left[\frac{(target - arr[low]) \times (high - low)}{(arr[high] - arr[low])} \right]

- low / high: The current start and end indices.
- arr[low] / arr[high]: The values at those indices.
- Logic: It calculates the fraction of the way the target is between the low and high values, and applies that fraction to the indices.

14.7.3 Example

List: [10, 20, 30, 40, 50, 60, 70, 80, 90, 100] (Indices 0–9)

Target: 90

- Binary Search: Would start at index 4 (Value 50), then move right.
- Interpolation Search:
  - Target (90) is near the end value (100).
  - The formula calculates pos \approx 8.
  - It checks index 8 immediately. Value is 90. Found in 1 step.

14.7.4 Complexity Analysis

- Best Case: O(1) (If the formula guesses the exact index instantly).
- Average Case: O(\log (\log n))
  - This is extremely fast. For n = 4 billion, Binary Search takes ~32 steps. Interpolation Search takes ~5 steps.
- Worst Case: O(n)
  - This happens if the data is not uniformly distributed (e.g., 1, 2, 4, 8, 1000, 2000...). The formula fails to predict correctly, and it degrades to a linear scan.

Check Your Progress 14.1

MCQ:
Which searching technique divides the search interval in half repeatedly to find an element in a sorted array?
a) Linear Search
b) Binary Search
c) Hashing
d) Sequential Search

NOTE

True/False:
In hashing, a hash function maps a key to a specific index in a hash table.

Short Questions:

1. Define hashing in computer science.

.....

.....

2. Name two collision resolution techniques in hashing.

.....

.....

14.8 HASH TABLES (HASHING)

14.8.1 Introduction

A Hash Table (or Hash Map) is one of the most efficient data structures used in computer science. It is designed to solve the problem of searching, inserting, and deleting data in constant time, O(1) on average.

While Linear Search takes O(n) and Binary Search takes O(\log n), a Hash Table aims to find an item instantly, regardless of the data size. It achieves this by using a specific function—called a Hash Function—to calculate the exact index where an element should be stored.

Think of a library where, instead of looking up a book in a catalog, you plug the book's title into a calculator (the hash function). The calculator gives you a number, say "42". You go to shelf 42, and the book is right there. No searching required.

14.8.2. Components of a Hash Table

1. Key: The input data you want to store (e.g., a username "Alice").
2. Hash Function: A mathematical algorithm that takes the Key and converts it into an integer (hash code).
    - *Example:* len("Alice") % 10 might return index 5.
3. Hash Table (Array): The actual array (or "buckets") where the data is stored using the index generated by the hash function.

14.8.3. The Collision Problem

Ideally, every unique key would produce a unique index. However, in reality, the hash table size is limited, and the number of possible keys is infinite.

NOTE

Sometimes, two different keys (e.g., "John" and "Jane") might produce the same index from the hash function. This event is called a Collision.

Since two items cannot occupy the exact same slot in an array, we need a strategy to handle this conflict. These strategies are called Collision Resolution Techniques.

14.9 COLLISION RESOLUTION TECHNIQUES

There are two primary methods to handle collisions: Separate Chaining (Open Hashing) and Open Addressing (Closed Hashing).

14.9.1 Separate Chaining (Open Hashing)

In this method, each slot (bucket) in the hash table does not store the item itself. Instead, it stores a pointer to a secondary data structure, usually a Linked List.

- How it works: When a collision occurs (two keys map to index 5), simply append the new item to the linked list at index 5.
- Searching: To find an item, go to the hashed index, then traverse the linked list at that index until you find the match.
- Pros:
    - Simple to implement.
    - The table never technically "fills up" (you can just keep adding to the lists).
- Cons:
    - Requires extra memory for pointers.
    - If many keys hash to the same index, the list becomes long, degrading search speed to O(n) (Linear Search).

14.9.2 Open Addressing (Closed Hashing)

In this method, all elements are stored directly inside the hash table array. No linked lists are used. If a collision occurs (the calculated slot is occupied), the algorithm searches for the next available empty slot within the array itself.

There are three common "probing" strategies to find that next slot:

14.9.2.1. Linear Probing

If the calculated index i is occupied, check i+1, then i+2, then i+3, and so on, wrapping around to the beginning if necessary.

- Formula: Index = (Hash(Key) + i) \% Size

NOTE

- Problem: Causes Clustering. If a block of slots gets filled, any new key that hashes into that block extends the block further, making future searches slower.

14.9.2.2. Quadratic Probing

Instead of checking the very next slot, we space out the probes to reduce clustering. If index i is busy, we check i + 1^2, i + 2^2, i + 3^2...

- Formula: Index = (Hash(Key) + i^2) \% Size
- Benefit: Reduces the primary clustering problem of Linear Probing.

14.9.2.3. Double Hashing

This is one of the most effective methods. If a collision occurs, a second hash function is used to calculate the step size.

- Formula: Index = (Hash1(Key) + i \times Hash2(Key)) \% Size
- Benefit: The step size depends on the key itself, so even if two keys collide at the first index, they will likely have different step sizes and won't collide again.

Activity 1: "The Collision Course" – Building a Hash Table

Objective:

To build a Hash Table from scratch to understand how constant-time O(1) lookups are achieved. The primary focus is on implementing a Hash Function and solving the inevitable problem of Collisions (when two different keys map to the same index) using Chaining (Linked Lists) or Linear Probing.

Materials / Tools / Software Required:

- Computer: Standard development setup.
- Programming Language: Python, C++, or Java.
- Dataset: A list of 50 strings (e.g., names or product codes).
- Visual Aid: A diagram showing an Array of "Buckets" (Linked Lists).

Expected Outcome:

By the end of this activity, students will be able to:

1. Design a Hash Function: Write a simple modulo-based function (e.g., index = sum(ASCII values) % Table Size) to map string keys to array indices.
2. Implement Chaining: Create the table as an array of Linked Lists. When a collision occurs, they will append the new data to the list at that index rather than overwriting existing data.

NOTE

3. Perform Fast Lookups: Write a get(key) function that hashes the key, goes to the correct bucket, and traverses the short list to find the value.
4. Analyze Load Factor: Experiment by shrinking the table size to force more collisions, observing how the lookup speed degrades from O(1) to O(n) as the chains get longer.

14.10 SUMMARY

This unit provides a comprehensive overview of fundamental algorithms used for retrieving information from data structures, ranging from simple sequential checks to complex hashing mechanisms.

Linear Search is introduced as the most intuitive method, operating by checking every element in a list one by one until a match is found. While highly flexible because it works on unsorted data, it suffers from O(n) time complexity, making it inefficient for large datasets. In contrast, Binary Search employs a "Divide and conquer" strategy. By requiring the data to be pre-sorted, it can eliminate half of the remaining search space in every step, achieving a significantly faster logarithmic time complexity of O(\log n).

The unit further explores searching within Ordered Tables. Techniques like Indexed Sequential Search function as a hybrid, using a small auxiliary index to guide the search to a specific block of data, optimizing performance for disk-based storage. Interpolation Search improves upon Binary Search for uniformly distributed data by mathematically predicting the target's position (similar to finding a name in a phone book), potentially achieving O(\log (\log n)) time complexity.

Finally, the unit covers Hashing, a technique designed to achieve constant time complexity, O(1), for search, insertion, and deletion. A Hash Table uses a Hash Function to map keys (like usernames) to specific indices in an array. A major challenge in hashing is Collision—when two keys map to the same index. The unit details two primary resolution strategies: Separate Chaining (Open Hashing), where collisions are handled by storing multiple items in a linked list at the same index, and Open Addressing (Closed Hashing), which finds the next available slot within the array using techniques like Linear Probing, Quadratic Probing, or Double Hashing

14.11 GLOSSARY

1. Linear Search: A simple algorithm that sequentially checks each element of a list until a match is found or the list ends.

NOTE

2. Binary Search: An efficient algorithm for sorted lists that repeatedly divides the search interval in half to locate a target value.
3. Time Complexity: A measure of the amount of time an algorithm takes to complete as a function of the length of the input.
4. Divide and Conquer: A problem-solving strategy that breaks a problem into smaller sub-problems (used in Binary Search).
5. Interpolation Search: A variation of Binary Search that estimates the position of a target value based on the values at the ends of the search interval.
6. Hash Function: A mathematical algorithm that converts an input key into a specific integer index for storage in a Hash Table.
7. Collision: An event in hashing where two different input keys produce the same output index from the hash function.
8. Separate Chaining: A collision resolution technique where each slot in the hash table points to a linked list containing all entries that hashed to that index.
9. Open Addressing: A collision resolution technique where all elements are stored within the hash table array itself, finding alternative slots when collisions occur.
10. Linear Probing: A specific Open Addressing method where, upon collision, the algorithm checks the next sequential index (i+1) for an empty slot.

14.12 ANSWER TO CHECK YOUR PROGRESS-POSSIBLE ONE

Check Your Progress 14.1

MCQ Answer:

b) Binary Search – divides the search interval repeatedly.

True/False Answer:
True – a hash function maps keys to indices in a table.

Short Answers:

1. Hashing is a technique to map keys to specific locations for fast access.
2. Collision resolution techniques: Chaining and Open Addressing

NOTE

14.13 SUGGESTED READINGS – OER

Sr. No.	Title	URL	QR Code
1	Data Structure Using C++	http://books.aisectonline.com/User/UnitUrls?unitId=3722	

14.14 REFERENCE

1. Data Structures and Algorithms in C++ – Adam Drozdek – ISBN: 9781133608426 – 2012
2. Introduction to Algorithms – Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein – ISBN: 9780262046305 – 2022 (4th Edition)
3. Data Structures and Algorithms Made Easy – Narasimha Karumanchi – ISBN: 9781468108860 – 2011
4. Data Structures: Concepts and Implementation – Anuradha A. Puntambekar – ISBN: 9789333223911 – 2020
5. Advanced Data Structures – Reema Thareja & S. Rama Sree – ISBN: 9780199487172 – 1st Ed, 2023
6. Algorithms Unlocked – Thomas H. Cormen – ISBN: 9780262518802 – 2013
7. Algorithms + Data Structures = Programs – Niklaus Wirth – ISBN: (varies by edition) – 1976 / later editions
8. Data Structures – An Algorithmic Approach – Ikvinderpal Singh (C/C++ context) – ISBN: 9789380016690

14.15 EXERCISE

Part 1

A. Multiple choice question

1. Which of the following is a strict requirement for using Binary Search?

A) The list must be a Linked List.

B) The list must contain integers only.

C) The list must be sorted.

D) The list size must be a power of 2.

2. What is the worst-case time complexity of a Linear Search?

NOTE

A) O (1)

B) O (\log n)

C) O (n)

D) O (n^2)

3. In Interpolation Search, if the data is not uniformly distributed, the time complexity can degrade to:

A) O (1)

B) O (\log n)

C) O (n)

D) O (n \log n)

4. Which collision resolution technique involves using a secondary data structure (like a Linked List) at each index?

A) Linear Probing

B) Separate Chaining

C) Quadratic Probing

D) Double Hashing

5. If a Hash Table has a poor hash function that maps all keys to the same index, the search time complexity becomes:

A) O (1)

B) O (log n)

C) O (n)

D) O (\sqrt{n})

Answers with Explanations:

1. Answer: C) The list must be sorted.
Explanation: Binary search works by repeatedly dividing the search space in half, which is only possible when the elements are in sorted order.

2. Answer: C) O (n)
Explanation: In the worst case, linear search checks every element in the list before finding the target or concluding it is absent.

3. Answer: C) O (n)
Explanation: Interpolation search assumes uniform distribution; if this assumption fails, it may behave like linear search in the worst case.

NOTE

4. Answer: B) Separate Chaining
Explanation: Separate chaining stores multiple elements at the same hash index using a linked list or similar structure to handle collisions.

5. Answer: C) O (n)
Explanation: If all keys collide at one index, the hash table effectively becomes a linear list, making search operations linear in time.

B. Small answer type question

1. Briefly explain why Linear Search is preferred over Binary Search for small, unsorted datasets.
2. Define the "Collision" problem in Hash Tables and name the two primary categories of resolving it.
3. What is the fundamental difference in how Interpolation Search determines the "split point" compared to Binary Search?
4. Why is the Space Complexity of a Recursive Binary Search considered O(\log n) while the Iterative version is O(1)?
5. Describe the "Clustering" problem associated with Linear Probing in Open Addressing.

Part 2

C. Long answer type question (answer in 200 words)

1. Compare Linear Search and Binary Search in terms of data requirements, complexity, and best-use scenarios. When would you strictly choose one over the other?
2. Explain the concept of Indexed Sequential Search. How does using an auxiliary index table improve performance compared to a standard Linear Search?
3. Describe the mechanism of "Separate Chaining" (Open Hashing). How does it handle searching for an element when multiple keys have hashed to the same index?
4. Discuss the "Double Hashing" technique in Open Addressing. How does the use of a second hash function prevent primary clustering?
5. Analyze the Best, Average, and Worst-case time complexities of a Hash Table. What specific conditions lead to the worst-case scenario?

Part 3

D. Long answer type question (answer in 300 words)

1. Provide a detailed step-by-step walkthrough of a Binary Search finding the value 60 in the array [10, 20, 30, 40, 50, 60, 70]. Contrast this with how a Linear Search would process the same request.
2. Explain the mathematical logic behind Interpolation Search. Provide the formula used to calculate the probe position and

NOTE

discuss why this method is incredibly fast (O(\log (\log n))) for uniform data but risky for non-uniform data.

3. Write a comprehensive explanation of Open Addressing (Closed Hashing). detailed the logic of Linear Probing, Quadratic Probing, and Double Hashing, highlighting the advantages and disadvantages of each approach.
4. Critically evaluate the trade-offs between using a Hash Table versus a Sorted Array with Binary Search. Consider factors like memory usage, initialization time, and speed of insertion/deletion.
5. Write the pseudocode for both the Iterative and Recursive implementations of Binary Search. Explain the logic of the low, high, and mid variables in the context of the Divide and Conquer strategy.

UNIT 15
INTRODUCTION TO FILE STRUCTURE

UNIT STRUCTURE

15.1 Introduction

15.2 Objective

15.3 Introduction to File Structure

15.4 Objectives of File Organization

15.5 Terminology Used in File Handling

15.6 File Organization Concepts

15.7 Summary

15.8 Glossary

15.9 Answer to check your progress

15.10 Suggested readings - OER

15.11 Reference

15.12Exercise

INTRODUCTION

In computer science, data is not only processed in main memory but also stored permanently on secondary storage devices such as hard disks, solid-state drives, and magnetic tapes. The method used to organize and manage this data on storage devices is known as **File Structure**. File structure plays a vital role in determining how efficiently data can be stored, retrieved, updated, and maintained over time.

Unlike main memory, secondary storage is slower and requires careful organization to minimize access time and storage wastage. File structures provide a systematic way to store records in files and define the operations that can be performed on them, such as creation, reading, writing, updating, and deletion. They act as a bridge between the logical view of data seen by users and the physical layout of data on storage devices.

Understanding file structures is essential for designing efficient file systems, database systems, and large data-handling applications. Different file organization techniques—such as sequential, direct, and indexed file structures—are used based on application requirements

NOTE

like access speed, storage efficiency, and ease of modification. This unit introduces the fundamental concepts, terminology, objectives, and methods related to file structure and file organization.

15.2 OBJECTIVE

After completing this unit, you will be able to:

1. Understand the concept and importance of file structure in computer systems.
2. Explain the hierarchy of data organization including fields, records, and files.
3. Identify and describe basic file operations such as read, write, seek, and update.
4. Differentiate between various types of file organization techniques.
5. Analyze the objectives of file organization with respect to efficiency, access speed, and storage utilization.

15.3 INTRODUCTION TO FILE STRUCTURE

In the world of computer science, how we organize data *inside* the computer's memory (RAM) is often the focus of algorithms and data structures. However, how we organize data outside of main memory—on persistent storage devices like hard disks, SSDs, or magnetic tapes—is the domain of File Structures.[1]

A File Structure is a combination of representations for data in files and a set of operations for accessing that data.[2] It acts as the bridge between the logical definition of data (what the user sees) and the physical storage device (blocks, tracks, and sectors).

Why File Structure Matters

The primary goal of a good file structure is efficiency.[3] Unlike RAM, accessing data from a disk is slow—often thousands of times slower than accessing memory.

- Minimizing Disk I/O: The most expensive operation in computing is the "trip" to the disk. A good file structure minimizes the number of trips required to find a specific record.[5]
- Storage Management: It ensures data is stored compactly but can expand as new records are added.
- Data Integrity: It provides a reliable format that allows data to be retrieved accurately even after the program that created it has closed.

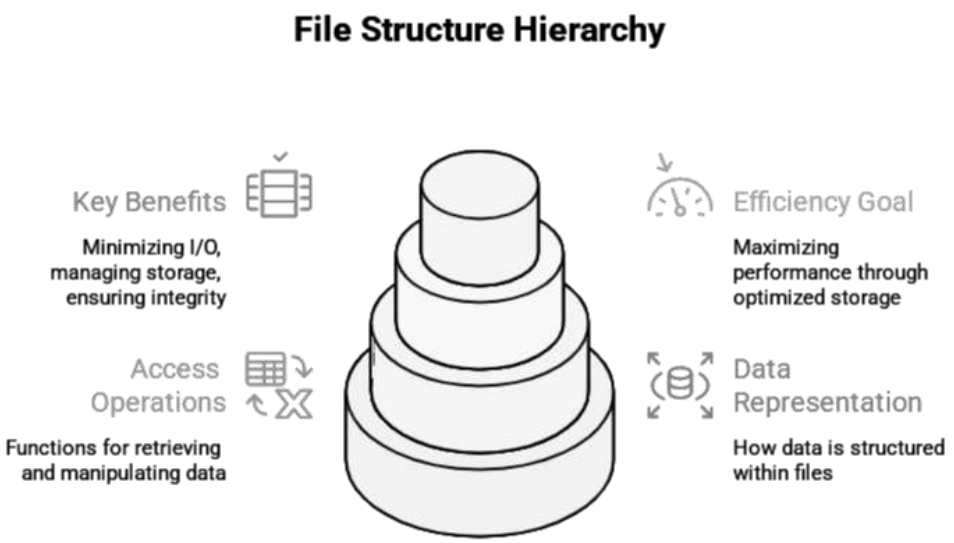


Fig 15.1: File Structure Hierarchy

15.3.1. The Anatomy of a File

To understand file structures, one must understand the hierarchy of data organization:

1. Field: The smallest unit of meaningful data (e.g., a "Name" or "Employee ID"). Fields can be Fixed Length (always 10 bytes) or Variable Length (separated by delimiters like commas).
2. Record: A collection of related fields that represents a single entity (e.g., all data for one employee).
    - *Logical Record:* How the user views the data.
    - *Physical Record (Block):* How the OS views the data. Often, multiple logical records are "blocked" together into one physical block to improve I/O efficiency.
3. File: A collection of records.[6]

15.3.2. Fundamental Operations

Regardless of the structure used, most file systems support a standard set of operations.[7] These are the "verbs" of file processing:

- Creation: Allocating space for a new file in the directory.
- Open: Preparing the file for access (often loading the directory entry into RAM).[8]
- Read: Transferring data from the storage device to a memory buffer.
- Write/Update: Transferring data from memory to the storage device.
- Seek: Moving the read/write head (or pointer) to a specific location within the file without reading the preceding data.[9]

NOTE

- Close: Releasing the file access and ensuring all buffers are flushed to the disk.[10]

15.3.3. Types of File Organization

The method chosen to organize records determines the efficiency of the file. The three classic methods are Sequential, Direct, and Indexed Sequential.[11]

A. Sequential Organization

This is the simplest form of file structure. Records are stored one after another in a specific order (usually entered chronologically or sorted by a Key).[12]

- Mechanism: To find the 100th record, the system must read the first 99 records.
- Use Case: Ideal for batch processing where every record needs to be processed (e.g., generating monthly payroll for all employees).
- Pros: Simple to implement; efficient for "read-all" operations.
- Cons: Terribly inefficient for searching for a specific record (Random Access). Time complexity is O(n).

B. Direct (Random) Access Organization

This method allows the system to jump directly to a specific record without reading the others.[13] It typically uses a Hashing Algorithm.[14]

+1

- Mechanism: The Record Key (e.g., ID 105) is passed through a hash function (math formula) which calculates the exact physical address (block number) where that record is stored.[15]
    - *Formula:* Address = H(Key)
- Use Case: Online transaction systems (e.g., retrieving a specific flight booking).
- Pros: Immediate access.[16] Time complexity is effectively O(1).
- Cons: "Collisions" occur when two keys hash to the same address; space can be wasted if the data is sparse.[17]

C. Indexed Sequential Access Method (ISAM)

This is a hybrid approach that combines the best of both worlds. It keeps the data in a sequential file but creates a separate Index File.

- Mechanism:
    1. The Data File: Contains records sorted sequentially.[18]

NOTE

2. The Index: A smaller file containing pairs of (Key, Address).19

   To find a record, the computer looks up the Key in the index, gets the address, and jumps to that location in the data file

- Use Case: Databases and modern file systems (like NTFS or ext4) use variations of this (often using B-Trees).
- Pros: Supports both fast individual lookups and efficient batch processing.
- Cons: Requires extra storage for the index; the index must be updated every time a record is added or deleted.[21]

15.4 OBJECTIVES OF FILE ORGANIZATION

While "File Structure" refers to the specific representation of data (like the blueprint), File Organization refers to the strategy used to store that data to meet specific goals.

We do not organize files simply to keep them tidy; we organize them to solve the physical limitations of storage hardware. The hard disk is a mechanical device in an electronic world—it is slow, prone to failure, and limited in capacity. Therefore, every file organization technique (Sequential, Indexed, Hash) is designed to balance a competing set of critical objectives.

Below are the primary objectives that drive the design of any file organization system.

15.4.1. Minimizing Input/Output (I/O) Operations

This is the single most important objective. In computing, the CPU operates in nanoseconds, while disk access operates in milliseconds. Accessing the disk is roughly 100,000 to 1,000,000 times slower than accessing RAM.

- **The Goal:** To locate and retrieve the required record with the fewest number of disk accesses possible.
- The "One-Access" Ideal: The theoretical "perfect" file organization would allow you to find any record in the world with exactly one trip to the disk (1 I/O). Hashing comes close to this, whereas Sequential organization fails this objective completely for random searches.
- Blocking: A key strategy here is *Blocking*. By grouping multiple logical records into a single physical block, we can retrieve many records in a single I/O operation, maximizing the data transfer rate.

NOTE

15.4.2. Optimal Record Retrieval (Access Speed)

Different applications require different "types" of speed. A file organization must be chosen based on which type of retrieval is the priority:

- Fast Random Access: The ability to jump to a specific record immediately (e.g., "Find Student ID #998"). Direct/Hash organization achieves this best.
- Fast Sequential Access: The ability to process large batches of data quickly in order (e.g., "Print the payroll for all 5,000 employees"). Sequential organization achieves this best.
- The Trade-off: It is rarely possible to maximize both. Optimizing for random access often fragments the data, making sequential processing slower.

15.4.3. Storage Efficiency (Space Utilization)

Disk space, while cheaper today, is not infinite. Efficient file organization seeks to store the maximum amount of data in the minimum amount of space.

- Minimizing Fragmentation:
    - *Internal Fragmentation:* Wasted space inside a block (e.g., a 512-byte block holding a 300-byte record leaves 212 bytes wasted).
    - *External Fragmentation:* Wasted space outside blocks (gaps between files that are too small to use).
- Handling Variable Length Records: If records vary in size (e.g., one email is 50 words, another is 5,000), the file organization must pack them tightly rather than reserving the maximum possible space for every record, which would waste huge amounts of storage.

15.4.4. Ease of Update (Insertion and Deletion)

Data is rarely static. A major objective is to allow records to be added, removed, or modified without requiring a complete rewrite of the entire file.

- The Insertion Problem: If a file is sorted alphabetically, adding "Charlie" between "Beta" and "Delta" requires shifting every record after "Beta" down by one spot. This is expensive.
- The Deletion Problem: When a record is deleted, does it leave a "hole" (wasted space)? A good file organization must have a strategy to reuse that space immediately or mark it for later reclamation (Garbage Collection).

NOTE

15.4.5. Data Integrity and Reliability

File organization is responsible for ensuring that the data stored is the data retrieved.

- Consistency: If the system crashes in the middle of writing a record, the file organization should ensure the file doesn't become corrupted.
- Redundancy: Some organizations (like RAID or distributed file systems) intentionally duplicate data to meet the objective of reliability—if one part of the disk fails, the data can be recovered from the copy.

15.4.6. Security and Access Control

While often handled by the Operating System, the file organization itself can support security objectives.

- Authorization: The structure may include metadata headers that define who can read, write, or execute the file.
- Encryption: The organization may support storing data in an encrypted format, ensuring that even if the physical disk is stolen, the raw bytes are unreadable without the key.

15.5 TERMINOLOGY USED IN FILE HANDLING

In file handling, precise terminology is the foundation of understanding. The subject bridges software (your program) and hardware (the disk), so the vocabulary often switches between "logical concepts" (how we imagine data) and "physical concepts" (how the machine stores it).

Below is the essential glossary of file handling, categorized by function and hierarchy.

15.5.1. The Data Hierarchy (Logical View)

These terms describe how data is organized from the user's or programmer's perspective.

- Field (Data Item):

  The smallest unit of meaningful data. It represents a single attribute of an entity.

  - *Example:* "Employee Name", "Date of Birth", or "Salary".
  - *Fixed-Length Field:* Allocating exactly 20 bytes for a name, regardless of the actual name length.
  - *Variable-Length Field:* Using a delimiter (like a comma) to separate data, so "Tom" takes 3 bytes and "Christopher" takes 11.

NOTE

- Record:

  A collection of related fields treated as a single unit. A record represents one specific entity or object.

  - *Example:* All information about one specific employee (Name + ID + Salary).
  - *Record Key:* A specific field used to identify the record (see Section 3).

- File:

  A collection of related records stored as a single named unit on secondary storage.

  - *Example:* Employees.dat (containing 5,000 employee records).

- Database:

  A collection of interrelated files (tables) designed to minimize redundancy and allow for complex relationships between data.

15.5.2. Physical Storage Terms (Hardware View)

These terms describe how the Operating System and the Disk Drive physically handle the data.

- Sector:

  The smallest physical storage unit on a hard disk platter. Traditionally 512 bytes. You cannot write "half a sector"—you must write the whole thing.

- Track:

  A concentric circle on the disk platter containing a sequence of sectors.

- Cylinder:

  The set of all tracks that are at the same distance from the center across all platters in a disk drive. When the read/write arm moves, it accesses a "cylinder" of data.

- Block (Physical Record):

  This is the most critical term in file handling performance.

  - The disk talks in "Sectors", but the Operating System (OS) talks in "Blocks".
  - A block is the smallest unit of data transferred between memory and disk in a single I/O operation.
  - Often, one Block = Multiple Records.

NOTE

- Blocking Factor (B_f):

  The number of logical records stored in a single physical block.

  - *Formula:*

    Blocking\ Factor = \frac{Block\ Size}{Record\ Size}

  - If a Block is 4096 bytes and a Record is 100 bytes, the Blocking Factor is roughly 40.

15.5.3. Operational Terminology (Processing)

These terms describe the mechanisms used when a program interacts with a file.

- Buffer:

  A temporary storage area in Main Memory (RAM).

  - *Why it's needed:* The CPU is fast; the Disk is slow. We cannot write to the disk byte-by-byte (it would be too slow). instead, we fill a "Buffer" in RAM, and when it is full, we dump the whole thing to the disk in one go.

- Stream:

  An abstract flow of data.

  - *Input Stream:* Data flowing from the disk (source) into your program.
  - *Output Stream:* Data flowing from your program to the disk (destination).

- File Pointer (Seek Pointer):

  An internal marker that tracks the current position in the file.

  - When you open a file, the pointer is at 0 (the beginning).
  - If you read 10 bytes, the pointer moves to 10.
  - **Seek:** The operation of manually moving this pointer to a specific location (e.g., "Jump to byte 500").

- File Handle (File Descriptor):

  When you request to open a file, the OS gives your program a unique reference number (or object) called a Handle. You don't refer to the file by name (data.txt) after opening it; you refer to it by its Handle.

- EOF (End of File):

  A marker or signal indicating that no more data exists to be read.

15.5.4. Identification & Access Terms

These terms relate to how we find specific data within a file.

NOTE

- Key (Key Field):

  A field used to identify or search for a record.

  - Primary Key: A field that *uniquely* identifies a record (e.g., Student ID). No two records can have the same Primary Key.
  - Secondary Key: A field used for searching that might not be unique (e.g., "Last Name"—there might be five "Smiths").
  - Sort Key: The field used to sequence the records (e.g., sorting by Date).

- Index:

  A separate file or structure (like a table of contents) that contains Keys and their corresponding physical addresses (pointers). It speeds up search operations.

- Hashing:

  A mathematical technique used to calculate the address of a record directly from its Key, without using an index or searching sequentially.

- Collision:

  A situation in Hashing where two different Keys calculate to the same address (storage location).

15.5.5. File Attributes (Metadata)

Information *about* the file, rather than the contents of the file itself.

- Metadata: Data describing the file structure, such as file size, owner, permissions, and date of creation.
- Extension: The suffix (e.g., .txt, .csv, .exe) that indicates the file type and which application should open it.

Path: The string specifying the unique location of a file in the directory hierarchy (e.g., C:/Users/Docs/file.txt).

Check Your Progress 15.1

MCQ:
Which of the following describes a file in computer systems?
a) A sequence of records stored on secondary memory
b) A temporary variable in RAM
c) A stack of elements in memory
d) A pointer to a node in a linked list

True/False:
Files are always stored in the main memory (RAM).

Short Questions:

NOTE

1. Define a file in computer science.

.....

.....

2. Name two purposes of using files.

.....

.....

15.6 FILE ORGANIZATION CONCEPTS

File Organization refers to the logical structuring of records in a file. While the *Physical Storage* deals with blocks and tracks, the *File Organization* deals with how we map a logical record (like "User: John") to that physical space.

The choice of organization concept dictates how we can access the data. It is essentially a trade-off between flexibility (ease of random access) and efficiency (speed of sequential processing).

15.6.1. The Two Primary Access Paradigms

Before understanding the organizations, one must understand the two ways we access data, as these drive the design of the concepts below.

1. Sequential Access: Processing data in a strict order (Record 1, then Record 2, then Record 3...). This is like a cassette tape.
2. Random (Direct) Access: Retrieving specific data immediately without scanning predecessors. This is like a vinyl record or a Spotify playlist.

15.6.2. Sequential File Organization

This is the oldest and simplest concept. Records are stored one after another.

A. Pile (Heap) File

This is the "unorganized" organization.

- Concept: Records are collected in the order they arrive. If a new user registers, their record is simply appended to the very end of the file.
- Searching: To find a record, you must perform a linear search (scan from the start).
- Pros: Extremely fast insertion (O(1)).
- Cons: Terrible search speed (O(n)).

NOTE

B. Sorted Sequential File

- Concept: Records are sorted based on a Primary Key (e.g., Student ID). If you add a record "ID: 50", it must be inserted physically between "ID: 49" and "ID: 51".
- Searching: Allows for Binary Search (if fixed length), reducing complexity to O(\log n).
- Pros: efficient for searching; excellent for "Next" operations (Find next student).
- Cons: Insertion is very expensive because inserting a record in the middle requires shifting all subsequent records down.

15.6.3. Direct (Relative) File Organization

This concept is designed purely for speed in Random Access. It eliminates the need to "search" by calculating exactly where the data lives.

- Concept: The physical address of the record is determined directly from its Key using a mathematical formula.

  Address = f(Key)

- Relative Addressing: The system doesn't generate a hard disk track number (e.g., Track 9, Sector 2). Instead, it generates a "Relative Record Number" (e.g., the 50th slot in the file). The OS translates "Slot 50" to the physical disk location.

Activity 1: The "Digital Filing Cabinet" – Fixed-Length Record Management

Objective:

To understand how data is organized persistently on a disk. Students will move beyond simple text files (variable length) to implementing Fixed-Length Records in a binary file. This demonstrates the foundation of database storage, allowing them to calculate exactly where a specific record lives in a file (Random Access) without reading everything before it.

Materials / Tools / Software Required:

- Computer: Standard development setup.
- Programming Language: C++ (preferred for struct/byte handling) or Python (using the struct module).
- Hex Editor: A tool to view the raw binary content of a file (e.g., HxD, Hex Fiend, or a VS Code Hex Dump extension).
- Concept Guide: A diagram showing "Fields," "Records," and "Padding."

NOTE

Expected Outcome:

By the end of this activity, students will be able to:

5. Define a Record Structure: Create a class/struct (e.g., Student) with fixed sizes (e.g., ID is 4 bytes, Name is exactly 20 bytes).
6. Write Binary Data: Write several records to a file in Binary Mode. They will observe that the file size is exactly N \times \text{Record Size}.
7. Inspect with Hex Editor: Open the generated file in a Hex Editor to see how strings are padded with null bytes and how integers look in little-endian/big-endian format, demystifying the "black box" of file storage.
8. Implement Random Access: Write a function readRecord(index) that uses a "seek" operation (moving the file pointer to index * RecordSize) to retrieve the 50th record instantly, avoiding the need to read the first 49.

15.7 SUMMARY

File structure refers to the systematic arrangement of data in storage devices so that it can be stored, accessed, and managed efficiently. It defines how records and fields are organized within a file, which directly affects data retrieval speed and storage utilization. A well-designed file structure ensures smooth data processing and is essential in database systems, operating systems, and business applications.

The main objectives of file organization are to enable quick data access, efficient storage utilization, and easy updating of records. It aims to minimize retrieval time, reduce redundancy, and support operations such as insertion, deletion, and modification without disrupting the system. Proper file organization also ensures data integrity, security, and scalability, allowing systems to handle growing data volumes effectively.

In file handling, common terminology includes fields (individual data items), records (collection of related fields), files (collection of records), keys (unique identifiers), primary key, secondary key, and file access methods. Understanding these terms is important for managing and manipulating data efficiently.

File organization concepts include sequential, indexed, and direct (or random) file organization. Sequential organization stores records in order, suitable for batch processing. Indexed organization uses an index to locate records quickly. Direct organization allows immediate access

NOTE

using a key value. Each method is chosen based on application requirements and performance needs.

15.8 GLOSSARY

1. **File Structure:** The organization of data on secondary storage devices (like hard disks) including the operations used to access that data.
2. **Field:** The smallest unit of meaningful data (e.g., "Employee Name"), which can be fixed or variable in length.
3. **Record:** A collection of related fields treated as a single unit (e.g., all details of one employee).
4. **Blocking Factor (B_f):** The number of logical records that fit into a single physical block on the disk.
5. **Seek Time:** The time taken to move the disk's read/write head to the specific track or cylinder containing the data.
6. **Sequential Access:** Accessing data in a strict order, one record after another (similar to a cassette tape).
7. **Direct Access (Random Access):** Retrieving a specific record immediately using its address, without scanning previous records.
8. **Hashing:** A mathematical technique used to calculate the direct physical address of a record based on its Key.
9. **ISAM (Indexed Sequential Access Method):** A file organization technique that uses a separate index file to point to records stored sequentially in a data file.
10. **Metadata:** Data that describes the file itself rather than the content (e.g., file size, permissions, creation date).

15.9 ANSWER TO CHECK YOUR PROGRESS-POSSIBLE ONE

Check Your Progress 15.1

MCQ Answer:
a) A sequence of records stored on secondary memory.

True/False Answer:
False – files are stored in secondary storage like disks.

Short Answers:

1. A file is a collection of related records stored permanently in secondary memory.

NOTE

2. Purposes: Data persistence, organized storage for easy access and retrieval.

15.10 SUGGESTED READINGS – OER

Sr. No.	Title	URL	QR Code
1	Data Structure Using C++	http://books.aisectonline.com/User/UnitUrls?unitId=3723	

15.11 REFERENCE

- File Structures: An Object-Oriented Approach with C++ – Michael J. Folk, Bill Zoellick, Greg Riccardi – ISBN: 9788177583731 – 2012
- File Organization and Processing – Alan L. Tharp – ISBN: 9788126518685 – 2008
- Data Management and File Structures – Mary E. S. Loomis – ISBN: 9788120306714
- Database System Concepts – Avi Silberschatz, Henry F. Korth, S. Sudarshan – ISBN: 9780078022159 – 7th Edition (2019)
- Database Systems: The Complete Book – Hector Garcia-Molina, Jeffrey D. Ullman, Jennifer Widom – ISBN: 9780131873254 – 2nd Edition (2008)
- File Organization for Database Design – Author Unknown (McGraw-Hill Computer Science Series) – ISBN: 0070701334 – ~2004

15.12 EXERCISE

Part 1

C. Multiple choice question

1. **Which of the following is the primary bottleneck that File Structures aim to minimize?**

A) CPU Processing Time

B) RAM Usage

C) Disk I/O Operations

D) Network Bandwidth

NOTE

2. **What is the term for the number of logical records stored in one physical block?**

A) Cluster Count

B) Blocking Factor

C) Sector Size

D) File Pointer

3. **Which file organization method is best suited for generating a monthly payroll report for *all* employees?**

A) Direct/Hashed Organization

B) Sequential Organization

C) Pile Organization

D) Inverted List Organization

4. **In Hashing, what is the term for the situation where two different keys calculate to the same address?**

A) Fragmentation

B) Collision

C) Overflow

D) Seek Error

5. **Which modern data structure is most commonly used in databases to handle large files efficiently?**

A) Linked Lists

B) Stacks

C) B-Trees / B+ Trees

D) Queues

Answers with Explanations:

1. Answer: C) Disk I/O Operations
Explanation: Disk access is much slower than CPU and RAM operations, so file structures are designed to reduce the number of disk reads and writes.

2. Answer: B) Blocking Factor
Explanation: Blocking factor refers to how many logical records fit into a single physical disk block, improving storage efficiency.

3. Answer: B) Sequential Organization
Explanation: Payroll reports require processing all records in order, which makes sequential organization efficient for batch processing tasks.

NOTE

4. Answer: B) Collision
Explanation: A collision occurs when a hash function produces the same storage address for two different keys.

5. Answer: C) B-Trees / B+ Trees
Explanation: B-Trees and B+ Trees maintain sorted data and allow efficient searching, insertion, and deletion in large database files.

B. Small answer type question

1. Define "File Structure" and explain its primary relationship with secondary storage devices.
2. Differentiate between a Logical Record and a Physical Record (Block), and explain the concept of "Blocking."
3. List and briefly describe the fundamental operations (the "verbs") associated with file processing.
4. Distinguish between "Data Structure" and "File Structure" based on storage location, persistence, and focus.
5. Explain the terminology related to file identification: Primary Key, Secondary Key, and Sort Key.

Part 2

C. Long answer type question (answer in 200 words)

1. Describe Sequential File Organization, distinguishing between a "Pile" (Heap) and a "Sorted Sequential File," and discuss its pros and cons.
2. Explain the mechanism of Direct (Random) Access Organization using Hashing, including how the address is calculated and what "Collisions" are.
3. Analyze the Indexed Sequential Access Method (ISAM). How does it combine the features of sequential and direct organization?
4. Discuss the primary objectives of File Organization, specifically focusing on "Minimizing Input/Output (I/O) Operations" and "Optimal Record Retrieval."
5. Compare Sequential, Direct, and Indexed Sequential organizations in terms of Search Efficiency, Storage Efficiency, and Ease of Update.

Part 3

D. Long answer type question (answer in 300 words)

1. Describe the hierarchy of Physical Storage Terms (Sector, Track, Cylinder, and Block) and explain how the "Blocking Factor" is calculated.
2. Trace the modern evolution of file structures from the Tape Era to the Database Era (B-Trees).

NOTE

3. Explain the concept of Multi-Key (Inverted) File Organization and how "Inverted Lists" are used to search by secondary attributes like City or Department.
4. Discuss the specific challenges related to "Ease of Update" (Insertion and Deletion) in file organization and how different structures handle these operations.
5. Explain the limitations of ISAM that led to the development of B-Trees and B+ Trees, and how these tree structures optimize disk usage.

UNIT 16
TYPES OF FILE ORGANIZATION AND INDEXING

NOTE

UNIT STRUCTURE

16.1 Introduction

16.2 Objective

16.3 Sequential Files: A Comprehensive Guide

16.4 Direct (Random) File Organization: A Comprehensive Guide

16.5 Indexed Sequential File Organization (ISAM): A Comprehensive Guide

16.6 Summary

16.7 Glossary

16.8 Answer to check your progress

16.9 Suggested readings - OER

16.10 Reference

16.11Exercise

INTRODUCTION

File organization is a fundamental concept in data management that determines how records are stored, accessed, and maintained in secondary storage devices. As the volume of data grows, the efficiency of data retrieval and update operations becomes critically dependent on the method of file organization used. Unit 16 focuses on the major types of file organization and indexing techniques that have been traditionally used in computer systems and continue to influence modern database technologies.

This unit provides an in-depth understanding of Sequential File Organization, Direct (Random) File Organization, and Indexed Sequential File Organization (ISAM). Sequential files emphasize simplicity and efficiency in batch processing by storing records in a linear order. Direct file organization introduces hashing techniques to enable fast, real-time access to individual records. Indexed sequential organization combines the strengths of both methods by maintaining

NOTE

sorted data along with indexes to support both sequential and random access.

By studying these file organization techniques, learners gain insight into how data storage structures impact system performance, storage utilization, and reliability. This unit also highlights practical use cases such as payroll systems, banking applications, airline reservation systems, and inventory management, helping learners connect theoretical concepts with real-world applications. Overall, Unit 16 lays a strong foundation for understanding advanced database systems and file management strategies.

16.2 OBJECTIVE

After completing this unit, you will be able to:

1. Understand the concept and working principles of different types of file organization techniques.
2. Explain the structure, operations, advantages, and limitations of Sequential File Organization.
3. Analyze Direct (Random) File Organization using hashing techniques and collision resolution methods.
4. Describe the architecture and functioning of Indexed Sequential File Organization (ISAM).
5. Compare sequential, direct, and indexed sequential file organizations and identify their appropriate real-world applications.

16.3 SEQUENTIAL FILES: A COMPREHENSIVE GUIDE

A Sequential File is the most basic and enduring form of file organization in computer science. In this system, records are stored one after another in a contiguous or logical sequence. To access a specific record, the system must traverse the file from the very beginning until the desired data is found.

Think of a Sequential File like a cassette tape: to hear the 5th song, you must fast-forward through the first four. You cannot simply "jump" to the song instantly as you would on a vinyl record or CD (Random Access).

16.3.1 Core Philosophy

- Linearity: Data is processed in a linear stream (Record 1 \rightarrow Record 2 \rightarrow ... \rightarrow Record N).
- Simplicity: No complex addressing or indexing algorithms are required.

NOTE

- Dependency: The position of a record is determined by the position of the record preceding it.

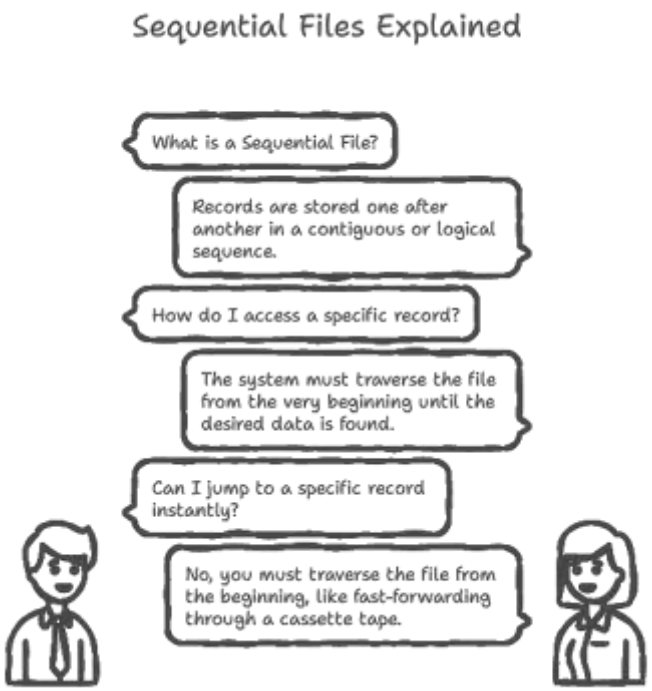


Fig 16.1: Sequential Files

16.3.2 Structure and Organization

Sequential files are typically stored on serial access devices like magnetic tapes, though they are frequently implemented on random-access disks (HDDs/SSDs) for specific performance benefits.

There are two primary methods for organizing records within a sequential file:

16.3.2.1 Pile File Method (Unsorted)

- Structure: Records are stored in the exact order they arrive.
- Insertion: New records are simply appended to the end of the file.
- Search: Searching requires a complete linear scan.
- Use Case: Ideal for data collection logs, error dumps, or chronological event recording.

16.3.2.2 Sorted File Method (Ordered)

- Structure: Records are sorted based on a Key Field (e.g., Employee ID or Date).

NOTE

- Insertion: Inserting a new record is expensive. The system must locate the correct position and shift all subsequent records to make space (or rewrite the file).
- Search: More efficient than Pile. If searching for ID 500 and you reach 501, you can stop searching because you know 500 doesn't exist. Binary search is possible *only* if the storage medium supports direct addressing (like a hard disk), not on tapes.

16.3.3 Key Operations

Handling sequential files requires a different mindset than modern databases. Operations are optimized for Batch Processing rather than real-time interactivity.

16.3.3.1 Reading

The system utilizes a "Next Record" pointer.

1. Open the file.
2. Read the first record into a memory buffer.
3. Process the data.
4. Advance the pointer to the next record automatically.
5. Repeat until the EOF (End of File) marker is reached.

16.3.3.2 Writing (Creation)

Writing is fast because new data is written to the end of the stream. There is no need to calculate disk addresses or update index trees.

16.3.3.3 Updating (The Master-Transaction Model)

Updating a sequential file is its most complex operation. You cannot easily "overwrite" a record in the middle of a file because the new data might be longer or shorter than the old data, shifting everything else.

Instead, the standard industry practice is the Batch Update Process:

1. Old Master File: The current version of the file (sorted by Key).
2. Transaction File: A separate file containing all changes (updates, additions, deletions), also sorted by the same Key.
3. Process: A program reads both files simultaneously.
    - If keys match: Apply the change.
    - If key exists in Transaction but not Master: Add new record.
    - If key exists in Master but not Transaction: Keep original record.

NOTE

4. New Master File: A completely new file is created with the merged/updated data.

16.3.4 Common Use Cases

Despite the dominance of SQL and NoSQL databases, sequential files remain critical in specific domains:

16.3.4.1 Batch Processing Systems

- Payroll**:** Calculating salaries for 10,000 employees at the end of the month. Since you have to pay *everyone*, accessing records one by one in order is the fastest method.
- Billing**:** Generating electricity bills for a whole city.

16.3.4.2 Logging and Auditing

- Server logs, error logs, and clickstreams are naturally sequential. They are written as "Pile Files" (time-ordered) and rarely modified, only read for analysis.

16.3.4.3 Backups and Archival

- Long-term storage of data (Cold Storage) is often done on magnetic tapes using sequential organization because it is the cheapest and most durable way to store petabytes of data that are rarely accessed.

16.3.5 Technical Summary (For Developers)

- Complexity**:** O(n) for search; O(1) for appending (Pile).
- Storage efficiency**:** Close to 100% (headers/footers are negligible).
- Bottleneck**:** I/O bound. The speed is limited by how fast the device can stream data, not CPU processing.

16.4 DIRECT (RANDOM) FILE ORGANIZATION: A COMPREHENSIVE GUIDE

Direct File Organization (often called Random Access or Hashed File Organization) is a method of storing data where records can be accessed directly, without physically traversing the preceding records.[1]

Think of a Direct File like a music CD or a Spotify playlist: if you want to play track #8, the laser jumps immediately to that specific location. You do not need to fast-forward through tracks 1 through 7.

16.4.1 Core Philosophy

- Non-Linearity**:** Data is not necessarily stored in adjacent memory blocks.

NOTE

- Mathematical Mapping**:** The physical location of the record is determined by a mathematical calculation performed on the record's Key (e.g., ID number).[2]
- Independence**:** Accessing Record B does not depend on Record A.

16.4.2 Structure: How It Works (The Hashing Algorithm)

In sequential files, the "address" is determined by order.[3] In Direct files, the address is calculated. This relies heavily on a technique called Hashing.

The Process

The system uses a Hash Function (mapping function) to translate a record's Key into a physical storage address.[4]

Address = f(Key)

1. Input: You provide the Key (e.g., Student ID 1045).
2. Process: The function calculates the address (e.g., Divide 1045 by the file size).
3. Output: The system yields a specific Block or Bucket number (e.g., Block 45).
4. Action: The read/write head jumps instantly to Block 45 to deposit or retrieve the data.

16.4.3 The Challenge: Handling Collisions

The biggest downside of Direct File Organization is the Collision.

A collision occurs when two different keys calculate to the same address.

- *Example:* If your function is Key % 10, then Key 25 and Key 35 both map to address 5. They cannot both occupy the same slot.

To manage this, Direct Files use **Collision Resolution Techniques**:

16.4.3.1 Linear Probing (Open Addressing)

If the calculated address is occupied, the system looks at the **next** available slot.[6]

- *Scenario:* Address 5 is full. Try 6. If 6 is full, try 7. Place record in 8.

16.4.3.2 Chaining (Linked Lists)

Instead of storing the record directly in the slot, the slot contains a "pointer" to a linked list.

- *Scenario:* Address 5 contains a list. Both Key 25 and Key 35 are attached to that list "hanging" off Address 5.

NOTE

16.4.3.3 Overflow Areas

The main file is kept clean. Any colliding records are thrown into a separate "Overflow File." When searching, if the record isn't in the primary address, the system checks the Overflow area.

16.4.4Key Operations

16.4.4.1 Inserting

1. Apply Hash Function to the new record's Key.
2. Check the calculated address.
    - Empty? Write the record.
    - Occupied? Apply collision resolution (e.g., find next spot).

16.4.4.2 Searching (Retrieval)

This is where Direct Files shine.

1. Apply Hash Function to the Key.
2. Go to the calculated address.
3. Read the record.
    - *Result:* This is typically an O(1) operation—instant access regardless of file size.

16.4.4.3 Updating

Unlike sequential files, you do not need to rewrite the whole file.

1. Jump to the address.
2. Read the record into memory.
3. Modify the data.
4. Rewrite the record back to the same location (assuming the record length hasn't exceeded the block size).

16.4.4.4 Deleting

Deletion can be tricky. You cannot simply remove a record if you are using Linear Probing, because that record might be a "bridge" to finding another record that collided earlier. Instead, records are usually marked with a logical flag called a "Tombstone" (e.g., isDeleted = True), allowing future searches to skip over it but eventually overwrite it.

16.4.5 Common Use Cases

Direct file organization is the backbone of systems requiring real-time interactivity:

NOTE

1. Online Banking Systems (OLTP)

When you check your account balance at an ATM, the system uses your Account Number (Key) to hash directly to your record. It cannot afford to scan millions of other accounts to find yours.

2. Airline Reservation Systems

When you book a seat, the system must instantly check the status of a specific flight ID. Speed is critical to prevent double-booking.

3. Key-Value Stores (NoSQL)

Modern databases like Redis or DynamoDB utilize concepts derived from Direct File Organization (Distributed Hash Tables) to serve data at massive scale with low latency.

16.4.6 Technical Summary (For Developers)

- Complexity: O(1) average case; O(n) worst case (if severe collisions occur).
- Efficiency Factor: depends heavily on the "Load Factor" (7α).[8]
    - $\alpha = \frac{\text{Number of Records}}{\text{Total Slots}}$
    - If α gets too high (> 0.7 or 70% full), collisions skyrocket and performance degrades to sequential speeds.
- Re-hashing: When a direct file becomes full, you often have to create a larger file and "re-hash" (move) every single record to the new structure.

Check Your Progress 16.1

MCQ:
Which file organization stores records one after another in a sequence?
a) Sequential File
b) Indexed File
c) Direct/Hashed File
d) B-Tree File

True/False:
Indexed file organization allows faster access to records compared to sequential files.

Short Questions:

1. Define sequential file organization.

.....

NOTE

.....

2. Name two advantages of indexed file organization.

.....

.....

16.5 INDEXED SEQUENTIAL FILE ORGANIZATION (ISAM): A COMPREHENSIVE GUIDE

Indexed Sequential File Organization—often referred to as ISAM (Indexed Sequential Access Method)—is a hybrid approach that attempts to combine the best features of both Sequential and Direct (Random) file organizations.[1]

Think of an Indexed Sequential file like a textbook:

- The Pages (Data): The content is written sequentially from Chapter 1 to Chapter 10. You can read it effectively from start to finish.
- The Index (Map): At the back of the book, there is an alphabetical list of topics with page numbers. If you want to find "Gravity," you check the index, jump to page 145, and then read just that page. You don't read pages 1-144.

16.5.1 Core Philosophy

- Duality: It supports Sequential Access (for batch processing) and Random Access (for individual queries) equally well.[2]
- Sorted Data: The actual data records are stored in sorted order (usually ascending) based on a Primary Key.
- Index Support: A separate file (the Index) is maintained to track the physical addresses of the data blocks.

16.5.2 Structure and Architecture

Unlike the "Pile" or "Hash" methods, ISAM relies on a hierarchical structure consisting of three distinct distinct areas.[3]

A. The Data Area (Sequential)

This is where the actual records reside.

- Records are stored in blocks (groups of records).
- Within each block, records are sorted by the Key (e.g., Employee ID).[4]

NOTE

- There is usually empty space left in these blocks initially to allow for easy insertions.

B. The Index Area

This acts as the "directory" for the data. It does not contain the data itself, but pointers to where the data is.[5] ISAM often uses a hierarchy of indexes:

1. Master Index: The top-level summary. It points to a specific cylinder or large section of the disk.
2. Cylinder/Block Index: A more detailed index that points to the specific track or block where the record key *should* be.

Note: The index is typically "Sparse." It doesn't list every single record; it lists the highest key in each block. If you are looking for Key 45, and the index says Block A ends with 40 and Block B ends with 50, you know to search Block B.

C. The Overflow Area

This is a critical safety valve.

- Because the main Data Area is sorted, inserting a new record is difficult if a block is full.
- Instead of rewriting the whole file (as in Sequential), the new record is placed in a separate Overflow Area, and a pointer is linked from the original block to this new location.

16.5.3 Key Operations

16.5.3.1 Searching (Random Retrieval)

This process is logarithmic rather than linear.

1. Consult the Index: The system looks up the Key in the Index file.
2. Locate Block: The index provides the address of the specific data block that contains the range of keys including the target.[6]
3. Scan Block: The system goes to that block and scans it sequentially (in memory) to find the specific record.
    - *Efficiency:* Instead of searching 1,000,000 records, you might search an index of 1,000 entries, jump to a block, and scan 10 records.

16.5.3.2 Sequential Processing

If you need to generate a full report (e.g., "Print all Student Grades"), the system ignores the index entirely. It simply opens the Data Area and reads from the first record to the last, enjoying the high speed of sequential access.

16.5.3.3 Insertion and Maintenance

Insertion is the "Achilles' Heel" of ISAM.

1. Locate Position: Find the correct block where the Key belongs.
2. Insert:
    - *If space exists:* Shift records in that block and insert the new one.
    - *If block is full:* Move the displaced record into the Overflow Area and update the pointers.
3. Degradation: As more records are pushed into Overflow, the file becomes fragmented ("spaghetti code" for data). Retrieval slows down because the read-head has to jump back and forth between the Data Area and Overflow Area.[7]
4. Reorganization: Periodically, the file must be "reorganized" (flattened)—the overflow records are merged back into the main sequence, and the index is rebuilt.

16.5.4 Common Use Cases

ISAM was the standard for business computing (mainframes) for decades before modern SQL databases took over, and the concept survives in modern database engines (like InnoDB in MySQL).

1. Supermarket Inventory

- Interactive: A cashier scans a barcode (Key). The system uses the Index to find the price instantly.
- Batch: At night, the manager runs a report listing "All items with stock < 10." The system reads the file sequentially to generate the list.

2. Library Management

- Interactive: A student searches for a specific book ID to see if it is available.
- Batch: The librarian runs a yearly audit of all books ordered by acquisition date.

3. Student Record Systems

- Universities often need to look up a single student's transcript (Random) but also generate a sorted list of all graduating seniors (Sequential/Range).

NOTE

Activity 1: The "Speed Dial" – Implementing an Index File

Objective: To demonstrate the efficiency of Indexed Sequential organization compared to pure Sequential access. Students will simulate a database system by creating two separate files: a large "Master File" containing full data records, and a lightweight "Index File" containing only Keys and Byte Offsets, allowing for rapid retrieval without scanning the entire dataset.

Materials / Tools / Software Required:

- Computer: Standard development setup.
- Programming Language: Python or C++ (languages with easy file seeking capabilities).
- Dataset: A text or CSV file containing 1,000 entries (e.g., ID, Name, Address, Phone), sorted by ID.
- Visual Guide: A diagram showing the mapping between an Index Key and the physical location in the Master File.

Expected Outcome: By the end of this activity, students will be able to:

1. Generate the Master File: Write a program to store the full dataset sequentially on the disk.
2. Build the Index: Write a script that scans the Master File and extracts just the Primary Key (e.g., ID) and its Byte Offset (location using tell() or ftell()), saving these pairs into a separate, much smaller "Index File."
3. Perform an Indexed Search: Implement a search function that:
    - First, loads the small Index File into memory (or binary searches it on disk).
    - Finds the correct offset for a requested ID.
    - Uses seek() to jump directly to that spot in the Master File to retrieve the full record.
4. Compare Costs: Observe that searching the index requires reading only a few bytes, whereas a sequential search would require reading megabytes of data to find a record near the end of the file.

16.6 SUMMARY

File organization is a critical concept in computer science that determines how records are stored, accessed, and managed on storage devices. The most fundamental type is Sequential File Organization, which functions analogously to a cassette tape. In this system, records are stored continuously in a linear order, meaning that to access a

NOTE

specific item, the computer must scan through all preceding records. While this O(N) access time makes it inefficient for finding single items, it is highly optimized for batch processing tasks—such as processing payroll or analyzing logs—where high volumes of data are read in order. However, updating these files is resource-intensive, often requiring a "Master-Transaction" model where the entire file is rewritten to insert or modify data.

To overcome the slowness of linear searches, Direct (Random) File Organization allows for immediate data retrieval, similar to selecting a specific track on a music CD. By using hashing algorithms, the system mathematically calculates the exact physical address of a record based on its key (like an ID number), achieving near-instant O(1) access. This speed makes it the standard for real-time applications like online banking and airline reservations. The downside is complexity and storage waste; the system requires extra "breathing room" to avoid collisions—instances where two keys calculate to the same address—and must employ complex resolution techniques like chaining or probing when they occur.'

Finally, the Indexed Sequential Access Method (ISAM) serves as a hybrid solution, combining the best features of the previous two. Functioning like a textbook with a dedicated index, ISAM stores data sequentially to support efficient range queries and batch reports, while simultaneously maintaining a separate index file that points to data blocks for fast random access.

16.7 GLOSSARY

1. Sequential File: A storage method where records are arranged linearly. To access the Nth record, you must traverse the previous N-1 records.
2. Direct (Random) Access: A method allowing immediate access to a specific record without scanning predecessors, typically using a mathematical formula to find the address.
3. ISAM (Indexed Sequential Access Method): A hybrid organization using a sequential data file combined with a separate Index file to support both batch and random processing.
4. Hashing: An algorithm used in Direct files that calculates a record's physical disk address (Key \rightarrow Address) based on its unique Key.
5. Collision: An error in hashing where two different Keys calculate to the same physical address (slot), requiring resolution techniques.
6. Overflow Area: A reserve storage space in ISAM used to hold new records when the main data block is full, preventing the need to rewrite the whole file.

NOTE

7. Master File: The primary, permanent up-to-date file in a sequential system containing the current status of all records.
8. Transaction File: A temporary file containing changes (updates/deletions) intended to be applied to the Master File.
9. Linear Probing: A collision resolution technique where, if a target slot is full, the system searches the very next available slot sequentially.
10. Activity Ratio: A metric measuring the percentage of records accessed in a run; high activity favors Sequential files, while low activity favors Direct files.

16.8 ANSWER TO CHECK YOUR PROGRESS-POSSIBLE ONE

Check Your Progress 16.1

MCQ Answer:
a) Sequential File – stores records in order, one after another.

True/False Answer:
True – indexed files provide faster access using an index.

Short Answers:

1. Sequential file organization stores records in a logical order, one after another.
2. Advantages: Faster search using index, easy record retrieval without scanning entire file.

16.9 SUGGESTED READINGS – OER

Sr. No.	Title	URL	QR Code
1	Data Structure Using C++	http://books.aisectonline.com/User/UnitUrls?unitId=3724	

16.10 REFERENCE

- File Structures: An Object-Oriented Approach with C++ – Michael J. Folk, Bill Zoellick, Greg Riccardi – ISBN: 9788177583731 – 2012

NOTE

- File Organization and Processing – Alan L. Tharp – ISBN: 9788126518685 – 2008
- Data Management and File Structures – Mary E. S. Loomis – ISBN: 9788120306714
- Database System Concepts – Avi Silberschatz, Henry F. Korth, S. Sudarshan – ISBN: 9780078022159 – 7th Edition (2019)
- Database Systems: The Complete Book – Hector Garcia-Molina, Jeffrey D. Ullman, Jennifer Widom – ISBN: 9780131873254 – 2nd Edition (2008)
- File Organization for Database Design – Author Unknown (McGraw-Hill Computer Science Series) – ISBN: 0070701334 – ~2004

16.11 EXERCISE

Part 1

A. Multiple choice question

1. Which file organization method is most analogous to a Cassette Tape?

A) Direct File

B) Indexed Sequential File (ISAM)

C) Sequential File

D) B-Tree

2. In Direct File Organization, what is the primary cause of a "Collision"?

A) The file size is too small.

B) Two different keys hash to the same physical address.

C) The hard disk has a bad sector.

D) The user tries to read and write simultaneously.

3. Which component of ISAM acts as a "directory" pointing to the location of data blocks?

A) The Overflow Area

B) The Master File

C) The Transaction Log

D) The Index

4. Which organization type is theoretically fastest (O(1)) for retrieving a single specific record?

A) Sequential

NOTE

B) Pile File

C) Direct (Hashed)

D) ISAM

5. Why are "Overflow Areas" used in ISAM?

A) To store the Index file separately.

B) To keep a backup of deleted records.

C) To store new records when the sorted data block is full.

D) To increase the speed of sequential batch processing.

Answers with Explanations:

1. Answer: C) Sequential File
Explanation: A cassette tape must be played from the beginning to reach a specific song, just like a sequential file must be read from the start to find a particular record.

2. Answer: B) Two different keys hash to the same physical address.
Explanation: In hashing, a function converts keys into storage addresses. A collision occurs when two different keys generate the same address.

3. Answer: D) The Index
Explanation: The index stores key values along with pointers to corresponding data blocks, functioning like a directory for quick lookup.

4. Answer: C) Direct (Hashed)
Explanation: Direct file organization uses a hash function to compute the storage address directly, allowing constant-time (O(1)) access in ideal cases.

5. Answer: C) To store new records when the sorted data block is full.
Explanation: When the main data block is full, additional records are placed in an overflow area to maintain sorted order without reorganizing the entire file.

B. Small answer type question

1. Define Sequential File Organization and explain the "Master-Transaction" model used for updating these files.
2. Differentiate between the "Pile File" (Unsorted) and "Sorted File" methods within sequential organization.
3. Explain the core concept of Direct (Random) File Organization and how it differs from Sequential Organization in terms of data access.
4. Define "Collision" in the context of Direct Files and list two common techniques used to resolve them.

NOTE

5. Briefly explain the three distinct areas that make up the architecture of an ISAM (Indexed Sequential Access Method) file

Part 2

C. Long answer type question (answer in 200 words)

1. Analyze the advantages and disadvantages of Sequential File Organization. Why is it still considered ideal for "Batch Processing" tasks like payroll?
2. Describe the mechanism of "Hashing" in Direct File Organization. How does the system translate a logical Key into a physical address?
3. Discuss the "Insertion" and "Maintenance" challenges in ISAM, specifically focusing on the role of the "Overflow Area" and the need for periodic reorganization.
4. Compare Sequential, Direct, and ISAM organizations based on three criteria: Search Cost, Storage Efficiency, and Best Use Case.
5. Explain why Direct File Organization is preferred for Online Transaction Processing (OLTP) systems like banking, while ISAM is often used for general-purpose business applications.

Part 3

D. Long answer type question (answer in 300 words)

1. Explain the "Master-Transaction" update model used in Sequential File Organization. Why is a new file created instead of modifying the existing one?
2. Describe the specific algorithms used in Hashing: Division-Remainder, Folding, and Mid-Square. How does each generate an address from a Key?
3. Compare "Linear Probing" (Open Addressing) and "Chaining" (Linked Lists) as techniques for resolving collisions in Direct Files.
4. Distinguish between the "Pile File" (Heap) and "Sorted Sequential File" regarding their Insertion and Search complexities.
5. Detail the three structural components of an ISAM file (Data Area, Index Area, and Overflow Area) and the specific function of each component.

DR. C.V. RAMAN UNIVERSITY

Chhattisgarh, Bilaspur / AN AISECT GROUP UNIVERSITY

Approved by : PCI | AICTE | NCTE | BCI | Member of : AIU | Joint Committee : (UGC | DEB | AICTE) | Recognized by : UGC | A NAAC Accredited University

DR. C. V. RAMAN UNIVERSITY (C.G.)

SLM FEEDBACK FORM

PROGRAMME : _____

COURSE : _____

BLOCK : _____

UNIT : _____

Dear Student,

While studying the units of this block, you may have found certain portions of the text difficult to comprehend. We wish to know your difficulties and suggestions, in order to improve the course. Therefore, we request you to fill up and send us the following questionnaire, which pertains to this block. If you find the space provided insufficient, kindly use a separate sheet.

Please mail to aisectcontent@aisect.org/ for online submission please scan QR code:

Questionnaire

1. **Name** : _____

2. **Enrolment No.** : | | | | | | | | | | | |

3. **How many hours did you need for studying the units?** : | |

4. **Please give your reactions to the following items based on your reading of the block:**

Items	Excellent	Very Good	Good	Poor	Give specific Examples (If poor)
Presentation Quality					
Language and Style					
Illustrations					
Used (diagrams, tables, etc.)					
Conceptual Clarity					
Check Your Progress Questions					
Suggested Readings - OER					

5. **Any other comments:**

_____

_____

_____

_____

***