

MCA

DATA STRUCTURES AND ALGORITHMS

How to Use Self-Learning Material?

The pedagogy used to design this course is to enable the student to assimilate the concepts with ease. The course is divided into modules. Each module is categorically divided into units or chapters. Each unit has the following elements:

-  **Table of Contents:** Each unit has a well-defined table of contents. *For example: “1.1.1. (a)” should be read as “Module 1. Unit 1. Topic 1. (Sub-topic a)” and 1.2.3. (iii) should be read as “Module 1. Unit 2. Topic 3. (Sub-topic iii).”*
-  **Aim:** It refers to the overall goal that can be achieved by going through the unit.
-  **Instructional Objectives:** These are behavioural objectives that describe intended learning and define what the unit intends to deliver.
-  **Learning Outcomes:** These are demonstrations of the learner's skills and experience sequences in learning, and refer to what you will be able to accomplish after going through the unit.
-  **Self-Assessment Questions:** These include a set of multiple-choice questions to be answered at the end of each topic.
-  **Did You Know?:** You will learn some interesting facts about a topic that will help you improve your knowledge. A unit can also contain Quiz, Case Study, Critical Learning Exercises, etc., as metacognitive scaffold for learning.
-  **Summary:** This includes brief statements or restatements of the main points of unit and summing up of the knowledge chunks in the unit.
-  **Activity:** It actively involves you through various assignments related to direct application of the knowledge gained from the unit. Activities can be both online and offline.
-  **Bibliography:** This is a list of books and articles written by a particular author on a particular subject referring to the unit's content.
-  **e-References:** This is a list of online resources, including academic e-Books and journal articles that provide reliable and accurate information on any topic.
-  **Video Links:** It has links to online videos that help you understand concepts from a variety of online resources.

LEADERSHIP KLEF



President
Er. Koneru Satyanarayana



Vice Chancellor
Dr. G. Pardha Saradhi Varma



Pro-Vice Chancellor
Dr. N. Venkatram



Registrar
Dr. K. Subbarao

CREDITS

Author

M. M. Poornima

Director CDOE

C. Shanath Kumar

Instructional Designer

Nabina Das

Content Writer

P. Manjusha

Graphic Designer

B. Suchitra



First Edition, 2023.

KL Deemed to be University-CDOE has full copyright over this educational material. No part of this document may be produced, stored in a retrieval system, or transmitted, in any form or by any means.

Author's Profile



M. M. Poornima

M. M. Poornima, M.Sc and M.Tech, working as an Assistant Professor in the Department of CSA, Koneru Lakshmaiah Educational Foundation, has 10 years of teaching experience. Her research areas of interest include Computer Vision, Cloud Computing, IoT, and Artificial Intelligence.



Data Structures and Algorithms

Course Description

The field of Data Structures and Algorithms (DSA) includes studying methods for efficiently arranging and storing data and the development of algorithms intended to resolve issues involving these data structures. Comprehending DSA is a crucial ability for all computer science students, as individuals with a solid grasp of these ideas frequently demonstrate proficiency as programmers and ace interviews with prominent tech firms such as Google, Microsoft, Amazon, and Meta. This lesson attempts to provide a quick and clear introduction to DSA, an essential computer science component for efficiently maintaining and processing data. They greatly improve the pace at which tasks are executed and handle big datasets, among other frequent software difficulties.

The first module covers the fundamentals of time and space concerns, introducing flowcharts, algorithms, and algorithm complexity analysis. In addition to learning about static and dynamic memory allocation, students will comprehend and categorise data structures. The three main subjects are strings, pointers, and arrays. We discuss pointers, which are used to access memory locations dynamically, as well as one-dimensional and multidimensional arrays. The discussion of string manipulation and string library functions offers a strong basis for comprehending the organisation and manipulation of data within memory.

Stacks and queues are two significant linear data structures that are the subject of the second module. Students will grasp how to implement stacks, carry out operations, and investigate their uses in expression evaluation and backtracking, among other applications. Stacks operate on the Last-In-First-Out (LIFO) principle. Additionally, the idea of several layers is presented. The remaining queues in the module operate on a First-In-First-Out (FIFO) basis. Students will investigate the use of queues and other processes. Different queues, such as circular, deque, and priority, are described, each with practical implications for scheduling and buffering.

Students in the third module study dynamic data structures like linked lists and trees. A linked list supports singly, doubly, and circularly linked lists and facilitates effective insertion and deletion. After that, the topic of tree structures is presented to the students. Starting with some fundamental vocabulary, they go on to several kinds of trees, such as binary, binary search, and AVL trees. Students grasp about pre-order, in-order, and post-order tree traversal approaches. More intricate tree structures, such as B trees and B+ trees, which are crucial for file systems and databases, are also covered in this session.

Graphs, a data structure that represents networks and relationships between things, are covered in the fourth module. Pupils interpret traversal methods such as Depth-First Search (DFS), Breadth-First Search (BFS), and graph representations. The shortest path algorithms—such as Bellman-Ford and Dijkstra's—have also been shown. Algorithms for sorting and searching are

covered next in the module. In addition to interpreting binary and linear search, students will grasp about fast, merge, and bubble sorting strategies. Finally, the curriculum covers hashing, where students investigate hash functions, collision resolution strategies, and the concept of perfect hashing for efficient data retrieval.



The course '**Data Structures and Algorithms**' is divided into **four** modules.

MODULE 1: INTRODUCTION TO DATA STRUCTURES

Algorithms and Flowcharts, Basics Analysis of Algorithm, Complexity of Algorithm, Introduction and Definition of Data Structure, Classification of Data, Arrays, Various Types of Data Structure, Static and Dynamic Memory Allocation, Function, and Recursion. Arrays, Pointers and Strings: Introduction to Arrays, Definition, One-Dimensional Array and Multidimensional Arrays, Pointer, Pointer to Structure, various Programs for Array and Pointer. Strings. Introduction to Strings, Definition, Library Functions of Strings.

MODULE 2: LINKED LISTS, STACKS, AND QUEUE

Introduction, Representation and Operations of Linked Lists, Singly Linked List, Doubly Linked List, Circular Linked List, And Circular Doubly Linked List. Stacks and Queue: Introduction to Stack, Definition, Stack Implementation, Operations of Stack, Applications of Stack and Multiple Stacks. Implementation of Multiple Stack Queues, Introduction to Queue, Definition, Queue Implementation, Operations of Queue, Circular Queue, De-queue and Priority Queue.

MODULE 3: TREES

Introduction to Tree, Tree Terminology Binary Tree, Binary Search Tree, Strictly Binary Tree, Complete Binary Tree, Tree Traversal, Threaded Binary Tree, AVL Tree, B Tree, B+ Tree.

MODULE 4: GRAPHS, SEARCHING, SORTING, AND HASHING

Introduction, Representation to Graphs, Graph Traversals Shortest Path Algorithms. Searching and Sorting: Searching, Types of Searching, Sorting, Types of sorting like quick sort, bubble sort, merge sort, and selection sort. Hashing: Hash Function, Types of Hash Functions, Collision, Collision Resolution Technique (CRT), Perfect Hashing.

Table of Contents

MODULE 1

INTRODUCTION TO DATA STRUCTURES

Unit 1.1 Overview of Algorithm

Unit 1.2 Overview of Data Structure

Unit 1.3 Functions and Arrays

Unit 1.4 Pointer and Strings

MODULE 2

LINKED LISTS, STACKS AND QUEUE

Unit 2.1 Overview of Linked Lists

Unit 2.2 Outline of Stack

Unit 2.3 Fundamental Concept of Queue

Unit 2.4 Types of Queues

MODULE 3

TREES

Unit 3.1 Exploring Trees

Unit 3.2 Tree Traversal and Balanced Trees

MODULE 4

GRAPHS, SEARCHING, SORTING, AND HASHING

Unit 4.1 Graphs

Unit 4.2 Searching and Sorting

Unit 4.3 Hashing

DATA STRUCTURES AND ALGORITHMS

MODULE 1

Introduction to Data Structures



(DEEMED TO BE UNIVERSITY)

Module Description

A data structure is a way to organise and store information so that it may be updated and accessed quickly. It efficiently processes, retrieves, and saves data on a computer in an organised manner. Almost every developed software system or program uses some form of data structure, both basic and sophisticated, so it is crucial to grasp these ideas well.

Introduction to Data Structures provides a basic interpretation of algorithms and data organisation. At the beginning of the subject, algorithms and flowcharts assist students in tackling problems. After that, the fundamentals of algorithm analysis are covered, emphasising time and space complexity to gauge an algorithm's performance efficiency.

In this section, data structures are introduced and defined, along with an explanation of how to organise and store data for quick access. Students will also examine static and dynamic memory allocation to interpret how memory is managed during program execution.

The section then explores arrays, introducing one-dimensional and multidimensional arrays essential for tabulating and organising homogeneous data in linear formats. Next, the function of pointers is examined, namely their reference to memory locations and their employment in dynamic data structures as pointers to structures.

The last section of the module covers strings, including how to define, work with, and utilise them, as well as important library functions. These make it possible to handle and analyse text-based data efficiently. Students will use practical programming examples throughout the module to interpret how arrays, pointers, and strings are implemented in various data manipulation scenarios.

As a prerequisite for more complex data management and optimisation subjects, this module gives students a firm grasp of fundamental data structures and algorithmic analysis.

The module consists of **four** units.

Unit 1.1: Overview of Algorithm

Unit 1.2: Overview of Data Structure

Unit 1.3: Functions and Arrays

Unit 1.4: Pointer and Strings

MODULE 1

Introduction to Data Structures

Unit 1

Overview of Algorithm



☰ Unit Table of Contents

Unit 1.1 Overview of Algorithm

Aim	09
Instructional Objectives	09
Learning Outcomes	09
1.1.1 Algorithms and Flowcharts	10
Self-Assessment Questions	23
1.1.2 Analysis and Complexity of Algorithm	24
Self-Assessment Questions	26
Summary	27
Terminal Questions	27
Answer Keys	28
Activity	28
Glossary	29
Bibliography	29
e-References	29
Video Links	30
Image Credits	30
Keywords	30



Aim

To provide students with a comprehensive interpretation of algorithms, their characteristics, flowcharts, and the fundamentals of asymptotic analysis.



Instructional Objectives

This unit is designed to:

- Describe the rules and characteristics of algorithms
- Create flowcharts using standard symbols to represent the sequence of steps in an algorithm visually
- Explain asymptotic analysis to evaluate the time complexity of algorithms using big-O, omega, and theta notations



Learning Outcomes

At the end of the unit, the student is expected to:

- Define an algorithm and explain its characteristics and rules
- Describe the steps involved in designing a flowchart for an algorithm
- Explain the types of asymptotic notations, big-O, omega, and theta, and their significance in evaluating time complexity

1.1.1 Algorithms and Flowcharts

Algorithm

An algorithm is a set of sequences and finite steps followed to accomplish a particular task.

Rules:

1. Steps must be finite.
2. All steps must lead to the solution.
3. A step must be executed in a finite time.
4. Can take 0 or more inputs.
5. Must produce at least one output.
6. It should be deterministic, giving the same output for the same input case.
7. The algorithm can be written to solve the problem, which humans can solve with paper and pencil in a finite amount of time.
8. Two algorithms can be compared based on speed and accuracy.

Dataflow of an Algorithm

- **Problem:** A problem can be a real-world problem or any instance of a real-world problem for which we need to create a program or a set of instructions. The set of instructions is known as an algorithm.
- **Algorithm:** An algorithm will be designed for a problem using a step-by-step procedure.
- **Input:** After designing an algorithm, the required and the desired inputs are provided to the algorithm.
- **Processing unit:** The input will be given to the processing unit, and the processing unit will produce the desired output.
- **Output:** The output is the program's outcome or result.

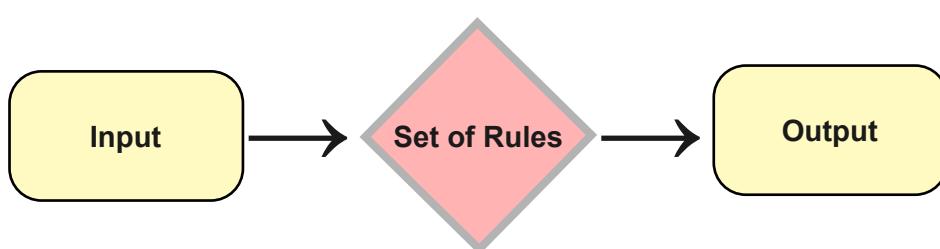


Fig. 1: Main components of an algorithm

Why do we need Algorithms?

Algorithms play a crucial role in computer science. The best algorithm ensures that the computer completes the task efficiently. When it comes to efficiency, a good algorithm is essential. An algorithm is extremely important for optimising a computer program.

- Grasping the fundamental concept of the problem
- To comprehend the problem's flow
- It gives the designer a concise definition of the problem's needs and aims.
- Discover a solution to the situation
- Enhance the effectiveness of existing methods
- It compares the algorithm's performance to that of other approaches.
- It is the easiest way to describe something without going into too much detail about how it works.
- To assess the techniques' performance in all circumstances (best cases, worst cases, and average cases)
- To determine the algorithm's resource requirements (memory, input-output cycles).
- We can evaluate and assess the complexity (in terms of time and space) of input size problems without developing and executing them, ultimately lowering the cost.

We will have to recognise and decide on trade-offs regularly. In addition to our problem-solving ability, we must acquire solution evaluation techniques. A problem can be solved in several ways. So, an algorithm is necessary for finding a solution and then assessing whether it is acceptable for repeatedly addressing a particular issue.

Characteristics of an algorithm

Standard instructions are followed while writing an algorithm. The conditions for a set of instructions to be considered an algorithm are:

- **Specified Input:** Those inputs should be well-defined if an algorithm requires inputs.
- **Specified Output:** The algorithm must explicitly indicate what output will be produced and should be well-defined.
- **Clear and Unambiguous:** The algorithm should be simple and comprehensive. Each of its steps should be distinct and lead to a single conclusion.
- **Finiteness:** Algorithms must come to a halt after a certain number of steps.
- **Feasible:** The algorithm must be simple, general, and practical. It must be able to be run with the available resources, and it should not incorporate any futuristic technology.
- **Independent:** Step-by-step instructions should be included in an algorithm and independent of any programming code.
- **Examples of Algorithms:**

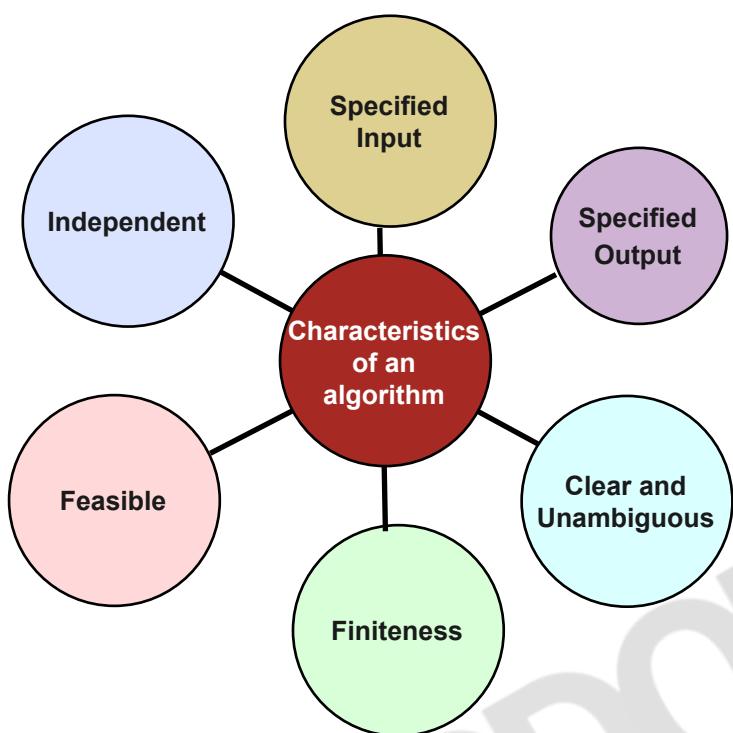


Fig. 2: Characteristics of an algorithm

Examples of Algorithms:

1. Algorithm to find the average of 3 numbers
 - **Step 1:** Start
 - **Step 2:** Declare variables num1, num2, num3, sum, average.
 - **Step 3:** Read values num1, num2, and num3.
 - **Step 4:** Add num1, num2, and num3 and assign the result to sum.
 - **sum←num1+num2+num3**
 - **Step 5:** Divide the sum by three and assign the result to the average.
 - **average←sum/3**
 - **Step 7:** Print Average
 - **Step 6:** Stop

Algorithm to find the product of 2 numbers

- **Step 1:** Start
- **Step 2:** Declare variables num1, num2, and product.
- **Step 3:** Read values num1, num2.

- **Step 4:** Multiply num1, and num2 and assign the result to product.
- **product←num1*num2**
- **Step 7:** Print the Product
- **Step 6:** Stop

Flowchart

A flowchart is a graphical representation of an algorithm. Programmers often use it as a program-planning tool to solve a problem. It uses symbols connected among themselves to indicate the flow of information and processing. Drawing a flowchart for an algorithm is known as “flowcharting”.

Basic Symbols Used in Flowchart Designs

- **Terminal:** The oval symbol indicates Start, Stop and Halt in a program's logic flow. A pause/halt is generally used in a program logic under some error conditions. Terminal is the first and last symbol in the flowchart.
- **Input/Output:** A parallelogram denotes any input/output type function. Program instructions that take input from input devices and display output on output devices are indicated with a parallelogram in a flowchart.
- **Processing:** A box represents arithmetic instructions. All arithmetic processes, such as adding, subtracting, multiplication, and division, are indicated by an action or process symbol.
- **The Decision:** The diamond symbol represents a decision point. Diamonds indicate decision-based operations such as yes/no questions or true/false in the flowchart.
- **Connectors:** Connectors are useful to avoid confusion whenever a flowchart becomes complex or extends over more than one page. A circle represents a connector.
- **Flow lines:** Flow lines indicate the exact sequence in which instructions are executed. Arrows represent the direction of the flow of control and the relationship among different flowchart symbols.

Symbol	Name	Function
	Start/end	An oval represents a start or end point
	Arrows	A line is a connector that shows relationships between the representative shapes
	Input/Output	A parallelogram represents input or output
	Process	A rectangle represents a process
	Decision	A diamond indicates a decision

Fig. 3: Symbols used in flowchart designs

Rules For Creating Flow chart

1. The flowchart must always begin with the keyword 'start.'
2. The flowchart must always end with the keyword 'end.'
3. An arrow should connect every symbol in the flowchart.
4. The decision-making symbol in the flowchart must be linked with an arrow line.

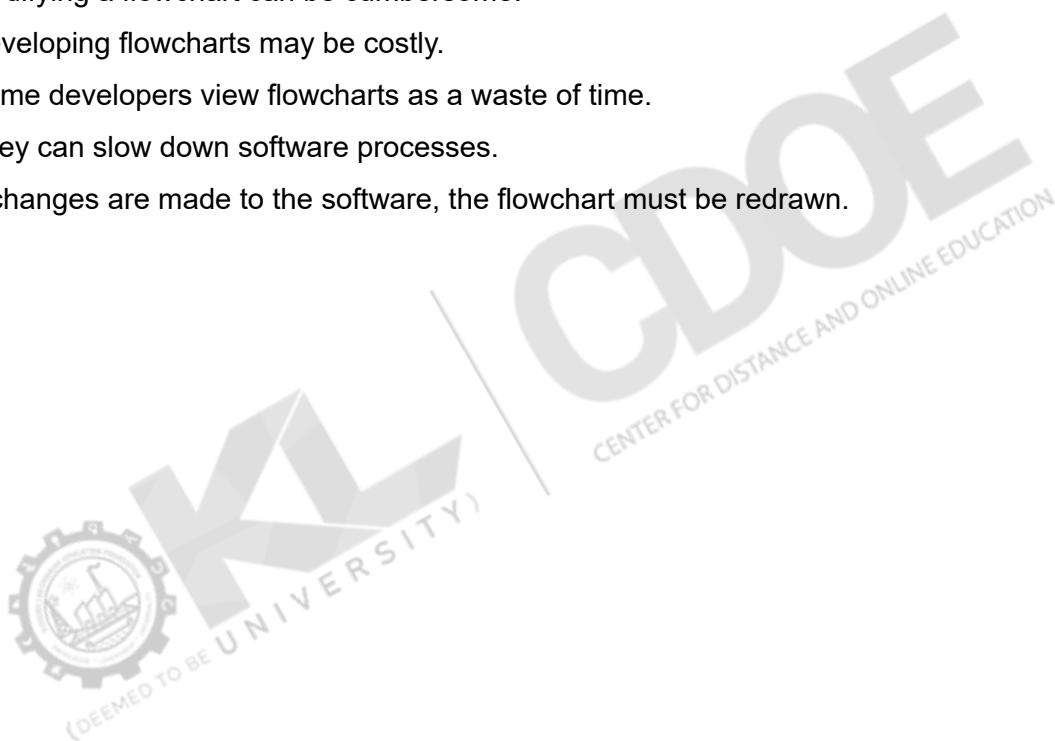
Advantages of Flowchart:

- Flowcharts provide a clear way to communicate the logic of a system.
- They act as a guide or blueprint during program design.
- Flowcharts assist in the debugging process.
- They help in analysing programs more effectively.
- Flowcharts enhance documentation.
- They serve as excellent documentation tools.

- They make it easier to trace errors in the software.
- Flowcharts are simple to comprehend.
- They can be reused for convenience in future scenarios.
- Flowcharts help ensure logical correctness in programs.

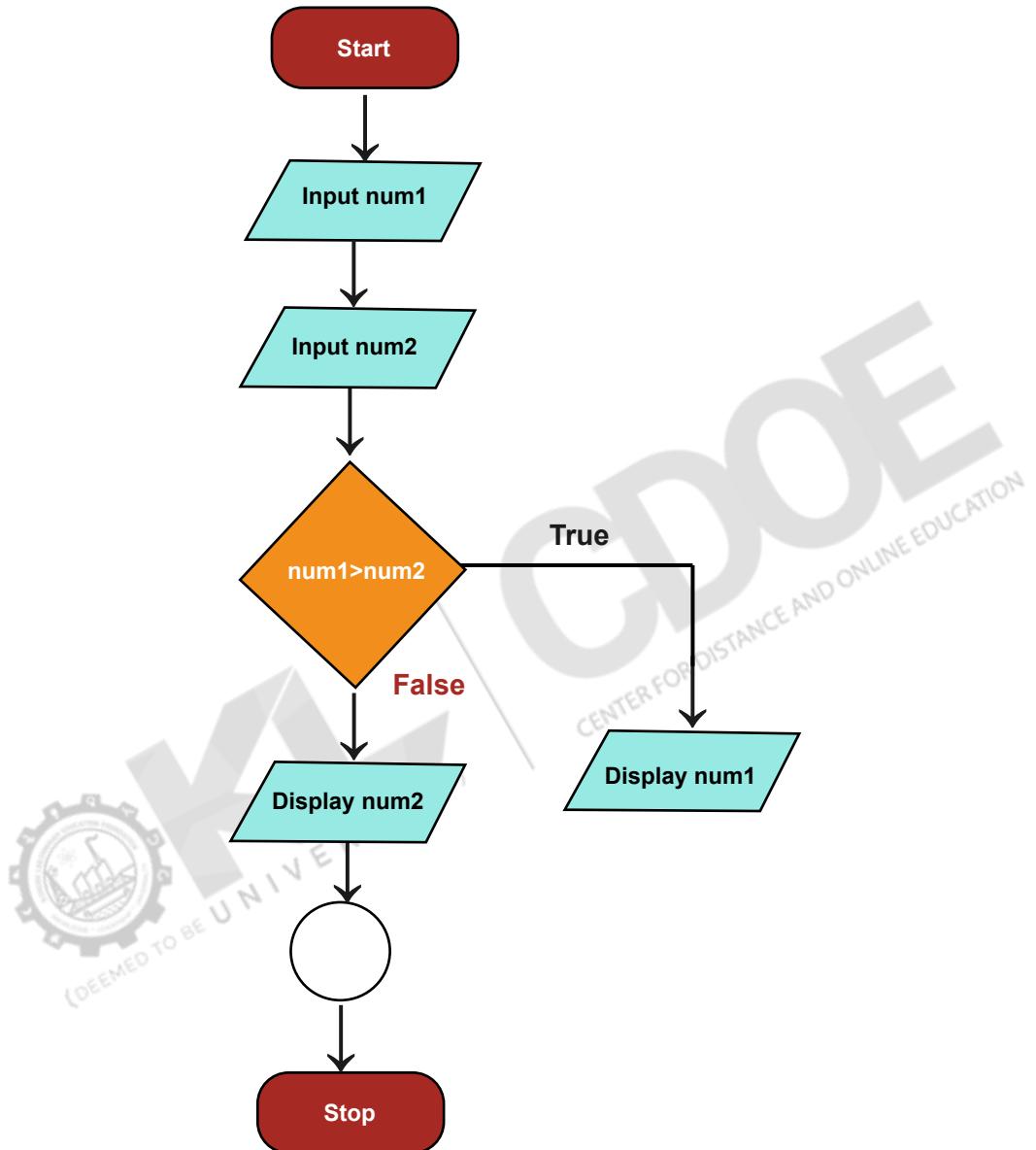
Disadvantages of Flowchart:

- Creating flowcharts for large or complex programs can be challenging.
- There is no universal standard for determining the level of detail.
- Reproducing flowcharts can be difficult.
- Modifying a flowchart can be cumbersome.
- Developing flowcharts may be costly.
- Some developers view flowcharts as a waste of time.
- They can slow down software processes.
- If changes are made to the software, the flowchart must be redrawn.



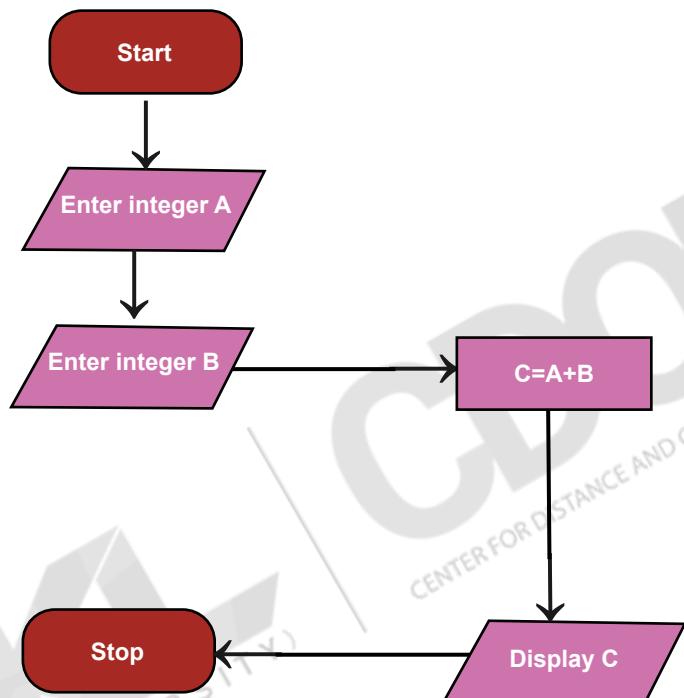
Example

Draw a flowchart to input two numbers from the user and display the largest of two numbers.



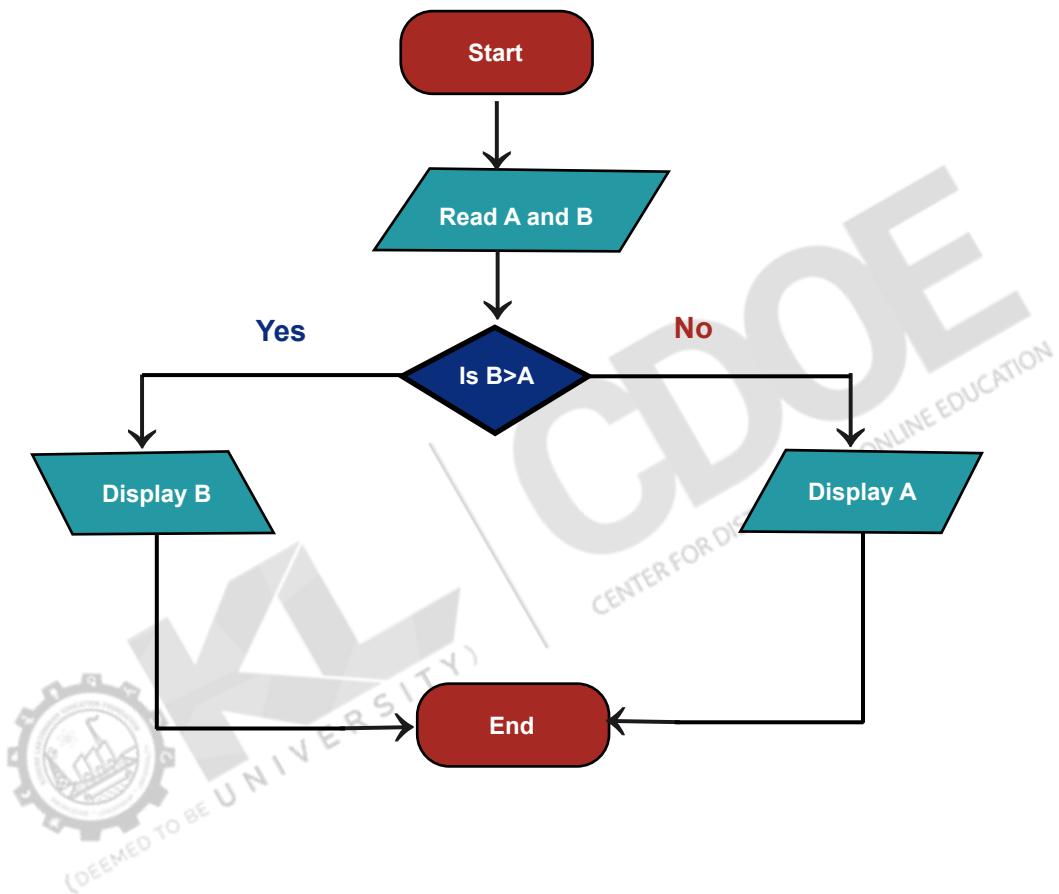
Example 1: Find the Sum of Two Numbers Entered

- **Step 1:** Read the Integer A.
- **Step 2:** Read Integer B.
- **Step 3:** Perform the addition using the formula: $C = A + B$.
- **Step 4:** Print the Integer C.



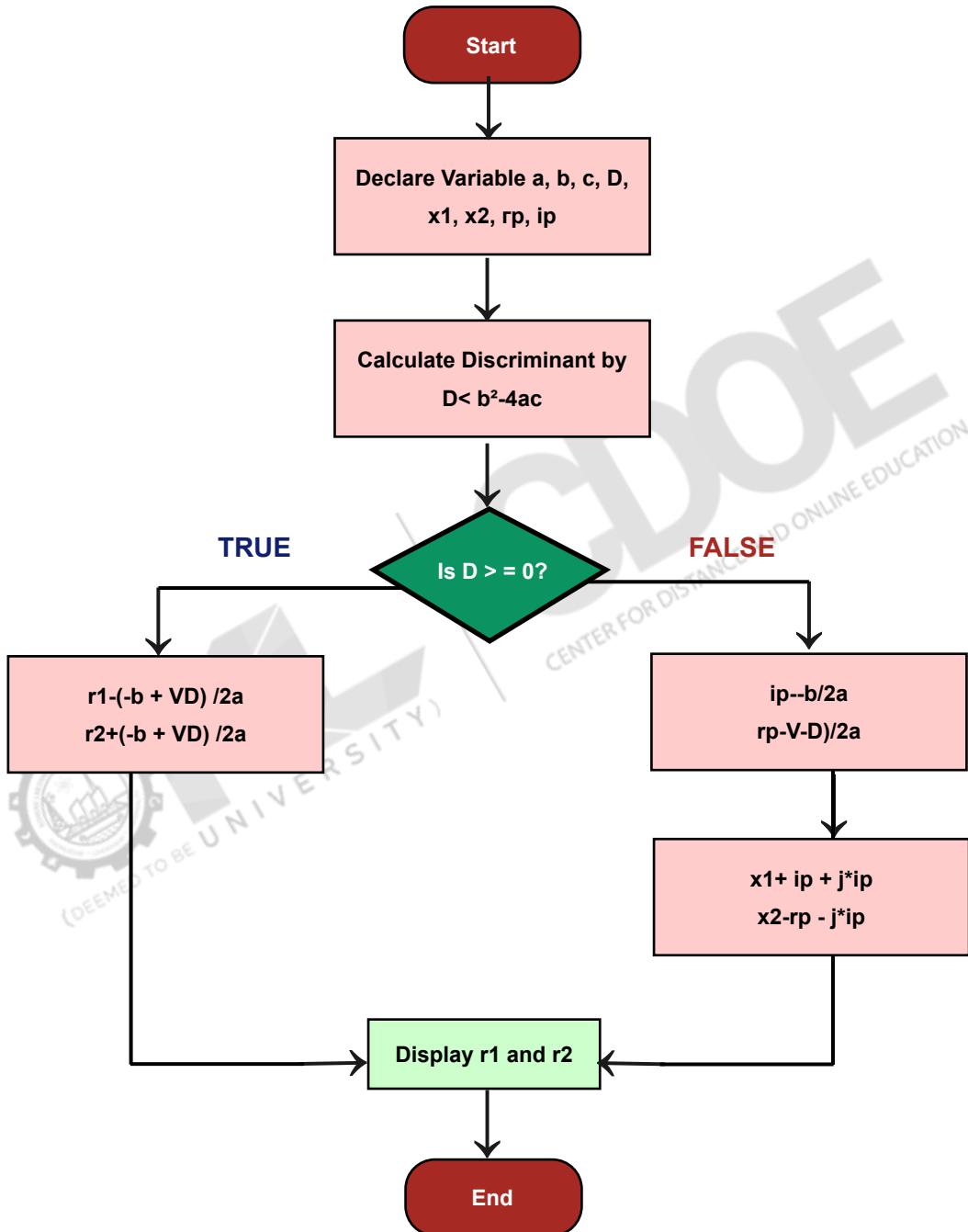
Example 2: Determining the Largest Number Among All the Entered Integers

- **Step 1:** Read the Integer A.
- **Step 2:** Read Integer B.
- **Step 3:** If B is greater than A, then print B, else A.



Example 3: Work Out All the Roots of a Quadratic Equation $ax^2 + bx + c = 0$

- **Step 1:** Enter the variables a, b, c, D, x1, x2, rp, and ip.
- **Step 2:** Evaluate the discriminant by using the formula: $D = b^2 - 4ac$
- **Step 3:** Print rp and ip.

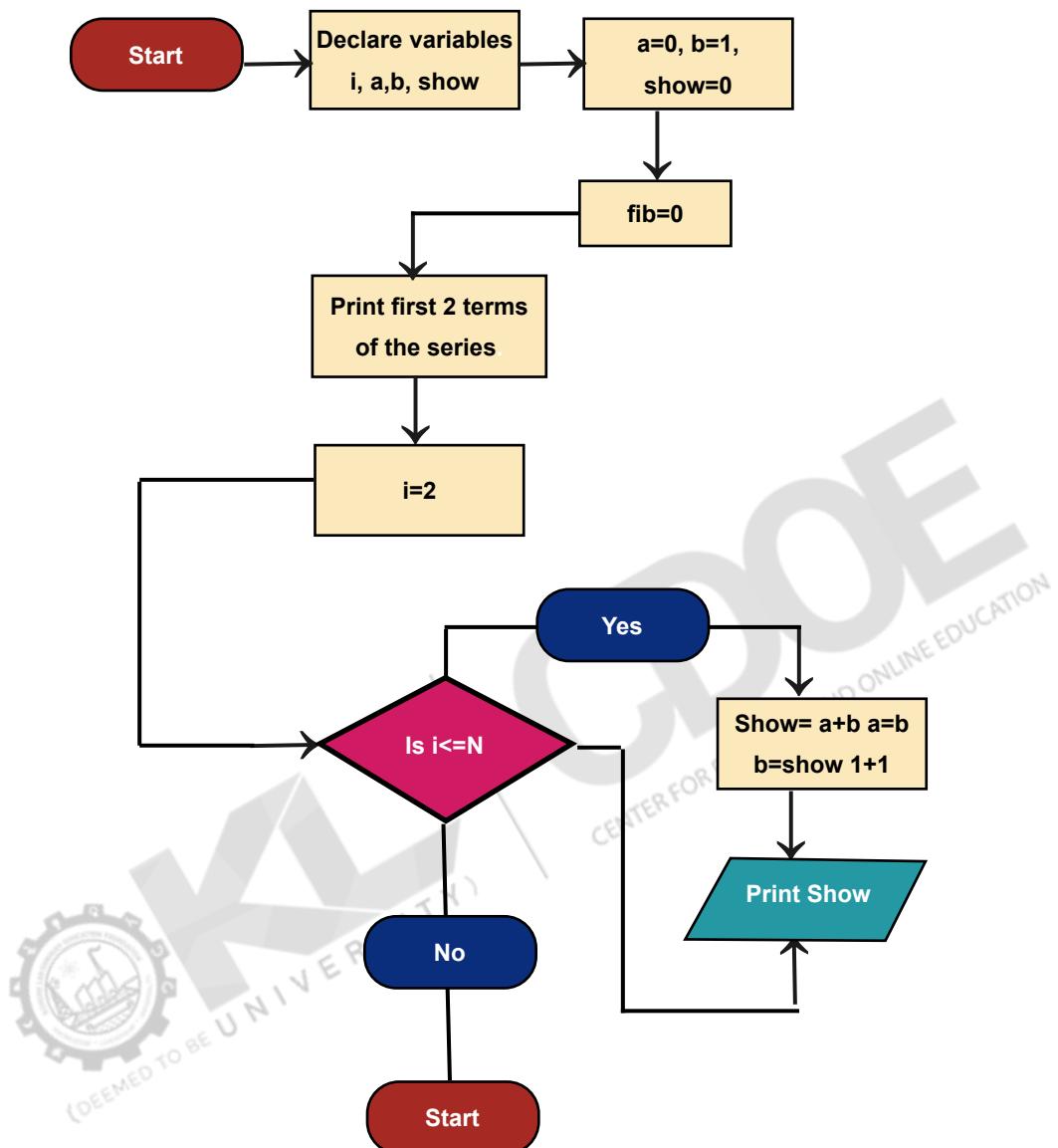


Work Out All the Roots of a Quadratic Equation $ax^2 + bx + c = 0$.

Example 4: Find the Fibonacci series till term ≤ 1000

- **Step 1:** Declare the variables i, a, b, show.
- **Step 2:** Enter the values for the variables, a=0, b=1, show=0
- **Step 3:** Enter the terms of the Fibonacci series to be printed, i.e., 1000.
- **Step 4:** Print the first two terms of the series.
- **Step 5:** Loop the following steps:
 - (i). Show = a + b
 - (ii). a= b
 - (iii). b = show
 - (iv). Add 1 to the value of i each time.
 - (v). Print Show

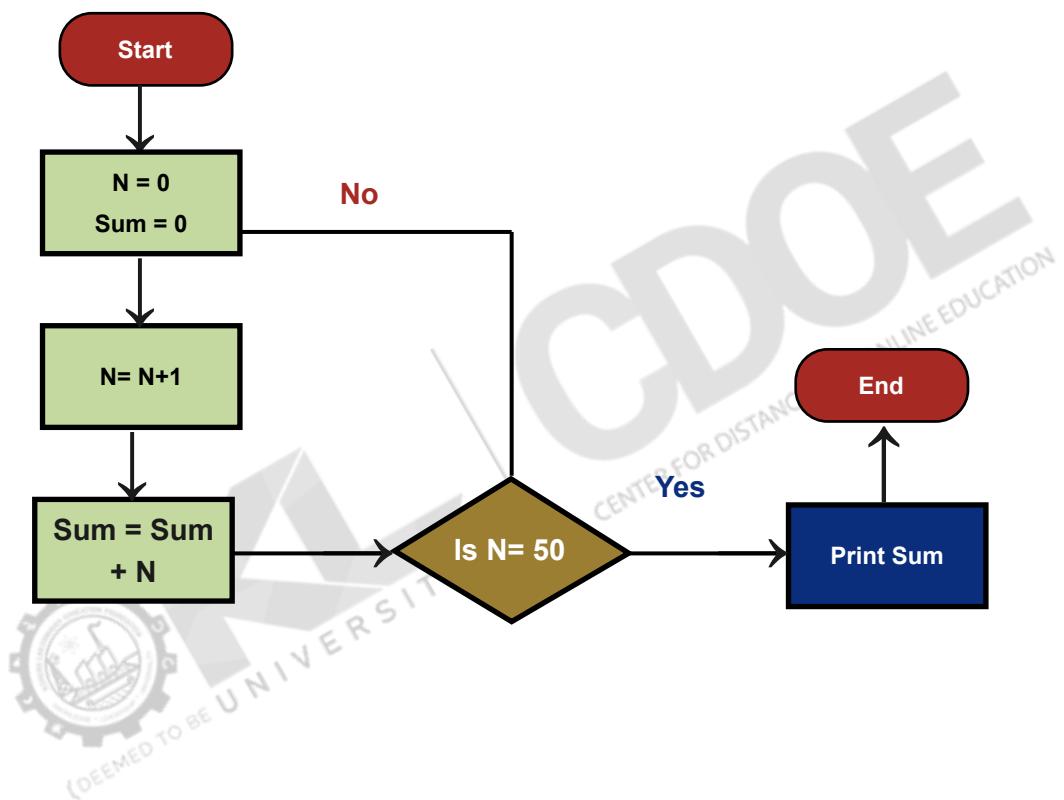




Find the Fibonacci series till term ≤ 1000

Example 5: Calculate the Sum of The First 50 Numbers

- **Step 1:** Declare number N= 0 and sum= 0
- **Step 2:** Determine N by N= N+1
- **Step 3:** Calculate the sum by the formula: Sum= N+m.
- **Step 4:** Add a loop between steps 2 and 3 until N= 50.
- **Step 5:** Print Sum.





Self-Assessment Questions

1. What symbol is used in a flowchart to represent the start or end of a process?

- a) Parallelogram
- b) Oval
- c) Rectangle
- d) Diamond

2. Which symbol is used for decision-making in a flowchart?

- a) Diamond
- b) Oval
- c) Parallelogram
- d) Rectangle

3. Flowcharts are easy to modify, even for large and complex programs.

- a) True
- b) False

4. Algorithms must always produce the same output for the same input.

- a) True
- b) False

5. Algorithms are often compared based on their speed and_____.

- a) Efficiency
- b) Accuracy
- c) Complexity
- d) Input size

1.1.2 Analysis and Complexity of Algorithm

Asymptotic analysis defines the mathematical foundation /framing of an algorithm's run-time performance. Using asymptotic analysis, we can very well conclude an algorithm's best-case, average-case, and worst-case scenarios.

Asymptotic analysis is input-bound, i.e., if the algorithm does not receive input, it is concluded to work constantly. Other than the "input," all other factors are considered constant.

Asymptotic analysis refers to computing the running time of any operation in mathematical computation units. Usually, the time required by an algorithm falls under three types:

- Best Case - Minimum time required for program execution.
- Average Case - Average time required for program execution.
- Worst Case - maximum time required for program execution.

Asymptotic Notations

Following is the commonly used asymptotic notations to calculate the running time complexity of an algorithm.

1. Big-O Notation (O -notation)
2. Omega Notation (Ω -notation)
3. Theta Notation (Θ -notation)

1. Theta Notation (Θ -Notation):

- Theta notation encloses the function from above and below. Since it represents the upper and lower bounds of an algorithm's running time, it is used to analyse its average-case complexity.
- **Theta (Average Case):** You add the running times for each possible input combination and take the average in the average case.
- Let g and f be the function from the set of natural numbers to itself. The function f is said to be $\Theta(g)$, if there are constants $c_1, c_2 > 0$ and a natural number n_0 such that $c_1 * g(n) \leq f(n) \leq c_2 * g(n)$ for all $n \geq n_0$

Mathematical Representation of Theta notation:

$\Theta(g(n)) = \{f(n) : \text{there exist positive constants } c_1, c_2 \text{ and } n_0 \text{ such that } 0 \leq c_1 * g(n) \leq f(n) \leq c_2 * g(n) \text{ for all } n \geq n_0\}$

2. Big-O Notation (O-notation)

Big-O notation represents the upper bound of the running time of an algorithm. Therefore, it gives the worst-case complexity of an algorithm.

- It is the most widely used notation for Asymptotic analysis.
- It specifies the upper bound of a function.
- The maximum time required by an algorithm or the worst-case time complexity.
- It returns a given input's highest possible output value(big-O).
- Big-Oh(Worst Case) It is defined as the condition that allows an algorithm to complete statement execution in the longest amount of time possible.

If $f(n)$ describes the running time of an algorithm, $f(n)$ is $O(g(n))$ if there exists a positive constant C and n_0 such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$. It returns a given input's highest possible output value (big-O).

The execution time is an upper bound on the algorithm's time complexity.

3. Omega Notation (Ω -Notation):

- Omega notation represents the lower bound of the running time of an algorithm. Thus, it provides the best-case complexity of an algorithm.
- The execution time is lower on the algorithm's time complexity.
- It is defined as the condition that allows an algorithm to complete statement execution in the shortest amount of time.
- Let g and f be the function from the set of natural numbers to itself. The function f is said to be $\Omega(g)$ if there is a constant $c > 0$ and a natural number n_0 such that $c*g(n) \leq f(n)$ for all $n \geq n_0$.



Self-Assessment Questions

6. Which of the following represents the average-case complexity of an algorithm?
- a) Big-O notation
 - b) Omega notation
 - c) Theta notation
 - d) Sigma notation
7. Omega notation is used to describe the worst-case performance of an algorithm.
- a) True
 - b) False
8. Asymptotic analysis only considers the input size when evaluating an algorithm's performance.
- a) True
 - b) False
9. The best-case complexity of an algorithm is represented by _____ notation.
- a) Big-O (O)
 - b) Theta (Θ)
 - c) Omega (Ω)
 - d) Sigma (Σ)
10. Theta notation is used to describe the _____ case performance of an algorithm.
- a) Best
 - b) Worst
 - c) Constant
 - d) Average



Summary

- An algorithm is a sequence of well-defined steps to solve a particular problem. It must have a finite number of steps, lead to a solution, be deterministic, and can take inputs and produce outputs.
- The data flow of an algorithm involves defining a problem, designing an algorithm to solve it, providing inputs, processing those inputs, and producing an output.
- Algorithms are critical in computer science for optimising tasks. They help comprehend the problem, evaluate solutions, and determine the resources required for solving problems.
- A flowchart is a graphical representation of an algorithm that helps visualise a system's logic.
- Flowcharts indicate flow control using symbols like terminals (for start/end), input/output symbols, processing symbols, decision symbols, and connectors.
- While flowcharts are useful for analysis, debugging, and documentation, they can be challenging to create for complex programs and may require frequent redrawing when changes are made.
- Asymptotic analysis provides a mathematical framework to evaluate an algorithm's performance in terms of run-time efficiency.
- It helps categorise algorithms based on their best-case, average-case, and worst-case execution times. Three common asymptotic notations describe algorithms' time complexity: Theta (Θ), Big-O (O), and Omega (Ω). Theta notation represents the average-case performance, Big-O notation provides an upper bound representing the worst-case scenario, and Omega notation defines the lower bound for the best-case scenario.



Terminal Questions

1. Explain the data flow of an algorithm with an example.
2. What is the significance of using flowcharts in algorithm design?
3. List the rules for constructing flowcharts. What are the advantages and disadvantages of using flowcharts?
4. What is asymptotic analysis, and why is it important in evaluating an algorithm's performance?
5. Define Big-O, Theta, and Omega notations. What are their differences?
6. Explain how an algorithm's best-case, worst-case, and average-case time complexities are computed.
7. The mathematical representation of Theta notation and its significance.



Answer Keys

Self-Assessment Questions	
Question No.	Answer
1	B
2	A
3	B
4	A
5	B
6	C
7	B
8	A
9	C
10	D



Activity

Activity Type: Offline

Duration: 1 hour

In Asymptotic Analysis, consider two algorithms with different time complexities, Algorithm A with a runtime of $O(n^2)$ and Algorithm B with a runtime of $O(n \log n)$. Discuss which algorithm would perform better for large input sizes and why. How would you analyse the scalability of these algorithms as the input size grows exponentially?



Glossary

- **Deterministic:** The property of an algorithm to produce the same output for the same input.
- **Decision Symbol:** A diamond-shaped symbol in flowcharts is used to represent decision-making steps.
- **Time Complexity:** A measure of the time required for an algorithm to run as a function of the input size.
- **Input-Bound:** A characteristic of algorithms whose performance is analysed based on the input size, with other factors considered constant.
- **Efficiency:** A measure of how well an algorithm optimises time and resources.



Bibliography

Textbooks

- Balagurusamy, E. (2017). *Data Structures*. McGraw Hill Education.
- Sedgewick, R., & Wayne, K. (2014). *Algorithms II*. Pearson Education.
- Sridhar, S. (2014). *Design and Analysis of Algorithms*. Oxford University Press.
- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (1989). *Introduction to Algorithms*. Prentice-Hall.



e-References

- **Algorithm and Flowchart:** <https://www.javatpoint.com/algorithm-vs-flow-chart>
- **Complete Guide on Complexity Analysis:** <https://www.geeksforgeeks.org/complete-guide-on-complexity-analysis/>



Video Links

Video	Links
Algorithm & Flowchart	https://www.youtube.com/watch?v=uxN7kttwkJI
Asymptotic Notations	https://www.youtube.com/watch?v=ffEMxC-9ksU



Image Credits

- **Fig. 1:** Self-Made
- **Fig. 2:** Self-Made
- **Fig. 3:** Self-Made



Keywords

- Algorithm
- Flowchart
- Asymptotic Analysis
- Big-O Notation (O)
- Omega Notation (Ω)
- Theta Notation (Θ)

MODULE 1

Introduction to Data Structures

Unit 2

Overview of Data Structure



■ Unit Table of Contents

Unit 1.2 Overview of Algorithm

Aim	33
Instructional Objectives	33
Learning Outcomes	33
1.2.1 Introduction and Definition of Data Structure	34
Self-Assessment Questions	37
1.2.2 Classification of Data and Arrays	38
Self-Assessment Questions	42
1.2.3 Various Types of Data Structure	43
Self-Assessment Questions	47
Summary	48
Terminal Questions	48
Answer Keys	49
Activity	49
Glossary	50
Bibliography	50
e-References	50
Video Links	51
Image Credits	51
Keywords	51



Aim

To provide students with a thorough interpretation of different types of data structures, their classifications, and how they are utilised in computer programming.



Instructional Objectives

This unit is designed to:

- Explain the concept of data structures and their significance in programming
- Discuss the structure and operations of stacks and queues, including their specific principles (LIFO and FIFO)
- Describe the structure, operations, and usage of arrays, including one-dimensional, two-dimensional, and multi-dimensional arrays



Learning Outcomes

At the end of the unit, the student is expected to:

- Classify data structures into linear and non-linear categories
- Analyse the usage of arrays and perform operations such as insertion, deletion, and traversal for one-dimensional, two-dimensional, and multi-dimensional arrays
- Evaluate the structure of trees and graphs, distinguishing between types like binary trees, AVL trees, and different types of graphs

1.2.1 Introduction and Definition of Data Structure

A data structure refers to a method of organising and storing data in a computer system, both mathematically and logically. The mathematical view focuses on formulas that help access specific memory locations, while the logical view emphasises how data is arranged and visualised.

Data Structure, a core area of Computer Science, helps us grasp the organisation of data and the flow of data management, leading to greater efficiency in processing or programming tasks. A data structure defines how data is stored in the computer's memory, allowing for easy retrieval and efficient usage when necessary. This organisation can follow various logical or mathematical patterns, depending on the type of data being managed.

Two key factors determine the scope of a particular data structure

- First, it must be rich enough to capture the relationships between the data and real-world entities.
- Second, the structure should be simple enough to allow efficient data processing when required.

Examples of data structures include arrays, linked lists, stacks, queues, trees, etc. They are widely used in Computer Science fields, such as Compiler Design, Operating Systems, Graphics, and Artificial Intelligence. These structures are critical in many computer algorithms, enabling programmers to manage data effectively. They also enhance program performance, aligning with the primary goal of storing and retrieving data as quickly as possible.

Importance of Data Structure

Data structures and algorithms are closely linked; data structure directly impacts how efficiently algorithms can operate on it. Data must be represented in a way that is intuitive and easy to work with, ensuring developers and users can carry out operations effectively.

Data structures offer several advantages:

- Simplified modification of data.
- Time-efficient processing.
- Optimised use of storage space.

- Clear representation of data.
- Quick access to large datasets.

Characteristics of Data Structures

A data structure is a systematic way of organising data. Key characteristics include:

- **Linear or Non-Linear:** This refers to the arrangement of data in a sequence (like arrays or graphs).
- **Static or Dynamic:** Static structures have fixed sizes and formats, whereas dynamic structures can change at runtime.
- **Time Complexity:** The time required to execute an algorithm should be as minimal as possible, which helps improve system performance.
- **Correctness:** The data structure must be implemented correctly, with appropriate interfaces that define how the data is structured.
- **Space Complexity:** Efficient use of memory is critical. The less space a data structure occupies, the better it functions in terms of performance.

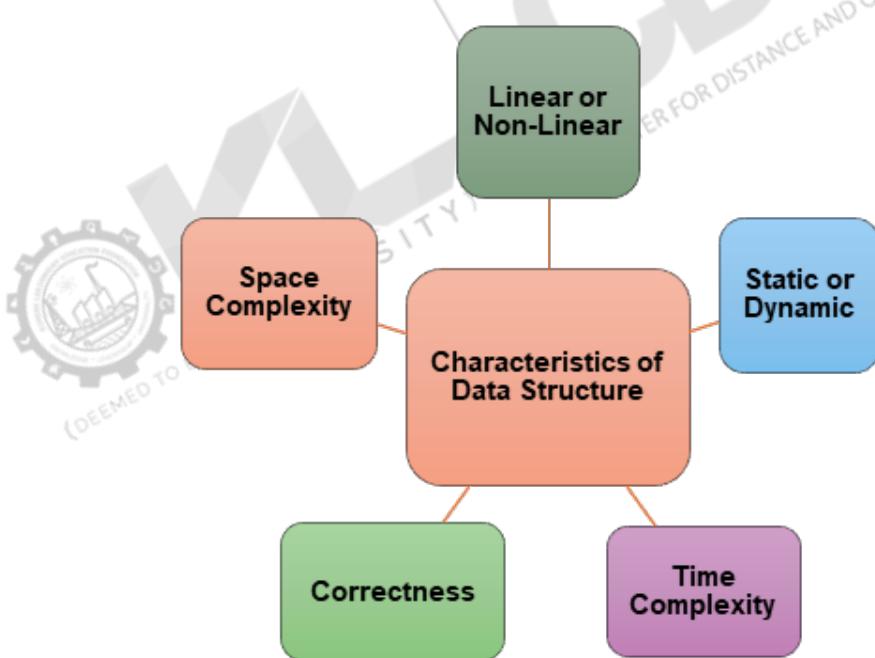


Fig. 1: Data structure characteristics

Applications of Data Structures

Software architects often use algorithms in combination with data structures such as lists, queues, and mappings of values. These combinations are crucial for a variety of applications, such as:

- **Data Storage:** Data structures help store data efficiently, as seen in the use of record collections and corresponding structures for storing database records.
- **Resource and Service Management:** Operating systems utilise data structures like linked lists for memory allocation, file directories for file management, and trees for structuring processes and scheduling tasks.
- **Data Exchange:** Data structures define how data is exchanged between programs, such as the organisation of TCP/IP packets.
- **Sorting and Ordering:** Structures like binary search trees, also known as ordered binary trees, allow efficient data sorting and enable prioritisation through priority queues.
- **Indexing:** Advanced data structures, like B-trees, are necessary to efficiently index database items.
- **Searching:** Complex structures, like B-trees, facilitate quick searching through large datasets in databases.
- **Scalability:** In large-scale data applications, data structures ensure efficient storage and management of data across distributed systems. Frameworks like Apache Spark utilise these structures to represent underlying database records and enable scalable queries.



Self-Assessment Questions

1. Which of the following is NOT a type of data structure?
 - a) Arrays
 - b) Linked Lists
 - c) Compiler
 - d) Queues

2. Which data structure is commonly used for memory allocation in operating systems?
 - a) Stacks
 - b) Linked Lists
 - c) Arrays
 - d) Trees

3. Arrays, linked lists, stacks, and queues are all examples of data structures.
 - a) True
 - b) False

1.2.2 Classification of Data and Arrays

A data type is a classification of data that determines what information a variable can hold. Data types are essential in computer programming, allowing variables to store specific data types. When a variable is created, it is assigned a data type that dictates the data it can contain. A data type is a way of interpreting a sequence of bits. There are many different types of data, and these types of help make storing and processing data more efficient.

Data types refer to the data that can be held in variables. Every programming language includes a set of predefined data types. This allows the programmer to define variables that store data of specific types and provides a set of operations to manipulate these variables meaningfully. Some data types, such as integers and characters, are straightforward to implement, as they are integrated into the computer's machine language instruction set.

Other data types may require more complex implementations. Some languages provide features that allow programmers to create custom combinations of basic types, such as structures in the C programming language. Mechanisms that enable the creation of more complex data types beyond those built into the language are essential. These new types should also support meaningful operations.

Basic or Primitive Data Types

The most common intrinsic or primitive data types include:

- **Integer:** A whole number which can be positive or negative and does not include any decimal or fractional part.
- **Real:** A number that includes a decimal point or fractional component.
- **Boolean:** A data type that stores one of two possible values, typically TRUE or FALSE.
- **Character:** Any individual letter, number, punctuation mark, or space, usually stored as a single unit, like a byte.
- **String:** Sometimes called “text,” a string is a sequence of characters, including alphabetic or numeric data.

Structured or Non-Primitive Data Types

In addition to primitive data types, there are also user-defined structured data types. These data types can hold collections of data values, generally consisting of primitive data types. Examples include arrays, records, lists, trees, and files. These types, created by programmers, are crucial for developing complex data structures. They focus on organising homogeneous (similar) or heterogeneous (different) data elements.

Abstract Data Types (ADT)

Apart from the built-in data types, abstract data types specify the data structure along with its possible operations. These are called abstracts because they specify what to do independent of implementation.

Arrays

An array is a fundamental data structure that stores a fixed-size sequential collection of elements of the same type. Arrays are widely used in algorithms due to their simplicity and efficiency in accessing elements via indices. In arrays, elements are stored in contiguous memory locations, making access to any element very fast ($O(1)$ time complexity).

An array is a collection of similar data elements stored in consecutive memory locations and can be accessed by indices. The index starts from 0, meaning that the first element is at index 0, the second at index 1, and so on.

Types of Arrays

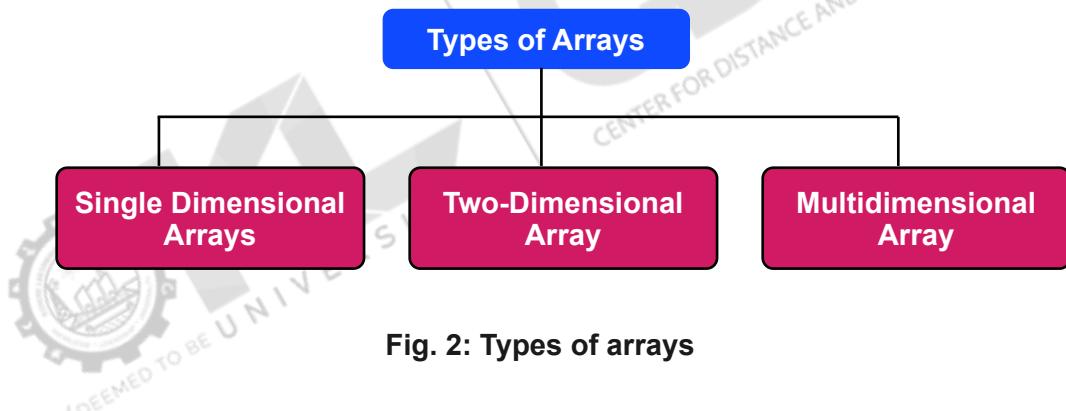


Fig. 2: Types of arrays

1. Single Dimensional Arrays

An array is a group of identically typed data elements that a common name may retrieve.

A two-dimensional array is like a table, and a one-dimensional array is similar to a list. The C language does not limit the number of dimensions in an array, however certain implementations might.

When the number of dimensions is unknown or irrelevant, some texts use the broad term array and refer to one-dimensional arrays as vectors and two-dimensional arrays as matrices.

Syntax

```
data_type array_name[array_size];
```

Where:

- **data_type**: Specifies the data type stored in each array element.
- **array_name**: The name used to refer to the array.
- **array_size**: The number of elements the array will contain.

Example:

```
int nums[5];
```

2. Two-Dimensional Array in C

The two-dimensional array can be defined as an array of arrays. It is organised as matrices, which can be represented as a collection of rows and columns. However, 2D arrays are created to implement a relational database-like data structure. They provide the ease of holding the bulk of data at once, which can be passed to any number of functions wherever required.

Syntax

```
data_type array_name[rows][columns];
```

Where:

- **data_type**: Specifies the type of data stored in each array element.
- **array_name**: The name used to refer to the array.
- **rows**: The size of the first dimension (number of rows).
- **columns**: The size of the second dimension (number of columns).

Example:

```
int nums[5][10];
```

3. Multidimensional Arrays in C

A multidimensional array is an array of arrays with tabular data representing homogeneous data. Data is typically kept in memory in row-major order in multidimensional arrays.

Syntax

The syntax for declaring a three-dimensional array is:

data_typearray_name[depth][rows][columns];

Where:

- **data_type**: Specifies the type of data stored in each array element.
- **array_name**: The name used to refer to the array.
- **depth**: The size of the first dimension (depth).
- **rows**: The size of the second dimension (number of rows).
- **columns**: The size of the third dimension (number of columns).

Example:

```
int nums[5][10][2];
```



Self-Assessment Questions

4. Which of the following is a primitive data type?

- a) Integer
- b) Array
- c) Record
- d) Tree

5. What is the time complexity for accessing an element in an array?

- a) $O(n)$
- b) $O(\log n)$
- c) $O(1)$
- d) $O(n^2)$

6. In a one-dimensional array, elements are stored in non-contiguous memory locations.

- a) True
- b) False

7. In C, multi-dimensional arrays must have constant sizes defined at compile-time.

- a) True
- b) False

1.2.3 Various Types of Data Structure

A data structure refers to the portion of memory allocated for a model where the necessary data can be structured. Data structures are broadly classified into two main types:

- (i) Primitive Data Structures
- (ii) Non-Primitive Data Structures

1. Primitive Data Structures

Machine-level instructions can directly operate upon primitive data structures. Examples include integers, real numbers, characters, pointers, and their respective memory storage representations. Programmers use these data types when defining variables in their code. For instance, a programmer might declare a variable “z” as a real data type, allowing it to store a real number.

2. Non-Primitive Data Structures

Non-primitive data structures, unlike primitive types, are not predefined by the programming language. Instead, they are created by the programmer and are often referred to as reference variables, as they point to memory locations where the actual data is stored. These data structures are built upon primitive data types. Examples include arrays, stacks, queues, linked lists, trees, graphs, and hash tables.

Types of Non-Primitive Data Structures

Non-primitive data structures are further divided into two categories:

a) Linear Data Structures

The elements are arranged sequentially in linear data structures, meaning they are stored list-like. This structure exhibits a linear relationship between the elements, and the simplest form is a one-dimensional array. However, due to its limitations, lists are often preferred for handling various data types. The following are examples of linear data structures:

A. Array: An array is a collection of elements of the same data type stored in consecutive memory locations and accessed by a common name. Arrays can be classified into different types:

- **One-dimensional Array:** A simple list of elements stored in contiguous memory locations.
- **Two-dimensional array:** A matrix-like structure with rows and columns for data storage.
- **Multidimensional Array:** An array with more than two dimensions, essentially an array of arrays.

B. Stack: A stack is a linear data structure that follows the Last-In-First-Out (LIFO) or First-in-last-out (FILO) principle, where elements are added and removed only from one end, called the top. The primary operations in the Stack are as follows:

- **Push:** Insert an element into the stack.
- **Pop:** Removes the top element from the stack.

C. Queue: A queue is a linear structure that follows the First-In-First-Out (FIFO) or Last-In-Last-Out (LILO) principle, where elements are inserted at the rear and removed from the front. The following are the primary operations of the Queue:

- **Enqueue:** Adds an element to the rear of the queue.
- **Dequeue:** Removes an element from the front of the queue.

D. Linked List: A linked list is a collection of data elements of the same type, where each element, called a node, is connected to the next one, but they do not need to be stored in consecutive memory locations. It is a linear but non-contiguous data structure. Linked lists can be classified into:

- **Singly Linked List:** A list where each node points to the next node in one direction.
- **Doubly Linked List:** A list where each node has pointers to the previous and next nodes, allowing traversal in both directions.
- **Circular Linked List:** A list where the last node points back to the first node, forming a loop.

b) Non-Linear Data Structures

In non-linear data structures, elements are stored in a manner that reflects hierarchical relationships between the data. These structures do not exhibit a sequential relationship between elements. Non-linear data structures include:

A. Tree: A tree is a data structure representing data with hierarchical relationships. It maintains a parent-child relationship between different elements, which are called nodes. Trees can be classified into different types:

- **Binary Tree:** A tree where each node has at most two children.
- **Binary Search Tree:** A tree where the left child is less than the parent, and the right child is greater.
- **AVL Tree:** A self-balancing binary search tree with balance factors to maintain balance.
- **B-Tree:** A self-balancing search tree where nodes can have multiple keys and children.

B. Graph: A graph is a structure that represents data with relationships between pairs of elements, which may not necessarily be hierarchical. It maintains connections between various elements, often called nodes and edges. Graphs represent electrical grids, communication networks, airline routes, and project planning flowcharts.

By organising data in such structures, complex operations can be performed efficiently, enabling programmers to develop algorithms that process data optimally. Graphs can be categorised based on the arrangement of vertices and edges as follows:

- **Null Graph:** No edges between vertices.
- **Trivial Graph:** Only one vertex.
- **Simple Graph:** No self-loops or multiple edges.
- **Multi Graph:** Multiple edges without self-loops.
- **Pseudo Graph:** Contains both self-loops and multiple edges.
- **Non-Directed Graph:** Edges without direction.
- **Directed Graph:** Edges with a direction between vertices.
- **Connected Graph:** Every pair of vertices has a path.
- **Disconnected Graph:** At least one pair of vertices lacks a path.
- **Regular Graph:** All vertices have the same degree.
- **Complete Graph:** Every vertex is connected to every other vertex.
- **Cycle Graph:** A graph where vertices form a closed loop.
- **Cyclic Graph:** Contains at least one cycle.
- **Acyclic Graph:** Has no cycles.
- **Finite Graph:** Contains a finite number of vertices and edges.
- **Infinite Graph:** Contains an infinite number of vertices and edges.
- **Bipartite Graph:** Vertices can be divided into two sets, with edges only between sets.
- **Planar Graph:** Can be drawn in a plane without intersecting edges.
- **Euler Graph:** All vertices have even degrees.
- **Hamiltonian Graph:** Contains a Hamiltonian circuit (a cycle visiting every vertex exactly once).

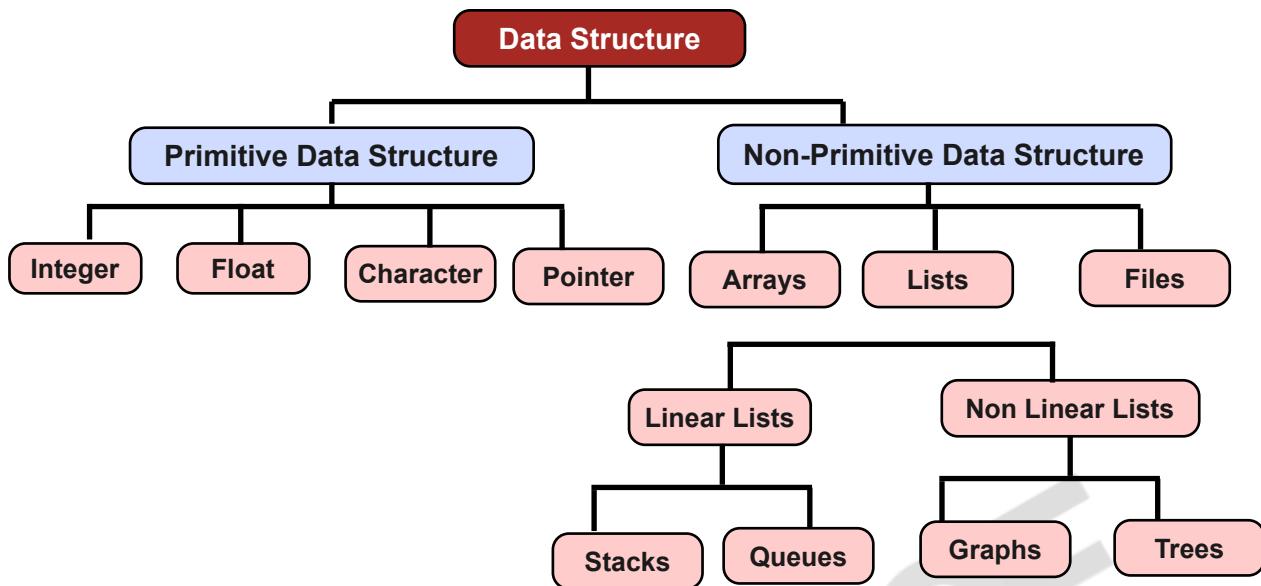


Fig. 3: Classification of data structures



Self-Assessment Questions

8. What is the primary operation to insert an element in a stack?
- a) Pop
 - b) Enqueue
 - c) Push
 - d) Dequeue
9. A _____ tree is a self-balancing binary search tree.
- a) Binary Tree
 - b) AVL
 - c) B-Tree
 - d) Red-Black Tree
10. A _____ graph has no self-loops or multiple edges between two vertices.
- a) Complete
 - b) Multi
 - c) Euler
 - d) Simple



Summary

- A Data Structure refers to organising and storing data in a computer system, focusing on mathematical and logical perspectives.
- Data structures play a crucial role in enhancing the efficiency of programming tasks by defining how data is stored in memory for quick and efficient retrieval.
- Primitive data types include integers, real numbers, Booleans, characters, and strings. Arrays are a key data structure that stores elements of the same type in contiguous memory locations.
- Abstract data types (ADTs) extend the concept of data types by specifying operations along with the data structure.
- Arrays play a significant role in algorithm efficiency, allowing for $O(1)$ time complexity for accessing elements.
- Multi-dimensional arrays allow for more complex data organisation, which is essential in applications like relational databases and mathematical computations.
- Data structures are methods of organising and storing data efficiently. They are classified into two main types: primitive data structures and non-primitive data structures.
- Primitive Data Structures include basic data types like integers, real numbers, characters, and pointers that the machine can directly process.
- Linear Data Structures, where data elements are stored sequentially, such as arrays, stacks, queues, and linked lists.
- Non-Linear Data Structures, where data elements are organised hierarchically or through relationships, including trees and graphs.



Terminal Questions

1. What is a data structure, and why is it important in computer science? Discuss the different characteristics of data structures.
2. Explain the concept of structured or non-primitive data types and provide examples.
3. What is the difference between a single-dimensional and a two-dimensional array?
4. How are multi-dimensional arrays declared and used in C?
5. What is a data structure? Explain the two main types of data structures.
6. Define and differentiate between primitive and non-primitive data structures.



Answer Keys

Self-Assessment Questions	
Question No.	Answer
1	C
2	B
3	A
4	A
5	C
6	B
7	A
8	C
9	B
10	D



Activity

Activity Type: Offline

Duration: 1 hour

Imagine designing a new software application requiring efficient data storage and retrieval. How would you decide which type of data structure to use among arrays, linked lists, stacks, queues, trees, and graphs? Provide a detailed explanation of the factors you would consider, such as time complexity, space complexity, ease of implementation, and specific use cases. Discuss the trade-offs in selecting one data structure over another for different scenarios.



Glossary

- **Contiguous Memory:** Memory locations next to each other, allowing efficient data access.
- **Stack:** A linear data structure that follows the LIFO principle, allowing operations at only one end.
- **Queue:** A linear data structure that follows the FIFO principle, with insertion at the rear and removal from the front.
- **Linked List:** A collection of nodes where each node points to the next, allowing non-contiguous storage.
- **Tree:** A hierarchical data structure consisting of nodes connected by edges, representing parent-child relationships.



Bibliography

Textbooks

- Balagurusamy, E. (2017). *Data Structures*. McGraw Hill Education.
- Sedgewick, R., & Wayne, K. (2014). *Algorithms II*. Pearson Education.
- Sridhar, S. (2014). *Design and Analysis of Algorithms*. Oxford University Press.
- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (1989). *Introduction to Algorithms*. Prentice-Hall.



e-References

- **Introduction to Data Structure:** <https://dbrau.ac.in/wp-content/uploads/2023/05/Introduction-of-Data-structure.pdf>
- **Introduction to Data Structures:** <https://www.geeksforgeeks.org/introduction-to-data-structures/>
- **An Introduction to Data Structures:** <https://www.javatpoint.com/data-structure-introduction>
- **Types of Arrays:** <https://www.geeksforgeeks.org/types-of-arrays/>



Video Links

Video	Links
Introduction to Data Structures	https://www.youtube.com/watch?v=xLetJpcjHS0
Classification of Data Structures	https://www.youtube.com/watch?v=Plty_7OsM8g
Classification of Arrays	https://www.youtube.com/watch?v=whOSq_xv6yg



Image Credits

- **Fig. 1:** Self-Made
- **Fig. 2:** Self-Made
- **Fig. 3:** Self-Made



Keywords

- Static data structure
- Dynamic data structure
- Time complexity
- Space complexity
- Abstract data types
- Single-dimensional array
- Two-dimensional array
- Primitive data structure
- Non-primitive data structure

MODULE 1

Introduction to Data Structures

Unit 3

Functions and Arrays



■ Unit Table of Contents

Unit 1.3 Functions and Arrays

Aim	54
Instructional Objectives	54
Learning Outcomes	54
1.3.1 Static and Dynamic Memory Allocation	55
Self-Assessment Questions	57
1.3.2 Function and Recursion	58
Self-Assessment Questions	67
1.3.3 Introduction to Arrays	68
1.3.3.1 One-Dimensional Array	68
1.3.3.2 Multidimensional Arrays	72
Self-Assessment Questions	76
Summary	77
Terminal Questions	77
Answer Keys	78
Activity	78
Glossary	79
Bibliography	79
e-References	79
Video Links	80
Image Credits	80
Keywords	80



Aim

To familiarise students with interpreting the concepts of memory allocation, function and recursion, and arrays (both one-dimensional and multidimensional) in C programming.



Instructional Objectives

This unit is designed to:

- Describe the advantages and disadvantages of static and dynamic memory allocation in C
- Define and implement functions in C
- Explain the concept of one-dimensional arrays in C



Learning Outcomes

At the end of the unit, the student is expected to:

- Use functions like malloc(), calloc(), realloc(), and free() for dynamic memory management in C
- Analyse the two methods for passing data into the function
- Evaluate the multidimensional arrays (2D and 3D arrays)

1.3.1 Static and Dynamic Memory Allocation

Static Memory Allocation

Static memory allocation is a method of memory management where a fixed amount of memory is reserved for a variable at the compilation time. This memory remains constant throughout the program's execution. It is commonly used for global variables, static variables, and arrays.

Static variables are declared outside the main function and retain their values for the duration of the program. They are assigned memory during the compilation process. Global variables function similarly to static variables but can be accessed by all functions in the program. Arrays, too, have their memory allocated at compile time, and their size is fixed.

Properties

- The compiler allocates memory for static variables, which can be accessed by assigning their addresses to pointer variables.
- Static variables are allocated memory permanently.
- Memory is reserved during the compilation phase.
- Static memory allocation uses stack memory.
- Although static memory allocation is fast, it is less efficient.
- Once allocated, memory cannot be reused.
- The memory size cannot be changed after it has been initially assigned.
- The process of memory allocation is straightforward.
- Commonly used for arrays.

Advantages

- **Faster Access:** Since memory is allocated during compilation, static memory access is quicker than dynamic memory access. The memory address is determined during compilation.
- **No Overhead:** Allocating or freeing memory has no runtime overhead, making it more efficient than dynamic allocation.
- **Persistent Data:** Static variables and arrays maintain their data throughout the program's life, which can be beneficial when data must be shared between different functions.

Disadvantages

- **Limited Flexibility:** Static memory is rigid because the memory size is fixed at compile time. Changing the size requires recompiling the program.
- **Wasted Memory:** Static allocation can lead to wasted memory if the data structure size isn't known beforehand.

- **Limited Scope:** Static variables are accessible only within the function or globally if declared outside functions.

Dynamic Memory Allocation

Dynamic memory allocation refers to assigning memory at runtime, meaning that memory is allocated and deallocated as required during execution. It's often used for structures like linked lists, trees, and dynamic arrays. Functions like malloc(), calloc(), realloc(), and free() handle dynamic memory. malloc() allocates memory in bytes and returns a pointer, while calloc() initialises the memory to zero. realloc() adjusts the size of already allocated memory, and free() releases the memory.

Properties

- Pointer variables store values returned by various memory management functions.
- Memory is allocated at runtime.
- Memory can be reserved or released at any point during execution.
- Heap memory is used for dynamic allocation.
- While dynamic memory allocation is slower, it is more efficient.
- The implementation is more complex.
- Memory can be resized or reused during the program's execution.
- Often used for dynamic data structures such as linked lists.

Advantages

- **Flexible Memory Usage:** The size of data structures can be modified during execution, making it more adaptable than static memory allocation.
- **Efficient Memory Usage:** Memory is allocated only when necessary, leading to less wasted memory.
- **Global Access:** Dynamic memory can be accessed across different functions.

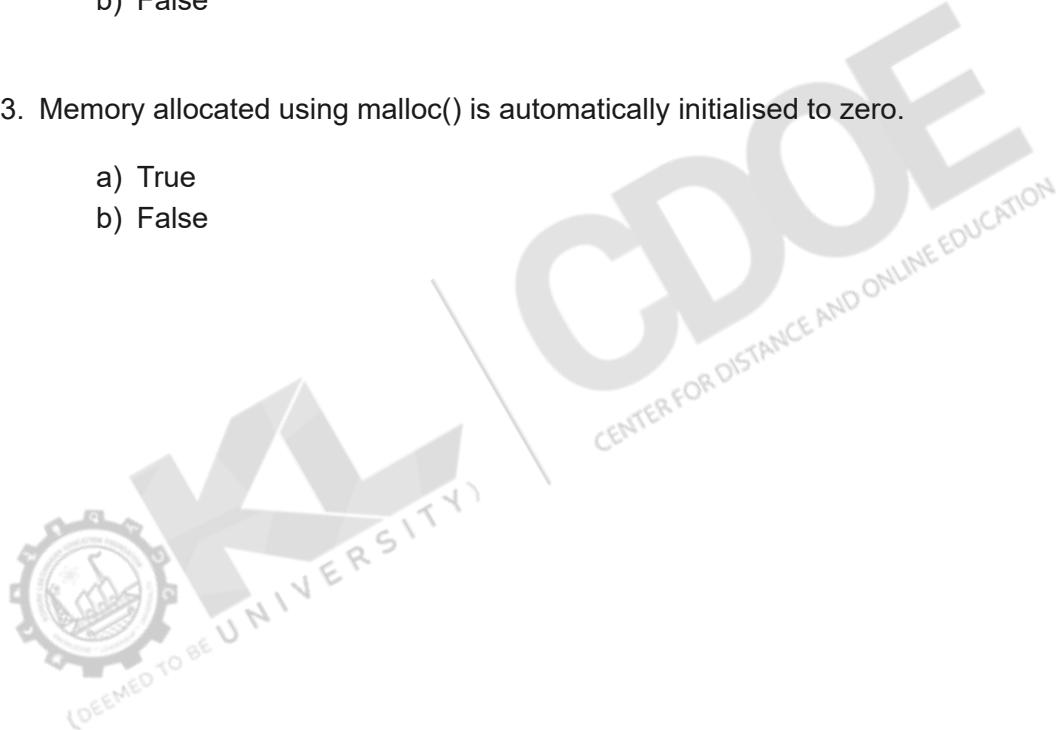
Disadvantages

- **Slower Access:** Dynamic memory access is slower since the memory address is determined during execution rather than compile time.
- **Memory Leaks:** Improper memory deallocation can cause memory leaks, leading to crashes or slow performance.
- **Fragmentation:** Incorrect memory allocation and deallocation may lead to fragmentation, where small unused memory spaces prevent larger blocks from being allocated.



Self-Assessment Questions

1. Static memory allocation occurs at runtime.
 - a) True
 - b) False
2. Dynamic memory allocation uses heap memory.
 - a) True
 - b) False
3. Memory allocated using malloc() is automatically initialised to zero.
 - a) True
 - b) False



1.3.2 Function and Recursion

A function is a block of code that only runs when it is called. You can pass data, known as parameters, into a function. Functions perform certain actions and are important for reusing code: Define the code once and use it many times.

Predefined Function

So, it turns out you already know what a function is. You have been using it the whole time while studying this tutorial!

For example, `main()` is a function which is used to execute code, and `printf()` is a function used to output/print text to the screen:

Example

```
int main()
{
    printf("Hello World!");
    return 0;
}
```

Create a Function

To create (often referred to as declare) your own function, specify the name of the function, followed by parentheses () and curly brackets {}:

Syntax

```
void myFunction() {
    // code to be executed
}
```

Example

`myFunction()` is the name of the function

Call a Function

Declared functions are not executed immediately. They are “saved for later use” and will be executed when called. To call a function, write the function’s name followed by two parentheses () and a semicolon. In the following example, `myFunction()` is used to print a text (the action), when it is called:

Example

Inside main, call myFunction():

```
// Create a function
void myFunction() {
    printf("I just got executed!");
}

int main() {
    myFunction(); // call the function
    return 0;
}

// Outputs "I just got executed!"
```

A function can be called multiple times:

C Function Declaration and Definition

Example

```
// Create a function
void myFunction() {
    printf("I just got executed!");
}

int main() {
    myFunction(); // call the function
    return 0;
}
```

A function consists of two parts:

- **Declaration:** the function's name, return type, and parameters (if any)
- **Definition:** the body of the function (code to be executed)

```
void myFunction() { // declaration
    // the body of the function (definition)
}
```

For code optimisation, it is recommended that the declaration and the definition of the function be separated.

You will often see C programs with function declaration above main(), and function definition below main(). This will make the code better organised and easier to read:

Example

```
// Function declaration  
  
void myFunction();  
  
// The main method  
  
int main() {  
    myFunction(); // call the function  
    return 0;  
}  
  
// Function definition  
  
void myFunction() {  
    printf("I just got executed!");
```

Call by value and Call by reference in C

In C language, there are two methods for passing data into the function: call by value and call by reference.

- Call by Value
- Call by Reference In C

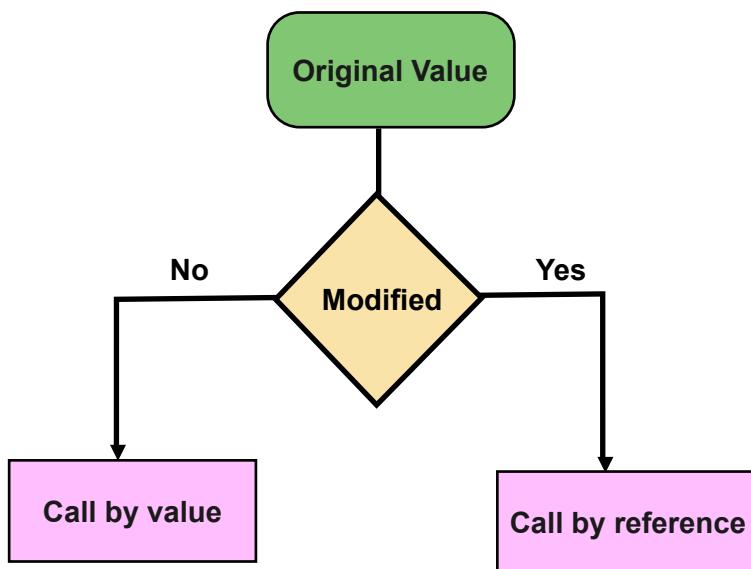


Fig. 1: Two methods for passing data into the function

Call by value in C

The value of the actual parameters is replicated into the formal parameters when using the call-by-value method. In another way, we can say that the call-by-value method calculates the variable's value in the function call. We cannot change the value of the formal parameter by the real parameter when using the call-by-value method. Since the value of the real parameter is copied into the formal parameter in a call-by-value, separate memory is allotted for the actual and formal parameters. The argument used in the function call is the actual parameter, whereas the argument used in the function specification is the formal parameter.

```

#include<stdio.h>
void change(int num) {
    printf("Before adding value inside function num=%d \n",num);
    num=num+100;
    printf("After adding value inside function num=%d \n", num);
}
  
```

```

int main() {
    int x=100;
    printf("Before function call x=%d \n", x);
    change(x);//passing value in function
    printf("After function call x=%d \n", x);
    return 0;
}

```

Results

- Prior to the function call, x=100
- Prior to including value into the function num=100
- Following value addition within function num=200
- Following the x=100 function call

Call by Value Example: Swapping the values of the two variables

```

#include <stdio.h>
void swap(int , int); //prototype of the function
int main()
{
    int a = 10;
    int b = 20;
    printf("Before swapping the values in main a = %d, b = %d\n",a,b); // printing the value of a and
    b in main
    swap(a,b);
    printf("After swapping values in main a = %d, b = %d\n",a,b); // The value of actual parameters
    do not change by changing the formal parameters in call by value, a = 10, b = 20
}

void swap (int a, int b)

{
    int temp;
    temp = a;
    a=b;
    b=temp;
}

```

```

printf("After swapping values in function a = %d, b = %d\n",a,b); // Formal parameters, a = 20,
b = 10
}

```

Results

- Prior to reversing the values in the primary
- Following a value swap in the function a = 20, b = 10.
- After exchanging values in main a = 10, b = 20

Refer to call by reference in C

When a function call is made via reference, the address of the variable is supplied as the real parameter. Since the address of the actual parameters is passed, the value of the actual parameters can be changed by altering the formal parameters. The memory allocation for formal and actual parameters in the call by reference is the same. The value stored at the address of the real parameters is the value on which all functions operate, and the updated value is also kept at the same address.

Call by reference Example: Swapping the values of the two variables

```

#include <stdio.h>
void swap(int *, int *); //prototype of the function
int main()
{
    int a = 10;
    int b = 20;
    printf("Before swapping the values in main a = %d, b = %d\n",a,b); // printing the value of a and
    b in main
    swap(&a,&b);
    printf("After swapping values in main a = %d, b = %d\n",a,b); // The values of actual parameters
    do change in call by reference, a = 10, b = 20
}

void swap (int *a, int *b)

{
    int temp;
    temp = *a;
    *a=*b;
}

```

```

*b=b;
printf("After swapping values in function a = %d, b = %d\n",*a,*b); // Formal parameters, a =
20, b = 10
}
  
```

Output

- Before swapping the values in main a = 10, b = 20
- After swapping values in function a = 20, b = 10
- After swapping values in main a = 20, b = 10

Difference between call by value and call by reference in c

No.	Call by value	Call by reference
1	A copy of the value is passed into the function	An address of value is passed into the function
2	Changes made inside the function are limited to the function only. Changing the formal parameters does not change the values of the actual parameters.	Changes made inside the function validate outside of the function, too. Changing the formal parameters does change the values of the actual parameters.
3	Actual and formal arguments are created at the different memory locations.	Actual and formal arguments are created at the same memory location.

Recursion in C

Recursion is the technique of making a function call itself. This technique provides a way to break complicated problems down into simple problems that are easier to solve. Recursion may be a bit difficult to interpret, but the best way to figure out how it works is to experiment with it.

Recursion Example

Adding two numbers together is easy, but adding a range of numbers is more complicated. In the following example, recursion is used to add a range of numbers together by breaking it down into the simple task of adding two numbers: Example

```

int sum(int k);
int main() {
    int result = sum(10);
    printf("%d", result);
    return 0;
}

int sum(int k) {
    if (k > 0) {
        return k + sum(k - 1);
    } else {
        return 0;
    }
}

```

Passing Array to Function in C

In C, various general problems require passing more than one variable of the same type to a function. For example, consider a function which sorts the 10 elements in ascending order. Such a function requires 10 numbers to be passed as the actual parameters from the main function. Here, instead of declaring 10 different numbers and passing them into the function, we can declare and initialise an array and pass that into the function. This will resolve all the complexity since the function will now work for any number of values.

As we notice, the array_name contains the address of the first element. Here, we must notice that we must pass only the name of the array in the function intended to accept an array. The array defined as the formal parameter will automatically refer to the array specified by the array name defined as an actual parameter.

Consider the following syntax to pass an array to the function.

functionname(arrayname);//passing array

Methods to declare a function that receives an array as an argument

There are 3 ways to declare the function intended to receive an array as an argument.

First way:

return_type function(type arrayname[])

Declaring blank subscript notation [] is the widely used technique.

Second way:

```
return_type function(type arrayname[SIZE])
```

Optionally, we can define size in subscript notation [].

Third way:

```
return_type function(type *arrayname)
```

You can also use the concept of a pointer.

C language passing an array to function example

```
#include<stdio.h>

int minarray(int arr[],int size){
    int min=arr[0];
    int i=0;
    for(i=1;i<size;i++){
        if(min>arr[i]){
            min=arr[i];
        }
    }//end of for
    return min;
}//end of function

int main(){
    int i=0,min=0;
    int numbers[]={4,5,7,3,8,9};//declaration of array
    min=minarray(numbers,6);//passing array with size
    printf("minimum number is %d \n",min);
    return 0;
}
```



Self-Assessment Questions

4. Which of the following functions is predefined in C?

- a) sum()
- b) myFunction()
- c) printf()
- d) add()

5. What results from the following recursive function call `sum(5)`?

- a) 5
- b) 10
- c) 25
- d) 15

6. Call by reference allows changes to the actual parameter.

- a) True
- b) False

7. Arrays in C can be passed to functions by passing individual elements.

- a) True
- b) False

1.3.3 Introduction to Arrays

1.3.3.1 One-Dimensional Array

An array is a group of identically typed data elements that may all be referenced with a single name.

A two-dimensional array is similar to a table, and a one-dimensional array is similar to a list. The C language does not limit the number of dimensions in an array, however certain implementations might.

When the number of dimensions is unknown or irrelevant, some texts use the broad term array and refer to one-dimensional arrays as vectors and two-dimensional arrays as matrices.

Array Declaration

The declaration of array variables is the same as that of variables of the same data type, with the exception that, for every array dimension, one pair of square [] brackets follow the variable name.

- The dimensions of uninitialised arrays, such as rows and columns, must be specified between square brackets.
- In C, dimensions used to declare arrays must be constant expressions or positive integral constants.
- In C, variables can be utilised if they have a positive value when the array is declared, but dimensions must still be positive integers.

Examples:

```
int i, j, intArray[ 10 ], number;  
float floatArray[ 1000 ];  
int tableArray[ 3 ][ 5 ]; /* 3 rows by 5 columns */  
const int NROWS = 100  
const int NCOLS = 200  
float matrix[NROWS ][ NCOLS ]
```

C Example:

```
int numElements:  
printf( "How big an array do you want?" );  
scanf( "%d", &numElements );  
if(numElements <= 0)  
{
```

```

printf("Error-Quitting\n")
exit(0);
}
double data numElements ];

```

Accessing Elements of One-Dimensional Array in C

Each element is identified by its index or position in a one-dimensional array. The index of the first element in the array is 0, and the index of the last element is arraySize - 1.

To access an element of a one-dimensional array in C programming language, we use the following syntax:

arrayName[index]

arrayName is the name of the array.

index is the index of the element we want to access.

Example of Accessing Elements of One-Dimensional Array in C

Let's take an example to interpret how to access elements of a one-dimensional array in C programming language.

```

#include <stdio.h>
int main() {
    int numbers[5] = {10, 20, 30, 40, 50};
    printf("The first element of the array is: %d\n", numbers[0]);
    printf("The third element of the array is: %d\n", numbers[2]);

    return 0;
}

```

Output:

- The first element of the array is: 10
- The third element of the array is: 30

Explanation:

In the above example, we have declared a one-dimensional array of integers named numbers. We have accessed the first element of the array using index 0 and the third element using index 2.

Accessing Elements of One-Dimensional Arrays

We can access individual elements of a one-dimensional array using their index, an integer value representing the element's position. The index of the first element in the array is 0, and the index of the last element is arraySize-1.

The syntax to access an element of a one-dimensional array in C is as follows:

arrayName[index]

arrayName is the name of the array.

index is the index of the element we want to access.

Modifying Elements of One-Dimensional Arrays

We can modify the value of individual elements of a one-dimensional array using their index. We simply assign a new value to modify an element using the assignment operator =.

Example of Modifying Elements of One-Dimensional Array in C

```
#include <stdio.h>
int main() {
    int numbers[5] = {10, 20, 30, 40, 50};
    printf("The third element of the array is %d\n", numbers[2]);
    numbers[2] = 35;
    printf("The third element of the array is now %d\n", numbers[2]);
    return 0;
}
```

Output:

- The third element of the array is 30
- The third element of the array is now 35

Explanation:

In the above example, we have declared a one-dimensional array of integers named numbers and initialised it with the values {10, 20, 30, 40, 50}. We have used the printf function to print the third element of the array, which is accessed using the index 2.

After that, we modified the value of the third element by assigning it a new value of 35. Finally, we used the print function again to print the new value of the third element.

Initialising One Dimensional Array in C

We can initialise a one-dimensional array in C programming language while declaring it or later in the program. We can initialise a one-dimensional array while declaring it by using the following syntax:

```
dataType arrayName[arraySize] = {element1, element2, ..., elementN};
```

- “`dataType`’ specifies the data type of the array.
- “`arrayName`’ is the name of the array.
- “`arraySize`’ specifies the number of elements in the array.
- “`{element1, element2, ..., elementN}`’ specifies the values of the elements in the array. The number of elements must be equal to “`arraySize`’.

Example of Initialising One Dimensional Array in C

Let’s take an example to interpret how to initialise a one-dimensional array in C programming language.

```
#include <stdio.h>
int main() {
    int numbers[5] = {10, 20, 30, 40, 50};
    for(int i=0; i<5; i++) {
        printf("numbers[%d] = %d\n", i, numbers[i]);
    }
    return 0;
}
```

Output of the code:

```
numbers[0] = 10
numbers[1] = 20
numbers[2] = 30
numbers[3] = 40
numbers[4] = 50
```

Explanation:

In the above example, we have declared a one-dimensional array of integers named `numbers` and initialised it with the values `{10, 20, 30, 40, 50}`. We have used a `for` loop to iterate over the elements of the array and print their values using the `printf` function.

We can also initialise a one-dimensional array later in the program by assigning values to its elements using the following syntax:

```
arrayName[index] = value;
```

- arrayName is the name of the array.
- index is the index of the element to which we want to assign a value.
- value is the value we want to assign to the element.

1.3.3.2 Multidimensional Arrays

A multidimensional array is an array of arrays with tabular data representing homogeneous data. Data is typically kept in memory in row-major order in multidimensional arrays.

Below is an example of how to declare N-dimensional arrays in general.

Syntax:

```
data_type array_name[size1][size2]....[sizeN];
```

data_type: Type of data to be stored in the array.
array_name: Name of the array.
size1, size2,..., sizeN: Size of each dimension.

Examples:

- Two dimensional array: int two_d[10][20];
- Three dimensional array: int three_d[10][20][30];

Size of Multidimensional Arrays:

Multiplying the dimensions' size can calculate the total number of elements that can be stored in a multidimensional array.

For example:

The array int x[10][20] can store total $(10 \times 20) = 200$ elements.

Similarly array int x[5][10][20] can store total $(5 \times 10 \times 20) = 1000$ elements.

To get the array's size in bytes, we multiply the size of a single element by the total number of elements in the array.

For example:

Size of array int $x[10][20] = 10 * 20 * 4 = 800$ bytes. (where int = 4 bytes)

Similarly, size of int $x[5][10][20] = 5 * 10 * 20 * 4 = 4000$ bytes. (where int = 4 bytes)

The most used forms of the multidimensional array are:

- Two-Dimensional Array
- Three-Dimensional Array
- Two-Dimensional Array in C

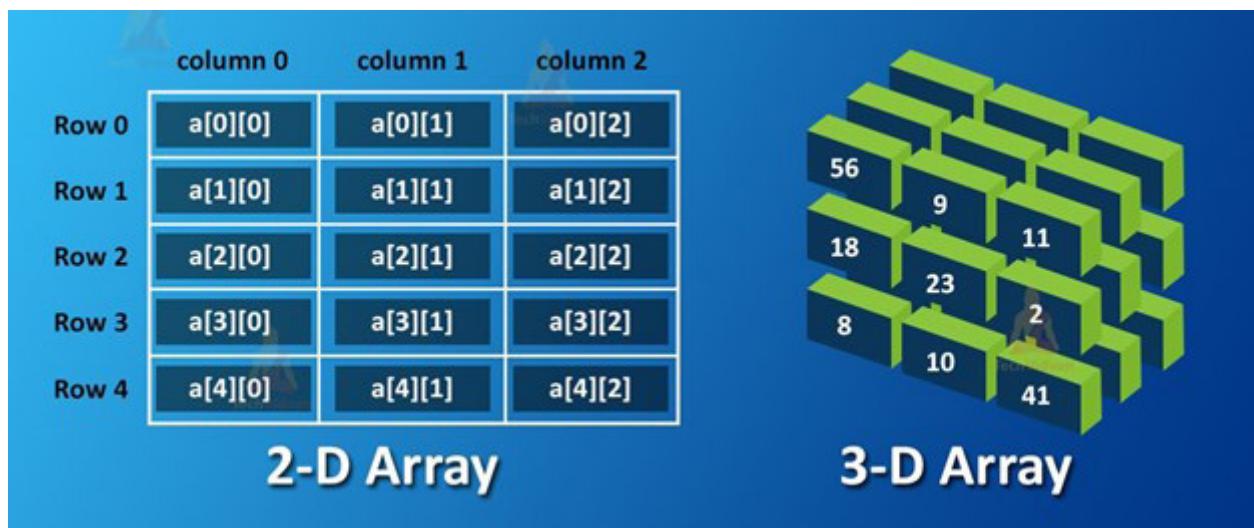


Fig. 2: Multidimensional array

A two-dimensional array

A two-dimensional array, or 2D in C, is the most basic multidimensional array type. It can be seen as an array of one-dimensional arrays stacked one on top of the other to create a table with "x" rows and "y" columns, where the row and column numbers vary from 0 to (x-1) and from 0 to (y-1).

Graphical Representation of Two-Dimensional Array of Size 3 x 3

Declaration of two-dimensional array in C

The basic form of declaring a 2D array with x rows and y columns in C is shown below.

Syntax:

```
data_type array_name[x][y];
```

where,

data_type: Type of data to be stored in each element.

array_name: name of the array

x: Number of rows.

y: Number of columns.

We can declare a two-dimensional integer array, say 'x' with 10 rows and 20 columns as:

Example:

```
int x[10][20];
```

Note: In this type of declaration, the array is allocated memory in the stack, and its size should be known at the compile time, i.e., fixed. We can also create an array dynamically in C by using the methods mentioned here.

Three-dimensional array in C

A Three-Dimensional Array, or 3D array in C, is a collection of two-dimensional arrays. It can be visualised as multiple 2D arrays stacked on each other.

graphical representation of a dimensional array

Graphical Representation of Three-Dimensional Array of Size 3 x 3 x 3

Declaration of Three-Dimensional Array in C

Using the syntax below, we can declare a 3D array with x 2D arrays, each with y rows and z columns.

Syntax:

```
data_type array_name[x][y][z];
```

data_type: Type of data to be stored in each element.

array_name: name of the array

x: Number of 2D arrays.

y: Number of rows in each 2D array.

z: Number of columns in each 2D array.

Example:

```
int array[3][3][3];
```

Initialisation of Three-Dimensional Array in C

Initialisation in a 3D array is the same as that of 2D arrays. The difference is that as the number of dimensions increases, the number of nested braces will also increase.





Self-Assessment Questions

8. What is the index of the first element in a one-dimensional array in C?

- a) 1
- b) 0
- c) -1
- d) Depends on the array

9. A multidimensional array stores data in a _____ order in memory.

- a) Row-major
- b) Column-major
- c) Circular
- d) Random

10. In C, array elements are accessed using their _____.

- a) Type
- b) Value
- c) Index
- d) Size



Summary

- Static memory allocation occurs at compile time, where a fixed amount of memory is reserved for global and static variables and arrays.
- In contrast, dynamic memory allocation occurs at runtime, allowing memory to be assigned and released as needed.
- Functions in C are blocks of code that can be reused multiple times by calling them. A function is composed of two parts: the declaration and the definition.
- In call by value, a copy of the value is passed to the function, and changes made do not affect the original value.
- In call by reference, the address of the variable is passed, allowing modifications made inside the function to affect the original variable.
- Recursion occurs when a function calls itself, breaking complex problems into smaller, more manageable tasks.
- A one-dimensional array (1D array) resembles a list, whereas a two-dimensional array (2D array) is like a table.
- A multidimensional array can store data in a tabular form and is declared by specifying multiple dimensions. A 2D array is like an array of 1D arrays; a 3D array is a collection of 2D arrays.



Terminal Questions

1. What is static memory allocation? Explain its properties and uses.
2. Compare and contrast static memory allocation with dynamic memory allocation.
3. What is the difference between call by value and call by reference in C?
4. How does recursion work, and what is a practical use case?
5. Describe how multidimensional arrays are represented and declared in C.



Answer Keys

Self-Assessment Questions	
Question No.	Answer
1	B
2	A
3	B
4	C
5	D
6	A
7	B
8	B
9	A
10	C



Activity

Activity Type: Offline

Duration: 1 hour

Imagine you are tasked with storing and processing the scores of 100 students in a class using a one-dimensional array. What would be the most efficient way to find the average score? How would you approach change if the number of students was unknown when writing the program?



Glossary

- **Heap Memory:** Memory space where dynamic memory allocation takes place.
- **Stack Memory:** Memory, such as local variables, is used for static memory allocation.
- **Memory Leak:** A condition where no longer needed memory is not released, leading to wasted memory.
- **Fragmentation:** The condition where small, unused memory blocks prevent allocating larger memory blocks.
- **Pointer:** A variable that stores the memory address of another variable.



Bibliography

Textbooks

- Balagurusamy, E. (2017). *Data Structures*. McGraw Hill Education.
- Sedgewick, R., & Wayne, K. (2014). *Algorithms II*. Pearson Education.
- Sridhar, S. (2014). *Design and Analysis of Algorithms*. Oxford University Press.
- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (1989). *Introduction to Algorithms*. Prentice-Hall.



e-References

- **Difference between Static and Dynamic Memory Allocation in C:** <https://www.naukri.com/code360/library/static-and-dynamic-memory-allocation>
- **C++ Functions:** https://www.w3schools.com/cpp/cpp_functions.asp
- **Difference Between Call by Value and Call by Reference in C:** <https://www.geeksforgeeks.org/difference-between-call-by-value-and-call-by-reference/>
- **Recursion in C:** <https://www.javatpoint.com/recursion-in-c>
- **One-Dimensional Arrays in C:** <https://www.geeksforgeeks.org/one-dimensional-arrays-in-c/>
- **Multidimensional Arrays in C – 2D and 3D Arrays:** <https://www.geeksforgeeks.org/multidimensional-arrays-in-c/>



Video Links

Video	Links
Static & Dynamic Memory Allocation	https://www.youtube.com/watch?v=udfbq4M2Kfc
Call By Value & Call by Reference in C	https://www.youtube.com/watch?v=HEiPxjVR8CU
Multidimensional Arrays	https://www.youtube.com/watch?v=36z4qgN3GWw



Image Credits

- **Fig. 1:** Self-Made
- **Fig. 2:** <https://techvidvan.com/tutorials/multidimensional-arrays-in-c/>



Keywords

- Static memory allocation
- Dynamic memory allocation
- Function
- Call by value
- Call by reference
- Predefined function
- One-dimensional array
- Two-dimensional array
- Multidimensional array
- Row-major order

MODULE 1

Introduction to Data Structures

Unit 4

Pointer and Strings



☰ Unit Table of Contents

Unit 1.4 Pointer and Strings

Aim	83
Instructional Objectives	83
Learning Outcomes	83
1.4.1 Pointer	84
1.4.1.1 Pointer to Structure	84
1.4.1.2 Various Programs for Array and Pointer	88
Self-Assessment Questions	94
1.4.2 Strings	95
1.4.2.1 Introduction and Definition of Strings	95
1.4.2.2 Library Functions of Strings	97
Self-Assessment Questions	103
Summary	104
Terminal Questions	104
Answer Keys	105
Activity	105
Glossary	106
Bibliography	106
e-References	106
Video Links	107
Image Credits	107
Keywords	107



Aim

To provide students with a comprehensive interpretation of the concepts of pointers and strings in C programming.



Instructional Objectives

This unit is designed to:

- Define a pointer and explain its significance in C programming
- Describe the different types of pointers, such as null and generic pointers
- Explain how to pass strings to functions and manipulate them using pointer notation



Learning Outcomes

At the end of the unit, the student is expected to:

- Perform basic arithmetic operations on pointers
- Analyse null and generic pointers in programs, including checking null pointer values and working with various types of data
- Compare and contrast between standard C library functions like strlen(), strcat(), strcpy(), etc

1.4.1 Pointer

1.4.1.1 Pointer to Structure

It may be defined as a variable that can store the address of another variable. The addresses of variables are always numbers so that pointers can store only numbers, i.e., garbage values (unpredictable values).



The pointer has the following advantages:

1. Accessing array elements.
2. Passing arguments to functions by reference (i.e. the arguments can be modified).
3. Passing arrays and strings to functions.
4. Creating data structures such as linked lists, trees, graphs, etc.
5. Obtaining memory from the system dynamically.

Declaration: datatype *variable name;

int *p; → here p is a pointer to integer
 char *p; → here p is a pointer to a character
 float *p; → here p is a pointer to float

Operators used in pointers:

1. Address operator (&)
2. value of address operator/Indirection operator (*)

1. Address operator (&): It returns the address of a specific variable.

- `Printf("%d",x);`→ 12
- `Printf("%d",&x);`→1000

2. Value of address operator: It returns the value stored at a specific address

- `x→12`
- `*(&x)→*(1000)→12`

Pointer Arithmetic:

As a pointer holds a variable's memory address, some arithmetic operations can be performed with pointers. C supports four arithmetic operators that can be used with a pointer, such as

- Addition +
- Subtraction -
- Incrementation ++
- Decrementation --

Pointers are variables. They are not integers, but they can be displayed as unsigned integers. The conversion specifier for a pointer is added and subtracted.

Examples:

Increment operator on pointer variable:

if x is the pointer variable

1. int *x
 x++;

The value x is incremented by 2 since the integer type requires 2 bytes long.

2. float *x;
 x++;

The value of x is incremented by 4 since the floating type requires 4 bytes long.

3. char *x;
 x++;

The value of x is incremented by 1 since the character requires 1 byte long.

Every time a pointer is incremented, it points to the immediate next location of its base type.

Decrement operator on a pointer variable:

```
int *x;  
x--;
```

Each time a pointer is decremented, it points to the location of the previous element.

A number can be added to the pointer:

```
int i, *j;  
i=5;  
j=&i;  
j=j+i;
```

A number can be subtracted from a pointer:

```
int i=5;  
int *j;  
j=&i;  
j=j-2;
```

Subtraction of one pointer from another is possible:

This is possible when both variables point to elements of the same array. The resulting value indicates the number of bytes separating the corresponding array elements.

```
main( )  
{  
    int a[ ] = {10, 20, 30};  
    int *i, *j;  
    i=&a[0];  
    j=&a[2];  
    printf("%d",j-i);  
}
```

j-i would print a value of 2 and not 6; i.e., the j and i are pointing to locations that are 2 integers apart.

Types of Pointers

- (i) Null Pointer
- (ii) Generic Pointer

(i) Null Pointer

A null pointer is a special pointer value known not to point anywhere. A NULL pointer does not point to any valid memory address. To declare a null pointer, you may use the predefined constant NULL.

```
int *ptr = NULL;
```

We can always check whether a given pointer variable stores the address of some variable or contains a null by writing,

```
if ( ptr == NULL)
{Statement block;
}
```

Null pointers are used when one of the program's pointers points somewhere sometimes but not always. In such situations, it is always better to set it to a null pointer when it doesn't point anywhere valid and test to see if it's a null pointer before using it.

(ii) Generic Pointer

A generic pointer is a pointer variable with void as its data type. The generic pointer can be pointed at variables of any data type. It is declared by writing:

```
void *ptr;
```

Before using, we need to cast a void pointer to another kind of pointer. Generic pointers are used when a pointer points to data of different types at different times. For example,

```
#include<stdio.h>

int main()
{
    int x=10;
    char ch = 'A';
    void *gp;
    gp = &x; printf("\n Generic pointer points to the integer value = %d", *(int*)gp);
    gp = &ch; printf("\n Generic pointer now points to the character %c", *(char*)gp);
    return 0;
}
```

OUTPUT:

Generic pointer points to the integer value = 10

Generic pointer now points to the character = A 1.4.1.2 Various Programs for Array and Pointer

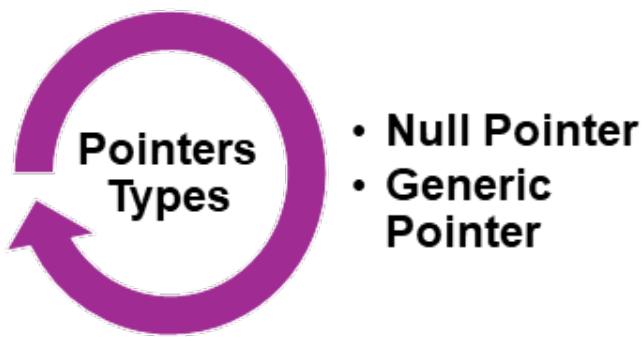


Fig. 1: Types of Pointers

1.4.1.2 Various Programs for Array and Pointer

(i) Pointers and Arrays

The concept of arrays is very much related to pointers. Array occupies consecutive memory locations.

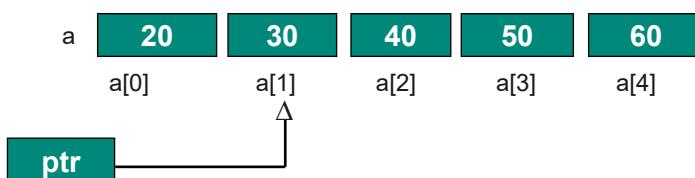
a	20	30	40	50	60
	65516	65518	65520	65522	65524
	a[0]	a[1]	a[2]	a[3]	a[4]

Array notation is a form of pointer notation. The address of the first element of the array is called the base address. The array's name is a pointer pointing to the array's first element. Let us use a pointer variable as:

```
int *ptr;
ptr=&arr[0];
```

Example:

```
int arr[5] = {1, 2, 3, 4, 5};
int *ptr = &arr[0];
ptr++;
printf("\n The value of the second element of the array is %d", *ptr);
```



(ii) Passing arrays to functions

An array can be passed to a function using pointers. The function that expects an array can declare the formal parameters in the following two ways:

Func(int arr[]);

OR

Func(int *arr);

When we pass the name of the array to the function, the address of the 0th element of the array is copied to the local pointer variable in the function. We can access all the array elements using the array name and index. The declaration should be as follows:

Func(int arr[], int n);

OR

Func(int *arr, int n);

(iii) Difference between Array Name and Pointer:

1. When memory is allocated for an array, its base address cannot be changed during program execution, which means the array name is an address constant. However, the pointer variable may change.
2. An array cannot be assigned to another array; however, one pointer variable can be assigned to another.
3. For the pointer, the address operator returns the address of the operand. For arrays, the array name (arr) and address of the array (&arr) give the same value.
4. For arrays, the size of the operator returns the number of bytes allocated. The sizeof the operator returns the number of bytes used for the pointers.

Array of pointers: An array of pointers can be declared as int *ptr[10]

The above statement declares an array of 10 pointers where each pointer points to an integer variable. For example, look at the code given below.

```
int *ptr[10];
int p=1, q=2, r=3, s=4, t=5;
ptr[0]=&p;
ptr[1]=&q;
ptr[2]=&r;
```

```
ptr[3]=&s;
ptr[4]=&t
```

Can you tell me what the output of the following statement will be?

```
printf("\n %d", *ptr[3]);
```

Yes, the output will be 4 because ptr[3] stores the address of integer variable s and *ptr[3] will therefore print the value of s that is 4.

Pointer and 2D Arrays: Individual elements of the array mat can be accessed using either:

mat[i][j] or *(*(mat + i) + j) or*(mat[i]+j);

- Pointer to a one-dimensional array can be declared as,

```
int arr[]={1,2,3,4,5};
int *parr;
parr=arr;
```

- Similarly, a pointer to a two-dimensional array can be declared as,

```
int arr[2][2]={{1,2},{3,4}};
int (*parr)[2];
parr=arr;
```

- Look at the code below, which illustrates using a pointer to a two-dimensional array.

```
#include<stdio.h>
main()
{
    int arr[2][2]={{1,2},{3,4}};
    int i, (*parr)[2];
    parr=arr;
    for(i=0;i<2;i++)
    {
        for(j=0;j<2;j++)
            printf(" %d", (*(parr+i))[j]);
    }
}
```

OUTPUT

1 2 3 4

Pointer and 3D Arrays:

- Pointer to a one-dimensional array can be declared as,

```
int arr[]={1,2,3,4,5};  
int *parr;  
parr=arr;
```

- Pointer to a two-dimensional array can be declared as,

```
int arr[2][2]={{1,2},{3,4}};  
int (*parr)[2];  
parr=arr;
```

- Pointer to a 3-dimensional array can be declared as,

```
int arr[2][2][2]={1,2,3,4,5,6,7,8};  
int (*parr)[2][2];
```

Function Pointers:

- This technique helps pass a function as an argument to another function.
- If we have declared a pointer to the function, then that pointer can be assigned to the address of the right sort of function just by using its name.
- When a pointer to a function is declared, it can be called using one of two forms:
`(*func)(1,2); OR func(1,2);`

```
#include <stdio.h>  
void print(int n);  
main()  
{  
    void (*fp)(int);  
    fp = print;  
    (*fp)(10);  
    fp(20);  
    return 0;  
}  
  
void print(int value)  
{printf("\n %d", value);  
}
```

(i) Comparing Function Pointers: Comparison operators == and != are used as usual.

```

if(fp >0)           // check if initialised
{
  if(fp == print)
    printf("\n Pointer points to Print");
  else
    printf("\n Pointer not initialised!");
}
  
```

- A function pointer can be passed as a function's calling argument. This is done when you want to pass a pointer to a callback function.

(ii) Array of Function Pointer: When an array of functions is made, the appropriate function is selected using an index. In the following way, we can define and use an array of function pointers in C:

1. Use `typedef` so that 'fp' can be used as

```
Typedef int (*fp) (int, int);
```

2. Define the array and initialise the elements to NULL. This can be done in two ways:

- `fp funcarr[10]={NULL};`
- `int (*funcarr[10]) (int, int) = {NULL};`

3. Assign the function address.

4. Call the function using an index to address the function pointer.

(i) Pointer and Strings: Consider the following program that prints a text.

```

#include<stdio.h>
main()
{
  char str[] = "Oxford";
  char *pstr = str;
  printf("\n The string is : ");
  while( *pstr != '\0')
  {
    printf("%c", *pstr);
    pstr++;
  }
}
  
```

- In this program, we declare a character pointer *pstr to show the string on the screen. We then “point” the pointer pstr at str. Then, we print each character of the string in the while loop. Instead of using the while loop, we could have straight away used the function puts(), like

- `puts(pstr);`

- Consider here that the function prototype for puts() is:

- `int puts(const char *s);` Here, the “const” modifier is used to assure the user that the function will not modify the contents pointed to by the source pointer. Note that the address of the string is passed to the function as an argument.

- (ii) Pointer to pointers:** We can use pointers that point to pointers—the pointers, in turn, point to data (or even to other pointers). Just add an asterisk (*) for each reference level to declare pointers to pointers.

For example, if we have:

```
int x=10;
int *px, **ppx;
px=&x;
ppx=&px;
```

Now if we write,

```
printf("\n %d", **ppx);
```

Then, it would print 10, the value of x.



Self-Assessment Questions

1. A _____ pointer does not point to any valid memory location.

- a) wild
- b) generic
- c) null
- d) void

2. To declare a generic pointer, we use the _____ data type.

- a) int
- b) char
- c) float
- d) void

3. Pointers to functions cannot be passed as arguments to other functions.

- a) True
- b) False

4. Pointers can be used for dynamic memory allocation.

- a) True
- b) False

5. A pointer is a variable that stores the _____ of another variable.

- a) Value
- b) Size
- c) Name
- d) Address

1.4.2 Strings

1.4.2.1 Introduction and Definition of Strings

A string is a null-terminated character array. This means that after the last character, a null character ('\0') is stored to signify the end of the character array.

- The general form of declaring a string is

```
char str[size];
```

- For example, if we write,

```
char str[] = "HELLO";
```

We declare a character array with 5 characters, namely, H, E, L, L and O. Besides, a null character ('\0') is stored at the end of the string. So, the internal representation of the string becomes- HELLO'\0'. Note that to store a string of length 5, we need 5 + 1 locations (1 extra for the null character).

The name of the character array (or the string) is a pointer to the beginning of the string.

str[0]	1000	H
str[1]	1001	E
str[2]	1002	L
str[3]	1003	L
str[4]	1004	O
str[5]	1005	/0

Reading of a string:

If we declare a string by writing

```
char str[100];
```

Then, str can be read from the user by using three ways

- use scanf function
- using gets() function
- using getchar()function repeatedly

The string can be read using scanf() by writing

```
scanf("%s", str);
```

The string can be read by writing

```
gets(str);
```

gets() and takes the starting address of the string that will hold the input. The string inputted using gets() is automatically terminated with a null character.

Writing of a string:

The string can be displayed on the screen using three ways

- use printf() function
- using puts() function
- using putchar()function repeatedly

The string can be displayed using printf() by writing

```
printf("%s", str);
```

The string can be displayed by writing

```
puts(str);
```

Suppressing input

scanf() can read a field without assigning it to any variable. This is done by preceding that field's format code with a *. For example, given:

```
scanf("%d*c%d", &hr, &min);
```

The time can be read as 9:05 as a pair. Here, the colon would be read but not assigned to anything.

Using a Scanset

- The ANSI standard added the new scanset feature to the C language. A scanset defines a set of characters that may be read and assigned to the corresponding string. A scanset is defined by placing the characters inside square brackets prefixed with a %.

```
int main()
{
    char str[10];
    printf("\n Enter string: " );
```

```

scanf("%[aeiou]", str );
printf( "The string is : %s", str);
return 0;
}

```

- The code will stop accepting characters when the user enters a character that is not a vowel.
- However, if the first character in the set is a ^ (caret symbol), then scanf() will accept any character not defined by the scanset. For example, if you write

```
scanf("%[^aeiou]", str );
```

1.4.2.2 Library Functions of Strings

The sscanf() function:

This function reads data from str and stores them according to the parameter format in the specific location. Consider the following:

```
sscanf(str, "%d", &num);
```

- a) The first parameter, str, contains the data to be converted.
- b) The second parameter, %d, determines how the string is converted.
- c) The third parameter specifies the memory location where the result of the conversion is placed.

String taxonomy

We can store a string either in a fixed or variable-length format.

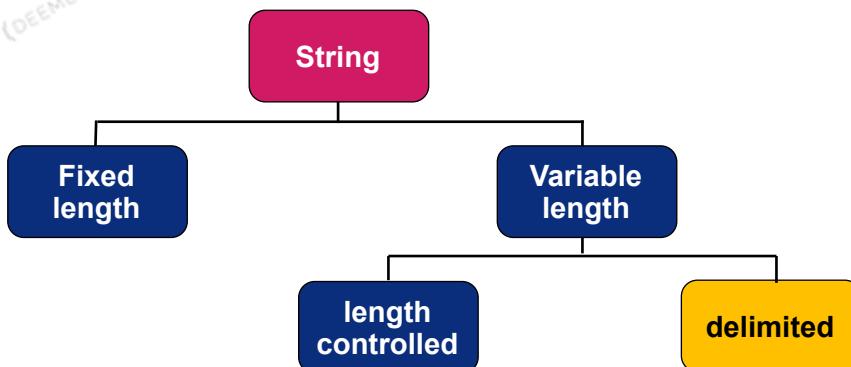


Fig. 2: String taxonomy

Fixed-length String: When storing a string in fixed-length format, we need to specify an approximate size for the string variable.

Variable Length String: In variable length format, the string is expanded or condensed to accommodate its elements. This can be done in two ways:

- **Length-controlled string:** In this format, we must specify the count, which is then used by the string manipulation functions.
- **Delimited string:** In this case, the string ends with a delimiter. The NULL character is the most used delimiter in the C language.

String Operation:

1. Length:

- The number of characters in the string constitutes the length of the string. For example, LENGTH("C PROGRAMMING IS FUN") will return 20. Note that even blank spaces are counted as characters in the string. LENGTH('0') = 0 and LENGTH("") = 0 because both the strings do not contain any character.

Step 1: [INITIALISE] SET I = 0
Step 2: Repeat Step 3 while STR[I] != '\0'
Step 3: SET I = I + 1
[END OF LOOP]
Step 4: SET LENGTH = I
Step 5: END

2. Converting to upper case:

- A character's ASCII code is stored in memory instead of real value. The ASCII code for A-Z varies from 65 to 91, and the ASCII code for A-Z ranges from 97 to 123. So, if we must convert a lower-case character into upper-case, then we need to subtract 32 from the ASCII value of the character.

Step1: [Initialise] SET I=0

Step 2: Repeat Step 3 while STR[I] != '\0'

Step 3: IF STR[1] > 'a' AND STR[I] < 'z'

SET Upperstr[I] = STR[I] - 32

ELSE

SET Upperstr[I] = STR[I]

[END OF IF]

[END OF LOOP]

Step 4: SET Upperstr[I] = '\0'

Step 5: EXIT

3. Converting to upper case:

- In memory the ASCII code of a character is stored instead of its real value. The ASCII code for A varies from 65 to 91, and the ASCII code for a ranges from 97 to 123. So, if we must convert a lowercase character into uppercase, we need to add 32 from the ASCII value of the character.

Step1: [Initialise] SET I=0

Step 2: Repeat Step 3 while STR[I] != '\0'

Step 3: IF STR[1] > 'A' AND STR[I] < 'Z'

SET Lowerstr[I] = STR[I] + 32

ELSE

SET Lowerstr [I] = STR[I]

[END OF IF]

[END OF LOOP]

Step 4: SET Lowerstr [I] = '\0'

Step 5: EXIT

4. Concatenating two strings:

If S1 and S2 are two strings, the concatenation operation produces a string containing the characters of S1 followed by the characters of S2.

Step 1. Initialise I =0 and J=0
Step 2. Repeat step 3 to 4 while I <= LENGTH(str1)
Step 3. SET new_str[J] = str1[I]
Step 4. Set I =I+1 and J=J+1
 [END of step2]
Step 5. SET I=0
Step 6. Repeat step 6 to 7 while I <= LENGTH(str2)
Step 7. SET new_str[J] = str1[I]
Step 8. Set I =I+1 and J=J+1
 [END of step5]
Step 9. SET new_str[J] = '\0'
Step 10. EXIT

5. Reversing a String:

If S1= “HELLO”, then the reverse of S1 = “OLLEH”. To reverse a string, we must swap the first character with the last, the second with the second last character, and so on.

Step1: [Initialise] SET I=0, J= Length(STR)
Step 2: Repeat Step 3 and 4 while I< Length(STR)
Step 3: SWAP(STR(I), STR(J))
Step 4: SET I = I + 1, J = J – 1
 [END OF LOOP]
Step 5: EXIT

Character and String functions.

Character Functions:

1.	islower()	Returns TRUE if the argument is a lowercase letter.
2.	isupper()	Returns TRUE if the argument is an uppercase letter.
3.	isascii()	Returns TRUE if the integer argument is in the ASCII range 0-127. Treats 128-255 as non-ASCII.
4.	toascii()	The argument (an arbitrary integer) is converted to a valid ASCII character number 0-127. c = toascii(500); //c gets number 116 // (modulus 500%128) c = toascii('d'); //c gets number 100
5.	tolower()	Converts the argument (an uppercase ASCII character) to lowercase. c = tolower('Q'); // c becomes 'q'
6.	toupper()	Converts the argument (a lowercase ASCII character) to uppercase. c = toupper('q'); //c becomes 'Q'

String Functions

1. **strcat ()**: The purpose of this function is to concatenate or combine two different strings.

- **Syntax**: `strcat(string1, string2);`
- **Example**: `strcat("HELLO", "FRIEND")`

2. **strcmp()**: This function compares two strings. If two strings are equal, it returns 0. If string1 is larger than string2, it returns a -ve value; if string2 is larger than string2, it returns a +ve value.

- **Syntax**: `strcmp(string1, string2)`
- **Example**: `strcmp("HELLO", "HAI");`

3. **strcpy()**: The purpose of this function is to copy one string to another.

- **Syntax**: `strcpy(string1, string2);`
- **Example**: `char name[30]; strcpy(name, "Rishi");`

4. `strlen()`: This function counts the number of characters in a string to find its length.

- **Syntax:** `n = strlen(string1)`
- **Example:** `n = strlen("Rishi")`

5. `strrev()`: The purpose of this function is to reverse a string. Here, the first character becomes the last, and the last character becomes the first character.

- **Syntax:** `strrev(string)`
- **Example:** `strrev("Rishi");`

6. `strlwr()`: This function converts the string into lowercase.

- **Syntax:** `strlwr(string)`
- **Example:** `strlwr("Rishi")`





Self-Assessment Questions

6. Which function can read a string from the user in C?

- a) printf()
- b) puts()
- c) gets()
- d) strrev()

7. What will be the output of strlen("HELLO") ?

- a) 6
- b) 7
- c) 4
- d) 5

8. What does strcmp() return if two strings are identical?

- a) 1
- b) 0
- c) -1
- d) It depends on the length of the strings

9. In C, string length includes the null character ('\0').

- a) True
- b) False

10. A scanset restricts the input to certain characters in scanf().

- a) True
- b) False



Summary

- A pointer is a variable that stores the memory address of another variable, allowing for more flexible manipulation of data structures and arrays.
- Pointers are essential in C programming, offering several advantages, such as accessing array elements, passing arguments by reference, creating dynamic data structures, and more.
- The syntax to declare a pointer is datatype *variable_name, and operators like the address operator & and indirection operator * are used to manipulate pointers.
- Pointer arithmetic allows the addition, subtraction, incrementing, and decrementing of memory addresses based on the data type.
- Types of pointers include null and generic pointers, with the latter being able to point to data of any type.
- A string in C is a null-terminated character array, meaning it ends with the unique null character ('\0').
- Strings can be declared using a simple syntax: char str[size], and when initialised, C automatically adds the null character at the end.
- Strings can be read from the user using various functions like scanf(), gets(), and getchar(), while they can be displayed using printf(), puts(), or putchar().
- Strings can be stored in two formats: fixed-length and variable-length.
- String operations are fundamental, such as calculating a string's length, converting between upper and lowercase, concatenating, and reversing strings.



Terminal Questions

1. What are the key advantages of using pointers in C programming?
2. Describe how arrays and pointers are related. How does pointer arithmetic work in a 2D array?
3. What is a string in C, and how is it represented in memory?
4. Explain the different ways to read a string from the user.
5. Differentiate between fixed-length and variable-length strings.
6. List and explain any four string functions in C.



Answer Keys

Self-Assessment Questions	
Question No.	Answer
1	C
2	D
3	B
4	A
5	D
6	C
7	D
8	B
9	D
10	A



Activity

Activity Type: Offline

Duration: 1 hour

Imagine you're working on a project that involves dynamic memory allocation. You're tasked with developing a program that manages student records. Each student record contains a unique ID, name, and grade. The number of students is unknown at compile time, so you use pointers to allocate memory for student records as the program runs dynamically.



Glossary

- **Null Character (\0):** A unique character that marks the end of a string in C.
- **Variables:** A variable is a named storage location in memory with a value.
- **Arithmetic Operations:** Arithmetic operations are mathematical calculations performed on numeric data types.
- **Array Notation:** Array notation is a way to access elements in an array using the array's name and the element's index.
- **Pointer Notation:** Pointer notation refers to the use of pointers to access the memory address of a variable.



Bibliography

Textbooks

- Balagurusamy, E. (2017). *Data Structures*. McGraw Hill Education.
- Sedgewick, R., & Wayne, K. (2014). *Algorithms II*. Pearson Education.
- Sridhar, S. (2014). *Design and Analysis of Algorithms*. Oxford University Press.
- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (1989). *Introduction to Algorithms*. Prentice-Hall.



e-References

- **C Pointers:** <https://www.geeksforgeeks.org/c-pointers/>
- **An array of Pointers in C:** <https://www.geeksforgeeks.org/array-of-pointers-in-c/>
- **Introduction to Strings – Data Structure and Algorithm:** <https://www.geeksforgeeks.org/introduction-to-strings-data-structure-and-algorithm-tutorials/>
- **String Library functions:** <https://www.tutorialspoint.com/explain-string-library-functions-with-suitable-examples-in-c>



Video Links

Video	Links
Introduction to pointers & pointer arithmetic	https://www.youtube.com/watch?v=kKKvGYAX_Zs
Basics of String Literals	https://www.youtube.com/watch?v=IlqiTmcK1Eg&list=PLBlnK6fEyqRhQbYrTDZYJaB4z1YgsAPW
String Functions in C Programming	https://www.youtube.com/watch?v=E7T2cnSjO3M



Image Credits

- **Fig. 1:** Self-Made
- **Fig. 2:** Self-Made



Keywords

- String
- Fixed-length string
- Variable-length string
- Pointer
- Indirection operator
- Generic pointer
- Null pointer
- Function pointer

DATA STRUCTURES AND ALGORITHMS

MODULE 2

Linked Lists, Stacks, and Queue



Module Description

This module introduces students to fundamental data structures critical for efficient data management and algorithm development. It begins with linked lists, a collection of nodes connected by pointers. Students will explore different types of linked lists, including singly linked lists, doubly linked lists, circular linked lists, and circular doubly linked lists. Each type has unique properties and use cases, and the module covers their representation, implementation, and basic operations, such as insertion, deletion, and traversal.

Students will examine the definition of a stack and explore various ways to implement it using arrays and linked lists. The core operations, such as push, pop, peek, and checking if the stack is empty or full, will be discussed in detail. The module also highlights real-world applications of stacks, including expression evaluation, function call management in recursion, and backtracking algorithms. Additionally, it covers multiple stacks, demonstrating how they can be used to solve more complex problems by dividing memory into several stack areas.

After defining the queue, the implementation using arrays and linked lists will be explained. Key operations will be covered, such as enqueue (adding elements), dequeue (removing elements), and checking if the queue is full or empty. Students will explore circular queues, which overcome the limitations of linear queues by efficiently using memory. The module then explains de-queues (double-ended queues), which allow insertion and deletion from both ends, and priority queues, where elements are processed based on priority rather than the insertion order.

The module consists of **four** units.

Unit 2.1: Overview of Linked Lists

Unit 2.2: Outline of Stack

Unit 2.3: Fundamental Concept of Queue

Unit 2.4: Types of Queues

MODULE 2

Linked Lists, Stacks, and Queue

Unit 1

Overview of Linked Lists



☰ Unit Table of Contents

Unit 2.1 Overview of Linked Lists

Aim	112
Instructional Objectives	112
Learning Outcomes	112
2.1.1 Linked Lists	113
2.1.1.1 Introduction	113
2.1.1.2 Representation	114
2.1.1.3 Operations	116
Self-Assessment Questions	119
2.1.2 Types of Linked Lists	120
2.1.2.1 Singly Linked List	120
2.1.2.2 Doubly Linked List	129
2.1.2.3 Circular and Doubly Linked List	130
Self-Assessment Questions	140
Summary	141
Terminal Questions	141
Answer Keys	142
Activity	142
Glossary	143
Bibliography	143
e-References	143
Video Links	144
Image Credits	144
Keywords	144



Aim

To provide students with a comprehensive interpretation of linked lists, their types, representation, and operations, along with the ability to implement and manipulate singly linked lists, doubly linked lists, circular linked lists, and circular doubly linked lists.



Instructional Objectives

This unit is designed to:

- Define the concept of linked lists and their importance in data structure
- Describe the essential operations (insertion, deletion, traversal, searching) on linked lists
- Explain the structure of a singly linked list and its one-directional node connection



Learning Outcomes

At the end of the unit, the student is expected to:

- Recognise how linked lists differ from arrays in structure and efficiency
- Illustrate how to visualise linked lists and their nodes in diagrams or code
- Analyse implementation of a doubly linked list in code, performing operations in forward and reverse direction

2.1.1 Linked Lists

2.1.1.1 Introduction

A linked list's members are linear data structures without contiguous memory addresses. Rather, pointers that connect one node to the next interconnect each component or node. In a linked list, every node comprises data and a reference to the node after it.

A linked list is a group of nodes. Each node consists of a data field plus a pointer (or link) to the following node in the sequence. Pointers connect these nodes, creating a structure that resembles a chain.

Basic terms in linked lists:

- **Head:** The head is the first node in a linked list, marking the starting point of the list.
- **Node:** A node is the fundamental building block of a linked list, containing both the data and a reference (or pointer) to the next node.
- **Data:** This is the portion of the node that holds the information or value.
- **Next Pointer:** This pointer points to the subsequent node in the list, allowing the linked list to connect each node sequentially.

The importance of linked lists

- Linked lists can dynamically allocate or deallocate memory at run-time, making them adaptable to different situations.
- When performing operations like insertion or deletion, linked lists only require adjusting pointers rather than shifting elements, simplifying these processes.
- They also use memory better since they grow or shrink as needed, reducing the risk of memory waste.
- Linked lists can implement more advanced data structures such as stacks, queues, graphs, and hash maps.

Linked lists advantages

- Linked lists allow for dynamic memory allocation, which adds flexibility.
- Memory utilisation is more efficient because the list size adjusts according to requirements, preventing unnecessary waste.
- Inserting or deleting nodes is simple and can be done at any position with minimal overhead.
- Linked lists are a foundation for implementing more complex data structures like stacks, queues, trees, and graphs.
- Linked lists can expand or contract in constant time.

Disadvantages of linked lists

- Pointers add memory overhead and make the structure more complex than arrays.
- Since linked lists store non-sequentially, accessing an element requires sequential traversal starting from the head, making random access impossible.
- Searching for a specific element can be slow, necessitating traversing the entire list, leading to $O(n)$ time complexity.
- Also, reverse traversal is not feasible in singly linked lists without making additional modifications.

Applications of linked lists

- They are fundamental in implementing linear data structures like stacks and queues and non-linear structures like hash maps and graphs.
- In dynamic memory allocation, linked lists manage free memory blocks. Graph representation through adjacency lists relies on linked lists to store adjacent vertices.
- In web browsers and text editors, doubly linked lists enable forward and backward navigation features.
- Circular doubly linked lists are also used in advanced data structures like Fibonacci heaps, offering further practical uses.

2.1.1.2 Representation

A linked list can be visualised as a chain of nodes, with each node containing a data field and a reference (or pointer) to the next node. The first node, referred to as the head, marks the beginning.

of the list, and the last node in the list points to NULL, indicating the end of the list.

Each node consists of:

- A data field for storing information.
- A pointer to the next node in the sequence.

Example Structure in C:

The following example demonstrates how to define and use a simple linked list in the C programming language:

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

```

int main() {
    // Initialise nodes
    struct node *head;
    struct node *one = NULL;
    struct node *two = NULL;
    struct node *three = NULL;

    // Allocate memory
    one = (struct node*)malloc(sizeof(struct node));
    two = (struct node*)malloc(sizeof(struct node));
    three = (struct node*)malloc(sizeof(struct node));

    // Assign data values
    one->data = 1;
    two->data = 2;
    three->data = 3;

    // Connect nodes
    one->next = two;
    two->next = three;
    three->next = NULL;

    // Assign head pointer
    head = one;

    return 0;
}
  
```

In this example, a linked list with three nodes is created. The nodes are connected in sequence, with the last node pointing to NULL, indicating the end of the list. The head pointer stores the reference to the first node, allowing traversal of the list.

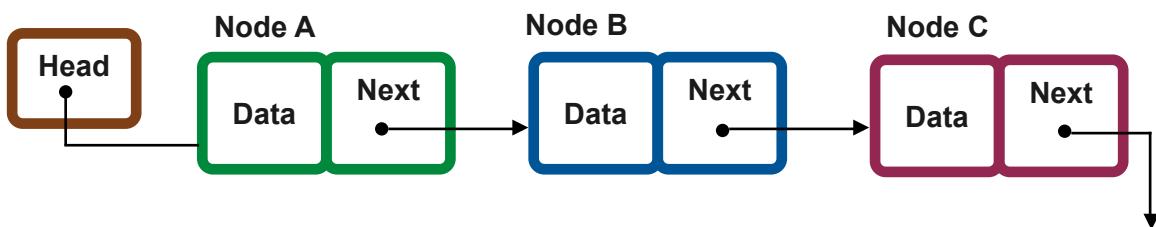


Fig. 1: Representation of linked list

2.1.1.3 Operations

Linked lists are dynamic data structures that allow flexible and efficient data manipulation. The following are the key operations that can be performed on a linked list:

1. Traversal

In this operation, each node of the linked list is accessed sequentially to process or retrieve data. Each node is visited from the head node until the end of the list (where the next pointer is null).

```

struct node *temp = head;
while (temp != NULL) {
    printf("%d --->", temp->data);
    temp = temp->next;
}
  
```

Output

List elements are -

1 ---> 2 ---> 3 --->

2. Insertion

New nodes are inserted into the list at various positions: at the beginning, at the end, or in the middle.

At the beginning, allocate memory, set data, point the new node to the head, and update the head.

```

newNode = malloc(sizeof(struct node));
newNode->data = 4;
newNode->next = head;
head = newNode;
  
```

At the End: Allocate memory, set data, traverse to last node, link new node.

```

newNode = malloc(sizeof(struct node));
newNode->data = 4;
newNode->next = NULL;
temp = head;
while (temp->next != NULL) temp = temp->next;
temp->next = newNode;
  
```

At the Middle: Traverse to the desired position and insert a new node.

```

for (int i = 2; i < position; i++) temp = temp->next;
newNode->next = temp->next;
temp->next = newNode;
  
```

3. Deletion

Remove nodes from the linked list at the beginning, the end, or from a specific position.

From Beginning: Point head to the second node.

```
head = head->next;
```

From End: Traverse to the second last node, set next to NULL.

```

while (temp->next->next != NULL) temp = temp->next;
temp->next = NULL;
  
```

From Middle: Traverse to the node before updating pointers.

```

for (int i = 2; i < position; i++) temp = temp->next;
temp->next = temp->next->next;
  
```

4. Search

The search operation traverses the linked list to find a specific node based on a given value or key. If the node is found, the corresponding data is retrieved.

```

bool searchNode(struct Node** head_ref, int key) {
    struct Node* current = *head_ref;
  
```

```
while(current != NULL) {  
    if(current->data == key) return true;  
    current = current->next;  
}  
  
return false;  
}
```

5. Sort

Based on their data values, sorting arranges the nodes in a specific order, typically ascending or descending. Sorting can be implemented using algorithms such as bubble sort or merge sort, which can be adjusted for linked list structures.

```
void sortLinkedList(struct Node** head_ref) {  
    struct Node *current = *head_ref, *index = NULL;  
    int temp;  
  
    if (head_ref == NULL) return;  
  
    while(current != NULL) {  
        index = current->next;  
  
        while(index != NULL) {  
            if(current->data > index->data) {  
                temp = current->data;  
                current->data = index->data;  
                index->data = temp;  
            }  
            index = index->next;  
        }  
  
        current = current->next;  
    }  
}
```



Self-Assessment Questions

1. In a linked list, each node contains a data field and a _____ to the next node.
 - a) Link
 - b) Function
 - c) Reference
 - d) Header

2. In a singly linked list, each node contains a pointer to the previous and next nodes.
 - a) True
 - b) False

3. The head of a linked list points to the _____ node.
 - a) Last
 - b) Middle
 - c) First
 - d) Random

4. Linked lists allow for _____ memory allocation, which enhances their flexibility.
 - a) Static
 - b) Dynamic
 - c) Sequential
 - d) Temporary

5. Insertion and deletion in a linked list are generally more straightforward and faster than in an array.
 - a) True
 - b) False

2.1.2 Types of Linked Lists

2.1.2.1 Singly Linked List

Data structures are crucial in computer programming because they make data handling and storage efficient. The linked list is a typical data structure. In this blog post, we will examine the concept of a single linked list in the C programming language. We'll review its operations, definition, and example code syntax and outputs.

Each node in a singly linked list has a data element and a reference to the node after it in the list, making it a linear data structure. The list's head node is the first node, while the last node points to NULL to denote the list's end.

A structure for each list node must be constructed to define a single linked list in C. The structure should have two fields: a data field for storing the actual data and a pointer field for holding a reference to the subsequent node.

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

To establish a single linked list, we must first initialise the head reference to NULL, which denotes an empty list. After that, we may add nodes to the list by dynamically allocating memory for each node and connecting them with the next pointer.

```
struct Node* head = NULL; // Initialising an empty list  
  
void insert(int value) {  
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));  
    newNode->data = value;  
    newNode->next = NULL;  
  
    if (head == NULL) {  
        head = newNode;  
    } else {
```

```

struct Node* current = head;
while (current->next != NULL) {
    current = current->next;
}
current->next = newNode;
}
}

```

To establish a single linked list, we must first initialise the head reference to NULL, which denotes an empty list. After that, we may add nodes to the list by dynamically allocating memory for each node and connecting them with the next pointer.

```

void traverse() {
    struct Node* current = head;
    while (current != NULL) {
        printf("%d ", current->data);
        current = current->next;
    }
}

```

A singly linked list can be searched for an element by traversing the list and comparing each node's data with the desired value. If a match is made, the relevant node is returned; if not, NULL is returned to show that the element is not present in the list.

```

struct Node* search(int value) {
    struct Node* current = head;
    while (current != NULL) {
        if (current->data == value) {
            return current;
        }
        current = current->next;
    }
    return NULL;
}

```

To remove a component from a singly linked list, locate the node that contains the desired value and modify the links to the preceding and following nodes accordingly. Additionally, the memory that the destroyed node used must be freed.

```
void delete(int value) {  
    if (head == NULL) {  
        printf("List is empty.");  
        return;  
    }  
  
    struct Node* current = head;  
  
    struct Node* previous = NULL;  
  
    while (current != NULL) {  
        if (current->data == value) {  
            if (previous == NULL) {  
                head = current->next;  
            } else {  
                previous->next = current->next;  
            }  
            free(current);  
            return;  
        }  
        previous = current;  
        current = current->next;  
    }  
  
    printf("Element not found in the list.");  
}
```

Operation Program for Single Linked List

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
    int data;
    struct node *link;
};
struct node *root=NULL;
int len;
void append(void);
void addatbegin(void);
void addatafter(void);
int length(void);
void display(void);
void delete(void);
void main()
{
    int ch;
    while(1)
    {
        printf("Single linked list Operations:\n");
        printf("1. Append\n");
        printf("2. Add at Begin\n");
        printf("3. Add at After\n");
        printf("4. Length\n");
        printf("5. Display\n");
        printf("6. Delete\n");
        printf("7. Exit");

        printf("Enter your Choice: \n");
        scanf("%d",&ch);
    }
}
```

```

switch(ch)
{
    case 1: append();
              break;
    case 2: addatbegin();
              break;
    case 3: addatafter();
              break;
    case 4: len=length();
              printf("length: %d\n\n",len);
              break;
    case 5: display();
              break;
    case 6: delete();
              break;
    case 7: exit(1);
    default: printf("Invalid choice \n");
}
}

}

//Create a new node
void append()
{
    struct node *temp;
    temp=(struct node*) malloc(sizeof (struct node));
    printf("Enter any value");
    scanf("%d",&temp->data);
    temp->link=NULL;
    if(root==NULL)
    {
        root=temp;
    }
}

```

```

else          //add newly create node at the end
{
  struct node *p;
  p=root;
  while(p->link!=NULL)
  {
    p=p->link;
  }
  p->link=temp;
}

//single linked list length find
int length( ) //local variable

{
  int count=0;
  struct node* temp;
  temp=root;
  while(temp!=NULL)
  {
    count++;
    temp=temp->link;
  }
  return count;
}

//add node at the first location
void addatbegin( )

{
  struct node *temp;
  temp=(struct node*) malloc(sizeof (struct node));
  printf("Enter any value");
  scanf("%d",&temp->data);
}

```

```
temp->link=NULL;
if(root==NULL)
{
    root=temp;
}
else
{
    temp->link=root; //right connection first and then left connection
    root=temp; //left side connection
}
void adddataafter()
{
    struct node *temp,*p;
    int loc,len,i=1;
    printf("Enter any Location");
    scanf("%d",&loc);
    len=length();
    if(loc>len)
    {
        printf("Invalid Location\n");
        printf("Currently list is having %d nodes only\n",len);
    }
    else
    {
        p=root;
        while(i<loc)
        {
            p=p->link;
            i++;
        }
```

```

    }

    temp=(struct node *)malloc(sizeof(struct node));
    printf("Enter any value\n");
    scanf("%d",&temp->data);

    temp->link=NULL;
    temp->link=p->link;//right side connection
    p->link=temp;//left side connection
}

}

void display()

{
    struct node *temp;
    temp=root;
    if(temp==NULL)
    {
        printf("List is Empty\n");
    }
    else
    {
        while(temp!=NULL)
        {
            printf("%d-->",temp->data);
            temp=temp->data;
        }
        printf("\n \n");
    }
}

void delete()

{
    struct node *temp;
    int loc;
}

```

```
printf("Enter Location to be deleted");
scanf("%d",&loc);
if(loc>length())
{
printf("Invalid location\n");
}
else if(loc==1)
{
temp=root;
root=temp->link;//remove left connection first next right connection
temp->link=NULL;
free(temp);
}
else
{
struct node *p=root, *q;
int i=1;
while(i<loc-1)
{
p=p->link;
i++;
}
q=p->link;
p->link=q->link;
q->link=NULL;
free(q);
}
```

2.1.2.1 2 Doubly Linked List

A doubly linked list (DLL) is a particular type of linked list in which each node contains a pointer to the previous node and the next node of the linked list.

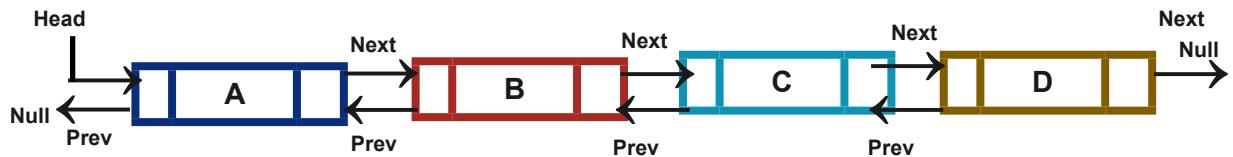


Fig. 2: Doubly linked list

```

// Node of a doubly linked list
class Node {
public:
    int data;

    // Pointer to next node in DLL
    Node* next;

    // Pointer to the previous node in DLL
    Node* prev;
};
  
```

Advantages of Doubly Linked List over the singly linked list:

- A DLL can be traversed in both forward and backward directions.
- The delete operation in DLL is more efficient if a pointer to the node to be deleted is given.
- We can quickly insert a new node before a given node.
- In a singly linked list, a pointer to the previous node is needed to delete a node. Sometimes, the list is traversed to get this previous node. In DLL, we can get the previous node using the previous pointer.

Disadvantages of Doubly Linked List over the singly linked list:

- Every node of DLL Requires extra space for a previous pointer. However, it is possible to implement DLL with a single pointer.
- All operations require an extra pointer before being maintained. For example, in insertion, we need to modify previous pointers together with the next pointers. For example, we need 1 or 2 extra steps to set the previous pointer in the following functions for insertions at different positions.

Applications of Doubly Linked List:

- Web browsers use it for backward and forward navigation of web pages
- LRU (Least Recently Used)/MRU (Most Recently Used) Cache are constructed using Doubly Linked Lists.
- Used by various applications to maintain undo and redo functionalities.
- In Operating Systems, a doubly linked list is maintained by a thread scheduler to keep track of processes being executed at that time.

2.1.2.3 Circular Linked List

A circular linked list is one in which all nodes are connected to form a circle. The first and last nodes are connected in a circular linked list, forming a circle. There is no NULL at the end.

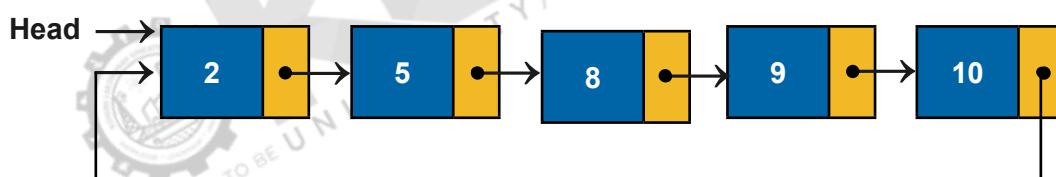


Fig. 3: Circular linked list

There are generally two types of circular linked lists:

Circular singly linked list: In a circular singly linked list, the last node of the list contains a pointer to the first node of the list. We traverse the circular singly linked list until we reach the same node where we started. The circular singly linked list has no beginning or end. No null value is present in the next part of any of the nodes.

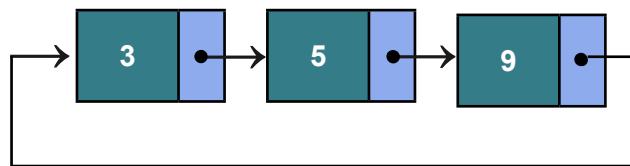


Fig. 4: Circular singly linked list

Circular Doubly Linked List: A circular doubly linked list has the properties of both a doubly linked list and a circular linked list. In this list, the previous and next pointers link or connect two consecutive elements. The last node points to the first node by the next pointer, and the first node points to the last node by the previous pointer.

Representation of circular doubly linked list

Note: We will use the singly circular linked list to represent the workings of the circular linked list.

Representation of circular linked list:

Circular linked lists are like single-linked lists except for connecting the last node to the first node.

Node representation of a Circular Linked List:

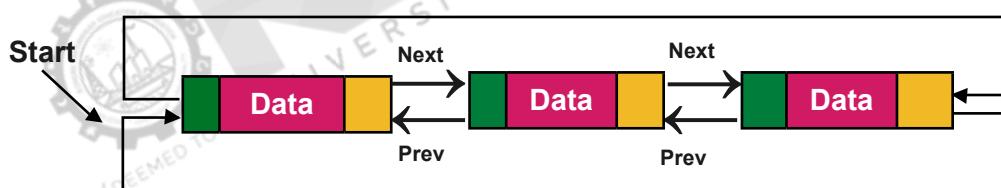


Fig. 5: Representation of a circular doubly linked list

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

Circular singly linked list:

Example of circular linked list

The above Circular singly linked list can be represented as:

```
Node* one = createNode(3);
```

```
Node* two = createNode(5);
```

```
Node* three = createNode(9);
```

```
// Connect nodes
```

```
one->next = two;
```

```
two->next = three;
```

```
three->next = one;
```

Explanation: In the above program, one, two, and three are the nodes with values 3, 5, and 9 respectively, which are connected circularly as:

- **For Node One:** The Next pointer stores the address of Node Two.

For Node Two: The Next stores the address of Node three

- **For Node Three:** The Next points to Node One.

Operations on the circular linked list:

We can do some operations on the circular linked list like the singly linked list, which is:

- Insertion
- Deletion

1. Insertion in the circular linked list:

A node can be added in three ways:

- Insertion at the beginning of the list
- Insertion at the end of the list
- Insertion in between the nodes

Insertion at the beginning of the list: To insert a node at the beginning of the list, follow these steps:

Create a node, say T.

Make $T \rightarrow \text{next} = \text{last} \rightarrow \text{next}$.

$\text{last} \rightarrow \text{next} = T$.

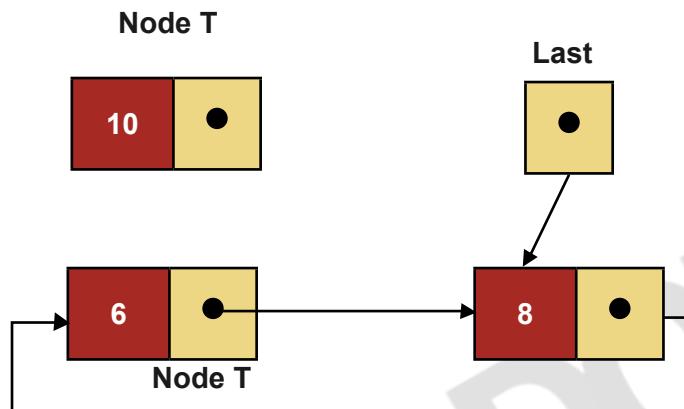


Fig. 6: Circular linked list before insertion

And then

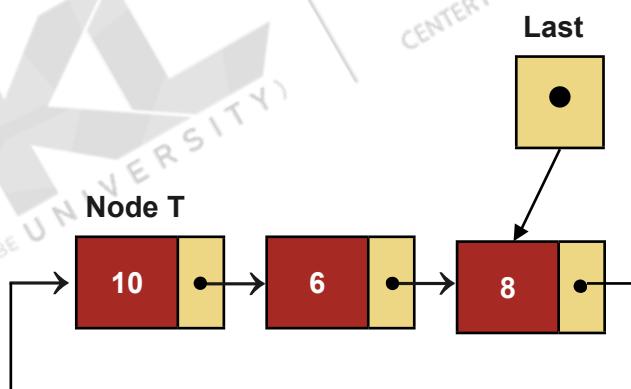


Fig. 7: Circular linked list after insertion

Insertion at the end of the list: To insert a node at the end of the list, follow these steps:

Create a node, say T.
Make $T \rightarrow \text{next} = \text{last} \rightarrow \text{next};$
 $\text{last} \rightarrow \text{next} = T.$
 $\text{last} = T.$

Circular linked list after insertion of node at the end

Insertion in between the nodes: To insert a node in between the two nodes, follow these steps:

Create a node, say T.
Search for the node after which T needs to be inserted, and say that the node is P.
Make $T \rightarrow \text{next} = P \rightarrow \text{next};$
 $P \rightarrow \text{next} = T.$
Suppose 12 needs to be inserted after the node has the value 10,

2. Deletion in a circular linked list:

a) Delete the node only if it is the only node in the circular linked list:

- Free the node's memory
- The last value should be NULL. A node always points to another node, so a NULL assignment is unnecessary.
- Any node can be set as the starting point.
- Nodes are traversed quickly from the first to the last.

b) Deletion of the last node:

- Locate the node before the last node (let it be temp)
- Keep the address of the node next to the last node in temp
- Delete the last memory
- Put temp at the end

c) Delete any node from the circular linked list:

We will be given a node, and our task is to delete it from the circular linked list.

Algorithm:

Case 1: The list is empty.

If the list is empty, we will simply return.

Case 2: The list is not empty

If the list is not empty, we define two pointers, curr and prev, and initialise the pointer curr with the head node.

```
#include <stdio.h>
#include <stdlib.h>

// Structure for a node
struct Node {
    int data;
    struct Node* next;
};

// Function to insert a node at the
// beginning of a Circular linked list
void push(struct Node** head_ref, int data)
{
    // Create a new node and make head
    // as next of it.
    struct Node* ptr1 = (struct Node*)malloc(sizeof(struct Node));
    ptr1->data = data;
    ptr1->next = *head_ref;

    // If linked list is not NULL then
    // set the next of last node
    if (*head_ref != NULL) {

        // Find the node before head and
        // update next of it.
        struct Node* temp = *head_ref;
```

```
while (temp->next != *head_ref)
    temp = temp->next;
    temp->next = ptr1;
}

else

    // For the first node
    ptr1->next = ptr1;

    *head_ref = ptr1;
}

// Function to print nodes in a given
// circular linked list
void printList(struct Node* head)
{
    struct Node* temp = head;
    if (head != NULL) {
        do {
printf("%d ", temp->data);
            temp = temp->next;
        } while (temp != head);
    }

    printf("\n");
}

// Function to delete a given node
// from the list
void deleteNode(struct Node** head, int key)
{
    // If linked list is empty
    if (*head == NULL)
        return;
```

```

// If the list contains only a
// single node
if ((*head)->data == key && (*head)->next == *head) {
    free(*head);
    *head = NULL;
    return;
}

struct Node *last = *head, *d;

// If head is to be deleted
if ((*head)->data == key) {

    // Find the last node of the list
    while (last->next != *head)
        last = last->next;

    // Point the last node to the next of
    // head, i.e. the second node
    // of the list
    last->next = (*head)->next;
    free(*head);
    *head = last->next;
}

return;
}

// Either the node to be deleted is
// not found or the end of list
// is not reached
while (last->next != *head && last->next->data != key) {
    last = last->next;
}

// If node to be deleted was found
if (last->next->data == key) {

```

```
d = last->next;  
last->next = d->next;  
free(d);  
}  
else  
printf("Given node is not found in the list!!!\n");  
}  
  
// Driver code  
int main()  
  
{  
// Initialise lists as empty  
struct Node* head = NULL;  
  
// Created linked list will be  
// 2->5->7->8->10  
push(&head, 2);  
push(&head, 5);  
push(&head, 7);  
push(&head, 8);  
push(&head, 10);  
  
printf("List Before Deletion: ");  
printList(head);  
  
deleteNode(&head, 7);  
  
printf("List After Deletion: ");  
printList(head);  
  
return 0;  
}
```

Output

- **List Before Deletion:** 10 8 7 5 2
- **List After Deletion:** 10 8 5 2
- **Time Complexity:** O(N). The worst case occurs when the element to be deleted is the last, and we need to move through the whole list.
- **Auxiliary Space:** O(1), As constant extra space is used.





Self-Assessment Questions

6. A _____ linked list consists of nodes where each node points to the next node and the last node points to NULL.
- a) Doubly
 - b) Singly
 - c) Circular
 - d) Triple
7. In a _____ linked list, each node contains references to both the next and previous nodes.
- a) Singly
 - b) Circular
 - c) Doubly
 - d) Linear
8. A singly linked list can be traversed in both forward and backward directions.
- a) True
 - b) False
9. Circular linked lists can be either singly or doubly linked.
- a) True
 - b) False
10. The primary operations of linked lists include insertion, deletion, traversal, and _____.
- a) Sorting
 - b) Merging
 - c) Splitting
 - d) Searching



Summary

- A linked list is a linear data structure where elements are not stored in contiguous memory locations.
- Instead, each component, called a node, contains data and a pointer to the next node in the sequence.
- Linked lists can grow or shrink dynamically, making them more memory-efficient and adaptable than arrays.
- The primary advantage is that node insertion and deletion are easier, requiring only pointer adjustments.
- Linked lists are fundamental to implementing other data structures like stacks, queues, and graphs.
- There are various linked lists, including singly linked lists, doubly linked lists, and circular linked lists.
- In a singly linked list, each node points to the next node and the last node points to NULL. This linear structure supports insertion, deletion, traversal, and searching operations.
- On the other hand, a doubly linked list contains nodes referencing the next and the previous nodes, enabling traversal in both directions.
- It offers more efficient deletion and insertion operations but requires extra memory for the additional pointer.
- Circular linked lists form a loop where the last node returns to the first node, creating a continuous structure without a NULL endpoint.



Terminal Questions

1. What is a linked list, and how is it structured?
2. What are the differences between singly and doubly linked lists?
3. Describe the basic operations performed on linked lists, such as traversal, insertion, and deletion.
4. Explain the structure of a singly linked list and its advantages over arrays.
5. Describe the characteristics of circular linked lists and their potential applications.
6. Discuss the operations commonly performed on linked lists, including insertion and deletion.



Answer Keys

Self-Assessment Questions	
Question No.	Answer
1	A
2	B
3	C
4	B
5	A
6	B
7	C
8	B
9	A
10	D



Activity

Activity Type: Offline

Duration: 1 hour

Examine the implementation of a doubly linked list in a programming language of your choice. How would you modify the insert and delete operations to handle edge cases, such as inserting/deleting at the beginning or end of the list? Consider the efficiency and complexity of your modifications.



Glossary

- **Node:** A fundamental unit of a linked list that holds data and a pointer to the next node.
- **Dynamic Memory Allocation:** The process of allocating memory at run-time, allowing a linked list to grow and shrink as needed.
- **Data Field:** The data field in a linked list or data structure refers to the part of the node that holds the actual data or information.
- **Pointer Field:** The pointer field in a linked list is the part of a node that contains a reference or pointer to the next node in the sequence.
- **Operating Systems:** An operating system (OS) manages a computer's hardware and software resources.



Bibliography

Textbooks

- Balagurusamy, E. (2017). *Data Structures*. McGraw Hill Education.
- Sedgewick, R., & Wayne, K. (2014). *Algorithms II*. Pearson Education.
- Sridhar, S. (2014). *Design and Analysis of Algorithms*. Oxford University Press.
- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (1989). *Introduction to Algorithms*. Prentice-Hall.



e-References

- **Linked List:** <https://www.javatpoint.com/singly-linked-list>
- **Types of Linked List:** <https://www.javatpoint.com/ds-types-of-linked-list>
- **DSA Linked List Types:** https://www.w3schools.com/dsa/dsa_data_linkedlists_types.php



Video Links

Video	Links
Introduction to Linked List	https://www.youtube.com/watch?v=R9PTBwOzceo&list=PLBlnK6fEyqRi3IvwLGzcaquOs5OBTCww
Creating the Node of a Single Linked List	https://www.youtube.com/watch?v=DneLxrPmmsw
Introduction to Doubly Linked List	https://www.youtube.com/watch?v=e9NG_a6Z0mg
Circular Doubly Linked List	https://www.youtube.com/watch?v=3ZrkixbHCTI



Image Credits

- **Fig. 1:** Self-Made
- **Fig. 2:** Self-Made
- **Fig. 3:** Self-Made
- **Fig. 4:** Self-Made
- **Fig. 5:** Self-Made
- **Fig. 6:** Self-Made
- **Fig. 7:** Self-Made



Keywords

- Linked List
- Singly Linked List
- Doubly Linked List:
- Circular Linked List
- Pointer
- Traversal
- Insertion

MODULE 2

Linked Lists, Stacks, and Queue

Unit 2

Outline of Stack



■ Unit Table of Contents

Unit 2.2 Outline of Stack

Aim	147
Instructional Objectives	147
Learning Outcomes	147
2.2.1 Stack	148
2.2.1.1 Definition and Introduction to Stack	148
2.2.1.2 Stack Implementation	149
2.2.1.3 Operations of Stack	156
Self-Assessment Questions	160
2.2.1.4 Applications of Stack and Multiple Stacks	161
2.2.1.5 Implementation of Multiple Stack Queues	164
Self-Assessment Questions	169
Summary	170
Terminal Questions	170
Answer Keys	171
Activity	171
Glossary	172
Bibliography	172
e-References	172
Video Links	173
Image Credits	173
Keywords	173



Aim

To provide students with a comprehensive interpretation of the stack's definition, introduction, implementation, operations, and applications.



Instructional Objectives

This unit is designed to:

- Define the Stack and explain its Last In First Out (LIFO) principle
- Explain the differences between static and dynamic stack implementations
- Discuss the conditions of stack overflow and underflow



Learning Outcomes

At the end of the unit, the student is expected to:

- Assess the fundamental operations of a Stack: push, pop, peek, and isEmpty
- Evaluate how Stacks are used in converting expressions from infix to postfix and vice versa
- Illustrate the implementation of multiple stacks and queues using a single array

2.2.1 Stack

2.2.1.1 Definition and Introduction to Stack

The stack data structure is a linear data structure that accompanies a principle known as LIFO (Last In First Out) or FILO (First In Last Out). Real-life examples of a stack are a deck of cards, piles of books, piles of money, and many more.

Stack is a linear data structure. It is an ADT (Abstract data type) that contains Homogeneous elements. Stack always follows the “LIFO manner,” i.e., “Last In First Out.”

In the Stack, the recently added element will be accessed first. Following is the diagram of Stack.

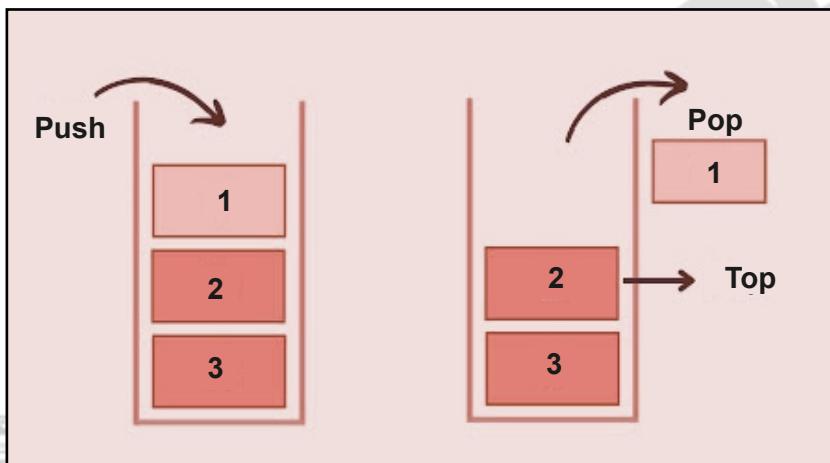


Fig. 1: Stack Data Structure

We can add and remove elements from the stack. Adding elements to the stack is called “PUSH,” and removing them from the stack is called “POP.” A stack is a single-end data structure where push and pop operations can be performed on a single end, known as the “TOP” end.

Benefits of Stacks

- Stacks are simple and easy to interpret, making them ideal for many applications.
- Push and pop operations are highly efficient, operating constantly ($O(1)$).
- The stack follows the Last-in, First-out (LIFO) principle, which is useful for tasks like function call management and expression evaluation.
- Memory usage is minimal since a stack only stores the elements that are pushed onto it.

Drawbacks of Stacks

- Access is limited since only the top element can be retrieved, making accessing or modifying elements in the middle difficult.
- When the stack exceeds its capacity, an overflow may lead to data loss.
- Stacks do not support random access to elements, making them unsuitable for scenarios requiring specific order access.
- Fixed capacity can be restrictive when the number of elements to be stored is unpredictable or variable.

Applications of Stack

- Used for converting infix expressions to postfixes or prefixes.
- Supports redo and undo features in editors and graphic software like Photoshop.
- Enables forward and backward navigation in web browsers.
- Essential for memory management in modern computers, as each running program has its own stack for memory allocation.
- Facilitates computer function calls, ensuring the most recent function is completed first.

2.2.1.2 Implementation of Stacks

A Stack can be implemented in many ways, such as using structures, pointers, arrays, linked lists, etc., but it can mainly be implemented in two ways.

1. Using arrays
2. Using LinkedList

1. Stack using Arrays:

We can implement Stacks using arrays, but it works only for a fixed number of data values. That means the amount of data must be specified at the beginning of the implementations.

```
#include<stdio.h>
#include<conio.h>
#define CAPACITY 5
int stack[CAPACITY];
int top=-1;
void push(void);
void pop(void);
void display(void);
```

```
int main()
{
    int value, choice,n;
    clrscr();
    printf("\n Enter the size of STACK[MAX=100]:");
    scanf("%d",&n);
    printf("\n\t STACK OPERATIONS USING ARRAY");
    printf("\n\t-----");
    printf("\n\t 1.PUSH\n\t 2.POP\n\t 3.DISPLAY\n\t 4.EXIT");
    do
    {
        printf("\n Enter the Choice:");
        scanf("%d",&choice);
        switch(choice)
        {
            case 1:
            {
                push();
                break;
            }
            case 2:
            {
                pop();
                break;
            }
            case 3:
            {
                display();
                break;
            }
            case 4:
            {
                printf("\n\t EXIT POINT ");
            }
        }
    }
}
```

```

break;
}
default:
{
printf ("\n\t Please Enter a Valid Choice(1/2/3/4)");
}

while(choice!=4);
return 0;
}

void push()
{
if(top==CAPACITY-1)
{
printf("\n\tSTACK is over flow");
}

else
{
printf(" Enter a value to be pushed.");
scanf("%d",&x);
top++;
stack[top]=x;
printf("Element Pushed");
}
}

void pop()
{
if(top== -1)
{
printf("\n\t Stack is under flow");
}
}

```

```

    }
  else
  {
    printf("\n\t The popped elements is %d",stack[top]);
    top--;
  }
}

void display()
{
  if(top>=0)
  {
    printf("\n The elements in STACK \n");
    for(i=top; i>=0; i--)
      printf("\n%d",stack[i]);
    printf("\n Press Next Choice");
  }
  else
  {
    printf("\n The STACK is empty");
  }
}

```

2. Stack Using Linked List:

The major problem with the Stack implemented using an array is it works only with a fixed number of data values, which means the amount of data must be specified at the beginning of the implementation. A stack implemented using an array is unsuitable when we don't know the size of the data. A Stack data structure can be implemented by using a linkedlist data structure. The Stack implemented using a linked list works for an unlimited number of values, which means the Stack implemented using a linked list works for the variable sizes of data. So, there is no need to fix the size at the beginning of the implementation.

In the linkedlist implementation of Stack, every new element is inserted as a “top” element. That means every newly inserted element is pointed by “top”. Whenever we want to remove an element from the stack, simply remove the node pointed by “Top” by moving the top to its next node on the list. The next field of the first element must always be “NULL”.

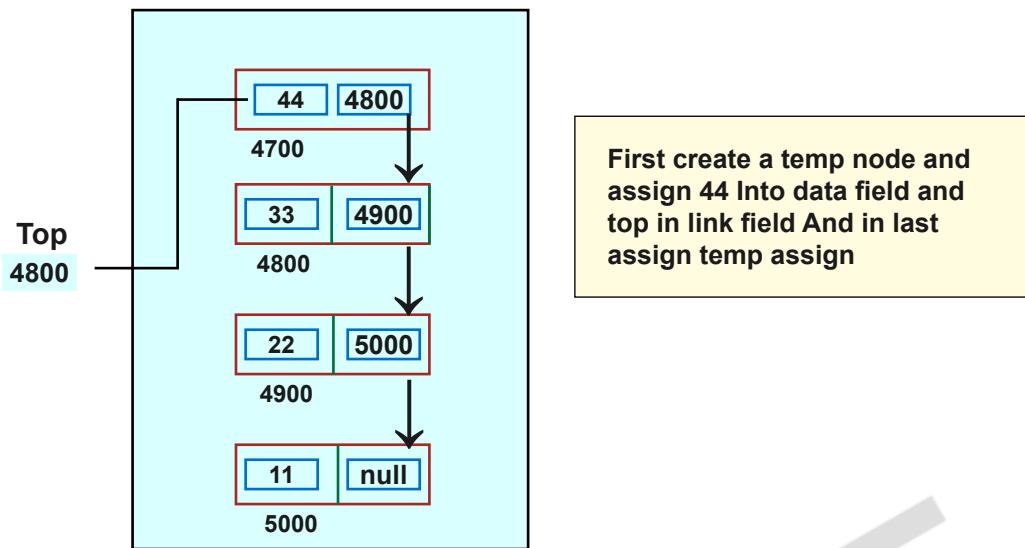


Fig. 2: Stack implementation using a linked list

Implement a stack using singly linked list

```
#include <stdio.h>
#include <stdlib.h>
#include<conio.h>
struct Node
{
    int data;
    struct Node *next;
};
struct Node *top=NULL;

void push(int value)
{
    struct Node *newnode;
    newnode = (struct Node *)malloc(sizeof(struct Node));
    newnode->data = value;
    if (top == NULL)
        top = newnode;
    else
        newnode->next = top;
    top = newnode;
}
```

```
{  
    newnode->next = NULL;  
}  
else  
{  
    newnode->next = top;  
}  
top = newnode;  
printf("Node is Inserted\n\n");  
}  
  
int pop()  
{  
    if (top == NULL)  
    {  
        printf("\nStack Underflow\n");  
    }  
    else  
    {  
        struct Node *temp = top;  
        printf("\n Deleted element is:%d",temp->data);  
        top = top->next;  
        free(temp);  
        return temp->data;  
    }  
}  
  
void display()  
{  
    if (top == NULL)  
    {  
        printf("\nStack Underflow\n");  
    }
```

```

    }
else
{
  struct Node *temp = top;
  while (temp->next != NULL)
  {
    printf("%d--->", temp->data);
    temp = temp->next;
  }
  printf("%d--->NULL\n\n", temp->data);
}

int main()
{
  int choice, value;
  clrscr();
  printf("\nImplementation of Stack using Linked List\n");
  while (1)
  {
    printf("1. Push\n2. Pop\n3. Display\n4. Exit\n");
    printf("\nEnter your choice : ");
    scanf("%d", &choice);
    switch (choice)
    {
      case 1:
        printf("\nEnter the value to insert: ");
        scanf("%d", &value);
        push(value);
        break;
      case 2:
        printf("Popped element is :%d\n", pop());
    }
  }
}

```

```
break;  
    case 3:  
        display();  
        break;  
    case 4:  
        exit(0);  
        break;  
    default:  
        printf("\nWrong Choice\n");  
    }  
}  
}  
}
```

2.1.3 Operations of Stack

Following are the operations of the Stack:

1. Creation
2. Insertion
3. Deletion
4. Traversal

These operations can be achieved by using the following function:

- Push()
- Pop()
- isFull()
- isEmpty()
- create()
- peek()
- travers()

- **push:** Pushing elements into the Stack.
- **pop:** It is a process of deleting elements from the top of the Stack.
- **isFull:** This function checks whether the Stack is full. If it is, it returns a true value (1); otherwise, it returns false.
- **isEmpty:** This function checks whether the Stack is empty. If empty, it returns true; otherwise, it returns false.
- **create:** It helps create an empty stack.
- **peek:** It is useful to get the top of the element from the Stack while deleting it.
- **size:** It returns the number of elements in the Stack.

Stack Overflow

When the number of elements exceeds the size of the stack, it is said to be a stack overflow. That means when a user tries to insert more elements even if the Stack is full, the situation is called Stack overflow.

Stack Underflow

When the Stack is empty and the user is still trying to delete it, the element (pop) is called a Stack underflow.

Push Operation in Stack Data Structure:

This operation inserts an item into the stack. If the stack has reached its maximum capacity, an overflow condition will result, preventing further insertion.

Algorithm for Push Operation:

1. First, verify if the stack is full.
2. If the stack is full (i.e., $\text{top} == \text{capacity} - 1$), the operation results in a Stack Overflow and no element can be added.
3. If the stack is not full, increase the top index by 1 ($\text{top} = \text{top} + 1$), and insert the new element at the updated top position.
4. Continue adding elements until the stack reaches its capacity.

Pop Operation in Stack Data Structure:

Following the Last In First Out (LIFO) principle, this operation removes an item from the stack. If the stack is empty, an Underflow condition occurs, meaning no element can be removed.

Algorithm for Pop Operation:

1. Before removing an element, check if the stack is empty.
2. If the stack is empty (i.e., $\text{top} == -1$), a Stack Underflow occurs, and no element can be removed.
3. Otherwise, store the element at the top index, reduce the top index by 1 ($\text{top} = \text{top} - 1$), and return the stored value.

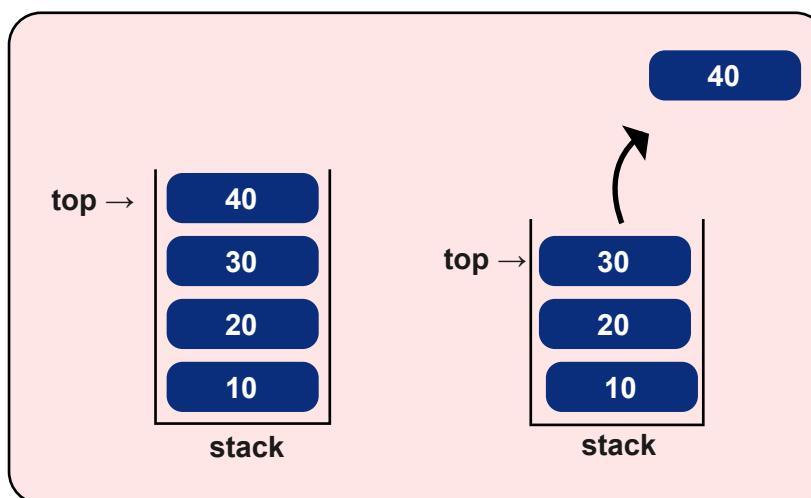


Fig. 3: POP operation in stack

Top/Peak Operation in Stack Data Structure:

This operation retrieves the element at the top of the stack without removing it.

Algorithm for Top/Peak Operation:

1. First, check if the stack is empty.
2. If the stack is empty (i.e., $\text{top} == -1$), print a message like Stack is empty.
3. Otherwise, return the element located at the current top index.

isEmpty Operation in Stack Data Structure:

This operation checks whether the stack is empty.

Algorithm for isEmpty Operation:

1. Inspect the value of the top.
2. If $\text{top} == -1$, return true, indicating the empty stack.
3. Otherwise, return false, as the stack still contains elements.

isFull Operation in Stack Data Structure:

This operation checks whether the stack has reached its maximum capacity.

Algorithm for isFull Operation:

1. Inspect the value of the top.
2. If top == capacity -1, return true, indicating the full stack.
3. Otherwise, return false, meaning the stack has space for more elements.





Self-Assessment Questions

1. What does the stack data structure follow the main principle?
 - a) First In First Out
 - b) First In Last Out
 - c) Last In First Out
 - d) Random Access

2. The push operation in a stack removes an element from the top.
 - a) True
 - b) False

3. A stack implemented with a linked list has no fixed size limit.
 - a) True
 - b) False

4. The _____ operation retrieves the top element without removing it.
 - a) pop
 - b) push
 - c) peek
 - d) dequeue

5. A stack implemented using _____ has a dynamic size.
 - a) Arrays
 - b) Linked list
 - c) Queues
 - d) Binary tree

2.2.1.4 Applications of Stack

Because a Stack is a restricted data structure, only limited operations will be performed. The elements are deleted from the Stack in the reverse order. The following are the applications of the Stack.

Expression evaluation

Stack is used to evaluate prefix, infix, and postfix expressions. Prefix notation or expression is nothing but an expression that contains the operator in front of the operands.

Eg: - +ab

Infix notation is a general expression mostly used by humans. In this notation, an operator is placed between the two operands.

Eg: - a+b

Postfix notation is when the operator will be placed after the operands.

Eg: - ab+

Conversion of expression:

The Stacks can be used to convert expressions from one form to another, such as infix to prefix, infix to postfix, etc.

Backtracking:

Suppose we are finding a path to solve a problem. We use a path, and after following it, we realise it is wrong. Now, we need to go back to the beginning of the path to start with a new path. This can be done with the help of Stack.

Parenthesis checking:

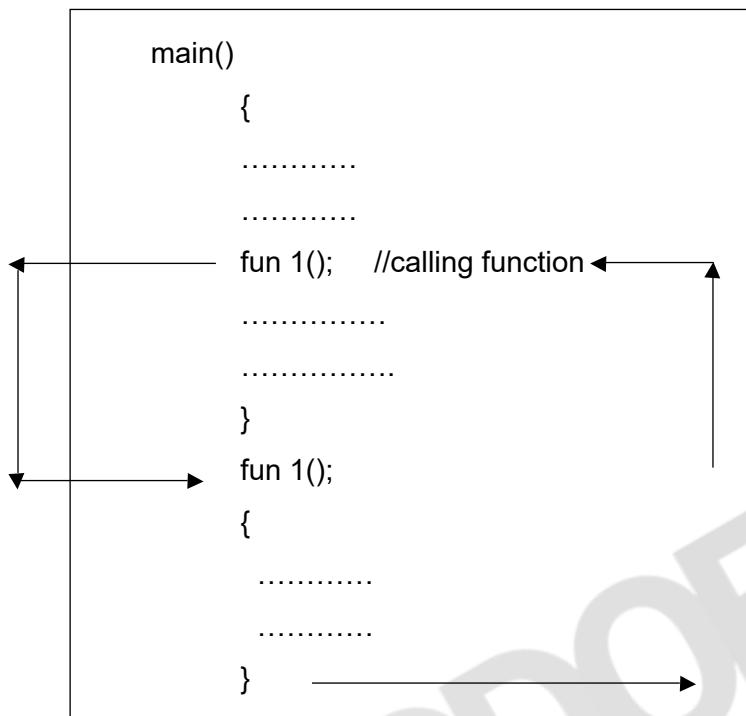
A stack is used to check the proper opening and closing of parenthesis.

String Rever

sal:

Stack is used to keep the information about the active functions and sub-routines.

When a function is called the control shifts to the function (called function) and executes all the statements there but to return to the point where it is called a Stack is used to remember the address of the calling statement.



Stacks are used to solve the towers of the Hanoi program.

Converting the infix expression to postfix expression:

We cannot convert the infix expression into a postfix expression.

Following is the algorithm:

1. Scan the characters from left to right.
2. when the left parenthesis (is encountered, it is pushed onto the Stack).
3. If an operand is encountered, send it directly to the output.
4. If the right parenthesis(is encountered, pop all the elements from the Stack and send it to the output excluding parenthesis.
5. If an operator is encountered, push it on the Stack.
6. When an operator is encountered, and the operator at the top of the Stack has lower priority than the operator already in the Stack, then pop out the previous operator and send it to the output.
7. Now, place the lower priority in the Stack.
8. Repeat the above process until the end of the expression.
9. When u reach the end, pop all the elements and print them.

The process of converting the expression $(a+b)^*c+d^*e$ into post-fix notation follows.

Symbol	Stack elements	Output
((-
A	(a
+	(+	a
B	(+	ab
)	Empty	ab+
*	*	ab+
C	*	ab+c
+	+	ab+c*
D	+	ab+c*d
*	**	ab+c*d
E	**	ab+c+de
End	Empty	ab+c*de*+

Evolution of postfix notation:

1. We can evaluate the postfix expression using the Stack mentioned in the above algorithm.
2. The expression can be from left to right. If you find a value, then immediately send it to the stack.
3. If an operator is found, pop two values from the Stack, operate the expression and then push the result on the Stack.
4. Repeat the above process (and) until the expression ends.
5. Finally, pop the results from the Stack and print.

Consider the postfix expression $23+4*56*+$. The evaluation of the expression is shown in the below table.

Symbol	Evaluation	Stack elements
2	-	2
3	-	2,3
+	$2+3$	5
4	-	5,4
*	$5*4$	20
5	-	20,5
6	-	20,5,6
*	$5*6$	20,30
+	$20+30$	50(Output)

2.2.1.5 Implementation of Multiple Stack Queues

To implement Multiple Stacks and Queues in C, you typically manage more than one stack or queue within a single array. Let's break this down and first look at the implementation separately for multiple stacks and queues.

Multiple Stacks

You must maintain separate top pointers to manage multiple stacks in a single array.

Example: Implementing Two Stacks in One Array

```
#include <stdio.h>
#include <stdlib.h>

#define MAX 10 // Size of the array

int stack[MAX];
int top1 = -1;
int top2 = MAX;
```

```

// Push an element into stack 1
void push1(int value) {
    if (top1 < top2 - 1) {
        stack[++top1] = value;
    } else {
        printf("Stack Overflow\n");
    }
}

// Push an element into stack 2
void push2(int value) {
    if (top1 < top2 - 1) {
        stack[--top2] = value;
    } else {
        printf("Stack Overflow\n");
    }
}

// Pop an element from stack 1
int pop1() {
    if (top1 >= 0) {
        return stack[top1--];
    } else {
        printf("Stack Underflow\n");
        return -1;
    }
}

// Pop an element from stack 2
int pop2() {
    if (top2 < MAX) {
        return stack[top2++];
    }
}

```

```

} else {
printf("Stack Underflow\n");
return -1;
}

int main() {
push1(10);
push1(20);
push2(30);
push2(40);

printf("Popped from stack 1: %d\n", pop1());
printf("Popped from stack 2: %d\n", pop2());

return 0;
}

```

Multiple Queues

You'll maintain separate front and rear indices for multiple queues in a single array.

Example: Implementing Two Queues in One Array

```

#include <stdio.h>
#include <stdlib.h>

#define MAX 10 // Size of the array

int queue[MAX];
int front1 = -1, rear1 = -1; // For Queue 1
int front2 = MAX, rear2 = MAX; // For Queue 2

// Enqueue an element into Queue 1
void enqueue1(int value) {
if (rear1 < front2 - 1) {
if (front1 == -1) {
front1 = 0;
}
}

```

```

    queue[++rear1] = value;
} else {
printf("Queue Overflow\n");
}
}

// Enqueue an element into Queue 2
void enqueue2(int value) {
if (rear1 < front2 - 1) {
if (front2 == MAX) {
front2 = MAX - 1;
}
queue[--rear2] = value;
} else {
printf("Queue Overflow\n");
}
}

// Dequeue an element from Queue 1
int dequeue1() {
if (front1 <= rear1 && front1 != -1) {
int value = queue[front1];
if (front1 == rear1) {
front1 = rear1 = -1; // Reset if the queue is empty
} else {
front1++;
}
return value;
} else {
printf("Queue Underflow\n");
return -1;
}
}

```

```

// Dequeue an element from Queue 2
int dequeue2() {
    if (rear2 <= front2 && front2 != MAX) {
        int value = queue[front2];
        if (front2 == rear2) {
            front2 = rear2 = MAX; // Reset if the queue is empty
        } else {
            front2--;
        }
        return value;
    } else {
        printf("Queue Underflow\n");
        return -1;
    }
}

int main() {
    enqueue1(10);
    enqueue1(20);
    enqueue2(30);
    enqueue2(40);

    printf("Dequeued from queue 1: %d\n", dequeue1());
    printf("Dequeued from queue 2: %d\n", dequeue2());

    return 0;
}

```



Explanation:

- **Multiple Stacks:** We use an array of size MAX and two pointers (top1 for the first stack and top2 for the second stack) to manage two stacks within the same array. The push1 and push2 functions add elements to stacks 1 and 2, respectively, while pop1 and pop2 remove elements.
- **Multiple Queues:** Similarly, for queues, we maintain front1, rear1 for the first queue and front2, rear2 for the second queue. The enqueue and dequeue functions manage the respective queues.



Self-Assessment Questions

6. Which of the following expressions is in postfix notation?

- a) $ab+$
- b) $+ab$
- c) $a+b$
- d) $ab-$

7. Which data structure is used to manage function calls in programming?

- a) Queue
- b) Linked List
- c) Stack
- d) Tree

8. Postfix notation places the operator after the operands.

- a) True
- b) False

9. Multiple stacks can be implemented in a single array using two front pointers.

- a) True
- b) False

10. In C programming, two stacks can be managed in one array using two _____ pointers.

- a) Front
- b) Rear
- c) Head
- d) Top



Summary

- The stack is a linear data structure that follows the Last In, First Out (LIFO) principle.
- Elements are added and removed from the top of the stack, with two primary operations: push (to add an element) and pop (to remove the top element).
- The stack can be implemented using arrays or linked lists. It has a fixed size when implemented using arrays, but a linked list allows dynamic resizing.
- The stack is widely used in tasks such as expression conversion, memory management, function call handling, and undo-redo operations. Basic stack operations include creation, insertion, deletion, and traversal.
- A stack is a restricted data structure in which elements are added and removed in a Last-in-First-out (LIFO) order.
- Stacks are useful for many applications, including expression evaluation (e.g., prefix, infix, and postfix), conversion of expressions (infix to prefix or postfix), backtracking in algorithms, parenthesis checking, and string reversal.
- Another key operation is converting infix expressions to postfix using a specific algorithm that processes the expression character by character, handling operators and operands.
- Evaluating postfix expressions involves pushing values onto the stack and applying operators.



Terminal Questions

1. Define a stack and explain how it follows the LIFO principle.
2. List the benefits and drawbacks of using a stack.
3. Differentiate between array-based and linked-list-based stack implementations.
4. What are the primary operations performed on a stack? Explain each.
5. Explain the steps to convert an infix expression to a postfix.
6. Describe how multiple stacks can be implemented in a single array in C.



Answer Keys

Self-Assessment Questions	
Question No.	Answer
1	C
2	B
3	A
4	C
5	B
6	A
7	C
8	A
9	B
10	D



Activity

Activity Type: Offline

Duration: 1 hour

Imagine you are designing a software system that requires frequent backtracking (such as a maze-solving application or a version control system). You have the option to use either a Stack or a Queue to manage the paths taken during the process.

- **Question:** Which data structure (Stack or Queue) would be more appropriate in this scenario, and why? Explain how the characteristics of the chosen data structure (LIFO or FIFO) affect the system's behaviour.



Glossary

- **Prefix Notation:** An expression where the operator is placed before the operands.
- **Infix Notation:** A general expression where the operator is placed between operands.
- **Postfix Notation:** An expression where the operator is placed after the operands.
- **Backtracking:** A method to solve a problem by exploring all possible paths.
- **Parenthesis Checking:** Ensuring that parentheses in an expression are correctly opened and closed.



Bibliography

Textbooks

- Balagurusamy, E. (2017). *Data Structures*. McGraw Hill Education.
- Sedgewick, R., & Wayne, K. (2014). *Algorithms II*. Pearson Education.
- Sridhar, S. (2014). *Design and Analysis of Algorithms*. Oxford University Press.
- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (1989). *Introduction to Algorithms*. Prentice-Hall.



e-References

- **Stack:** <https://www.javatpoint.com/data-structure-stack>
- **Stack Data Structure:** <https://www.geeksforgeeks.org/introduction-to-stack-data-structure-and-algorithm-tutorials/>
- **Applications of Stack in Data Structure:** <https://www.javatpoint.com/applications-of-stack-in-data-structure>



Video Links

Video	Links
Introduction to Stacks	https://www.youtube.com/watch?v=l37kGX-nZEI
Array Implementation of Stacks(Part 1)	https://www.youtube.com/watch?v=rS-ZKTqwi90
Array Implementation of Stacks (Part 2)	https://www.youtube.com/watch?v=MIv2fMvt9b4
Stack Operations	https://www.youtube.com/watch?v=SE-RkUlheoc



Image Credits

- **Fig. 1:** Self-Made
- **Fig. 2:** Self-Made
- **Fig. 3:** Self-Made



Keywords

- Stack
- Stack overflow
- LIFO (Last In First Out)
- FILO (First In Last Out)
- Stack underflow
- Array-based stack
- Linked list-based stack
- Multiple stacks
- Multiple queues
- Insertion

MODULE 2

Linked Lists, Stacks, and Queue

Unit 3

Fundamental Concept of Queue



Unit Table of Contents

Unit 2.3 Fundamental Concept of Queue

Aim	176
Instructional Objectives	176
Learning Outcomes	176
2.3.1 Queue	177
2.3.1.1 Definition and Introduction to Queue	177
2.3.1.2 Queue Implementation	179
2.3.1.3 Operations of Queue	185
Self-Assessment Questions	188
2.3.2 Circular Queue	189
Self-Assessment Questions	197
Summary	198
Terminal Questions	198
Answer Keys	199
Activity	199
Glossary	200
Bibliography	200
e-References	200
Video Links	201
Image Credits	201
Keywords	201



Aim

To familiarise students with a comprehensive interpretation of queues and circular queues, including their definitions, implementations, and operations.



Instructional Objectives

This unit is designed to:

- Describe the basic structure and behaviour of a queue, focusing on its First In, First Out (FIFO) principle
- Explain the basic operations of a queue, including enQueue and deQueue
- Discuss the implementation of a circular queue and manage circular behaviour using array indices



Learning Outcomes

At the end of the unit, the student is expected to:

- Assess the Implementation of a queue using arrays or linked lists in a programming language
- Illustrate the structural difference between a linear queue and a circular queue
- Analyse the application, advantages, and disadvantages of the queue

2.3.1 Queue

2.3.1.1 Definition and Introduction to Queue

A queue is a linear data structure containing homogeneous elements. In other words, it is an abstract data type that can be implemented as a linear list. It follows the “First In, First Out Method (FIFO).” Generally, a Queue has two ends: the front end and the rare end.

1. The front end is used for deletion.
2. The rare end is used for inserting values.

Hence, it follows a FIFO manner. We can access the first element inserted in the queue. We can add elements to the Queue. This process is called EnQueue(store). We can delete the elements from the Queue. This process is called deQueue(delete).

The front end always represents the first element of the Queue, and the back end represents the last element of the Queue.

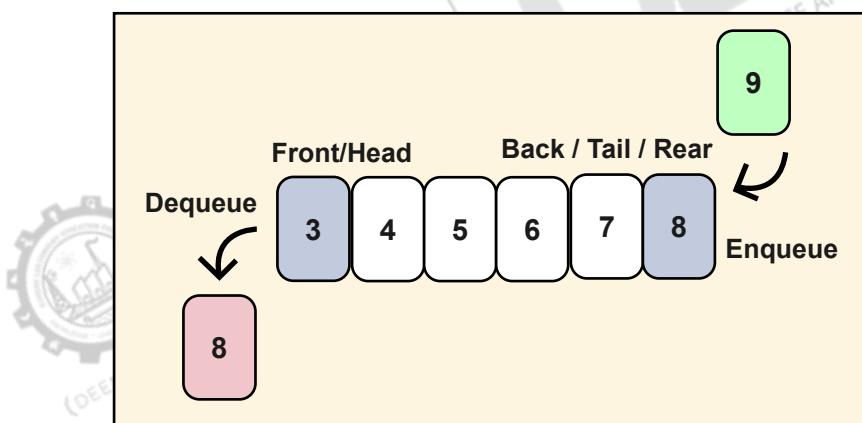


Fig. 1: Queue

Applications of Queue

A Queue always performs its operations in a “FIFO” manner. The following are some of the applications of Queues in computer systems.

1. Printing Using Computer:

When several documents are to be printed, they are done using the FIFO method only because all documents are arranged as a list using Queue.

2. Reservation System:

Reservations, such as air tickets or railway reservations, are made on a first-come, first-served basis, which is the FIFO method.

3. Computer Networks:

Computer networks are queues that give access from a server to several clients who are connected to the server using the FIFO method.

Advantages of Queue:

- Queue is used in many applications, such as printing documents.
- Queue is used in reservation-oriented information systems.
- It allows for the efficient management of large volumes of data.
- Operations like inserting and deleting elements are straightforward due to its First In, First Out (FIFO) principle.
- Queues are ideal for multiple users or processes needing a specific service.
- They provide fast communication between processes, enhancing performance.
- Queues can serve as a foundation for building other complex data structures.

Disadvantages of Queue

- Performing insertions or deletions from the middle of the queue can be time intensive.
- Searching for a particular element requires $O(N)$ time, which may be inefficient for large queues.
- In the case of array-based queues, the maximum size must be predetermined.

2.3.1.2 Queue Implementation

We can implement the Queues in Java by using two ways.

1. Queue implementation using arrays.
2. Queue implementation using LinkedList.

Algorithm to delete an element from the queue

If the front value is -1 or greater than the rear, write an underflow message and exit.

Otherwise, keep increasing the value of the front and return the item stored at the front end of the queue each time.

Algorithm

Step 1: IF FRONT = -1 or FRONT > REAR

 Write UNDERFLOW

 ELSE

 SET VAL = QUEUE[FRONT]

 SET FRONT = FRONT + 1

 [END OF IF]

Step 2: EXIT

C Function

```
int delete (int queue[], int max, int front, int rear)
{
    int y;
    if (front == -1 || front > rear)
    {
        printf("underflow");
    }
    else
    {
        y = queue[front];
        if(front == rear)
```

```
{  
    front = rear = -1;  
    else  
        front = front + 1;  
    }  
    return y;  
}  
}
```

Menu-driven program to implement queue using array

```
#include<stdio.h>  
#include<stdlib.h>  
#define maxsize 5  
void insert();  
void delete();  
void display();  
int front = -1, rear = -1;  
int queue[maxsize];  
void main ()  
{  
    int choice;  
    while(choice != 4)  
    {  
        printf("\n*****Main Menu*****\n");  
        printf("=====\n");  
        printf("\n1.insert an element\n2.Delete an element\n3.Display the queue\n4.Exit\n");  
        printf("\nEnter your choice ?");  
        scanf("%d",&choice);  
        switch(choice)
```

```

{
    case 1:
insert();
break;
    case 2:
delete();
break;
    case 3:
display();
break;
    case 4:
exit(0);
break;

    default:
printf("\nEnter valid choice??\n");
}
}
}

void insert()
{
    int item;
printf("\nEnter the element\n");
scanf("\n%d",&item);
if(rear == maxsize-1)
{
    printf("\nOVERFLOW\n");
return;
}
if(front == -1 && rear == -1)
{
    front = 0;
    rear = 0;
}
}

```

```
    }
else
{
    rear = rear+1;
}
queue[rear] = item;
printf("\nValue inserted ");

}

void delete()
{
    int item;
    if (front == -1 || front > rear)
    {
        printf("\nUNDERFLOW\n");
        return;
    }

    else
    {
        item = queue[front];
        if(front == rear)
        {
            front = -1;
            rear = -1 ;
        }
        else
        {
            front = front + 1;
        }
    }
    printf("\nvalue deleted ");
}
```

```

    }
}

void display()
{
    int i;
    if(rear == -1)
    {
        printf("\nEmpty queue\n");
    }
    else
    {
        printf("\nprinting values ....\n");
        for(i=front;i<=rear;i++)
        {
            printf("\n%d\n",queue[i]);
        }
    }
}

```

Output:

```

*****Main Menu*****
=====
1. insert an element
2. Delete an element
3. Display the queue
4. Exit

```

Enter your choice ?1

Enter the element

123

Value inserted

*****Main Menu*****

=====

1. Insert an element
2. Delete an element
3. Display the queue
4. Exit

Enter your choice ?1

Enter the element

90

Value inserted

*****Main Menu*****

=====

1. Insert an element
2. Delete an element
3. Display the queue
4. Exit

Enter your choice ?2

value deleted

*****Main Menu*****

=====

1. Insert an element
2. Delete an element
3. Display the queue
4. Exit

Enter your choice ?3

printing values

90

*****Main Menu*****

=====

1. Insert an element
2. Delete an element
3. Display the queue
4. Exit

2.3.1.3 Operations of Queue

We can perform the following Operations in Queue.

1. **qStore:** It is a process of adding elements at the end of Queue.

Algorithm:

qStore(Queue, element)

1. if isFull(Queue) == true
 Output "Queue is full, cannot add element"
 return
2. else
 rear = rear + 1 // Move rear to the next position
 Queue[rear] = element // Insert element at rear

2. **qDelete:** It is a process of removing elements from the front end of a Queue.

Algorithm:

qDelete(Queue)

1. if isEmpty(Queue) == true
 Output "Queue is empty, cannot delete element"
 return

2. else

```
element = Queue[front] // Get the element at the front
front = front + 1 // Move front to the next position
return element
```

3. isFull: It determines whether the Queue is full.

Algorithm:

```
isFull(Queue)
1. if rear == MAX_SIZE - 1
    return true
2. else
    return false
```

4. isEmpty: It determines whether the Queue is empty.

Algorithm:

```
isEmpty(Queue)
1. if front > rear
    return true
2. else
    return false
```

5. Create: It is used to create an empty Queue.

Algorithm:

```
CreateQueue(Queue, size)
1. Initialize front = 0
2. Initialize rear = -1
3. MAX_SIZE = size // Set the maximum size of the queue
4. for i = 0 to MAX_SIZE - 1

    Queue[i] = NULL // Initialize all positions to NULL
```

6. First: It is used to return the first element of the Queue.

Algorithm:

```
First(Queue)
1. if isEmpty(Queue) == true
    Output "Queue is empty"
    return NULL

2. else
    return Queue[front] // Return the front element
```

7. Last: It is used to return the last element of the Queue.

Algorithm:

```
Last(Queue)
1. if isEmpty(Queue) == true
    Output "Queue is empty"
    return NULL

2. else
    return Queue[rear] // Return the rear element
```



Self-Assessment Questions

1. Which operation is used to add an element to a queue?

- a) EnQueue
- b) DeQueue
- c) InsertQueue
- d) DeleteQueue

2. Where are elements inserted in a queue?

- a) Front
- b) Middle
- c) Rear
- d) Top

3. The process of adding an element to a queue is known as _____.

- a) deQueue
- b) enQueue
- c) insertQueue
- d) pushQueue

4. The _____ operation checks whether the queue is empty.

- a) isFull
- b) nQueue
- c) isEmpty
- d) deQueue

5. A queue is a linear data structure that follows the Last In First Out (LIFO) principle.

- a) True
- b) False

6. In the array implementation of a queue, the maximum size must be predetermined.

- a) True
- b) False

2.3.2 Circular Queue

A Circular Queue is a modified version of the standard queue where the last element is connected back to the first, forming a circular structure. The operations in a circular queue follow the FIFO (First In, First Out) principle and are often referred to as a Ring Buffer.

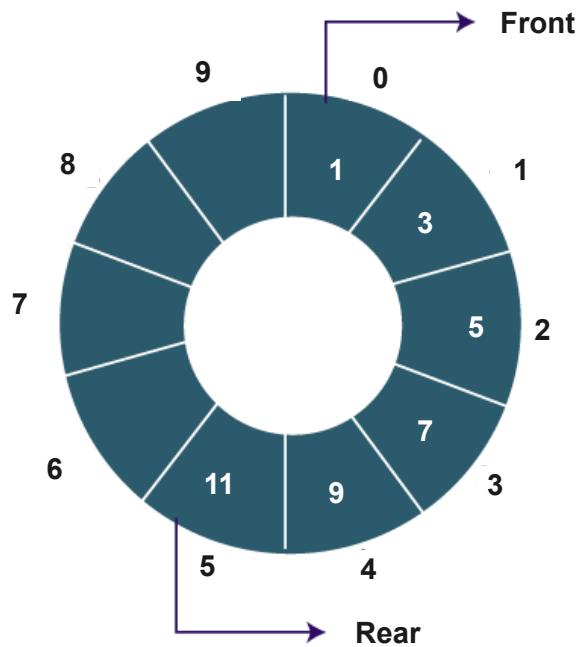


Fig. 2: Circular Queue

Operations on Circular Queue:

- **Front:** Retrieves the item at the front of the queue.
- **Rear:** Retrieves the item at the rear of the queue.
- **enQueue(value):** This operation adds an element to the rear of the circular queue. The new element is always added at the rear.

Before insertion, check if the queue is full (i.e., the rear is positioned just before the front circularly).

If it is full, then display Queue is full.

If the queue is not full, insert an element at the end of the queue.

- **deQueue()** This function deletes an element from the circular queue, always from the front position.
- Check whether the queue is Empty.

If it is empty, then display Queue is empty.

If the queue is not empty, then get the last element and remove it from the queue.

Illustration of Circular Queue Operations:

Refer to the image below to interpret better the enqueue and dequeue operations.

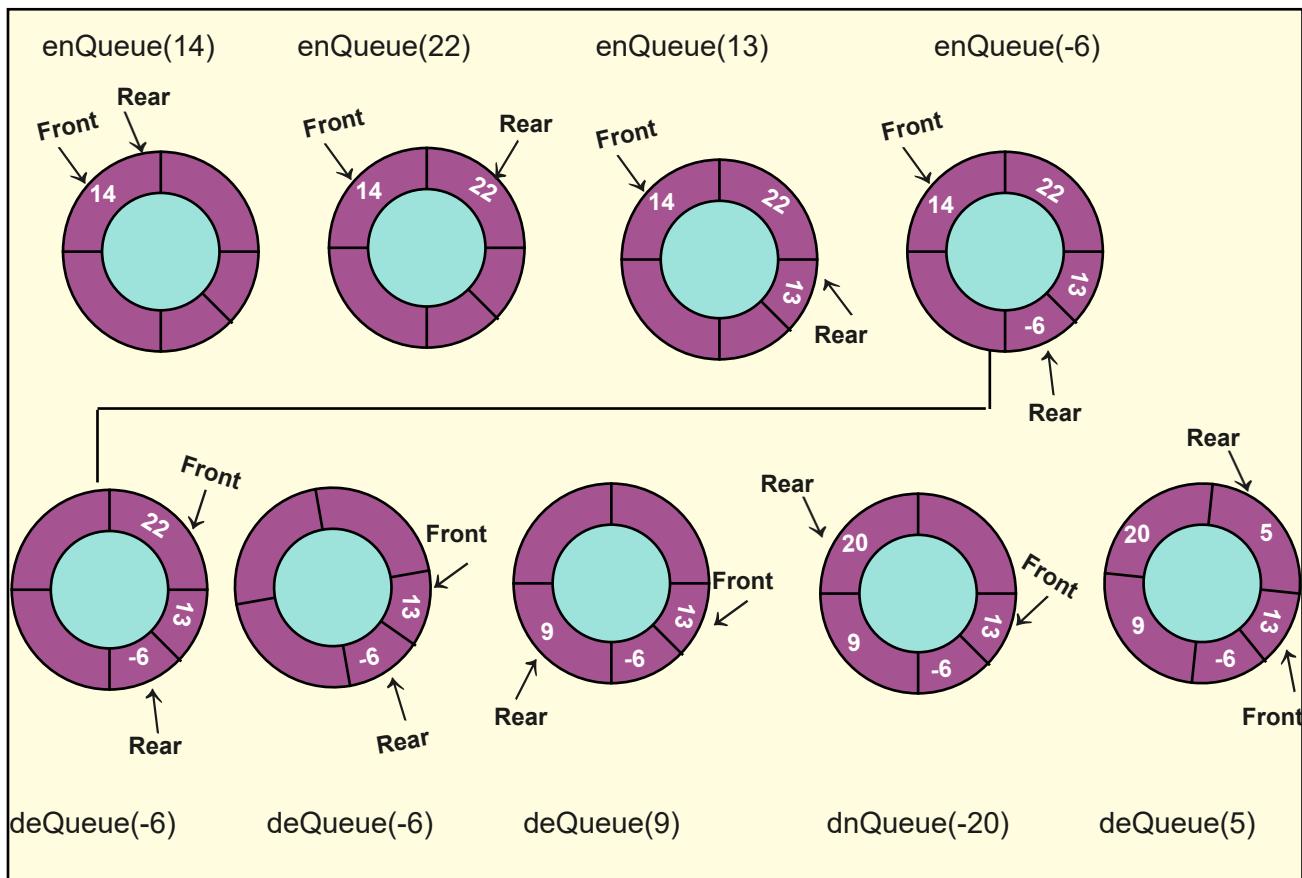


Fig. 3: Working process of circular queue operations

How to Implement a Circular Queue

A circular queue can be implemented using two data structures:

1. Array
2. Linked List

Implement Circular Queue using Array:

- Initialise an array queue of size n, where n is the maximum number of elements the queue can hold.
- Initialise two variables, front and rear, to -1.
- Enqueue: To enqueue an element x into the queue, do the following:
 - (i) Increment rear by 1.
 - (ii) If the rear equals n, set the rear to 0.
 - (iii) If the front is -1, set the front to 0.
 - (iv) Set queue[rear] to x.
- Dequeue: To dequeue an element from the queue, do the following:
 - (i) Check if the queue is empty by checking if the front is -1.
 - (ii) If it is, return an error message indicating that the queue is empty.
 - (iii) Set x to queue[front].
 - (iv) If the front equals the rear, set the front and rear to -1.
 - (v) Otherwise, increment the front by 1; if the front is equal to n, set the front to 0.
 - (vi) Return x.

```
// C or C++ program for insertion and
// deletion in Circular Queue
#include<bits/stdc++.h>
using namespace std;
```

```
class Queue
{
    // Initialise front and rear
    int rear, front;

    // Circular Queue
    int size;
    int *arr;

public:
    Queue(int s)
    {
        front = rear = -1;
        size = s;
        arr = new int[s];
    }

    void enQueue(int value);
    int deQueue();
    void displayQueue();
};

/* Function to create Circular queue */
void Queue::enQueue(int value)
{
    if ((front == 0 && rear == size-1) ||
        ((rear+1) % size == front))
    {
        printf("\nQueue is Full");
        return;
    }

    else if (front == -1) /* Insert First Element */
    {
        front = rear = 0;
```

```

arr[rear] = value;
}

else if (rear == size-1 &&front != 0)
{
    rear = 0;
    arr[rear] = value;
}

else
{
    rear++;
    arr[rear] = value;
}

// Function to delete element from Circular Queue
int Queue::deQueue()
{
    if (front == -1)
    {
        printf("\nQueue is Empty");
        return INT_MIN;
    }

    int data = arr[front];
    arr[front] = -1;

    if (front == rear)
    {
        front = -1;
        rear = -1;
    }
    else if (front == size-1)
        front = 0;
    else

```

```
front++;

return data;
}

// Function displaying the elements
// of Circular Queue
void Queue::displayQueue()
{
    if (front == -1)
    {
        printf("\nQueue is Empty");
        return;
    }
    printf("\nElements in Circular Queue are: ");
    if (rear >= front)
    {
        for (int i = front; i<= rear; i++)
        printf("%d ",arr[i]);
    }
    else
    {
        for (int i = front; i< size; i++)
        printf("%d ", arr[i]);

        for (int i = 0; i<= rear; i++)
        printf("%d ", arr[i]);
    }
}

/* Driver of the program */
int main()
{
    Queue q(5);
```

```

// Inserting elements in Circular Queue
q.enQueue(14);
q.enQueue(22);
q.enQueue(13);
q.enQueue(-6);

// Display elements present in Circular Queue
q.displayQueue();

// Deleting elements from Circular Queue
printf("\nDeleted value = %d", q.deQueue());
printf("\nDeleted value = %d", q.deQueue());

q.displayQueue();

q.enQueue(9);
q.enQueue(20);
q.enQueue(5);

q.displayQueue();

q.enQueue(20);
return 0;
}

```

Output

Elements in Circular Queue are: 14 22 13 -6
 Deleted value = 14
 Deleted value = 22
 Elements in Circular Queue are: 13 -6
 Elements in Circular Queue are: 13 -6 9 20 5
 Queue is Full

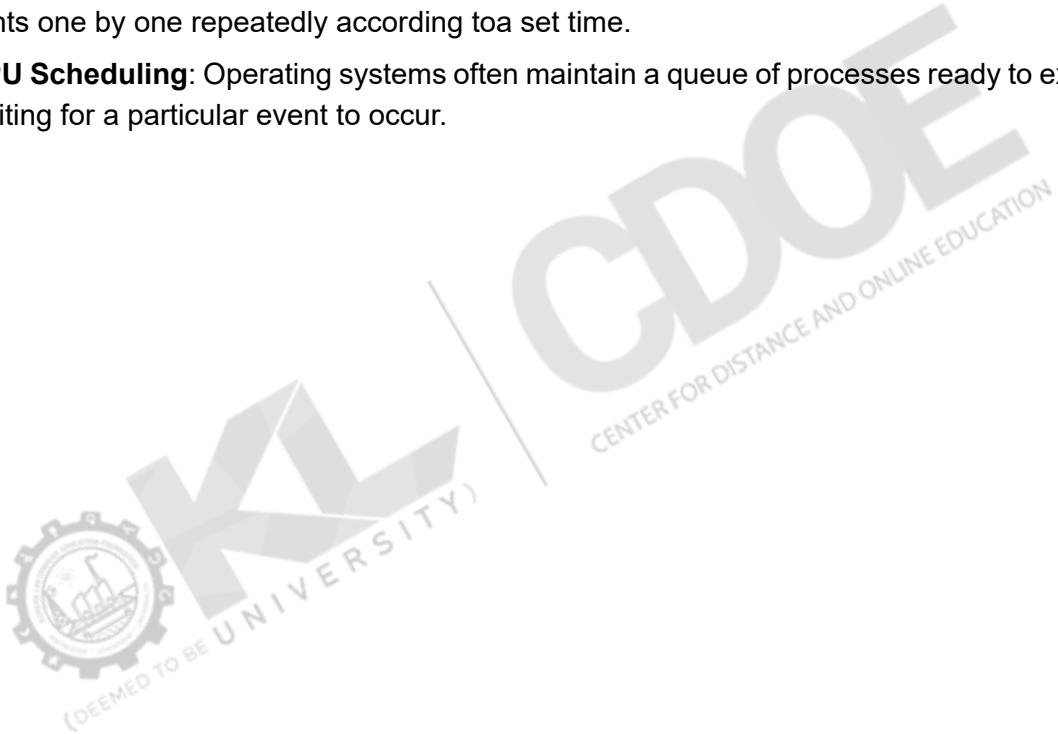
Complexity Analysis of Circular Queue Operations:

Time Complexity:

- **Enqueue:** O(1) because no loop is involved for a single enqueue.
- **Dequeue:** O(1) because no loop is involved for one dequeue operation.
- **Auxiliary Space:** O(N) as the queue is size N.

Applications of Circular Queue:

- **Memory Management:** Ordinary queues' unused memory locations can be utilised in circular queues.
- **Traffic system:** In a computer-controlled traffic system, circular queues switch on the traffic lights one by one repeatedly according to a set time.
- **CPU Scheduling:** Operating systems often maintain a queue of processes ready to execute or waiting for a particular event to occur.





Self-Assessment Questions

7. What is the time complexity of the enQueue operation in a circular queue?
- a) O(N)
 - b) O(1)
 - c) O(log N)
 - d) O(N²)
8. Which of the following is not an application of circular queues?
- a) Memory management
 - b) Traffic systems
 - c) Cpu scheduling
 - d) Sorting algorithms
9. In a circular queue, no more elements can be inserted once the queue becomes full until an element is deleted.
- a) True
 - b) False
10. Circular queues waste memory by not reusing empty spaces at the front of the queue.
- a) True
 - b) False
11. In a circular queue, the last element connects to the _____ element, forming a circle.
- a) Front
 - b) Middle
 - c) First
 - d) Rear
12. The process of adding an element to the rear of the queue is called _____.
- a) enQueue
 - b) deQueue
 - c) addQueue
 - d) pushQueue



Summary

- A queue is a linear data structure that follows the first-in-first-out (FIFO) principle, where elements are inserted from the rear and deleted from the front.
- It uses arrays or linked lists in various applications, including printing systems, reservation systems, and computer networks.
- Queue operations include inserting elements (EnQueue), deleting elements (DeQueue), checking if the queue is full or empty, and retrieving the first and last elements.
- Circular Queue follows the First In, First Out (FIFO) principle and can be implemented using arrays or linked lists.
- Critical operations include retrieving the front and rear items, adding elements to the rear (enQueue), and removing elements from the front (deQueue).
- When implemented with arrays, it involves managing indices that wrap around once the queue reaches the end.
- The time complexity for both enQueue and deQueue operations is $O(1)$, while the space complexity is $O(N)$, where N is the queue size.
- Circular queues are used in memory management to prevent unused spaces, traffic systems for managing traffic lights, and CPU scheduling to organise processes efficiently.



Terminal Questions

1. Define a queue. What is the principle that governs its operations?
2. What are the main applications, advantages and disadvantages of queues?
3. How can a queue be implemented in Java?
4. List and describe the main operations performed on a queue.
5. What is a circular queue, and how does it differ from a linear queue?
6. Explain how enQueue and deQueue operations work in a circular queue.



Answer Keys

Self-Assessment Questions	
Question No.	Answer
1	A
2	C
3	B
4	C
5	B
6	A
7	B
8	D
9	A
10	B
11	C
12	A



Activity

Activity Type: Offline

Duration: 1 hour

How would you design a Circular Queue implementation that minimises space complexity while ensuring efficient performance in enqueue and dequeue operations? Discuss the trade-offs between an array-based implementation versus a linked list-based implementation for a Circular Queue. How would you address challenges like handling overflow and underflow conditions in a Circular Queue?



Glossary

- **Rear:** The end of the queue where elements are inserted.
- **FIFO (First In First Out):** A method where the first element added is the first one to be removed.
- **Underflow:** A condition when attempting to delete an element from an empty queue.
- **Ring Buffer:** Another term for a circular queue, referring to its circular structure.
- **CPU Scheduling:** A process in operating systems that uses circular queues to manage processes.
- **Overflow:** A condition where no more elements can be added to the queue because it is full.



Bibliography

Textbooks

- Balagurusamy, E. (2017). *Data Structures*. McGraw Hill Education.
- Sedgewick, R., & Wayne, K. (2014). *Algorithms II*. Pearson Education.
- Sridhar, S. (2014). *Design and Analysis of Algorithms*. Oxford University Press.
- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (1989). *Introduction to Algorithms*. Prentice-Hall.



e-References

- **Queue Data Structure:** <https://www.shiksha.com/online-courses/articles/queue-data-structure-types-implementation-applications/>
- **Array representation of Queue:** <https://www.javatpoint.com/array-representation-of-queue>
- **A Comprehensive Look at Queue in Data Structure:** <https://www.simplilearn.com/tutorials/data-structure-tutorial/queue-in-data-structure>
- **Introduction to Circular Queue:** <https://www.geeksforgeeks.org/introduction-to-circular-queue/>



Video Links

Video	Links
Queue In Data Structure	https://www.youtube.com/watch?v=yzj0Ch01Exo
Queue Operations	https://www.youtube.com/watch?v=sDO9bPaBg6A
Queue Implementation Using Linked List	https://www.youtube.com/watch?v=QfRoQMSJ664
Array implementation of Queue	https://www.youtube.com/watch?v=okr-XE8yTO8
Circular Queue	https://www.youtube.com/watch?v=eKxWdc1DVFE



Image Credits

- **Fig. 1:** Self-Made
- **Fig. 2:** Self-Made
- **Fig. 3:** Self-Made



Keywords

- Queue
- Enqueue
- Dequeue
- Circular queue
- Time complexity
- Space complexity
- Memory management
- Traffic systems

MODULE 2

Linked Lists, Stacks, and Queue

Unit 4

Types of Queues

■ Unit Table of Contents

Unit 2.4 Types of Queues

Aim	205
Instructional Objectives	205
Learning Outcomes	205
2.4.1 De-Queue	206
Self-Assessment Questions	216
2.4.2 Priority Queue	217
Self-Assessment Questions	222
Summary	223
Terminal Questions	223
Answer Keys	224
Activity	224
Glossary	225
Bibliography	225
e-References	226
Video Links	226
Image Credits	226
Keywords	411



Aim

To provide students with a comprehensive interpretation of de-queue and priority queue concepts as specialised queue forms, highlighting their structure, types, operations, and applications.



Instructional Objectives

This unit is designed to:

- Describe the structure and function of a de-queue (double-ended queue) and differentiate it from a standard queue
- Discuss the priority queuedata structure and interpret how it differs from a standard queue
- Identify the types (input-restricted and output-restricted) and operationsof de-queue
- Explain the complexity of operations such as insertion and deletion



Learning Outcomes

At the end of the unit, the student is expected to:

- Analyse the time complexity of priority queue operations (add, remove, peek) in various data structures
- Evaluate the use of de-queues and priority queues in practical applications
- Implement a priority queue using arrays, linked lists, heaps, or binary search trees in a programming language

2.4.1 De-Queue

Deque (or double-ended queue)

A queue is a data structure that operates on the First-In-First-Out (FIFO) principle, meaning that the first element to enter is also the first to be removed. In a queue, elements are added at the rear (or tail) and removed from the front (or head). A common real-life illustration is a line outside a movie theatre, where the first person in line is the first to buy a ticket, and the last person in line buys their ticket.

What is a Deque (or double-ended queue)

A Deque, short for Double-Ended Queue, is a linear data structure that allows elements to be inserted and removed from both ends, making it a more flexible version of a traditional queue.

Although insertion and deletion in a deque can occur at both ends, it does not adhere to the FIFO principle. A deque can be visualised as follows:

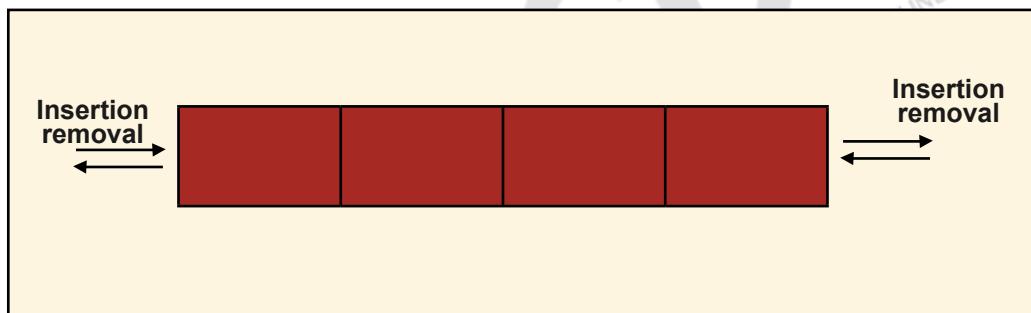


Fig. 1: Representation of deque

Types of Deques

There are **two** primary types of deques:

1. Input Restricted Queue
2. Output Restricted Queue

1. Input Restricted Queue:

In this type, elements can be inserted only from one end, while deletion can happen from both ends.

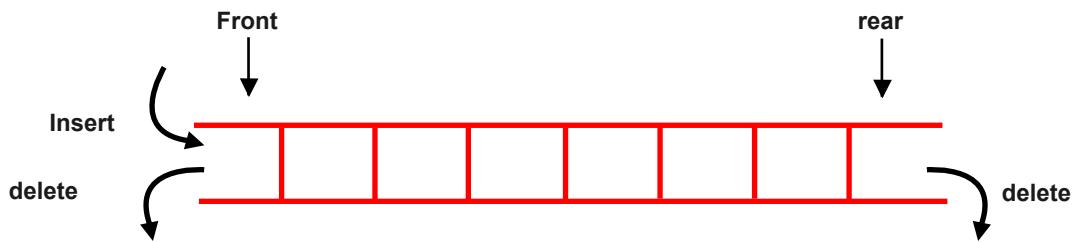


Fig. 2: Input-restricted double-ended queue

2. Output Restricted Queue:

In this type, elements can be removed only from one end, but insertion is allowed at both ends.

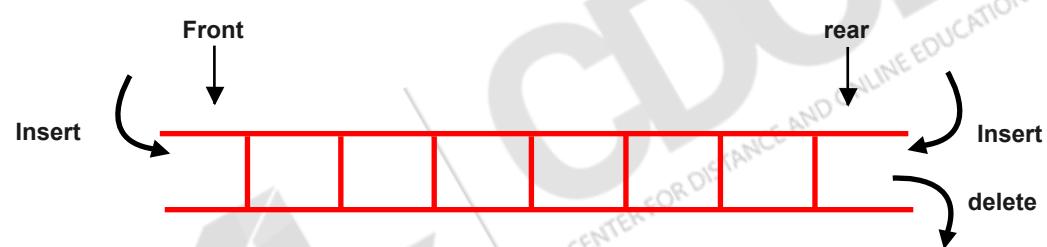


Fig. 3: Output-restricted double-ended queue

Operations on Deques

Several operations can be performed on a deque, such as:

- Inserting at the front
- Inserting at the rear
- Removing from the front
- Removing from the rear

In addition to these, peek operations are also supported. Using these, you can access the front and rear elements of the deque. Additional operations include:

- Retrieve the front element
- Retrieve the rear element

- Check if the deque is full
- Check if the deque is empty

Now, let's explore these operations with examples.

Insertion at the Front End

In this operation, an element is inserted at the front of the deque. Before proceeding, it's essential to verify if the deque is full. If not, the insertion is performed based on the following conditions:

- If the deque is empty, both the front and rear are initialised to 0, pointing to the first element.
- Otherwise, if the front is less than 1, set front = n - 1, moving it to the last index.

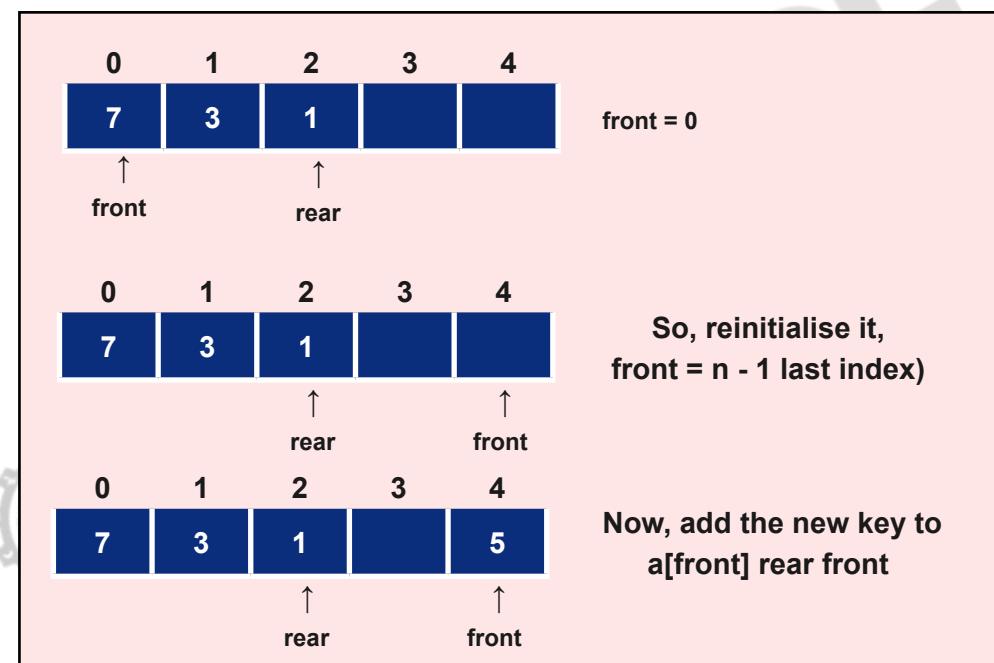


Fig. 4: Front end insertion

Insertion at the Rear End

This operation adds an element at the rear end of the deque. Before doing so, we check whether the deque is full. If there's space, the element is inserted as follows:

- If the deque is empty, both the front and rear are set to 0, pointing to the first element.
- Otherwise, increment the rear by 1. If the rear reaches the last index, reset it to 0.

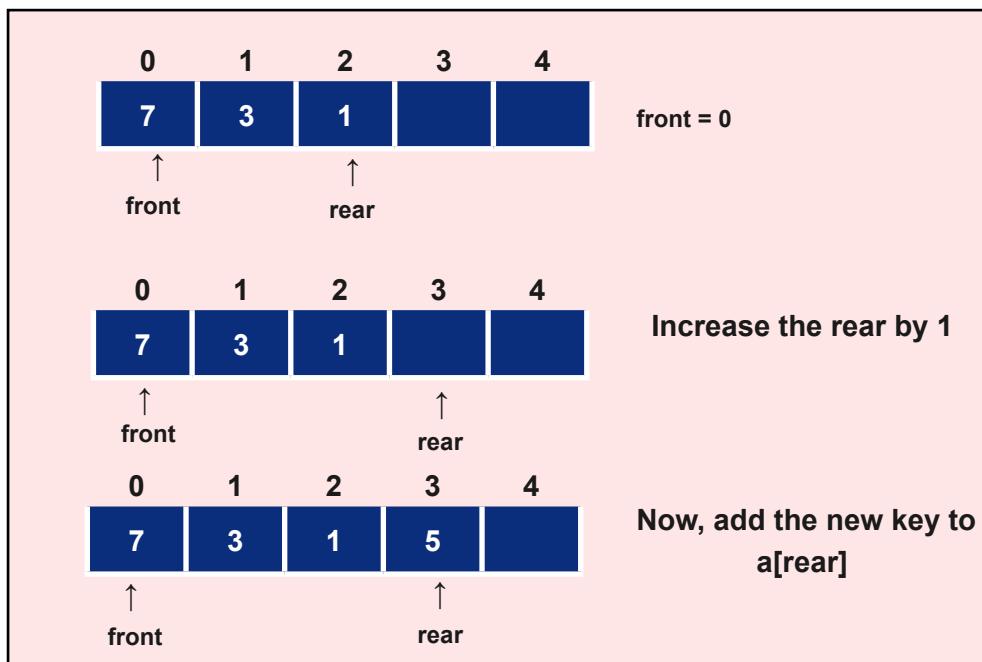


Fig. 5: Rear end insertion

Deletion at the Front End

This operation removes an element from the front. Before deletion, confirming that the deque isn't empty is essential. If it is, indicated by $\text{front} = -1$, the operation cannot proceed due to underflow. If not empty, deletion happens as follows:

- If there's only one element, set both $\text{rear} = -1$ and $\text{front} = -1$.
- If the front is at the last index, reset it to 0.
- Otherwise, increment the front by 1.

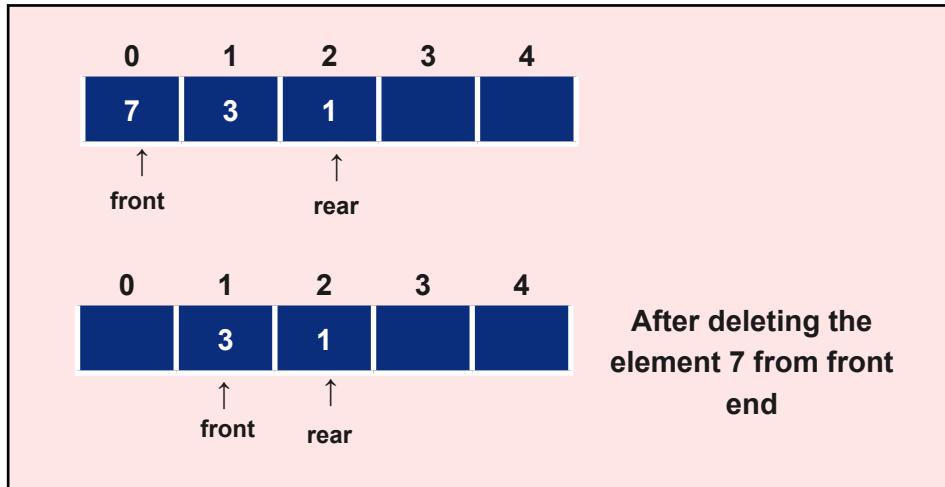


Fig. 6: Front end deletion

Deletion at the Rear End

Here, an element is removed from the rear. Before deletion, we must check if the deque is empty. If $\text{front} = -1$, it indicates underflow and deletion can't occur. Otherwise:

- If there's only one element, set both $\text{rear} = -1$ and $\text{front} = -1$.
- If $\text{rear} = 0$, reset rear to $n - 1$.
- Otherwise, decrement the rear by 1.

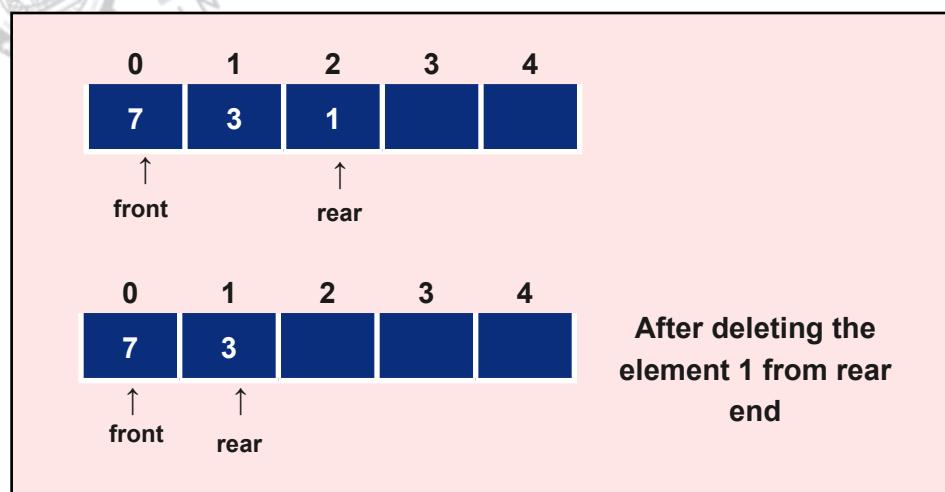


Fig. 7: Rear end deletion

Check if Empty

This operation verifies whether the deque is empty. If front = -1, the deque is empty.

Check if Full

This operation checks if the deque is full. If front = rear + 1 or front = 0 = rear = n - 1 = deque is full.

The time complexity for all deque operations is O(1), meaning they execute in constant time.

Applications of Deque

- A deque can function as a stack and a queue, allowing a wide range of operations.
- It can be used as a palindrome checker, where the string is read from both ends to verify if it's the same in reverse.

```
#include <stdio.h>
#define size 5
int deque[size];
int f = -1, r = -1;
// insert_front function will insert the value from the front
void insert_front(int x)
{
    if((f==0 && r==size-1) || (f==r+1))
    {
        printf("Overflow");
    }
    else if((f==-1) && (r==-1))
    {
        f=r=0;
        deque[f]=x;
    }
    else if(f==0)
    {
        f=size-1;
        deque[f]=x;
    }
    else
```

```

{
  f=f-1;
  deque[f]=x;
}

}

// insert_rear function will insert the value from the rear
void insert_rear(int x)
{
if((f==0 && r==size-1) || (f==r+1))
{
printf("Overflow");
}
else if((f==-1) && (r==-1))
{
r=0;
deque[r]=x;
}
else if(r==size-1)
{
r=0;
deque[r]=x;
}
else
{
r++;
deque[r]=x;
}

}

// display function prints all the value of deque.
void display()

```

```
{  
    int i=f;  
    printf("\nElements in a deque are: ");  
  
    while(i!=r)  
    {  
        printf("%d ",deque[i]);  
        i=(i+1)%size;  
    }  
    printf("%d",deque[r]);  
}  
  
// getfront function retrieves the first value of the deque.  
void getfront()  
{  
    if((f==-1) && (r==-1))  
    {  
        printf("Deque is empty");  
    }  
    else  
    {  
        printf("\nThe value of the element at front is: %d", deque[f]);  
    }  
}  
  
// getrear function retrieves the last value of the deque.  
void getrear()  
{  
    if((f==-1) && (r==-1))  
    {  
        printf("Deque is empty");  
    }  
}
```

```

else
{
printf("\nThe value of the element at rear is %d", deque[r]);
}

}

// delete_front() function deletes the element from the front
void delete_front()
{
if((f== -1) && (r== -1))
{
printf("Deque is empty");
}
else if(f==r)
{
printf("\nThe deleted element is %d", deque[f]);
f=-1;
r=-1;
}
else if(f==(size-1))
{
printf("\nThe deleted element is %d", deque[f]);
f=0;
}
else
{
printf("\nThe deleted element is %d", deque[f]);
f=f+1;
}
}

```

```
// delete_rear() function deletes the element from the rear
void delete_rear()
{
    if((f== -1) && (r== -1))
    {
        printf("Deque is empty");
    }
    else if(f==r)
    {
        printf("\nThe deleted element is %d", deque[r]);
        f=-1;
        r=-1;
    }
    else if(r==0)
    {
        printf("\nThe deleted element is %d", deque[r]);
        r=size-1;
    }
    else
    {
        printf("\nThe deleted element is %d", deque[r]);
        r=r-1;
    }
}

int main()
{
    insert_front(20);
    insert_front(10);
    insert_rear(30);
    insert_rear(50);
    insert_rear(80);
```

```
display(); // Calling the display function to retrieve the values of deque  
getfront(); // Retrieve the value at front-end  
getrear(); // Retrieve the value at rear-end  
delete_front();  
delete_rear();  
display(); // calling display function to retrieve values after deletion  
return 0;  
}
```





Self-Assessment Questions

1. In an Input Restricted Queue, elements can be inserted from only one end, while deletion is allowed from _____.
 - a) Rear end
 - b) Both ends
 - c) One end
 - d) Middle
2. Elements can be inserted and deleted in a deque from the front and rear ends.
 - a) True
 - b) False
3. The time complexity of all deque operations is _____.
 - a) $O(n)$
 - b) $O(\log n)$
 - c) $O(1)$
 - d) $O(n^2)$
4. In an Output Restricted Queue, insertion can occur at both ends, but deletion is restricted to _____.
 - a) Front end
 - b) Both ends
 - c) One end
 - d) Middle
5. What happens when the front of the deque reaches 0, and an element is inserted at the front?
 - a) The front is set to size-1
 - b) The rear is reset to 0
 - c) The front is incremented by 1
 - d) The rear is set to size-1

2.4.2 Priority Queue

A priority queue is an abstract data structure that functions similarly to a standard queue but with an added feature: each element has a priority level. In a priority queue, elements with higher priority are processed before others. The priority of each element determines the order of removal from the priority queue.

Only elements that can be compared are supported in a priority queue, meaning they can be arranged in ascending or descending order. For example, if we insert the values 1, 3, 4, 8, 14, and 22, the priority queue will arrange them in increasing order. Thus, 1 will have the highest priority, while 22 will have the lowest.

Features of a Priority Queue

A priority queue is an enhanced version of a regular queue and has the following characteristics:

- Each element has an associated priority.
- An element with a higher priority will be removed before those with lower priority.
- If two elements share the same priority, they will be processed in the order they were inserted, following the First-In-First-Out (FIFO) rule.

Example of a Priority Queue

Consider a priority queue containing the following values: 1, 3, 4, 8, 14, and 22. These values are organised in ascending order. Now, after performing a few operations, we can see how the priority queue behaves:

- **poll()**: This function removes the element with the highest priority. In this case, 1 is removed, as it has the highest priority.
- **add(2)**: This function adds the number 2 to the priority queue. Since 2 is now the smallest value, it will be assigned the highest priority.
- **poll()**: Number 2 will now be removed from the queue, as it is the highest priority.
- **add (5)**: This operation inserts 5 into the queue. Since 5 is greater than 4 but less than 8, it will have the third-highest priority.

Types of Priority Queue

There are two kinds of priority queues:

1. Ascending Order Priority Queue:

- Lower values are given higher priority in this type. For example, in a set of numbers like 1, 2, 3, 4, and 5, the smallest number, 1, is assigned the highest priority.

2. Descending Order Priority Queue:

- Higher values are given higher priority in this type. For instance, in a set of numbers like 5, 4, 3, 2, and 1, the largest number, 5, gets the highest priority.

Representation of a Priority Queue

We can represent a priority queue using a one-way list. The priority queue is structured using three lists:

- INFO list stores the data elements.
- PRN list contains the priority numbers associated with each element in the INFO list.
- LINK list holds the address of the next node.



Fig. 8: Representation of priority queue

Let's look at how we build a priority queue step by step:

- Step 1:** If the lowest priority number is 1 and its corresponding value is 333, this element is inserted first into the queue.
- Step 2:** Next, priority number 2 is higher; its associated values are 222 and 111. These elements are inserted in FIFO order, meaning 222 is added before 111.
- Step 3:** For priority number 4, the data elements are 444, 555, and 777. Based on the FIFO rule, these are also added, so 444 is inserted first, followed by 555 and 777.
- Step 4:** Finally, priority number 5 corresponds to the value 666, placed at the queue's end.

Implementing a Priority Queue

There are four common ways to implement a priority queue: arrays, linked lists, heap data structures, or binary search trees. The heap is considered the most efficient for this purpose, and we will explore how it works.

Analysis of complexities using different implementations

Implementation	add	Remove
Linked list	$O(1)$	$O(n)$
Binary heap	$O(\log n)$	$O(\log n)$
Binary search tree	$O(\log n)$	$O(\log n)$

What is Heap?

A heap is a tree-based data structure that forms a complete binary tree and satisfies the heap property. If A is a parent node of B, then A is ordered concerning the node B for all nodes A and B in a heap. It means that the value of the parent node could be more than or equal to the value of the child node, or the value of the parent node could be less than or equal to the value of the child node. Therefore, we can say that there are two types of heaps:

- **Max heap:** The max heap is a heap in which the value of the parent node is greater than the value of the child nodes.

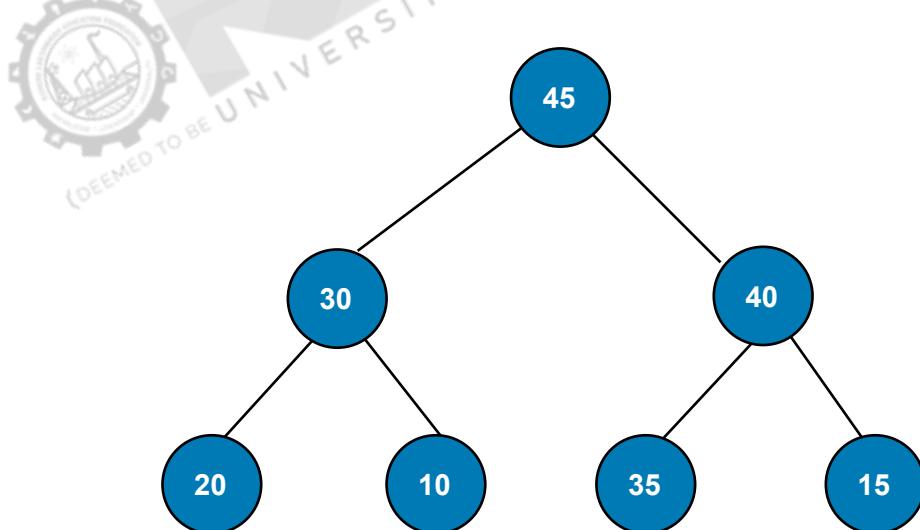


Fig. 9: Max heap

- **Min heap:** The min heap is a heap in which the value of the parent node is less than that of the child nodes.

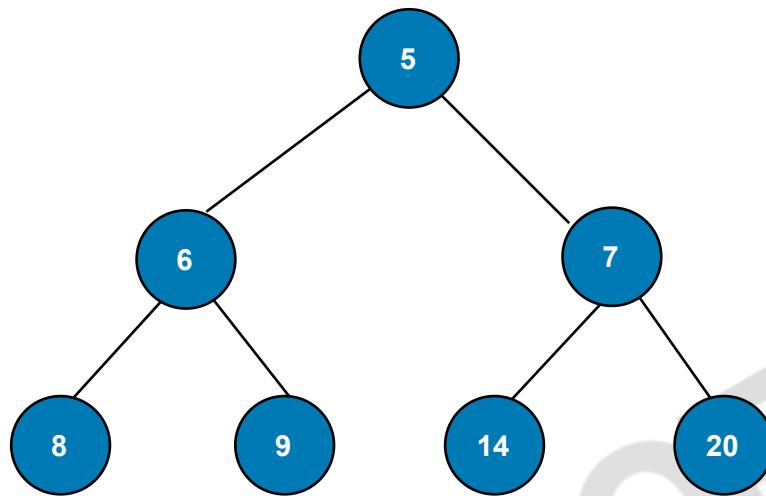


Fig. 10: Heap – Min

Both the heaps are the binary heap, as each has exactly two child nodes.

Priority Queue Operations

Insertion, deletion, and peek are common operations that we can perform on a priority queue. Let's see how we can maintain the heap data structure.

Inserting the element in a priority queue (max heap)

- If we insert an element in a priority queue, it will move to the empty slot by looking from top to bottom and left to right.
- If the element is not in the correct position, it is compared with the parent node; elements are swapped if found out of order. This process continues until the element is placed in the correct position.

Removing the minimum element from the priority queue

- As we know, the root node is the maximum element in a maxheap. When we remove the root node, an empty slot is created. The last inserted element will be added to this empty slot. Then, this element is compared with the child nodes, i.e., left-child and right-child, and swapped with the smaller of the two. It keeps moving down the tree until the heap property is restored.

Applications of Priority queue

Priority queues are widely used in various applications, including:

- Dijkstra's algorithm for finding the shortest path.
- Prim's algorithm for minimum spanning trees.
- Huffman coding in data compression techniques.
- Heap sort for sorting elements.
- They are also used in operating systems for priority scheduling, load balancing, and interrupt handling tasks.





Self-Assessment Questions

6. In a priority queue, each element has an associated_____.
- a) Position
 - b) Priority
 - c) Value
 - d) Task
7. In a max heap, the _____ node has a value greater than its children.
- a) Leaf
 - b) Child
 - c) Root
 - d) Middle
8. The priority queue can be implemented using arrays, linked lists, heaps, or_____.
- a) Binary search trees
 - b) Graphs
 - c) Queues
 - d) Stacks
9. In a priority queue, elements with lower priority are processed before elements with higher priority.
- a) True
 - b) False
10. Heaps are a tree-based data structure that implements efficient priority queues.
- a) True
 - b) False



Summary

- Deque (double-ended queue) is a versatile linear data structure that allows the insertion and deletion of elements from both ends, making it more flexible than a standard queue.
- Deques support several operations, such as insertion at the front or rear, deletion from both ends and peeking at the front or rear elements.
- There are two main types of deques: input-restricted (insertion at one end, deletion at both ends) and output-restricted (deletion at one end, insertion at both ends).
- Deques can be used in various applications, including stack operations and palindrome checking.
- The time complexity of deque operations is constant ($O(1)$), making it an efficient data structure.
- A Priority queue is a specialised data structure that works like a standard queue but assigns a priority to each element.
- Ascending order priority queue, where lower values are given higher priority, and Descending order priority queue, where higher values are given higher priority.
- Max heaps, where the parent node is more significant than its children, and Min heaps, where the parent node is smaller than its children.
- Priority queues are commonly used in algorithms like Dijkstra's and Prim's, as well as in Huffman coding, heap sort, and operating system scheduling.



Terminal Questions

1. What is a deque, and how is it different from a queue?
2. Describe the two primary types of deques.
3. List and explain the operations that can be performed on a deque.
4. What is a priority queue, and how does it differ from a regular queue?
5. Explain the difference between an ascending and a descending order priority queue.
6. Describe how a max heap implements a priority queue.



Answer Keys

Self-Assessment Questions	
Question No.	Answer
1	B
2	A
3	C
4	C
5	A
6	B
7	C
8	A
9	B
10	A



Activity

Activity Type: Offline

Duration: 1 hour

Imagine using a de-queue data structure to manage a line of customers at a bank. How would you prioritise customers arriving at the bank based on their needs and the required services? Discuss the factors you would consider in deciding the order in which customers should be served using the de-queue implementation. How might you handle situations where a high-priority customer arrives while others are still waiting? Brainstorm potential strategies to manage the customer line using the de-queue data structure efficiently.



Glossary

- **Constant Time Complexity ($O(1)$):** The time complexity for operations that execute in a fixed amount of time, regardless of the input size.
- **Dijkstra's Algorithm:** A graph algorithm that uses a priority queue to find the shortest path between nodes.
- **Huffman Coding:** A compression technique that uses a priority queue to assign variable-length codes to input characters.
- **Prim's Algorithm:** A minimum spanning tree algorithm that uses a priority queue.
- **Heap Sort:** A comparison-based sorting algorithm that builds a heap and then extracts elements in sorted order.



Bibliography

Textbooks

- Balagurusamy, E. (2017). *Data Structures*. McGraw Hill Education.
- Sedgewick, R., & Wayne, K. (2014). *Algorithms II*. Pearson Education.
- Sridhar, S. (2014). *Design and Analysis of Algorithms*. Oxford University Press.
- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (1989). *Introduction to Algorithms*. Prentice-Hall.



e-References

- **Deque Data Structure:** <https://www.programiz.com/dsa/deque>
- **Introduction to Priority Queue:** <https://www.geeksforgeeks.org/priority-queue-set-1-introduction/>



Video Links

Video	Links
Double-ended queue with examples	https://www.youtube.com/watch?v=RZxpblvnDC8
Priority queue in data structure	https://www.youtube.com/watch?v=YDo65gKDRPo



Image Credits

- **Fig. 1:** Self-Made
- **Fig. 2:** Self-Made
- **Fig. 3:** Self-Made
- **Fig. 4:** Self-Made
- **Fig. 5:** Self-Made
- **Fig. 6:** Self-Made
- **Fig. 7:** Self-Made
- **Fig. 8:** Self-Made
- **Fig. 9:** Self-Made
- **Fig. 10:** Self-Made



Keywords

- Deque (double-ended queue)
- Priority queue
- Input restricted queue
- Output restricted queue
- Max heap
- Min heap

DATA STRUCTURES AND ALGORITHMS

MODULE 3

Trees



CDOE
CENTER FOR DISTANCE AND ONLINE EDUCATION

Module Description

The module explores tree-based data structures, which are fundamental to efficient data organisation and retrieval. It begins with an Introduction to Trees, explaining basic concepts like nodes, edges, root, parent, child, and height, helping students grasp the hierarchical nature of trees. The module then explains the different types of trees used in various applications.

First, the Binary Tree is introduced, a structure where each node has at most two children, forming the basis for more advanced tree types. The binary search tree (BST) is a key focus, emphasising its properties that enable efficient searching, insertion, and deletion operations. These are pivotal in optimising performance in tasks like database querying.

The module also covers specialised trees such as the strictly binary tree, where every internal node has exactly two children, and the complete binary tree, which ensures a balanced structure and optimises space and operations. Various tree traversal techniques, such as In-order, Pre-order, and post-orders, are explained in detail, showing how these methods are used to process tree data effectively.

Advanced structures such as the threaded binary tree, which enhances traversal efficiency by using null pointers, and the self-balancing AVL Tree, which maintains balance after insertions and deletions, are also discussed. Finally, the module introduces B Trees and B+ Trees, widely used in databases and file systems for efficient multi-level indexing and storage.

The module consists of **two** units.

Unit 3.1: Exploring Trees

Unit 3.2: Tree Traversal and Balanced Trees

MODULE 3

Trees

Unit 1

Exploring Trees



■ Unit Table of Contents

Unit 3.1 Exploring Trees

Aim	231
Instructional Objectives	231
Learning Outcomes	231
3.1.1 Introduction and Terminology of Tree	232
Self-Assessment Questions	237
3.1.2 Binary Search Tree	238
Self-Assessment Questions	246
3.1.3 Strictly and Complete Binary Tree	247
Self-Assessment Questions	251
Summary	252
Terminal Questions	252
Answer Keys	253
Activity	253
Glossary	254
Bibliography	254
e-References	254
Video Links	255
Image Credits	255
Keywords	255



Aim

To enable students to interpret the fundamental concept of binary tree types, including binary trees, binary search trees, strictly binary trees, and complete binary trees.



Instructional Objectives

This unit is designed to:

- Define the basic concept of a binary tree and its various types
- Explain the structure, properties, and applications of a binary search tree
- Describe the characteristics and properties of a strictly binary tree (full binary tree)



Learning Outcomes

At the end of the unit, the student is expected to:

- Examine the different terminologies in tree
- Perform insertion, deletion, and search operations efficiently in a binary search tree
- Analyse the structure of complete binary trees and their importance in applications

3.1.1 Introduction and Terminology of Tree

A tree is a hierarchical data structure comprising nodes connected by edges. The data within a tree is organised hierarchically, making it a widely used abstract data type for representing hierarchical relationships. Each node in a tree can connect to multiple child nodes. Unlike linear data structures, a tree has a non-linear form, where edges link nodes. In other types of data structures, performing operations can become more complex as the data size grows. Still, trees allow for faster data access, making them an efficient option for handling large datasets.

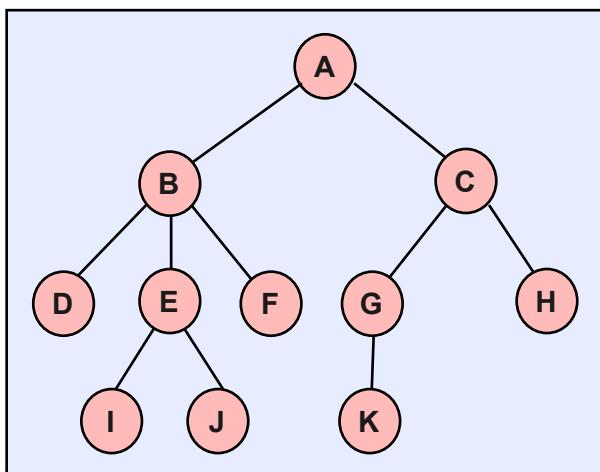


Fig. 1: Tree with 11 nodes and 10 edges

Tree Terminologies:

Root:

- The root node is the starting point of a tree from which all other nodes originate. A tree can only have one root node; multiple root nodes are not allowed.

Edge:

- An edge is the connection between two nodes in a tree. A tree with “n” nodes will have exactly “n-1” edges.

Parent:

- A parent node has branches leading to other nodes, meaning it has one or more child nodes.
A parent can have any number of children in a tree.

Child:

- A child node is any node that is a descendant of another node. All nodes in the tree, except the root, are child nodes.

Siblings:

- Nodes that share the same parent are called sibling nodes. They belong to the same parent.

Degree:

- The degree of a node refers to the number of child nodes it has. The node with the highest degree determines the degree of a tree.

Leaf Node:

- A node with no children is referred to as a leaf node. Leaf nodes are also known as external or terminal nodes.

Level:

- Each step from the top of the tree (root) downwards is called a level. Level numbering begins at 0 and increases by 1 with each step-down.

Height:

- The height of a node is the number of edges on the longest path from any leaf node to that node. The height of the tree is the height of its root node. All leaf nodes have a height of 0.

Depth:

- The depth of a node is defined as the number of edges between that node and the root. The depth of the tree is the total number of edges from the root to a leaf node along the longest path. The root node has a depth of 0. “Level” and “depth” are often used interchangeably.

Subtree:

- In a tree, each child of a node forms its own subtree. Every child node and its descendants form a subtree on its parent node.

Binary Tree

A binary tree is a tree data structure where each parent node can have no more than two children. Every node in a binary tree contains three elements:

- A data item
- A pointer to the left child
- A pointer to the right child

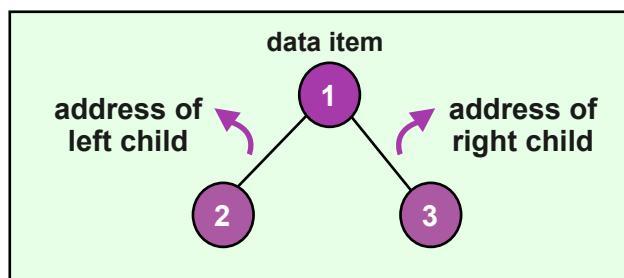


Fig. 2: Binary tree

Types of Binary Trees

1. Full Binary Tree:

- A full binary tree is a special binary tree where each internal or parent node has two or no children.

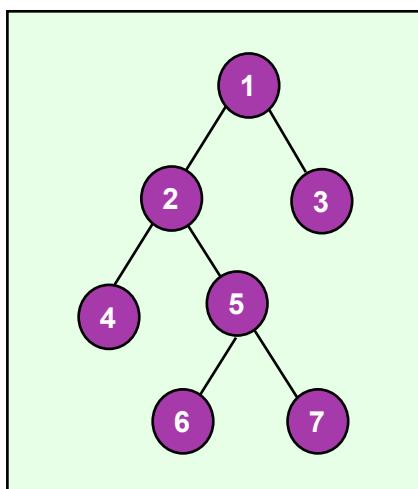


Fig. 3: Full binary tree

2. Perfect Binary Tree:

- A perfect binary tree is one in which all internal nodes have exactly two children and all the leaf nodes are at the same level.

3. Complete Binary Tree:

- A complete binary tree is like a full binary tree but with two key distinctions:
 - All levels must be fully filled.
 - Leaf nodes must be positioned toward the left, and the last leaf node may not have a right sibling. Therefore, a complete binary tree doesn't need a full binary tree.

4. Degenerate or Pathological Tree:

- In a degenerate or pathological tree, every parent node has only one child, either on the left or right.

5. Skewed Binary Tree:

- A skewed binary tree is a degenerate tree with nodes aligned to either the left or right side. This creates two types of skewed trees: left-skewed and right-skewed binary trees.

6. Balanced Binary Tree:

- A balanced binary tree is one in which the height difference between each node's left and right subtrees is 0 or 1.

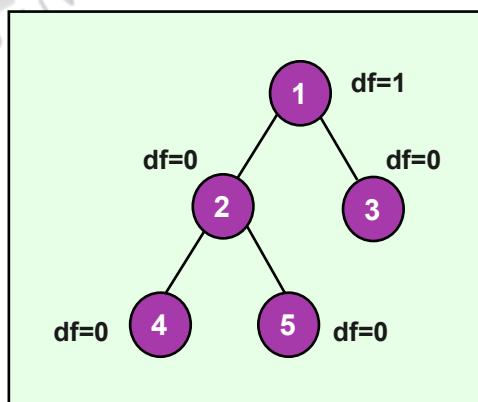


Fig. 4: Balanced binary tree

Binary Tree Representation

A node in a binary tree is represented by a structure that includes the data part and two pointers, each pointing to a node of the same type.

```
struct node
{
    int data;
    struct node *left;
    struct node *right;
};
```

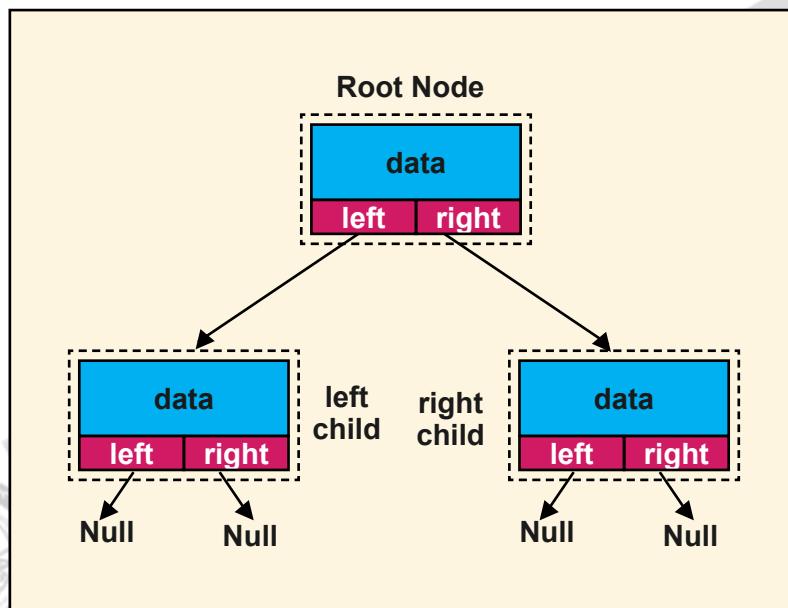


Fig. 5: Binary tree representation

Applications of Binary Trees

- To facilitate quick and efficient access to data
- In router algorithms
- To implement heap data structures
- In syntax trees



Self-Assessment Questions

1. The root node is the _____ node in a tree.
 - a) Leaf
 - b) Bottom-most
 - c) Top-most
 - d) Sibling

2. A node with no children is called a _____ node.
 - a) Root
 - b) Parent
 - c) Leaf
 - d) Sibling

3. In a binary tree, a node can have at most
 - a) 3
 - b) 2
 - c) 4
 - d) 1

3.1.2 Binary Search Tree (BST)

A binary search tree is a specialised tree structure in which each node follows a specific order: the left child must always have a value less than the parent, and the right child must always have a value greater than the parent. This pattern is consistently applied across the left and right subtrees.

Example of a BST

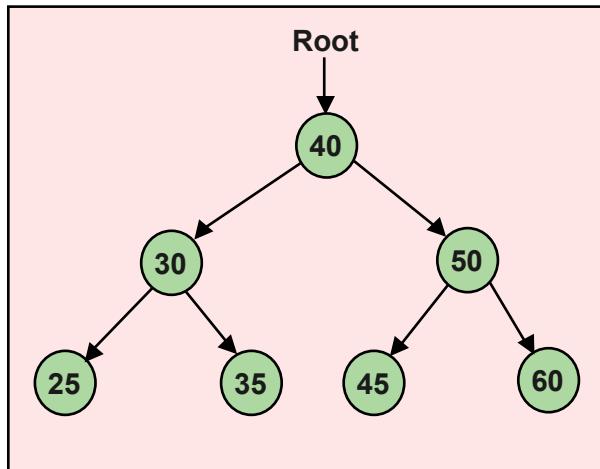
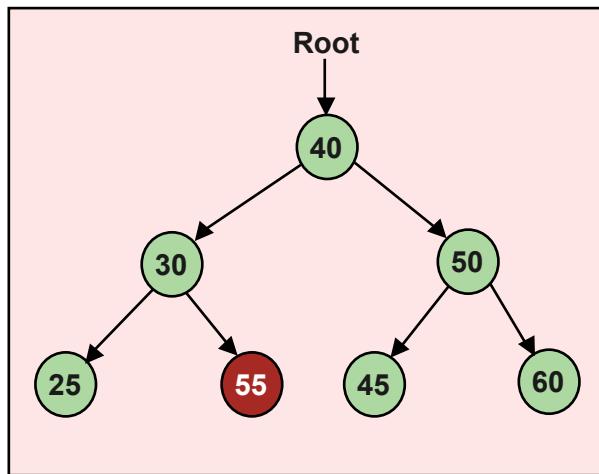


Fig. 6: Binary search tree

In the initial figure, the root node is 40, with the left subtree containing values smaller than 40 and the right subtree containing values larger than 40. The child of 30 (left of 40) also follows the rule—its left child is smaller, and its right child is larger. Hence, the structure in the figure is a valid binary search tree.

If we change the node with value 35 to 55 in the same tree, it no longer satisfies the properties of a binary search tree because 55 is larger than its parent (30) but is incorrectly placed in the left subtree. Thus, the tree becomes invalid as a binary search tree.



Advantages of Binary Search Trees:

- Searching is efficient because the tree's structure gives clues about the location of the target element.
- Insertion and deletion are faster than arrays and linked lists due to the hierarchical organisation.

Example of Creating a Binary Search Tree

Let's create a binary search tree with the following elements: 45, 15, 79, 90, 10, 55, 12, 20, 50.

1. First, insert 45 as the root.
2. Then, insert 15. Since 15 is less than 45, it becomes the left child of 45.
3. Insert 79: As 79 is greater than 45, it becomes the right child of 45.
4. Insert 90: Being greater than both 45 and 79, it becomes the right child of 79.
5. Insert 10: As 10 is less than 45 and less than 15, it becomes the left child of 15.
6. Insert 55: Since 55 is larger than 45 but smaller than 79, it is placed as the left child of 79.
7. Insert 12: 12 is smaller than 45 and 15 but greater than 10, so it becomes the right child of 10.
8. Insert 20: Since 20 is smaller than 45 but greater than 15, it is placed as the right child of 15.
9. Insert 50. Being greater than 45 but smaller than both 79 and 55, it becomes the left child of 55.

Once the binary search tree is fully constructed, we can perform insert, delete, and search operations.

Searching in a Binary Search Tree

Searching involves finding a specific element in the tree. In a BST, searching is straightforward because the values are arranged systematically. Here are the steps for searching:

1. Compare the target element with the root.
2. If the root matches the target, return the node.
3. If not, check whether the target is smaller than the root; move to the left subtree.
4. If larger, move to the right subtree.
5. Repeat this process recursively until the target is found or the tree is fully traversed.
6. If the target is not found, return NULL.

For example, to find node 20 in the tree, you would:

1. Compare 20 with the root (45) and move to the left subtree.
2. Compare 20 with 15; move to the right subtree.
3. Compare 20 with 20 (found).

Algorithm for Searching in a Binary Search Tree:

Search (root, item)

Step 1 - if (item = root → data) or (root = NULL)

return root

else if (item < root → data)

return Search(root → left, item)

else

return Search(root → right, item)

END if

Step 2 – END

Deletion in a Binary Search Tree

Deleting a node from a BST must maintain its structure. There are three cases for deletion:

- 1. The node is a leaf:** In this simplest case, replace the node with NULL and free its memory.
- 2. The node has one child:** Replace the node with its child and delete it.
- 3. The node has two children:** Replace it with its in-order successor (the smallest element in the right subtree), move it to a leaf position, and delete it.

Insertion in a Binary Search Tree

Inserting a new key always occurs at the leaf. The procedure is like searching:

1. Start at the root.
2. find an empty spot in the left subtree if the new key is smaller.
3. If the key is larger, search for a place in the right subtree and insert the value.

The complexity of the Binary Search tree

Let's explore the binary search tree's (BST) time and space complexity. We'll examine the time complexity for insertion, deletion, and search operations in the best, average, and worst-case scenarios.

1. Time Complexity

Operations	Best Case Time Complexity	Average Case Time Complexity	Worst Case Time Complexity
Insertion	$O(\log n)$	$O(\log n)$	$O(n)$
Deletion	$O(\log n)$	$O(\log n)$	$O(n)$
Search	$O(\log n)$	$O(\log n)$	$O(n)$

Where 'n' represents the number of nodes in the binary search tree.

2. Space Complexity

Operations	Space Complexity
Insertion	$O(n)$
Deletion	$O(n)$
Search	$O(n)$

The space complexity for any operation in a binary search tree is $O(n)$.

Implementation of BST:

```
#include <stdio.h>
#include <stdlib.h>

struct node {
    int data;
    struct node *right_child;
    struct node *left_child;
};

struct node* new_node(int x){
    struct node *temp;
    temp = malloc(sizeof(struct node));
    temp->data = x;
    temp->left_child = NULL;
    temp->right_child = NULL;
    return temp;
}

struct node* search(struct node * root, int x){
    if (root == NULL || root->data == x)
        return root;
    else if (x > root->data)
        return search(root->right_child, x);
    else
        return search(root->left_child, x);
}

struct node* insert(struct node * root, int x){
    if (root == NULL)
        return new_node(x);
    else if (x > root->data)
        root->right_child = insert(root->right_child, x);
```

```

else
    root -> left_child = insert(root->left_child, x);
return root;
}

struct node* find_minimum(struct node * root) {
if (root == NULL)
    return NULL;
else if (root->left_child != NULL)
    return find_minimum(root->left_child);
return root;
}

struct node* delete(struct node * root, int x) {
if (root == NULL)
    return NULL;
if (x > root->data)
    root->right_child = delete(root->right_child, x);
else if (x < root->data)
    root->left_child = delete(root->left_child, x);
else {
    if (root->left_child == NULL && root->right_child == NULL){
        free(root);
        return NULL;
    }
    else if (root->left_child == NULL || root->right_child == NULL){
        struct node *temp;
        if (root->left_child == NULL)
            temp = root->right_child;
        else
            temp = root->left_child;
        free(root);
        return temp;
    }
}

```

```
else {
    struct node *temp = find_minimum(root->right_child);
    root->data = temp->data;
    root->right_child = delete(root->right_child, temp->data);
}
}

return root;
}

void inorder(struct node *root){
if (root != NULL)
{
inorder(root->left_child);
printf(" %d ", root->data);
inorder(root->right_child);
}
}

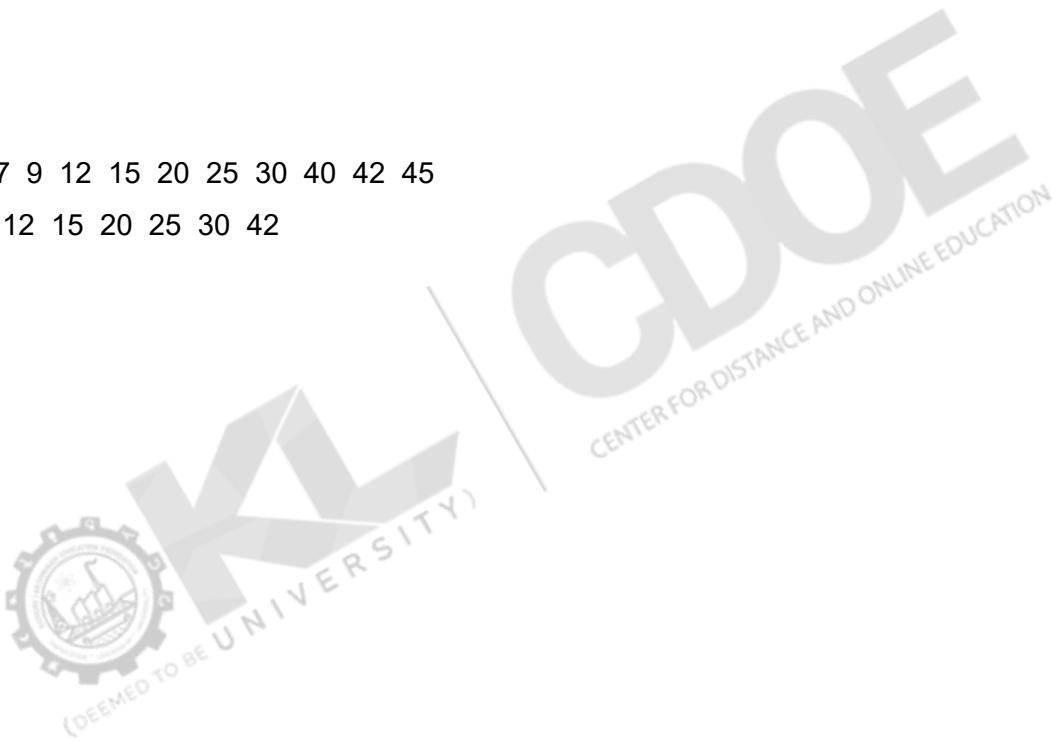
int main() {
struct node *root;
root = new_node(20);
insert(root, 5);
insert(root, 1);
insert(root, 15);
insert(root, 9);
insert(root, 7);
insert(root, 12);
insert(root, 30);
insert(root, 25);
insert(root, 40);
insert(root, 45);
insert(root, 42);
```

```
inorder(root);
printf("\n");

root = delete(root, 1);
root = delete(root, 40);
root = delete(root, 45);
root = delete(root, 9);
inorder(root);
printf("\n");
return 0;
}
```

Output

```
1 5 7 9 12 15 20 25 30 40 42 45
5 7 12 15 20 25 30 42
```





Self-Assessment Questions

4. In a binary search tree, searching, insertion, and deletion have a worst-case time complexity of _____.
- a) $O(\log n)$
 - b) $O(n)$
 - c) $O(n^2)$
 - d) $O(n \log n)$
5. The node with the smallest value in a binary search tree is in the _____ subtree of the
- a) tree.
 - b) Right
 - c) Root
 - d) Left
 - e) Both
6. In a binary search tree, if a node has two children, its _____ successor is used to replace it when the node is deleted.
- a) Pre-order
 - b) In-order
 - c) Post-order
 - d) Level-order
7. A binary search tree always guarantees a balanced structure.
- a) True
 - b) False

3.1.3 Strictly and Complete Binary Tree

Strictly Binary Tree

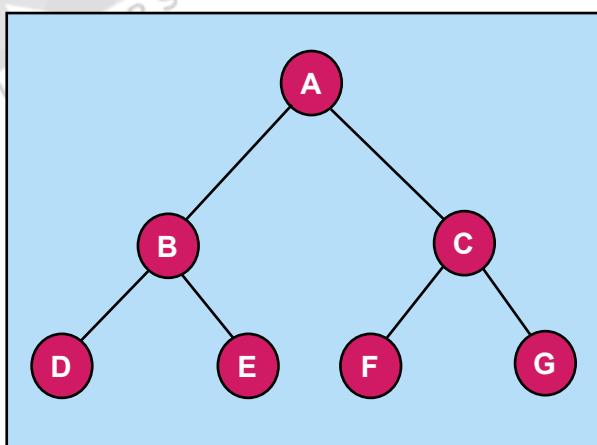
A Strictly Binary Tree, also known as a Full Binary Tree, is a binary tree where every node has exactly two or no children. This means that each internal node (non-leaf node) will always have two children, while the leaf nodes (the nodes without children) will have none. A strictly binary tree ensures that the tree is fully populated with no missing branches or gaps at any node level, except for the leaves at the last level.

Characteristics of Strictly Binary Tree:

- 1. Internal Nodes:** Every internal node must have exactly two children - one on the left and one on the right.
- 2. Leaf Nodes:** All the leaf nodes have zero children (no further branches).
- 3. Height Balance:** The tree may not be height-balanced (i.e., the depth of the left subtree and the right subtree may differ), but if each internal node has two children, it remains a strictly binary tree.
- 4. Tree Density:** Strictly binary trees are denser than other tree types because they fully utilise each level of the tree, excluding the last level (which only contains leaves).

Example:

Consider a strictly binary tree where every internal node has two children:



- In this tree, node A has two children (B and C).
- Nodes B and C each have two children: D, E for B and F, G for C.
- Nodes D, E, F, and G are leaf nodes, having no children.

This is a strictly binary tree because each internal node has exactly two children, and the leaf nodes have zero children.

Properties of Strictly Binary Tree:

- **Number of Nodes:** A strictly binary tree with n internal nodes will always have $2n + 1$ total nodes (including both internal nodes and leaves).
- **Leaf Nodes and Internal Nodes:** A strictly binary tree with I internal nodes has exactly $I + 1$ leaf nodes.
- **Height of the Tree:** The minimum height of a strictly binary tree is $\log(n + 1)$, where n is the number of nodes in the tree. The maximum height depends on the arrangement of the tree, but it adheres to the rule that each node has two children.

Advantages:

- Strictly binary trees are highly structured, which can simplify traversal algorithms and operations like searching, inserting, or deleting nodes.

Disadvantages:

- Strictly binary trees can be inefficient in certain use cases, especially when there is a need to balance the tree for optimal search time in AVL trees or Red-Black trees).

Complete Binary Tree

A Complete Binary Tree is a binary tree where all levels are fully filled except possibly for the last level, which must be filled from left to right. In other words, in a complete binary tree:

- Nodes fully occupy every level up to the second-last.
- The nodes on the last level are filled from left to right without any gaps in the structure.

The main characteristic of a complete binary tree is that it is “left-aligned,” meaning that any missing nodes must be on the far-right side of the last level, ensuring that there are no gaps before this.

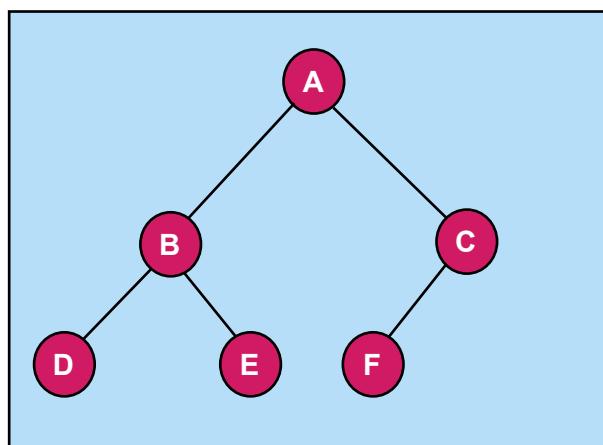
Characteristics of Complete Binary Tree:

- **Complete Levels:** All levels except the last are filled.
- **Left-Alignment:** Nodes in the last level are placed as far left as possible, with any missing nodes only appearing on the far right.

- Efficient Structure: Complete binary trees are efficient for storage and traversal because of their compact nature. They are often used to implement binary heaps (used in priority queues).

Example:

Consider a complete binary tree:



- The first two levels (root and second levels) are filled.
- The third level (the last level) has nodes D, E, and F, filled from the left without any gaps on the right.
- This is a complete binary tree because all the nodes are filled from the left, and the levels above the last one are fully populated.

Properties of Complete Binary Tree:

- Height:** The height h of a complete binary tree with n nodes is always $O(\log n)$, making it an efficient structure for various operations.
- Number of Nodes:** The number of nodes in a complete binary tree is at least 2^h and at most $2^{h+1} - 1$, where h is the tree's height.

- **Node Indices:** In a complete binary tree, if the nodes are numbered from 1 to n, then for a node at position i:
 - o The left child is located at position 2i.
 - o The right child is located at position 2i + 1.
 - o The parent is located at position floor(i / 2).

Applications:

Complete binary trees are often used in:

- **Binary Heaps:** A complete binary tree structure is ideal for binary heaps, which are used in heap sort and priority queues because of the tree's compactness and efficient heap property (either max-heap or min-heap).
- **Binary Search Trees:** A complete binary tree ensures balanced height and efficient searching.
- **Storage:** The array representation of complete binary trees is efficient because any node's left and right children can be directly calculated using simple arithmetic operations on the node's index.

Advantages:

- **Space Efficient:** Complete binary trees are highly space-efficient, minimising the use of empty nodes.
- **Height Minimisation:** The height of a complete binary tree is minimised, which leads to more efficient searching, insertion, and deletion operations compared to unbalanced trees.

Disadvantages:

- **Strict Balancing:** Complete binary trees are not always perfectly balanced, which can slightly reduce their efficiency for searching compared to perfectly balanced trees like AVL trees.



Self-Assessment Questions

8. Which operations can be efficiently implemented using a complete binary tree?

- a) Breadth-First Search
- b) Depth-First Search
- c) Merge Sort
- d) Heap Sort

9. A complete binary tree with 7 nodes has a height of:

- a) 3
- b) 2
- c) 7
- d) 4

10. In a strictly binary tree, each internal node must have exactly _____ children.

- a) One
- b) Two
- c) Three
- d) None



Summary

- A tree is a hierarchical, non-linear data structure with nodes connected by edges. Unlike linear data structures, trees provide efficient data access for large datasets.
- Various terminologies define tree components, such as root, edge, parent, child, sibling, degree, and subtree.
- Binary trees are a special type where each parent node can have no more than two children.
- Different binary tree types include full, perfect, complete, degenerate, skewed, and balanced binary trees.
- Binary trees are used in applications requiring efficient data access, like router algorithms, heap data structures, and syntax trees.
- A Binary Search Tree (BST) is a specialised hierarchical tree data structure in which each node follows a specific order: the left child's value is always less than its parent's, and the right child's value is always greater.
- A strictly binary tree is a binary tree where every node has exactly two children or none.
- The number of nodes is $2n + 1$, where n is the number of internal nodes, and the number of leaf nodes is $l + 1$, where l is the number of internal nodes.
- A complete binary tree is a binary tree where all levels are fully filled, except possibly the last one, which must be filled from left to right.
- Height is $O(\log n)$, and node indices are calculated so that the left child is at position $2i$ and the right child is at $2i + 1$.



Terminal Questions

1. Explain the concept of a balanced binary tree and its importance.
2. Describe the structure of a binary tree node and explain how binary trees are represented in programming.
3. What is a Binary Search Tree (BST)? Explain its structure and properties.
4. Describe the process of searching for an element in a Binary Search Tree.
5. Define a Strictly Binary Tree and explain its key characteristics.
6. What is a Complete Binary Tree? List its properties.



Answer Keys

Self-Assessment Questions	
Question No.	Answer
1	B
2	C
3	B
4	B
5	C
6	B
7	B
8	D
9	A
10	B



Activity

Activity Type: Offline

Duration: 1 hour

Imagine being given an unsorted list of numbers, for example [7, 3, 10, 1, 5, 8, 12, 4, 6] and asked to construct a binary search tree. How would you decide the root node of the tree? What criteria would you use to determine whether a number should be placed on a parent node's left or right-side during insertion? Discuss the advantages of using a Binary Search Tree over other data structures for searching and retrieving data efficiently.



Glossary

- **Time Complexity:** A measure of the time an algorithm takes to run as a function of the input size.
- **Space Complexity:** A measure of the amount of memory an algorithm requires as a function of the input size.
- **In-order Successor:** The node with the smallest value in the right subtree of a node, used in deletion operations.
- **Root Node:** The topmost node in a tree, serving as the starting point for traversal.
- **Internal Node:** A binary tree node with at least one child.



Bibliography

Textbooks

- Balagurusamy, E. (2017). *Data Structures*. McGraw Hill Education.
- Sedgewick, R., & Wayne, K. (2014). *Algorithms II*. Pearson Education.
- Sridhar, S. (2014). *Design and Analysis of Algorithms*. Oxford University Press.
- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (1989). *Introduction to Algorithms*. Prentice-Hall.



e-References

- **Introduction to Tree Data Structure:** <https://www.geeksforgeeks.org/introduction-to-tree-data-structure/>
- **Binary Tree:** <https://www.javatpoint.com/binary-tree>
- **Binary Search tree:** <https://www.javatpoint.com/binary-search-tree>
- **Difference Between Strictly and Complete Binary Tree:** <https://www.naukri.com/code360/library/what-is-the-difference-between-full-and-complete-binary-tree>



Video Links

Video	Links
Trees In Data Structure	https://www.youtube.com/watch?v=9oTV7fDEaCY
Binary Search Tree	https://www.youtube.com/watch?v=DU5g_dEbSyQ



Image Credits

- **Fig. 1:** Self-Made
- **Fig. 2:** Self-Made
- **Fig. 3:** Self-Made
- **Fig. 4:** Self-Made
- **Fig. 5:** Self-Made
- **Fig. 6:** Self-Made



Keywords

- Binary tree
- Binary search tree
- Recursive algorithm
- Strictly binary tree
- Complete binary tree
- Binary heap
- Node indices

MODULE 3

Trees

Unit 2

Tree Traversal and Balanced Trees



■ Unit Table of Contents

Unit 3.2 Tree Traversal and Balanced Trees

Aim	258
Instructional Objectives	258
Learning Outcomes	258
3.2.1 Tree Traversal	259
Self-Assessment Questions	263
3.2.2 Threaded Binary Tree	264
Self-Assessment Questions	270
3.2.3 Height Balanced Trees (AVL TREES)	271
Self-Assessment Questions	279
3.2.4 B Tree and B+ Tree	280
Self-Assessment Questions	292
Summary	293
Terminal Questions	293
Answer Keys	294
Activity	294
Glossary	295
Bibliography	295
e-References	295
Video Links	296
Image Credits	296
Keywords	297



Aim

To enable students to interpret tree traversal techniques, threaded binary trees, and height-balanced trees (AVL trees).



Instructional Objectives

This unit is designed to:

- Describe various tree traversal methods such as in-order, pre-order, and post-order traversals
- Discuss the insertion, deletion, and traversal operations in threaded binary trees
- Explain the AVL rotations (Left-of-Left, Right-of-Right, Right-of-Left, and Left-of-Right) and their implementation



Learning Outcomes

At the end of the unit, the student is expected to:

- Visualise and implement tree traversal algorithms using appropriate data structures
- Apply the concept of threaded binary trees and explain how they differ from traditional binary trees
- Assess the concept of height-balanced trees and how AVL trees ensure balance through rotations

3.2.1 Tree Traversal

Traversing a tree involves visiting every node in it. For instance, if you want to add up all the values or find the largest one, you must visit each node.

Unlike linear data structures such as arrays, stacks, queues, and linked lists, which have only one way to read their elements, a hierarchical tree can be traversed in multiple ways.

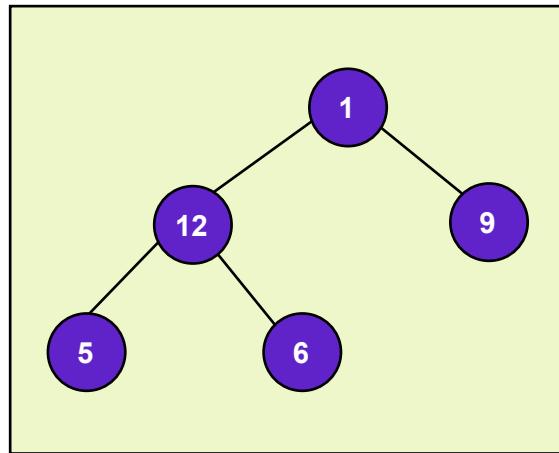


Fig. 1: Tree traversal

Let's consider how we might read the elements of the tree shown above:

Starting from the top and reading left to right:

1 -> 12 -> 5 -> 6 -> 9

Starting from the bottom and reading left to right:

5 -6 -> 12 -> 9 -> 1

Although this process is somewhat easy, it doesn't respect the tree's hierarchy, only the nodes' depth.

Instead, we use traversal methods that consider the basic structure of a tree, i.e.

```
struct node {
    int data;
    struct node* left;
    struct node* right;
}
```

This simple approach doesn't respect the tree's hierarchical structure, focusing only on node depth. Instead, we use tree traversal methods that align with the tree's structure.

A tree typically consists of:

- A node that holds data
- Two subtrees: Left Subtree and Right Subtree

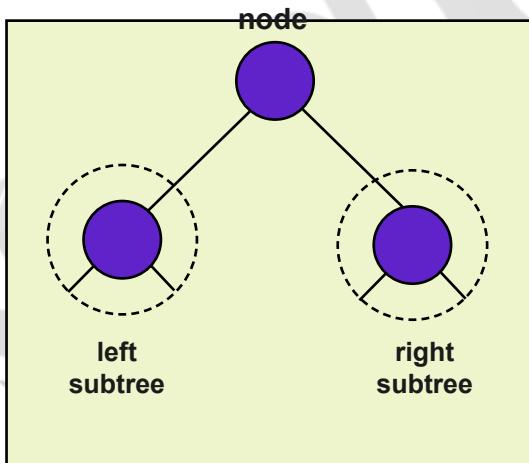


Fig. 2: Left and right subtree

Left and Right Subtree

Remember that our goal is to visit each node, so we need to visit all the nodes in the subtree, the root node, and all the nodes in the right subtree as well.

Given this structure, we must visit all the nodes in both subtrees and the root node. Depending on the order in which we visit the root and subtrees, there are three main traversal methods:

In- order Traversal

1. Visit all the nodes in the left subtree.
2. Visit the root node.
3. Visit all the nodes in the right subtree.

```
inorder(root->left)
display(root->data)
inorder(root->right)
```

Preorder Traversal

1. Visit the root node.
2. Visit all the nodes in the left subtree.
3. Visit all the nodes in the right subtree.

```
display(root->data)
preorder(root->left)
preorder(root->right)
```

CDOE

CENTER FOR DISTANCE AND ONLINE EDUCATION

Post order Traversal

1. Visit all the nodes in the left subtree.
2. Visit all the nodes in the right subtree.
3. Visit the root node.

```
postorder(root->left)
postorder(root->right)
display(root->data)
```

Visualising In-order Traversal

We begin at the root node and first traverse the left subtree.

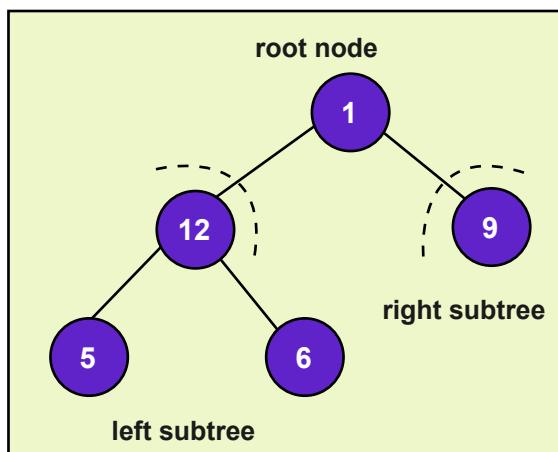


Fig. 1: Tree traversal

Once we've finished the left subtree, we must remember to return to the root node and traverse the right subtree. To manage this, we can use a stack.

1. Start with the root node on the stack.
2. Traverse the left subtree following the in-order rule:

left subtree -> root -> right subtree

Once we reach a leaf node, we can print it directly. For example, if node 5 has no children, we print it. Then, we print its parent node (12) and its right child (6).

By using a stack, we ensure that after completing the left subtree, we can return to the root and continue with the right subtree. After processing all nodes, the in-order traversal for this example would be:

5 -> 12 -> 6 -> 1 -> 9



Self-Assessment Questions

1. A tree traversal involves visiting every _____ in a tree structure.
 - a) Node
 - b) Edge
 - c) Level
 - d) Leaf

2. In-order traversal follows the sequence: left subtree->_____ ->right subtree.
 - a) Left child
 - b) Right child
 - c) Root node
 - d) Parent node

3. Pre-order traversal begins with the _____ node.
 - a) Left child
 - b) Root
 - c) Right child
 - d) Leaf

3.2.2 Threaded Binary Tree

A binary tree is represented using array representation or linked list representation. When a binary tree is represented using linked list representation, we use a NULL pointer in that position if the node does not have a child. There are more NULLs than actual pointers in any binary tree-linked list representation. Generally, in any binary tree linked list representation, if there are $2N$ reference fields, $N+1$ is filled with NULL ($N+1$ is NULL out of $2N$). This NULL pointer does not play any role except indicating no link (no child).

A new threaded binary tree uses NULL pointers to improve its traversal processes. In a threaded binary tree, NULL pointers are replaced by references to other nodes in the tree, called threads.

A. J. Perlis and C. Thornton

A threaded binary tree is defined as follows:

A binary tree is threaded by making all right child pointers that would normally be a null point to the node's in-order successor (if it exists) and all left child pointers that would normally be a null point to the node's in-order predecessor.

Why do we need a Threaded Binary Tree?

Binary trees waste much space: the leaf nodes have 2 null pointers. We can use these pointers to help us in-order traversals. A threaded binary tree makes tree traversal faster since we do not need stack or recursion.

Comparison between a normal binary tree and a threaded binary tree

Criteria	Normal Binary Tree	Threaded Binary Tree
Null Pointers	Null pointers remain unused.	Null pointers are used as threads.
Memory Utilisation	Wastage of memory due to null pointers.	Efficient use of memory by replacing null pointers with threads.
Traversal	Traversal is more complex, requiring stacks or recursion.	Traversal is easier and can be done without stack or recursion.
Structure	Simpler structure.	More complex structure.
Insertion & Deletion	Faster insertion and deletion operations.	Slower insertion and deletion due to complex structure.

Table 1: Difference between normal binary tree and threaded binary tree

Types of threaded binary trees:

Single-threaded

- Each node is threaded towards either the in-order predecessor or successor (left or right), which means all right null pointers will point to the in-order successor OR all left null pointers will point to the in-order predecessor.

Double-threaded

- Each node is threaded towards both the in-order predecessor and successor (left and right), which means all right null pointers will point to in-order success, AND all left null pointers will point to the in-order predecessor.

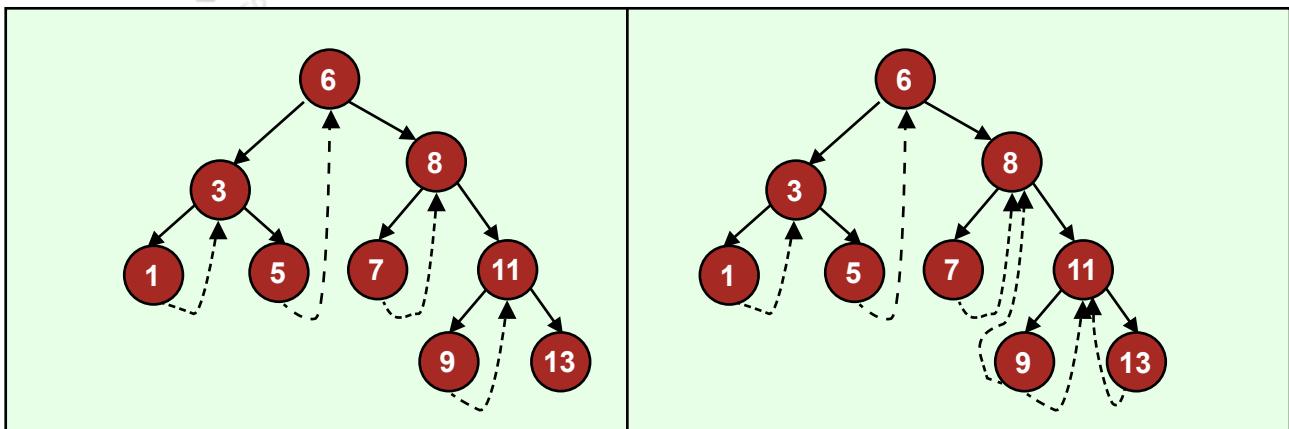
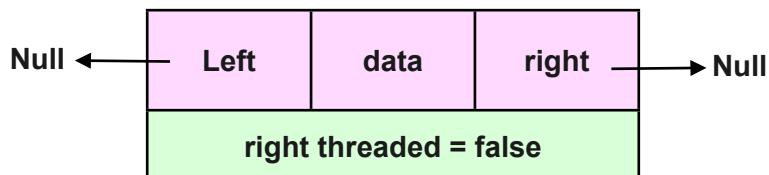


Fig. 4: Single and double-threaded binary trees

Implementation:

Let's see what the Node structure will look like



```

class Node {
    Node left;
    Node right;
    int data;
    boolean rightThread;
    public Node(int data){
        this.data = data;
        rightThread = false;
    }
}
    
```

In a normal BST node, we have left and right references and data, but we have a Boolean in the threaded binary tree. In a normal BST node, we have left and right references and data, but in the threaded binary tree, we have another field called "rightThreaded". This field will tell whether the node's right pointer points to its in-order successor and how we will see it further.

Operations:

- Insert node into the tree
- Print or traverse the tree.

Insert():

The insert operation will be quite like the Insert operation in the Binary search tree with few modifications. Our first task is to find the place to insert a node.

- Take current = root.
- Start from the current and compare root.data with n.

- Always keep track of the parent node while moving left or right.
- if current.data is greater than n, we go to the left of the root; if after moving to the left, the current = null, then we have found the place where we will insert the new node. Add the new node to the left of the parent node and make the right pointer point to the parent node and rightThreaded = true for the new node.

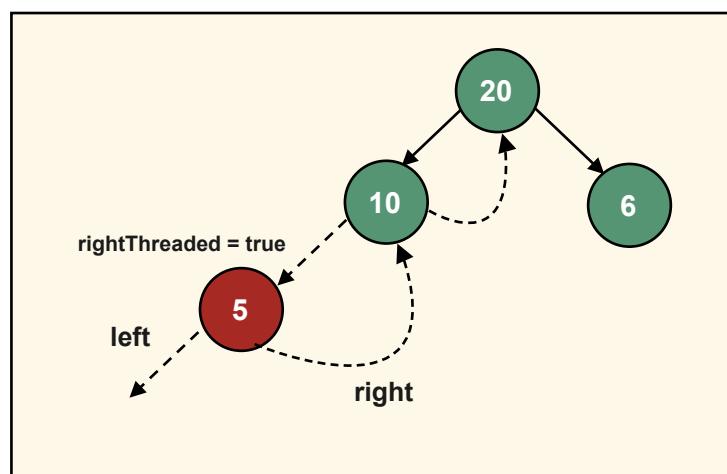


Fig. 5: Inserting node 5

- if current.data is smaller than n, we need to go to the right of the root while going into the right sub-tree and check rightThreaded for the current node, which means the right thread is provided. Points to the in-order successor; if rightThreaded = false, then and current reaches null, just insert the new node else, if right Thread = true. We need to detach the right pointer (store the reference; the new node's right reference will be a point to it) of the current node, make it a point to the new node, and make the right reference point to the stored reference.

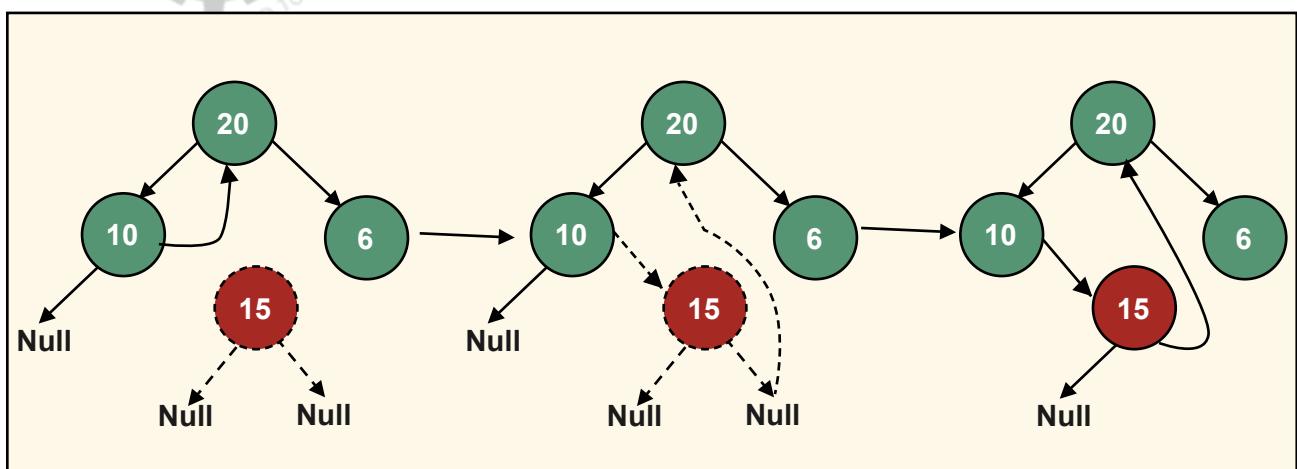


Fig. 6: Inserting node 15 into threaded binary tree

Traverse():

Traversing the threaded binary tree will be quite easy. There is no need for recursion or a stack to store the node. Go to the leftmost node and traverse the tree using the right pointer. Whenever right Thread = false again, go to the leftmost node in the right sub-tree.

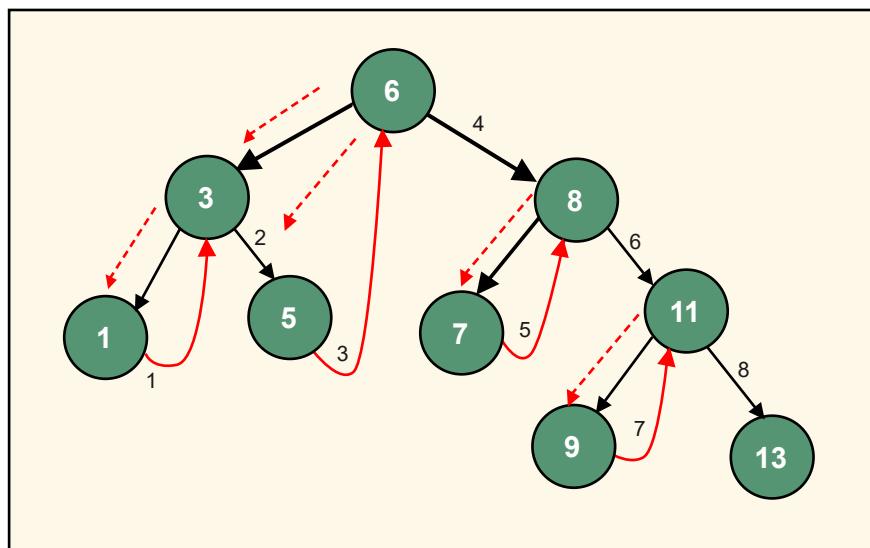


Fig. 7: Traversal of single-threaded binary tree

```

Node leftMost(Node n) {
    Node ans = n;
    if (ans == null) {
        return null;
    }
    while (ans.left != null) {
        ans = ans.left;
    }
    return ans;
}
void inOrder(Node n) {
  
```

```
Node cur = leftmost(n);
while (cur != null) {
    print(cur);
    if (cur.rightThread) {
        cur = cur.right;
    } else {
        cur = leftmost(cur.right);
    }
}
```





Self-Assessment Questions

4. A threaded binary tree replaces NULL pointers with _____ to improve traversal.
- a) References
 - b) Threads
 - c) Leaf nodes
 - d) Parent nodes
5. In a single-threaded binary tree, each node is threaded towards either the in-order _____ or successor.
- a) Parent
 - b) Leaf
 - c) Root
 - d) Predecessor
6. In a normal binary tree, NULL pointers are used only to represent missing children.
- a) True
 - b) False

3.2.3 Height Balanced Trees (AVL TREES)

Researchers Adelson-Velskii and Landis developed height-balanced trees, also called AVL trees. Height balancing attempts to maintain the balance factor of the nodes within a limit.

- **Height of the tree:** The height of a tree is the number of nodes visited in traversing a branch that leads to a leaf node at the deepest level of the tree.
- **Balance factor:** A node's balance factor is defined as the difference between the height of its left and right subtree.

Consider the following tree. The left height of the tree is 5 because traversing the branch that leads to a leaf node at the deepest level of this tree visits 5 nodes (45, 40, 35, 37, and 36).

$$\text{Balance factor} = \text{height of left subtree} - \text{height of the right subtree}$$

The following tree calculates and shows the balance factor for every node. For example, the balance factor of node 35 is $(0-2) = -2$.

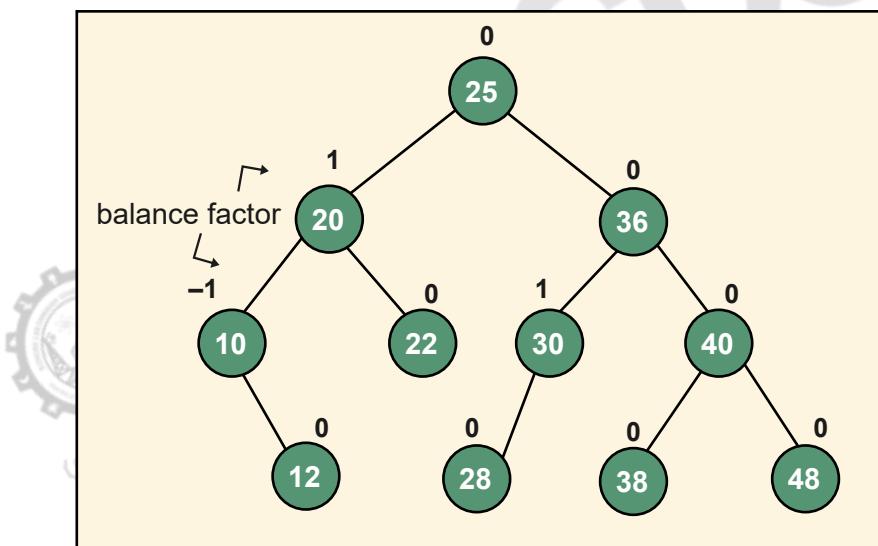


Fig. 8: Example of AVL tree

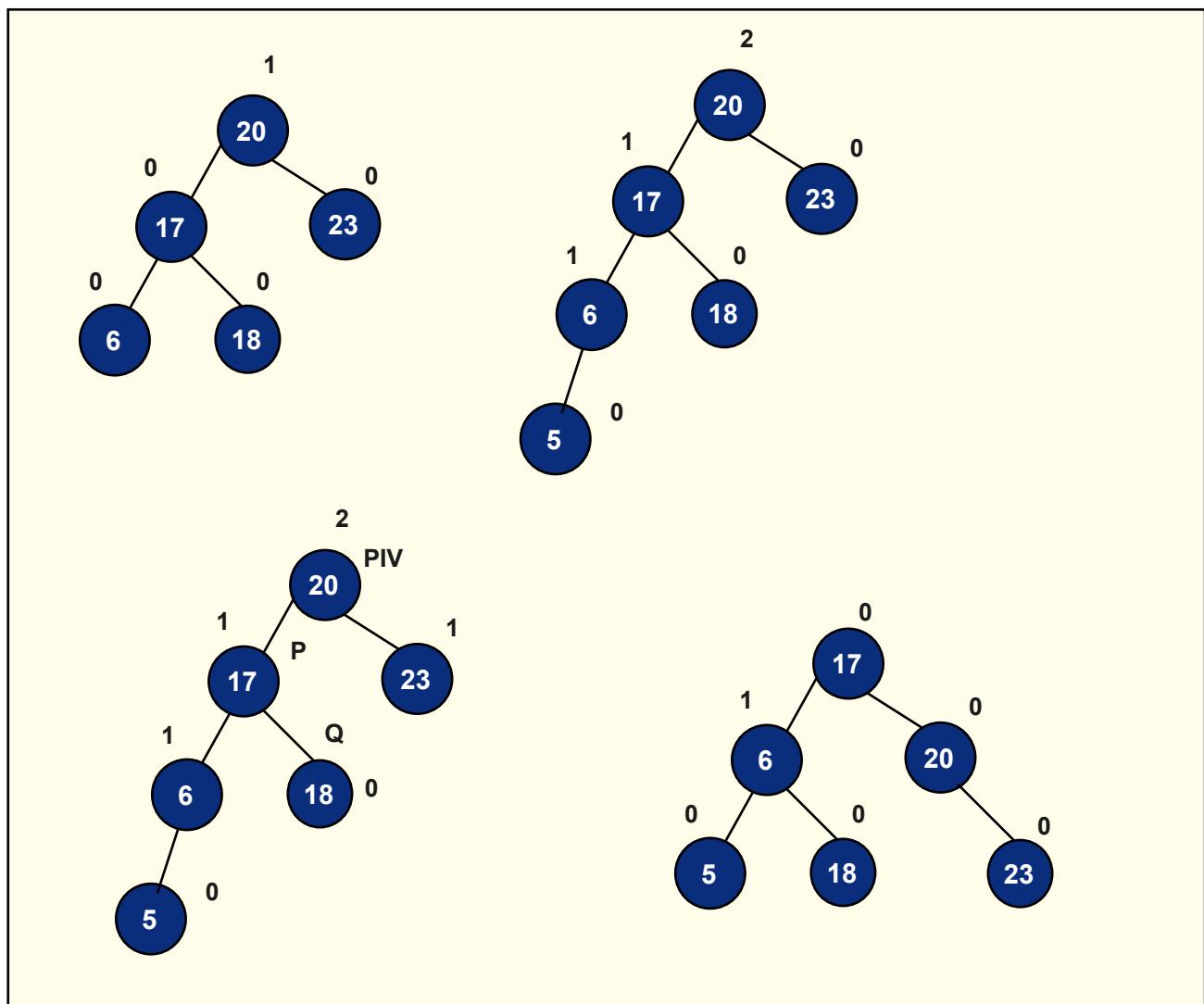
The purpose of going for a binary search tree is to make the search efficient. But when the elements are added to the binary search tree so that one side of the tree becomes heavier, the searching becomes inefficient. The very purpose of going for a binary search tree is not served. Hence, we try to adjust this unbalanced tree to have nodes equally distributed on both sides. This is achieved by rotating the tree using standard algorithms called AVL rotations. After applying AVL rotation, the tree becomes balanced, which is called the AVL or height-balanced tree.

The tree is considered balanced if each node has a balance factor, either -1 or 0, or 1. The tree is considered unbalanced if even one node has a balance factor deviated from these values. There are four types of rotations. They are:

1. Left-of-Left rotation
2. Right-of-Right rotation
3. Right-of-Left rotation
4. Left-of-Right rotation

1. Left-of-Left Rotation

Consider the following tree. Initially, the tree is balanced. Now, a new node 5 is added. Adding the new node makes the tree unbalanced, as the root node has a balance factor of 2. Since this node is disturbing the balance, it is called the pivot node for our rotation. It is observed that the new node was added as the left child to the left subtree of the pivot node. The pointers P and Q are created and made to point to the proper nodes as described by the algorithm. Then, in the next two steps, rotate the tree. The last two steps in the algorithm calculate the new balance factors for the nodes, and it is seen that the tree has become a balanced tree.



Algorithm

LEFT-OF-LEFT (pivot)

P = left(pivot)

Q = right(P)

Root = P

Right(P)=pivot

Left(pivot) = Q

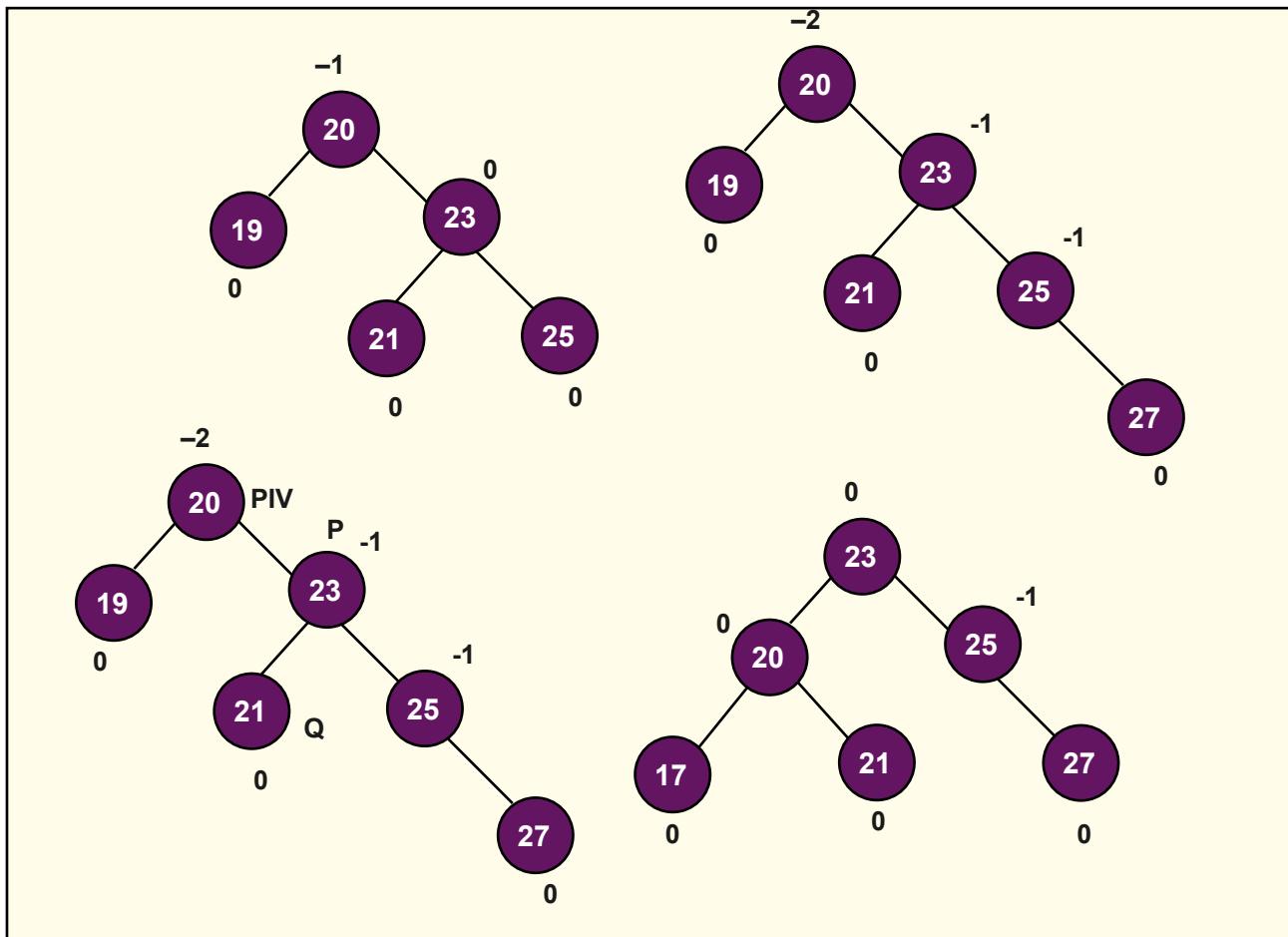
Bal(pivot) = 0

Bal(right(pivot)) = 0

End LEFT-OF-LEFT

2. Right-of-Right Rotation

In this case, the pivot element is fixed as before. The new node is found to be added as the right child to the right subtree of the pivot element. The first two steps in the algorithm set the pointers P and Q to the correct positions. In the next two steps, the tree is rotated to balance it. The last two steps calculate the new balance factor of the nodes.



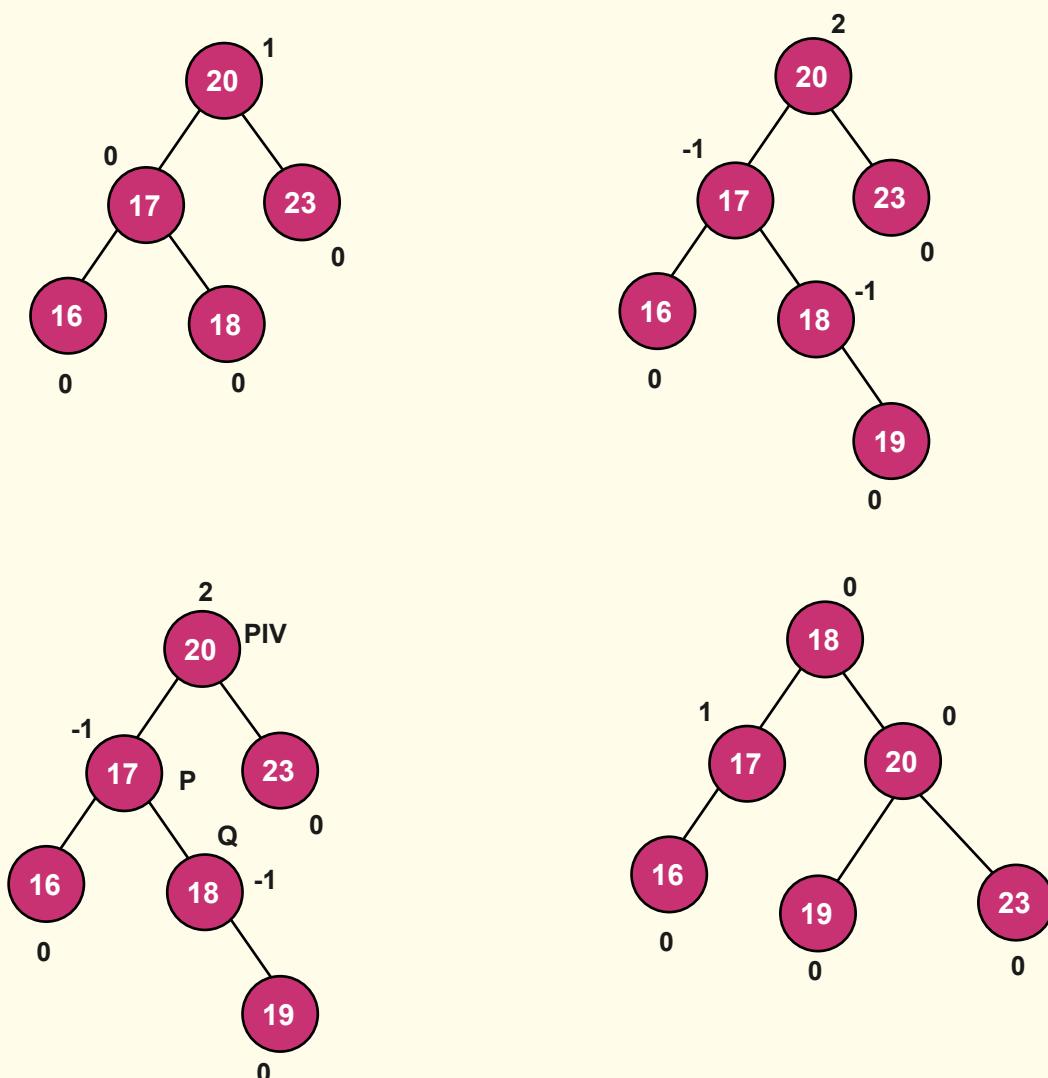
Algorithm

```

RIGHT-OF-RIGHT (pivot)
P = right(pivot)
Q = left(P)
Root = P
Left(P) = pivot
Right(pivot) = Q
Bal(pivot) = 0
Bal(left(pivot)) = 0
End RIGHT-OF-RIGHT
    
```

3. Right-of-Left Rotation

In the following tree, a new node, 19, is added as the right child to the left subtree of the pivot node. Node 20 is fixed as the pivot node, as it disturbs the balance of the tree. In the first two steps, the pointers P and Q are positioned. In the next four steps, the tree is rotated. In the remaining steps, the new balance factors are calculated.

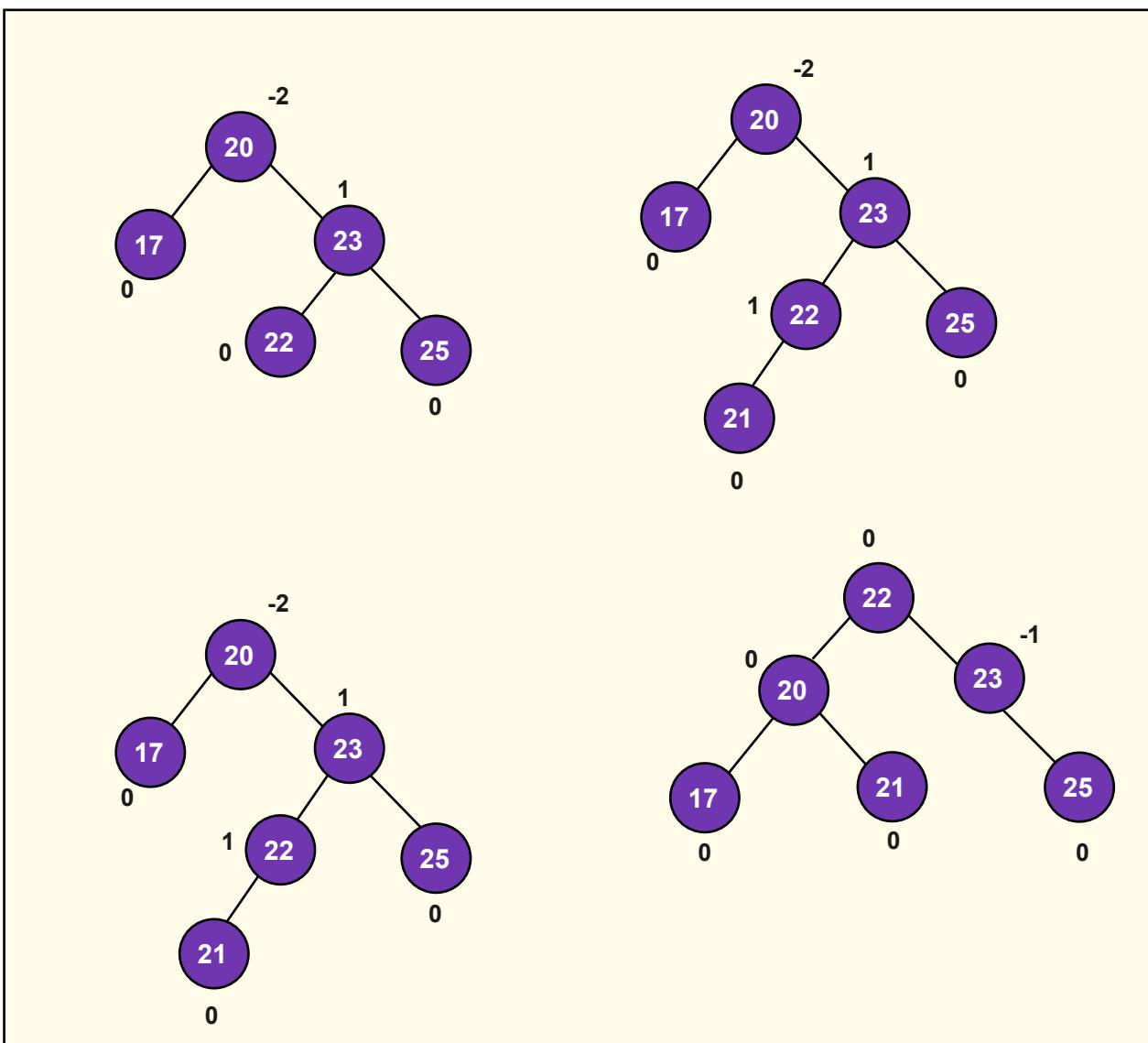


Algorithm

```
RIGHT-OF-LEFT (pivot)
P = left(pivot)
Q = right(P)
Root = Q
Left(Q) = P
Right(Q) = Pivot
Left(pivot) = right(Q)
Right(P) = left(Q)
    If Bal(pivot) = 0
        Bal(left(pivot)) = 0
        Bal(right(pivot)) = 0
    Else
        If Bal(pivot) = 1
            Bal(pivot) = 0
            Bal(left(pivot)) = 0
            Bal(right(pivot)) = -1
        Else
            Bal(pivot) = 0
            Bal(left(pivot)) = 1
            Bal(right(pivot)) = 0
        End if
    End if
End RIGHT-OF-LEFT
```

4. Left-of-Right

In the following tree, a new node 21 is added. The tree becomes unbalanced, and node 20 has a deviated balance factor and is hence fixed as the pivot node. Pointers P and Q are positioned in the algorithm's first two steps. The tree is rotated in the next four steps to make it balanced. The remaining steps calculate the new balance factors for the nodes in the tree.



Algorithm

LEFT-OF-RIGHT (pivot)

P = right(pivot)

Q = left(P)

Root= Q

Right(Q) = P

Left(Q) = Pivot

Right(pivot) = left(Q)

Left(P) = right(Q)

If Bal(pivot) = 0

Bal(right(pivot)) = 0

Bal(left(pivot)) = 0

Else

If Bal(pivot) = 1

Bal(pivot) = 0

Bal(right(pivot)) = 0

Bal(left(pivot)) = -1

Else

Bal(pivot) = 0

Bal(right(pivot)) = 1

Bal(left(pivot)) = 0

End if

End if

End LEFT-OF-RIGHT



Self-Assessment Questions

7. The balance factor of a node in an AVL tree is calculated as the height of the _____ subtree minus the height of the _____ subtree.

- a) Right, left
- b) Left, right
- c) Root, leaf
- d) Root, child

8. An AVL tree remains balanced when each node has a balance factor of _____.

- a) -1, 0, or 1
- b) 0, 1, or 2
- c) 1, 2, or 3
- d) 0, 1, or -2

9. A Right-of-Right rotation occurs when a new node is inserted into the _____ subtree of a node's right child.

- a) Left
- b) Root
- c) Right
- d) Parent

3.2.4 B Tree and B+ Tree

B –Tree

Multiway search tree (m-way search tree): A multiway search tree of order n is a tree in which any node may contain maximum n-1 values and have maximum n children.

Consider the following tree. Every node in the tree has one or more values stored in it. The tree shown is of order 3. Hence, this tree can have a maximum of 3 children, and each node can have a maximum of 2 values. Hence, it is an m-way search tree.

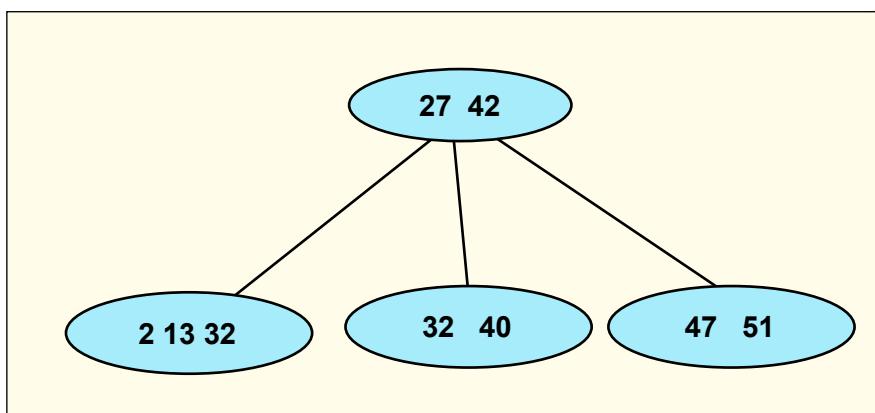
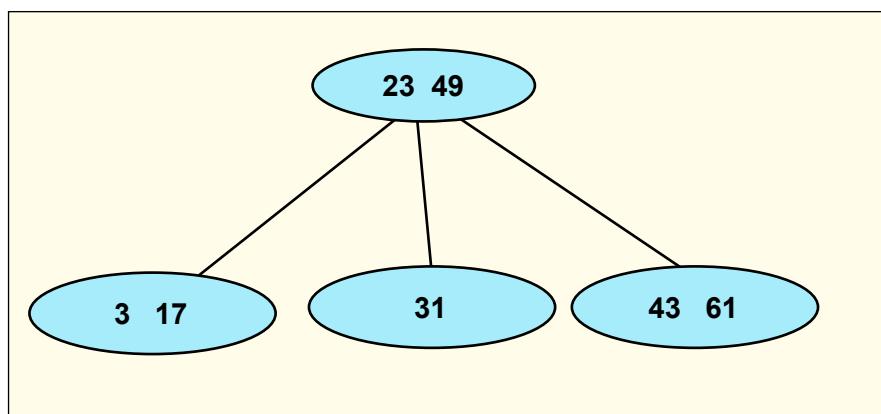


Fig. 9: Example of B-tree

B – tree is a m-way search tree of order n that satisfies the following conditions.

- All non-leaf nodes (except root node) have at least $n/2$ children and a maximum of n children.
- The non-leaf root node may have at least $n/2$ children and a maximum of n children.
- B-tree can exist with only one node, the root node, containing no children.
- A node with n children must have n-1 values.
- All the values on a node's leftmost child are smaller than that node's first value. All values on a node's right-most child are greater than that node's last values.
- If x and y are any two ith and (i+1)th the values of a node, where $x < y$, then all the values appearing on the (i+1)th sub-tree of that node are greater than x and less than y.
- All the leaf nodes should appear on the same level. All the nodes except the root node should have minimum $n/2$ values.

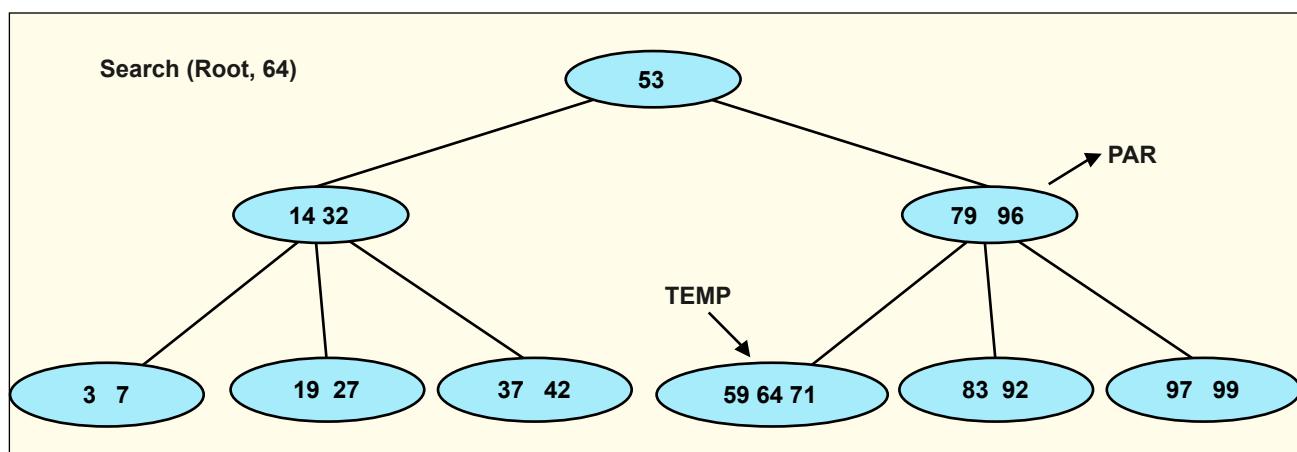
Consider the following tree. It is a m-way search tree of order 3. Let us check whether the above conditions are satisfied. The root node has 3 children and, therefore, has only 2 values stored in it. Also, the elements in the first child (3, 17) are less than the value of the root node's first element (23). The value of the elements in the second child (31) is greater than the value of the first element of the root node (23) and less than the value of the second element (39) in the root node. The value of the elements in the rightmost child (43, 61) is greater than the value of the rightmost element in the root node. All three leaf nodes are at the same level (level 2). Hence, all the conditions specified above are satisfied by the given m-way search tree. Therefore, it is a B-Tree.



Search Operation in a B-Tree

Let us say the number to be searched for is $k = 64$. A temporary pointer temp is made to point to the root node initially. The value $k = 64$ is now compared with each element in the node pointed by temp . If the value is found, then the address of the node where it is found is returned using the temp pointer. If the value of k is greater than an i th element of the node, then the temp is moved to the $i+1$ th node, and the search process is repeated. If the k value is less than the first value in the node, then the temp is moved to the first child. If the k value is greater than the last value of the node, then the temp is moved to the rightmost child of the node, and the search process is repeated.

After the node where the value is found is located (now pointed by temp), a variable LOC is initialised to 0, indicating the position of the value to be searched within that node. The value k is compared with every element of the node. When the value of the k is found within the node, then the search comes to an end position where it is found stored in LOC . If not found, the value of LOC is zero, indicating that the value is not found.



Algorithm

SEARCH(ROOT, k)

Temp = ROOT, i = 1, pos = 0

While i ≤ count(temp) and child[i](temp) ≠ NULL

If k = info(temp[i])

Pos = i

Return temp

Else

If k < info(temp[i])

SEARCH(child[i](temp), k)

Else

If i = count(temp)

Par = temp

Temp = child[i+1](temp)

Else

i = i + 1

End if

End if

End if

End While

While i ≤ count(temp)

```

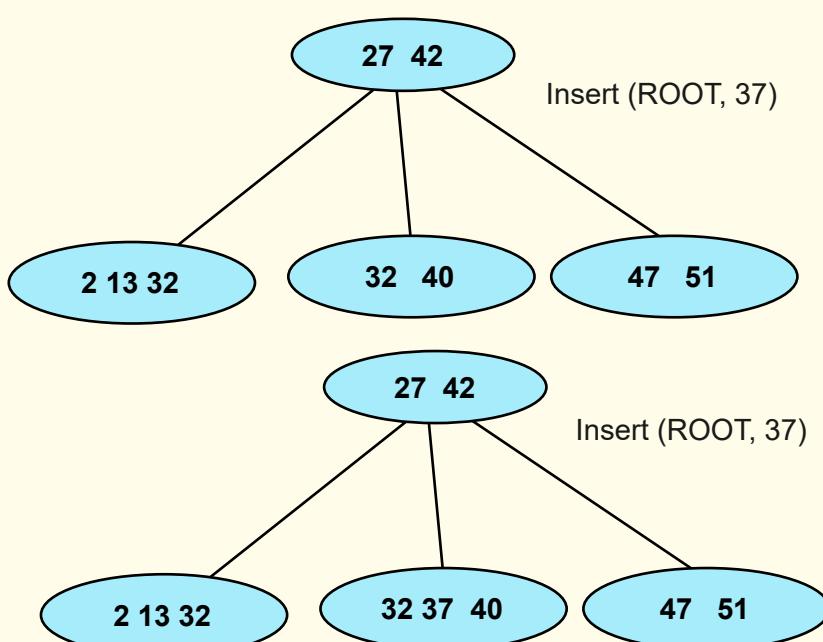
If k = info(temp[i])
Pos = i
Return temp
End if
End while
End SEARCH

```

Insert Operation in a B-Tree

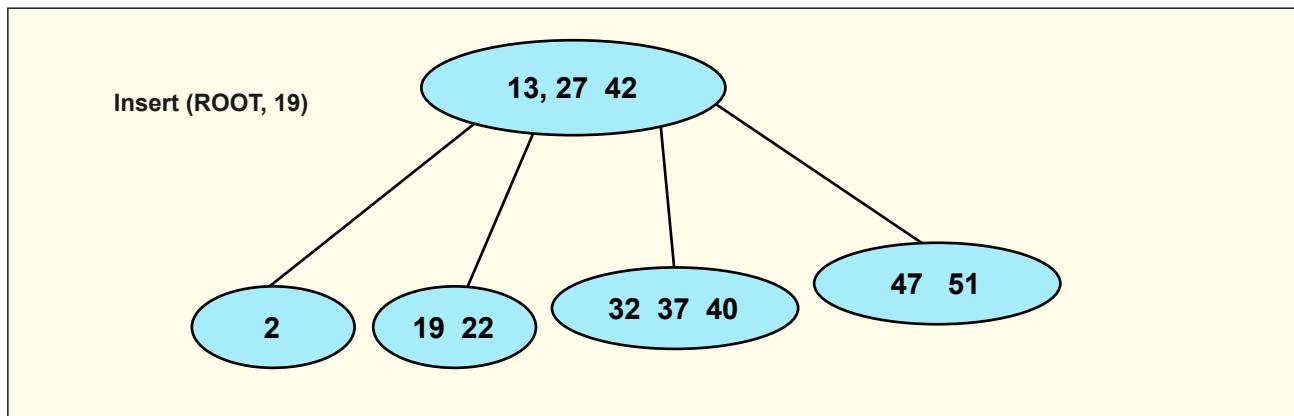
One of the conditions in the B-Tree is that the maximum number of values present in the node of a tree is $n - 1$, where n is the order of the tree. Hence, it should be taken care that, even after insertion, this condition is satisfied. There are two cases: In the first case, the element is inserted into a node that already has less than $n - 1$ values, and in the second case, the element is inserted into a node that already has exactly $n - 1$ values. The first case is a simple one. The insertion into the node does not violate any condition of the tree. But in the second case, if the insertion is done, then after insertion, the number values exceed the limit in that node.

Let us take the first case. In both cases, the insertion is done by searching for that element in the tree that will give the node where it will be inserted. While searching, if the value is already found, then no insertion is done as the B-tree is used for storing key values, and keys do not have duplicates. Now, the value given is inserted into the node. Consider the figure which shows how value 37 is inserted into correct place.



In the second case, insertion is done as explained above. But now, it is found that the number of values in the node after insertion exceeds the maximum limit. Consider the same tree shown above.

Let us insert value 19 into it. After the insertion of value 19, the number of values (2, 13, 19, 22) in that node has become 4. But it is a B-Tree of order 4 in which there should be only a maximum of 3 values per node. Hence, the node is split into two nodes, the first node containing the numbers starting from the first value to the value just before the middle value (first node: 2). The second node will contain the numbers starting just after the mid value till the last value (second node: 19, 22). Mid-value 13 is pushed into the parent.



Algorithm

```

INSERT( ROOT, k )
Temp = SEARCH( ROOT, k )
If count(temp) < n-1
Ins(temp, k)
Return
Else
Repeat for i = n/2 +1 to n-1
Info(R[i-n/2]) = info(temp[i])
Count(R) = count(R) + 1
End repeat
Count(temp) = n/2 – 1
Ins(par, info(temp[m/2]))
End if
INSERT(ROOT, k )
End INSERT

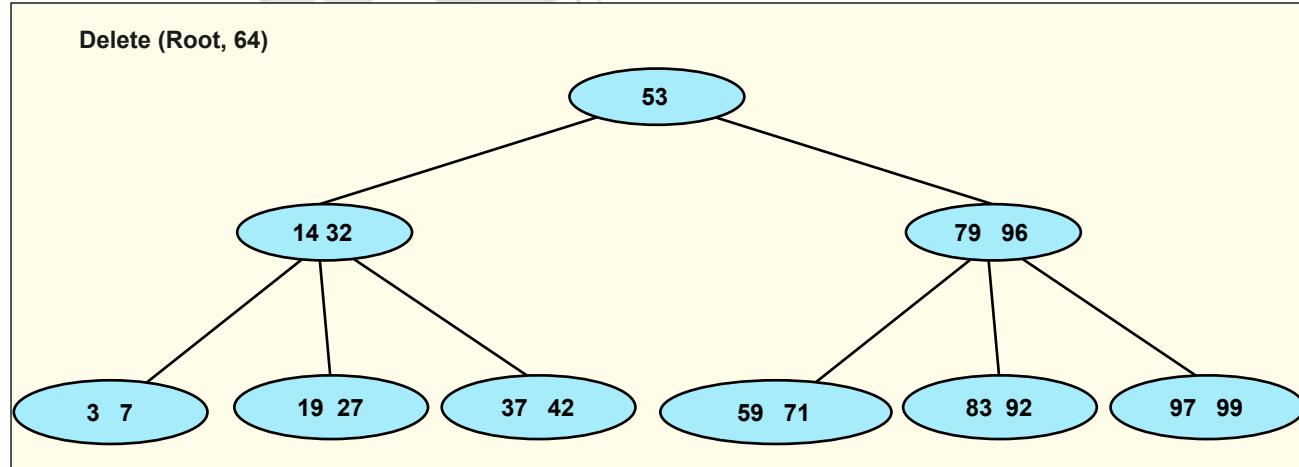
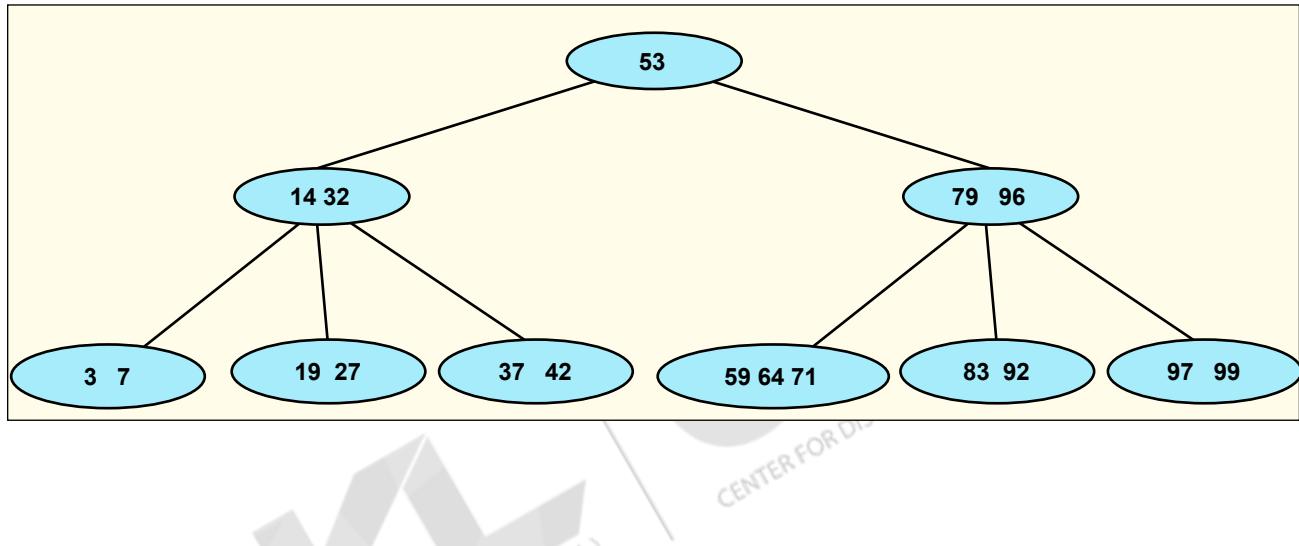
```

Delete Operation in B-Tree

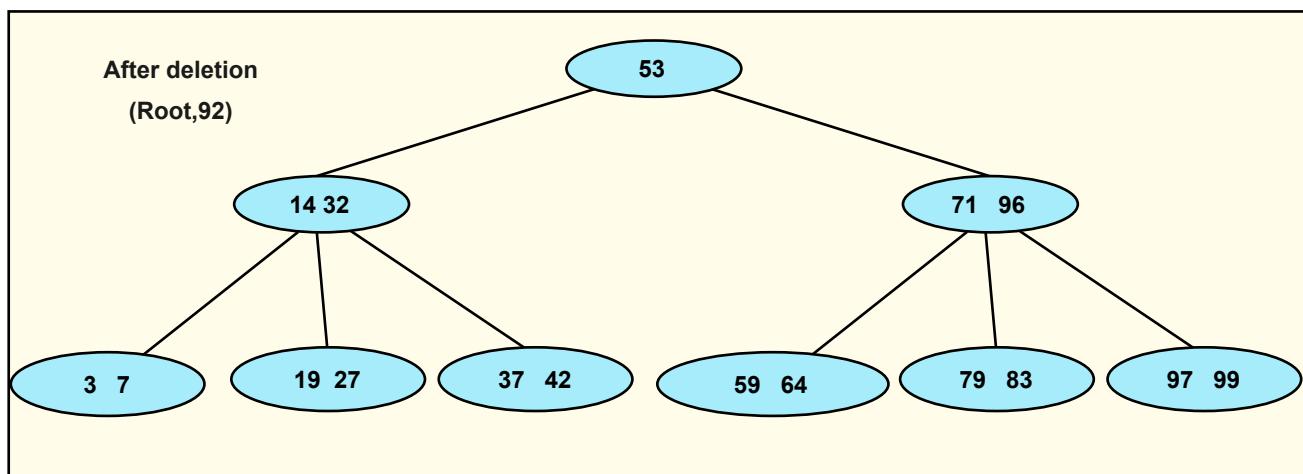
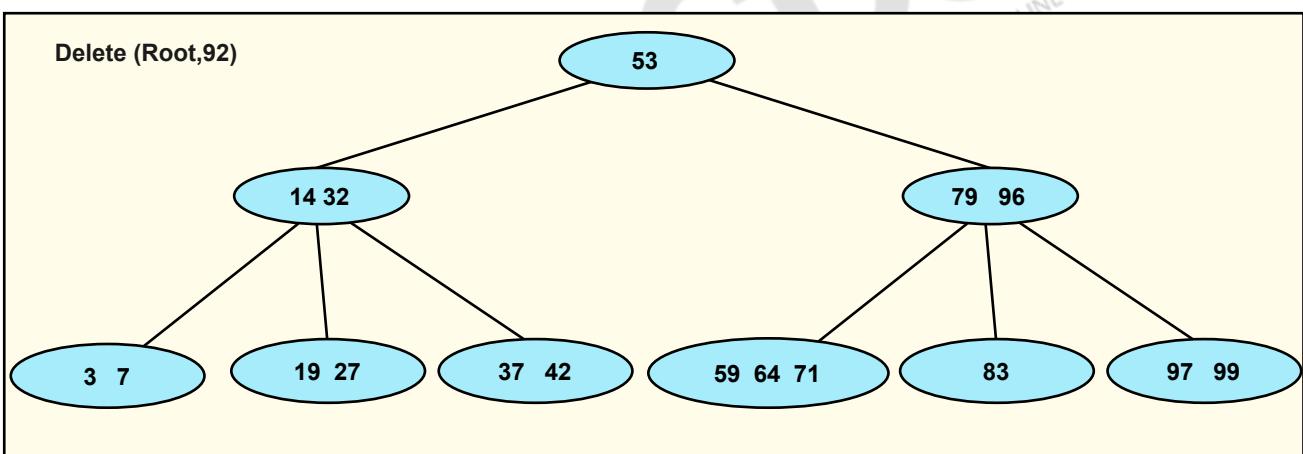
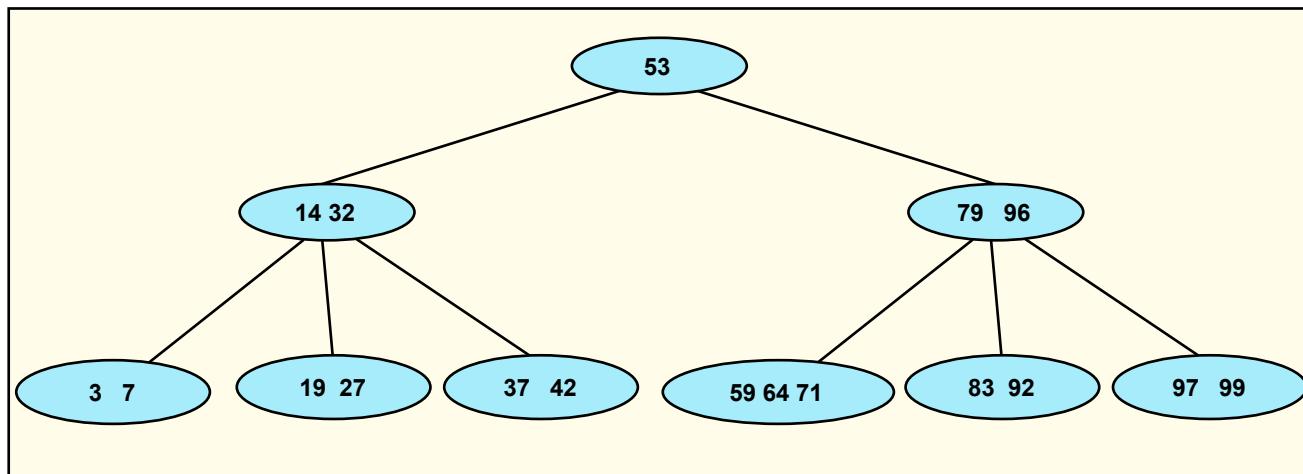
When the delete operation is performed, we should ensure that the node has a minimum $n/2$ value even after deletion, where n is the order of the tree.

There are three cases:

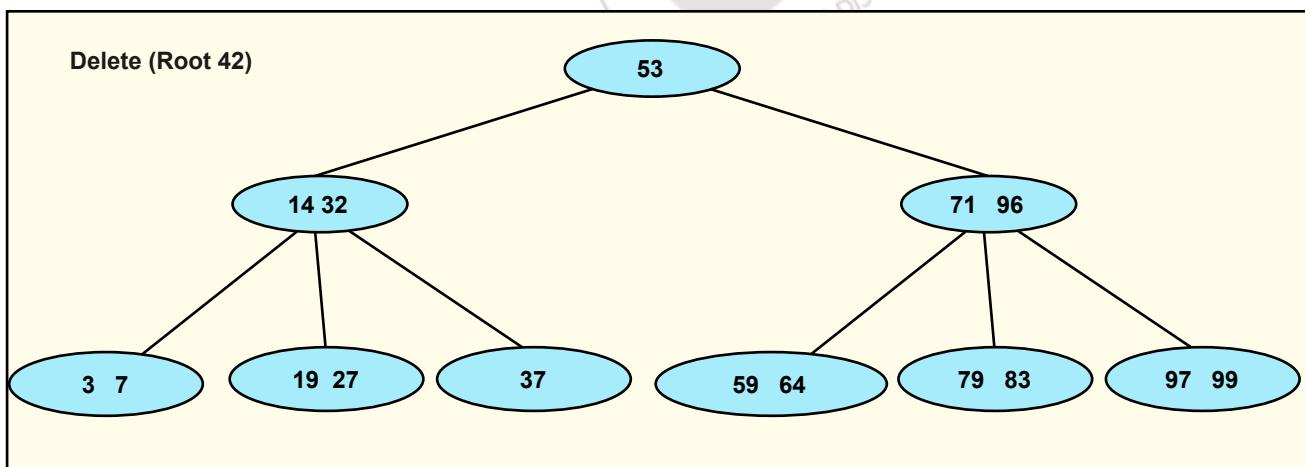
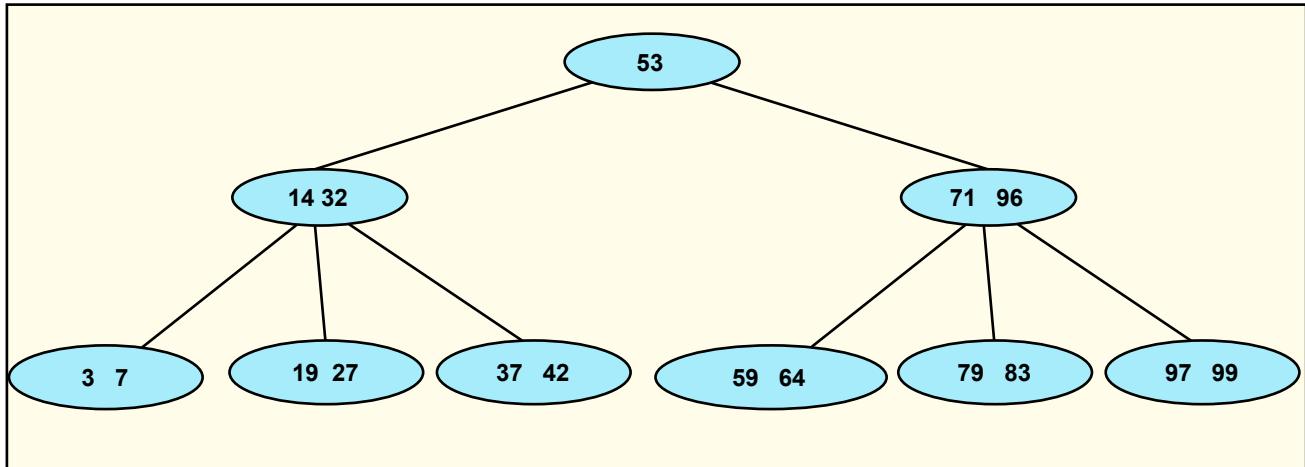
Case 1: The node from which the value is deleted has a minimum $n/2$ value even after deletion. Let us consider the following B-Tree of order 5. A value of 64 is to be deleted. Even after the deletion of value 64, the node has a minimum $n/2$ value (i.e., 2 values). Hence, the rules of the B-Tree are not violated.



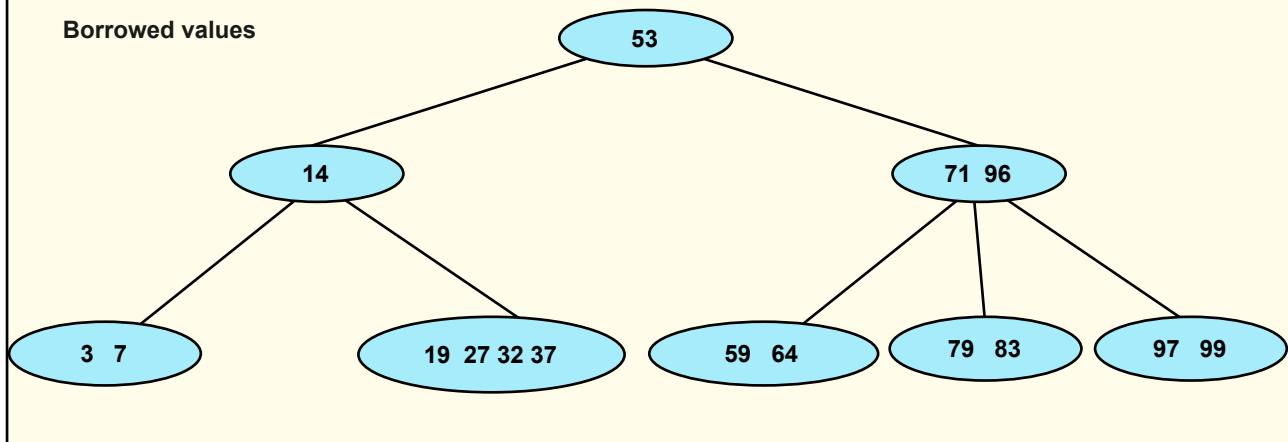
Case 2: In the second case, the node has less than minimum $n/2$ values after the deletion. Let us say we delete 92 from the tree. After 92 is deleted, the node has only one value, 83. However, a node adjacent to it has 3 values (i.e., extra values are in the adjacent node). Then, the last value in that node, 71, is pushed to its parent, and the first value in the parent, namely 79, is pushed into the node, which has values less than the minimum limit. Now, the node has obtained the minimum required values.



Case 3: In the previous case, there was an adjacent node with extra elements, so the adjustment was made easily. But if all the nodes have exactly the minimum required, and if a value is now deleted from a node in this, then no value can be borrowed from any of the adjacent nodes. Hence, as before, a value from the parent is pushed into the node (in this case, 32 is pushed down). Then, the nodes are merged. However, we see that the parent node has insufficient values. Hence, the same process of merging occurs recursively until the entire tree is adjusted.



Borrowed values



14 53 71 96

3 7

19 27 32 37

59 64

79 83

97 99

Algorithm

```

DELETE(ROOT, K )
Temp = SEARCH( ROOT, k ), DELETED = 0, i = 1
While i < = count(temp)
    If (k = info(temp[i]))
        DELETED = 1
        Delete temp[i]
    End if
End while
If DELETED = 0
Print "Item not found"
Return
Else
If count(temp) < n / 2
i = 1
While i <= count(par)
If count(child[i](par)) > n/2
s = child[i](par)
break
Else
i = i + 1
End if
End while
If info(temp[1]) > info(s[count(s)])
Ins(temp, info(par[1]))
Ins(par, info(s[count(s)]))
Else
Ins(temp, info(par[count(par)]))
Ins(par, info(s[1]))
End if
End if
End if
End DELETE

```

B+ Tree

A B+ Tree is a specialised version of the B-tree data structure where the data pointers are stored exclusively at the leaf nodes. The structure of a B+ tree distinguishes between internal and leaf nodes, with the leaf nodes holding an entry for each search key and a pointer to the corresponding data record. These leaf nodes are connected in a linked manner, ensuring sequential access to the data. Internal nodes primarily guide the search process, with certain key values from the leaf nodes replicated in these internal nodes.

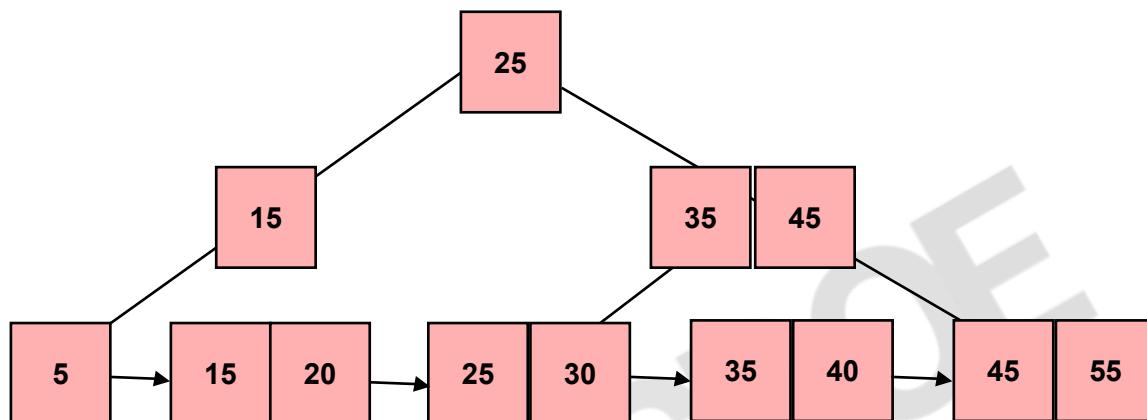


Fig. 10: Example of B+

Characteristics

- **Self-balancing:** B+ Trees automatically adjust their structure during insertions and deletions, ensuring consistent search performance.
- **Multi-level:** The tree has a hierarchy with the root at the top, internal nodes in the middle, and leaf nodes containing data at the bottom.
- **Ordered:** Keys are kept in sorted order, supporting an efficient range of queries.
- **High Fan-out:** Nodes can have many children, reducing tree height and improving search and indexing efficiency.
- **Cache-efficient:** Optimised for modern cache systems, enhancing performance.
- **Disk-efficient:** Designed for effective data storage and retrieval in disk-based systems.

Why B+ Trees Are Preferred

B+ Trees are ideal for systems where slow data access is a concern, as they reduce the number of I/O operations and optimise disk access. Their balanced nature and ability to perform an efficient range of queries make them popular in database systems and applications that demand quick data retrieval.

Difference between B Tree and B+ Tree

Parameter	B+ Tree	B Tree
Structure	Data is stored only in leaf nodes, which are internal nodes used for indexing.	Both keys and data are stored in every node.
Leaf Node Arrangement	Leaf nodes are linked, allowing efficient range-based queries.	Leaf nodes are not linked, making range queries less efficient.
Order	Higher order, capable of storing more keys.	Lower order, storing fewer keys.
Key Duplication	Allows key duplication, especially in leaf nodes.	Typically avoids key duplication.
Disk Access	Better disk access performance due to sequential reads in linked leaf nodes.	Requires more disk I/O with non-sequential reads in internal nodes.
Applications	Commonly used in databases and file systems for efficient range queries.	Used in general-purpose in-memory or database applications.
Performance	Optimised for range queries and bulk data retrieval.	Offers balanced performance across search, insert, and delete operations.
Memory	Requires more memory for internal nodes.	It uses less memory as keys and values are stored together.

Table 2: B Tree vs. B+ Tree



Self-Assessment Questions

10. In a B-Tree of order n, a node can have a maximum of _____ children.
- a) $n - 1$
 - b) n
 - c) $n + 1$
 - d) $n/2$
11. During a search in a B-Tree, if a value is _____ than all values, the search pointer moves to the leftmost child.
- a) Smaller
 - b) Greater
 - c) Equal
 - d) one of the above
12. In a B+ Tree, all data pointers are stored at the _____ nodes.
- a) Root
 - b) Internal
 - c) Leaf
 - d) Random



Summary

- Tree traversal involves visiting every node in a tree structure, ensuring each node is visited once. This process is essential for various operations like summing values or finding the largest node in a tree.
- A threaded binary tree improves the traversal process of a binary tree by utilising the unused NULL pointers.
- In a threaded binary tree, these NULL pointers are replaced with “threads” that point to the node’s in-order predecessor or successor.
- In Single-threaded, each node is threaded towards either the in-order predecessor or successor (left or right).
- In Double-threaded, each node is threaded towards both the in-order predecessor and successor (left and right).
- An AVL tree is a binary search tree (BST) in which the difference in height between each node’s left and right subtrees (called the balance factor) is maintained to be either -1, 0, or 1.
- A B-Tree is an m-way search tree of order n designed to maintain sorted data while allowing for efficient insertion, deletion, and search operations.
- Each node in the B-Tree can have a minimum of $n/2$ children and a maximum of n children, with all leaf nodes at the same level.
- A B+ Tree is a refined variant of the B-tree that stores all data pointers exclusively at the leaf nodes, differentiating it from a standard B-tree, which stores data and keys in all nodes.
- B+ Trees are designed to self-balance and are optimised for cache and disk efficiency, making them ideal for database and file system indexing.



Terminal Questions

1. How does in-order traversal differ from pre-order traversal?
2. Explain the three main types of tree traversal.
3. What is a threaded binary tree, and how does it differ from a normal binary tree?
4. Explain why threads can replace NULL pointers in a binary tree.
5. What is an AVL tree, and how does it maintain balance?
6. Explain the B+ Tree and how it differs from a B-tree.



Answer Keys

Self-Assessment Questions	
Question No.	Answer
1	A
2	C
3	B
4	B
5	D
6	A
7	B
8	A
9	C
10	B
11	A
12	C



Activity

Activity Type: Offline

Duration: 1 hour

Imagine you are traversing through a binary tree with the in-order traversal algorithm. At each node, you must decide whether to visit the left child, process the current node, or visit the right child. How would you determine the correct order of operations at each node to ensure a successful traversal of the entire tree using the in-order traversal method? Consider the importance of following a systematic approach and the implications of deviating from the correct sequence.



Glossary

- **Stack:** A data structure used to store nodes temporarily, ensuring correct traversal order, especially in recursive methods like in-order traversal.
- **Traversal:** The process of visiting all nodes in a tree systematically.
- **Thread:** A pointer that replaces a NULL value in a threaded binary tree to link nodes to their in-order predecessors or successors.
- **Balance Factor:** The difference between the heights of a node's left and right subtrees in an AVL tree.
- **Pivot Node:** The node at which the AVL tree is unbalanced and requires a rotation to restore balance.



Bibliography

Textbooks

- Balagurusamy, E. (2017). *Data Structures*. McGraw Hill Education.
- Sedgewick, R., & Wayne, K. (2014). *Algorithms II*. Pearson Education.
- Sridhar, S. (2014). *Design and Analysis of Algorithms*. Oxford University Press.
- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (1989). *Introduction to Algorithms*. Prentice-Hall.



e-References

- **Tree traversal:** <https://www.javatpoint.com/tree-traversal>
- **Threaded Binary Tree:** <https://www.javatpoint.com/threaded-binary-tree>
- **AVL Tree Data Structure:** <https://www.geeksforgeeks.org/introduction-to-avl-tree/>
- **B tree vs B+ tree:** <https://www.javatpoint.com/b-tree-vs-bplus-tree>



Video Links

Video	Links
Tree Traversals	https://www.youtube.com/watch?v=IpyCqRmaKW4
Threaded Binary Tree	https://www.youtube.com/watch?v=B7BI1VbFCI4
AVL tree	https://www.youtube.com/watch?v=vv0G7QgiGSQ
B Trees and B+ Trees	https://www.youtube.com/watch?v=aZjYr87r1b8



Image Credits

- **Fig. 1:** Self-Made
- **Fig. 2:** Self-Made
- **Fig. 3:** Self-Made
- **Fig. 4:** Self-Made
- **Fig. 5:** Self-Made
- **Fig. 6:** Self-Made
- **Fig. 7:** Self-Made
- **Fig. 8:** Self-Made
- **Fig. 9:** Self-Made
- **Fig. 10:** Self-Made
- **Table 1:** Self-Made
- **Table 2:** Self-Made



Keywords

- In-order traversal
- Pre-order traversal
- Post-order traversal
- Threaded binary tree
- Single-threaded binary tree
- Double-threaded binary tree
- AVL tree
- B tree
- B+ tree



DATA STRUCTURES AND ALGORITHMS

MODULE 4

Graphs, Searching, Sorting, and Hashing



Module Description

The module begins with an introduction to graphs, including various representation techniques such as adjacency lists and matrices. Core concepts like graph traversal are explored, covering depth-first and breadth-first search methods that enable efficient navigation and data extraction from graphs. The module also introduces key shortest-path algorithms, equipping students with methods to find minimum path lengths across nodes, which is crucial for network design and routing applications.

Moving to Searching and Sorting, the module examines several popular algorithms that facilitate data retrieval and arrangement. Students will examine searching techniques, focusing on sequential and binary search methods that optimise data access. A thorough review of sorting algorithms follows, covering quick sort, bubble sort, merge sort, and selection sort, each offering unique approaches and efficiencies for organising data sets.

Finally, Hashing is introduced as an efficient data storage and retrieval technique. Students will explore hash functions, their types, the essential concept of collisions, and how they can be resolved through Collision Resolution Techniques (CRT). Perfect hashing is presented as an idealised form for minimal collision scenarios. This module prepares students with practical skills and theoretical knowledge to implement and optimise data structures for improved computational performance.

The module consists of **three** units.

Unit 4.1: Graphs

Unit 4.2: Searching and Sorting

Unit 4.3: Hashing

MODULE 4

Graphs, Searching, Sorting, and Hashing

Unit 1

Graphs



■ Unit Table of Contents

Unit 4.1 Graphs

Aim	302
Instructional Objectives	302
Learning Outcomes	302
4.1.1 Introduction	303
Self-Assessment Questions	308
4.1.2 Representation of Graphs	309
Self-Assessment Questions	314
4.1.3 Graph Traversals Shortest Path Algorithms	315
Self-Assessment Questions	325
Summary	326
Terminal Questions	326
Answer Keys	327
Activity	327
Glossary	328
Bibliography	328
e-References	328
Video Links	329
Image Credits	329
Keywords	329



Aim

To provide students with a comprehensive interpretation of the fundamentals of graph theory, including graph representations, traversal methods, and spanning tree concepts.



Instructional Objectives

This unit is designed to:

- Define key terms and concepts related to graph theory, including vertices, edges, paths, and cycles
- Explain graph representations such as adjacency matrix and adjacency list
- Discuss the purpose and techniques of graph traversal, differentiating between Breadth-First Search (BFS) and Depth-First Search (DFS)



Learning Outcomes

At the end of the unit, the student is expected to:

- Analyse the BFS and DFS algorithms on given graphs
- Identify spanning trees within connected graphs
- Apply Kruskal's Algorithm to weighted graphs

4.1.1 Introduction

Graphs are non-linear data structures that are used to describe broad relationships among objects. A graph is a collection of vertices or nodes joined as a pair by the lines or edges. Graph G can be generally represented as $G(V,E)$, a finite set of vertex and edges where V is a set of vertex and E is a set of edges.

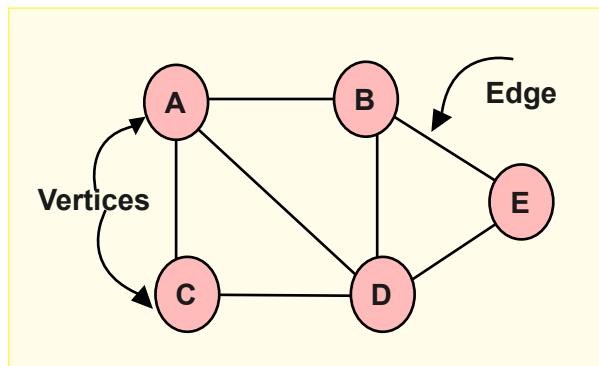


Fig. 1: Example of vertex and edges

In the above graph,

- The set of vertex is $\{A,B,C,D,E\}$ and
- The set of edges is $\{(A,B),(B,E),(C,A),(B,D),(D,E)\}$

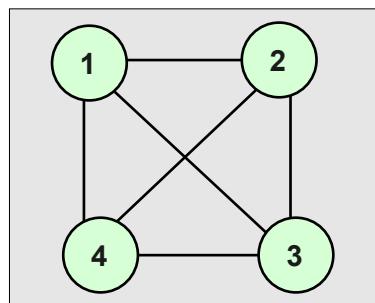
Types of Graphs:

There are many types of Graphs. Some of the important Graphs are listed below:

1. Directed Graph
2. Undirected Graph
3. Complete Graph
4. Connected Graph
5. Sub-Graph
6. Weighted Graph
7. Connected directed graph or strongly connected graph

1. Directed Graph:

- The graph is said to be a directed graph if it contains all the edges as directed edges.
- That is, a graph $G(V, E)$ is said to be a directed graph if E is a set of directed edges.

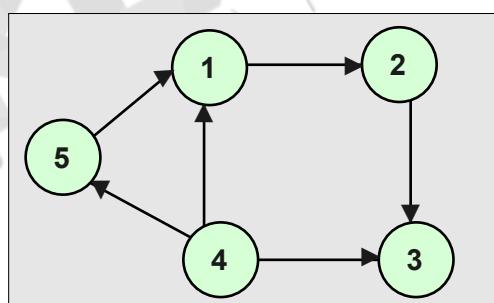


$$V(G) = \{1, 2, 3, 4\}$$

$$E(G) = \{(1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)\}$$

2. Undirected graph:

Graph $G(V, E)$ is said to be an undirected graph if it contains all the edges as undirected edges.

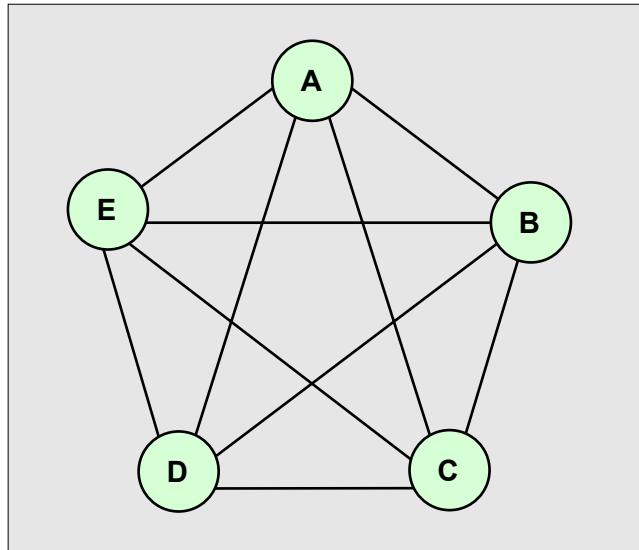


$$V(G) = \{1, 2, 3, 4, 5\}$$

$$E(G) = \{(1,2), (2,3), (4,3), (4,1), (4,5), (5,1)\}$$

3. Complete Graph:

A graph $G(V, E)$ is said to be complete if every vertex of V is adjacent to every other vertex of V in G . If a complete graph contains an “ n ” vertex, then it contains $n(n-1)/2$ edges.



There number of vertex ($n=5$)

$$\text{Number of edges} = n(n-1)/2$$

$$\begin{aligned}
 &= 5(5-1)/2 \\
 &= 5*4/2 \\
 &= 10
 \end{aligned}$$

The degree of the node represents several vertices connected as adjacent to the node.

In the above graph

$$\text{Degree of A} = 4$$

$$\text{Degree of B} = 4$$

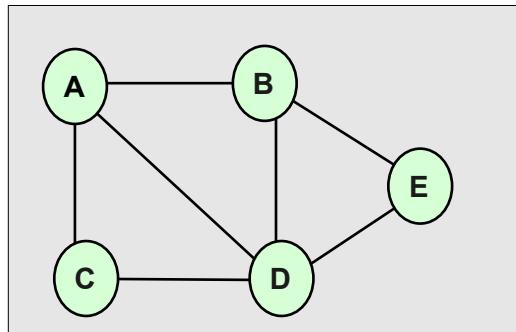
$$\text{Degree of C} = 4$$

$$\text{Degree of D} = 4$$

$$\text{Degree of E} = 4$$

4. Connected Graph:

Graph $G(V, E)$ is said to be connected if there is a path between every pair of vertices.

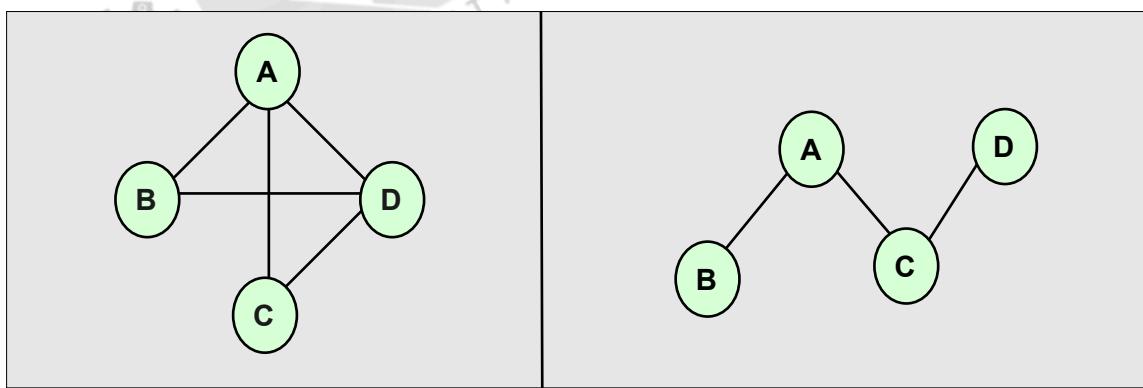


$$V = \{a, b, c, d, e\}$$

$$E = \{(a,b)(b,e)(a,c)(c,d)(d,e),(a,d)\}$$

5. Sub- Graph:

A graph $G|$ is said to be a subset of G where $G|(V|, E|) \subset G(V, E)$ that means a set of vertex of $G|$ should be a subset of the set of vertex of G [$G|(V|) \subset G(V)$] and the set of edges of $G|$ should be the subset of set of edges of G [$G|(E|) \subset G(E)$]



$$V = \{A, B, C, D\}$$

$$G \text{ of } E = \{(A,B)(A,D)(A,C)(B,D)(D,C)\}$$

$$V| = \{A, B, C, D\} \subset V$$

$$E| = \{(A,B), (A,C), (C,D)\} \subset E$$

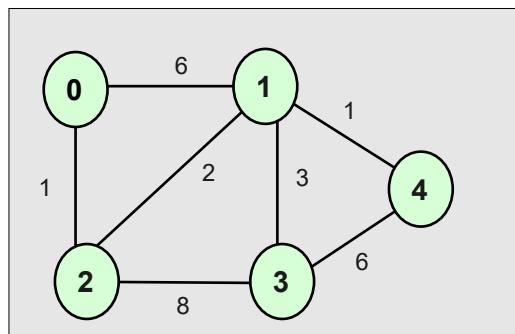
By observing the above

$$G|(V|) \subset G(V)$$

$$G|(E|) \subset G(E)$$

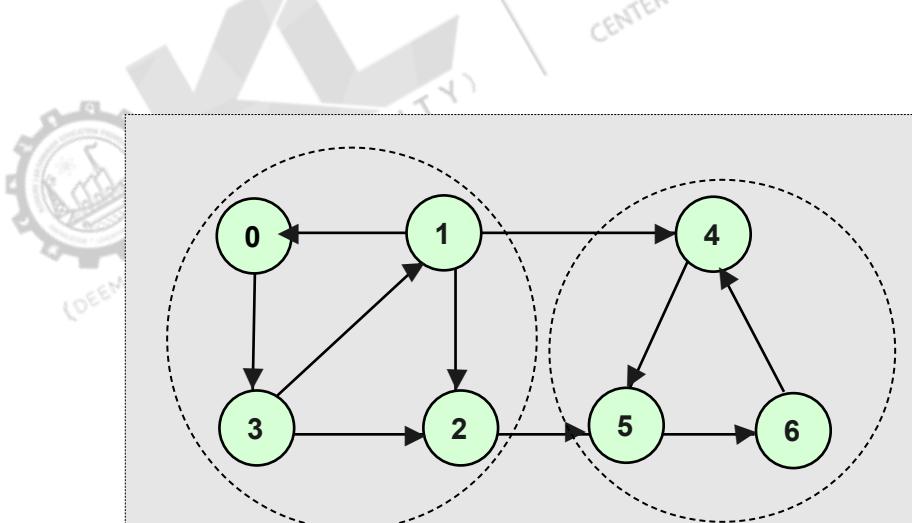
6. Weighted Graph:

Graph G is a weighted graph if each edge is assigned some position number, including zero, which is called a weighted graph.



7. Strongly Connected graph:

A strongly connected graph is a connected graph that contains the directed edges.





Self-Assessment Questions

1. A graph $G(V, E)$ is a collection of _____ and edges.
 - a) Vertices
 - b) Matrices
 - c) Numbers
 - d) Arrays

2. A graph is _____ if every vertex has a path to every other vertex.
 - a) Complete
 - b) Connected
 - c) Directed
 - d) Weighted

3. In a directed graph, the edges have a specific _____.
 - a) Colour
 - b) Shape
 - c) Weight
 - d) Direction

4. How many edges are in a complete graph with five vertices?
 - a) 5
 - b) 10
 - c) 15
 - d) 20

4.1.2 Representation of Graphs

A graph G is a collection of two sets V and E where V is the collection of vertices (nodes) v_0, v_1, \dots, v_{n-1} also called nodes, and E is the collection of edges e_1, e_2, \dots, e_n where an edge is an arc which connects two nodes. This can be represented as $G = (V, E)$

$$V(G) = (v_0, v_1, \dots, v_{n-1}) \text{ or set of vertices}$$

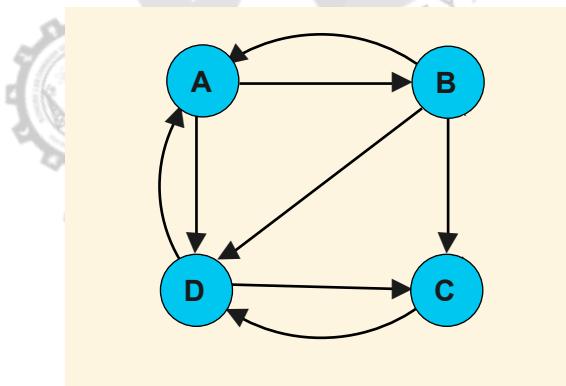
$$E(G) = (e_1, e_2, \dots, e_n) \text{ or set of edges}$$

There are two ways to represent the graph in computer memory. They are

1. Sequential representation.
2. Linked representation.

1) Sequential Representation or Adjacency Matrix:

An adjacency matrix is the matrix that keeps the information of adjacent nodes. In other words, we can say that this matrix keeps the information on whether this node is adjacent to any other node. We know very well that we can represent a matrix in a two-dimensional array of $n \times n$ or $\text{array}[n][n]$, where the first subscript will be a row, and the second subscript will be a column of that matrix. Suppose there are 4 nodes in the graph, then row1 represents the node1, row2 represents the node2, and so on. Similarly, column1 represents node1, column2 represents node2, and so on. The Adjacency matrix A of a graph $G = (V, E)$ with n nodes is an $n \times n$ matrix such that:



$$A[i][j] = \begin{cases} 1 & \text{if there is an edge from node } i \text{ to node } j \\ 0 & \text{if there is no edge from node } i \text{ to node } j \end{cases}$$

Hence, all the entries of this matrix

will be either 1 or 0.

Let us take a graph

The Corresponding adjacency matrix for this graph will be

$$\text{Adjacency Matrix } A = \begin{array}{c|cccc} & A & B & C & D \\ \hline A & 0 & 1 & 0 & 1 \\ B & 1 & 0 & 1 & 1 \\ C & 0 & 0 & 0 & 1 \\ D & 1 & 0 & 1 & 0 \end{array}$$

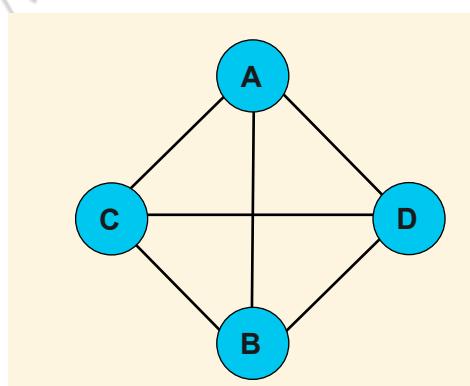
This adjacency matrix is maintained in the array $A[4][4]$.

Here, the entry of matrix $A[0][1] = 1$, representing an edge in the graph from node A to node B.

Similarly, $A[2][0] = 0$, indicating no edge from node C to node A.

Let us take an undirected graph: -The corresponding adjacency matrix for this graph will be

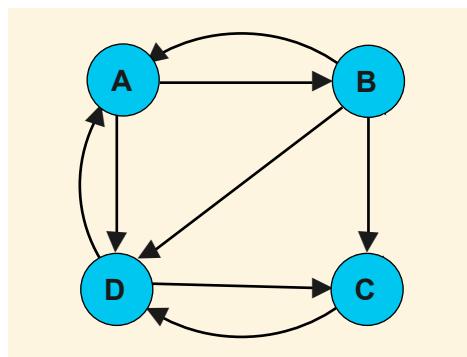
$$\text{Adjacency Matrix } A = \begin{array}{c|cccc} & A & B & C & D \\ \hline A & 0 & 1 & 1 & 1 \\ B & 1 & 0 & 1 & 1 \\ C & 1 & 1 & 0 & 1 \\ D & 1 & 1 & 1 & 0 \end{array}$$



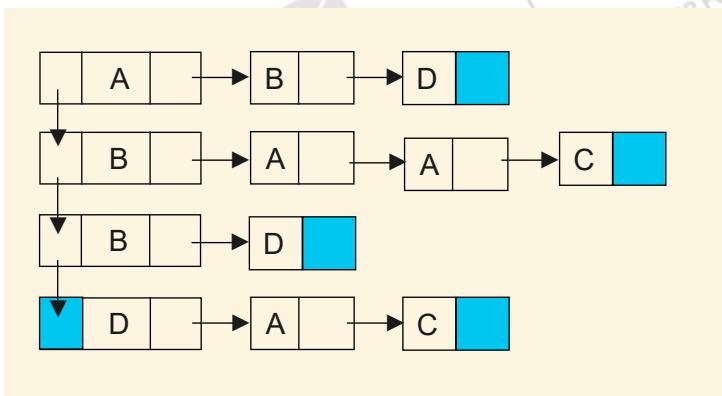
2) Adjacency List or Linked Representation:

If the graph's adjacency matrix is sparse, then it is more efficient to represent the graph through an adjacency list. In the adjacency list representation of the graph, we will maintain two lists. The first list will keep track of all the nodes in the graph, and the second list will maintain a list of adjacent nodes for each node. Suppose there are n nodes. Then, we will create one list that will keep information on all nodes in the graph, and then we will create n lists., where each list will keep information on all adjacent nodes of the node. Let us take the same graph

The adjacency list for this graph will be as follows:



The adjacency list for this graph will be as:



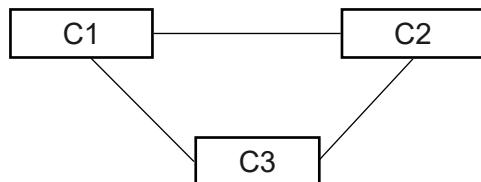
Application of Graphs:

- Graphs are mostly used in the following applications.
- Networks
- Topological sort
- Finding the shortest path
- Representation of Airline routes
- Analysis of electrical networks
- Study of molecular structure.

Networks:

The graphs can represent computer networks (or) Communication networks.

The vertices are the computers, and the edges are the communication lines.



$$V = \{c1, c2, c3\}$$

$$E = \{(c1, c3) (c3, c2) (c2, c1)\}$$

Weights are the bandwidth.

Topological Sort:

Graphs also sort the elements using a topological sort algorithm.

Finding the shortest path:

Graphs are used to find the shortest path between any two vertices when they are.

Representation of Airline routes:

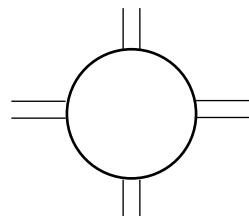
We can use graphs to represent airline route systems.

Here, the Airports are vertices, the flights are the edges, and the distance (or) ticket price is a weight.



Analysis of electrical networks:

Graphs are used to analysis the electrical networks. Graphs can represent electrical circuits.



Study of molecular structure:

The study of the molecular structure of chemical compounds required the graph.

Operations of Graphs:

By using the graphs, we can perform the following operations:

- Adding or removing a vertex.
- Adding or removing an edge.
- Check whether an edge exists between two vertices.
- Finding the successor of the given vertex.





Self-Assessment Questions

5. A _____ array represents an adjacency matrix.
- a) One-dimensional
 - b) Three-dimensional
 - c) Four-dimensional
 - d) Two-dimensional
6. An adjacency list is more efficient than an adjacency matrix when the graph is
- a) Sparse
 - b) Dense
 - c) Complete
 - d) Connected
7. In the adjacency matrix, an entry of 1 indicates that two nodes are_____.
- a) The same
 - b) Adjacent
 - c) Not adjacent
 - d) Isolated

4.1.3 Graph Traversals Shortest Path Algorithms

Graph Traversing Methods

Graph traversal methods are used to visit all the vertex of a group.

There are two types of graph traversal methods. They are

1. Breadth-first search (BFS) and
2. Depth-first search (DFS)

1. Breadth-first search (BFS)

Breadth-first search, popularly known as BFS, is an Algorithm for traversing branched data structures like Trees and Graphs.

The BFS Algorithm first traverses the start node (any node), then traverses the direct children or adjacent vertices of the start vertex and visits their children soon.

This approach is also known as the **wavefront traverse method**.

The BFS Algorithm is very simple and is widely used in problem-solving. It uses a Queue to hold the nodes waiting to be processed.

First, we examine the starting vertex A and then all its neighbours. We continue with the other nodes similarly, and no node is processed more than once.

Algorithm:

Step 1: Initialise all the nodes to the ready state.

Step 2: Start with a node and place it into the Queue.

Step 3: Remove the processed node from the Queue

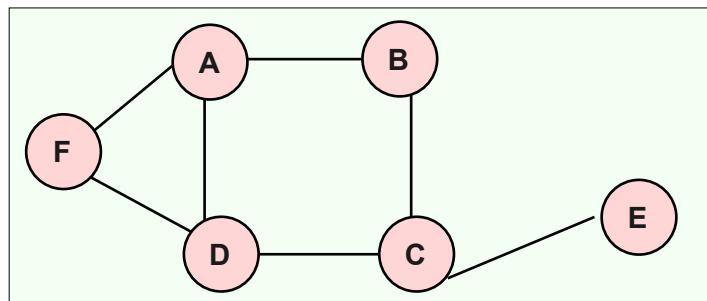
And print it.

Step 4: Place all the adjacent nodes of the processed Vertex into the Queue.

Step 5: Repeat Step3 and Step4 until all the elements Of the graph are visited.

Step 6: Delete all the elements of the Queue and print They are to complete BFS.

Example:



1. Visit a node A and add A into the Queue

Queue A

After processing A

Queue BDF

After processing F

Queue BD

After processing D

Queue CB

After processing B

Queue E

After processing C

Queue A

After processing E

Queue A empty

Output:

A F D B C E

2. Depth First Search (DFS)

It is simply known as DFS. It is a way of the traverse of the graph. It allows the visiting of the vertices of the graph only. But there are 100's of algorithms. Therefore, interpreting the principles of DFS is quite important to moving forward in graph theory.

In this search Algorithm, after visiting the initial node, it should be kept in the Stack [pushed into the Stack] and then checked for any other adjacent nodes. If yes, then go forward to visit a node, and the visited node should be pushed into the Stack and marked as visited.

This process will continue until all the vertex are visited and all the vertex are removed from the stack.

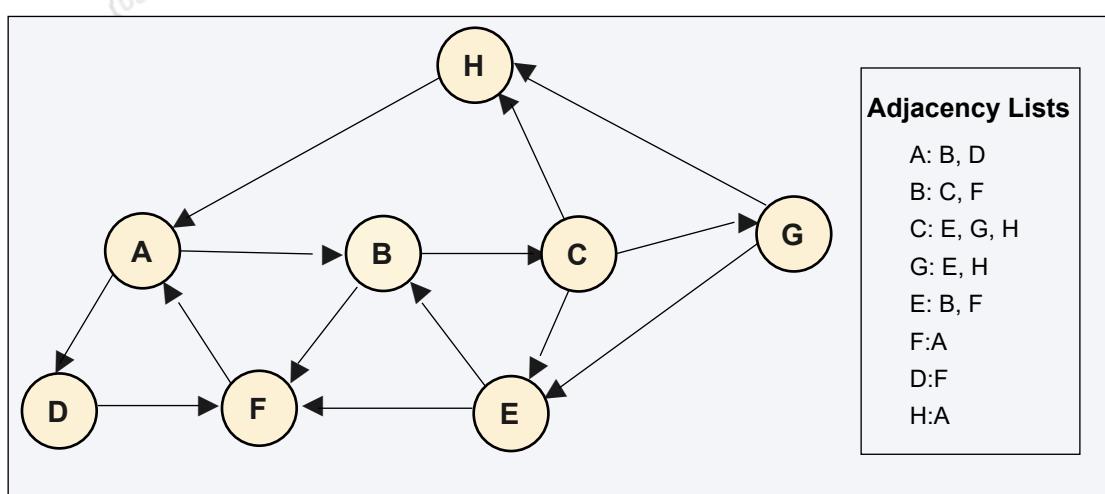
Once a node has no adjacent vertex, remove the vertex from the Stack and perform a backtrack to visit the previous vertex or node.

Algorithm:

- Step 1:** Initialise all the nodes into a ready state.
- Step 2:** Start with node A and push into the Stack.
- Step 3:** Check the adjacent nodes, if there is an adjacent Nodes visit one node and push into the Stack
- Step 4:** Otherwise, pop the node from the Stack.
- Step 5:** Repeat the steps 2 to 4 until all the vertices are Visited.
- Step 6:** Remove all the nodes from Stack.

Note: The visited node should be kept in the output sequence.

Consider graph G and its adjacency list, which is given in the figure below. Using the depth-first search (DFS) algorithm, calculate the order to print all the graph nodes starting from node H.



Push H onto the stack

STACK: H

POP the top element of the stack, i.e., H,

Print it and push all the neighbours of H onto the stack, which is ready.

Print H

STACK: A

Pop the top element of the stack, i.e., A, print it, and push all the neighbours of A onto the stack that are in a ready state.

Print A

Stack: B, D

Pop the top element of the stack, D, print it, and push all the neighbours of D onto the stack that are in the ready state.

Print D

Stack: B, F

Pop the top element of the stack, F, print it and push all the neighbours of F onto the stack that are in the ready state.

Print F

Stack: B

Pop the top of the stack, i.e., B and push all the neighbours

Print B

Stack: C

Pop the top of the stack, i.e., C, and push all the neighbours.

Print C

Stack: E, G

Pop the top of the stack, i.e., G, and push all its neighbours.

Print G

Stack: E

Pop the top of the stack, i.e., E, and push all its neighbours.

Print E

Stack:

Hence, the stack is empty, and all the graph nodes have been traversed.

The printing sequence of the graph will be:

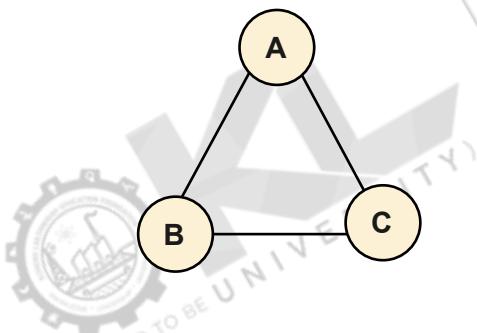
H → A → D → F → B → C → G → E

Spanning Tree:

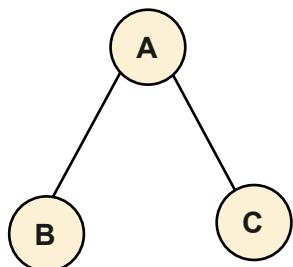
A spanning tree is a subset of graph $G(V,E)$, which contains all the vertex of G and contains the edges, which are a subset of $G(E)$.

A spanning tree should not form a cycle. The Graph (G) should be a connected graph. A graph may contain more than one spanning tree.

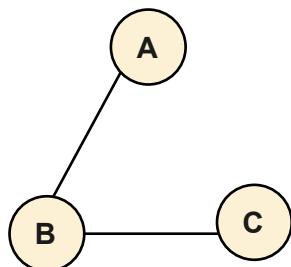
Connected graph:



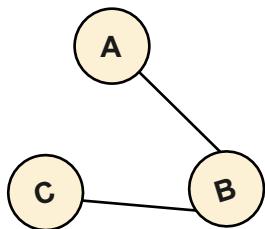
Spanning tree-1:



Spanning tree-1:



Spanning tree-1:



Minimum Spanning Tree:

A Minimum spanning tree is a tree with fewer weights than the other spanning trees of Graph G. We can find a minimum spanning tree only for the connected weighted graph. We can find the minimum spanning tree Algorithms.

The most popular minimum spanning tree algorithms are:

1. Kruskal's Algorithm
2. Prim's Algorithm

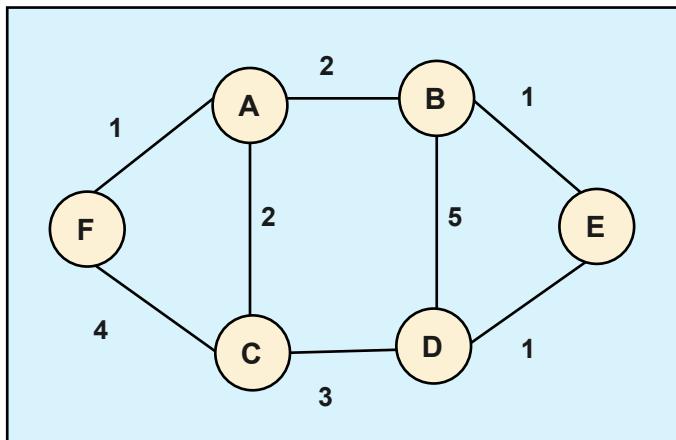
1. Kruskal's Algorithm

Following is the process of Kruskal's Algorithm

Algorithm:

1. List all the edges in non-decreasing order, which means ascending order.
2. Pick the smallest edge.
3. If the cycle formed, then discard it; otherwise, add it to the tree.
4. Repeat 2 & 3 steps until all the vertex are completed.

Example:



Step 1:

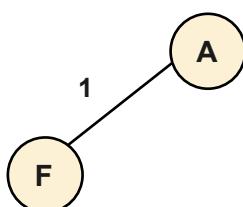
list all the edges in non-decreasing order as:

Edges	Weights
AF	1
BE	1
ED	1
AB	2
AC	2
CD	3
FC	4
BD	5

Step 2:

Pick the smallest edge <AF>

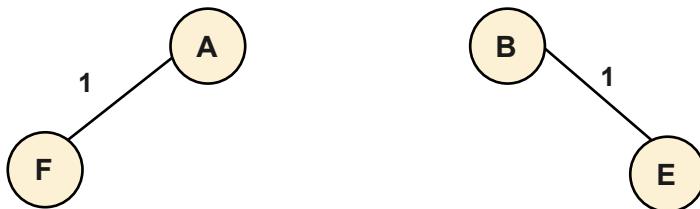
weight 1, Add it to the tree



Step 3:

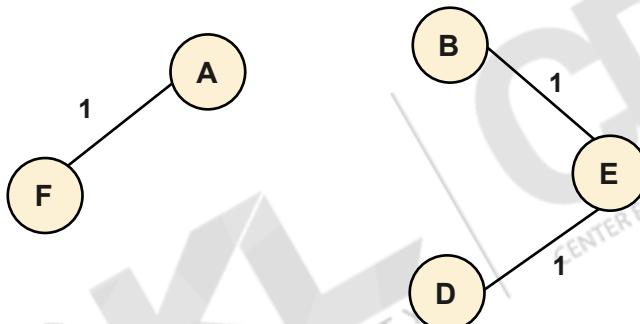
Pick the smallest edge from the remaining, i.e., $\langle BE \rangle$

Weight 1, Add it to the tree.

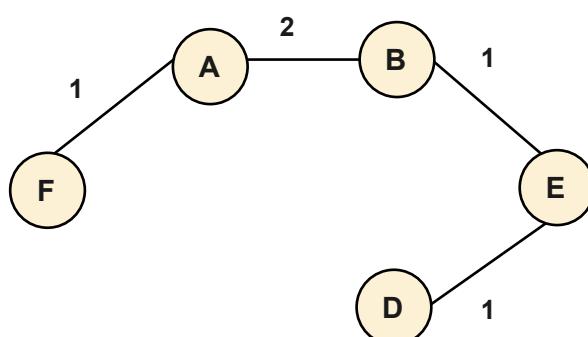

Step 4:

Pick the smallest edge from the remaining, i.e., $\langle ED \rangle$,

Weight 1, add it

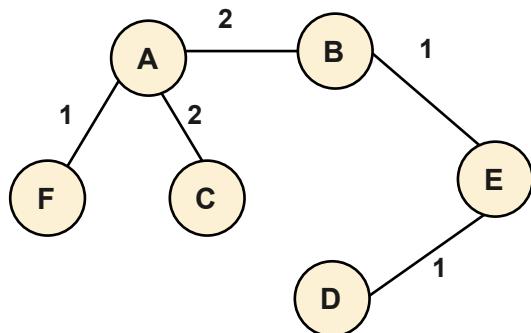

Step 5:

Pick the smallest edge from the remaining, i.e., $\langle AB \rangle$ weight 2, and dd it.

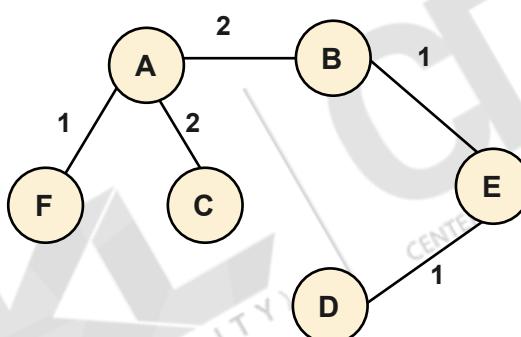


Step 6:

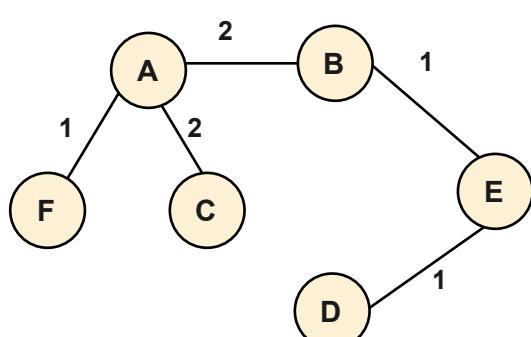
Pick the smallest edge from the remaining, i.e., $\langle AC \rangle$ weight 2, add it.


Step 7:

Pick the smallest edge from the remaining, i.e., $\langle CD \rangle$ weight 3, and discard it [cycle is formed].

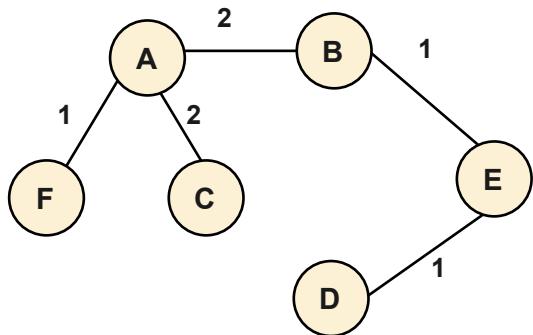

Step 8:

Pick the smallest edge from the remaining, i.e., $\langle FC \rangle$ weight 4, discard it [cycle is formed].



Step 9:

Pick the smallest edge from the remaining, i.e., $\langle BD \rangle$ weight 5, and discard it [cycle is formed].



The weight of the path is 7





Self-Assessment Questions

8. BFS is also known as the _____ traversal method.

- a) Edge-first
- b) Wavefront
- c) Depth-first
- d) Stack-based

9. In DFS, we use a _____ data structure to keep track of the nodes.

- a) Queue
- b) List
- c) Stack
- d) Array

10. Which graph traversal algorithm will likely yield the shortest path in an unweighted graph?

- a) DFS
- b) Kruskal's Algorithm
- c) Prim's Algorithm
- d) BFS



Summary

- Graphs are non-linear data structures designed to represent relationships among various objects.
- A graph, denoted as $G(V,E)$, is composed of vertices (V) and edges (E) connecting pairs of vertices.
- Each graph type has unique applications and is widely used in computer science to model and solve complex relationship problems.
- A graph G is a structure composed of vertices V (or nodes) and edges E , which can be represented as $G = (V,E)$. Graphs can be represented in memory in two primary ways:
- The sequential Representation (Adjacency Matrix) approach uses a 2D array to record adjacency between nodes. Each element in the matrix indicates whether an edge exists between a pair of nodes (represented by 1 if adjacent, 0 if not).
- Linked Representation (Adjacency List) Efficient for sparse graphs, an adjacency list uses separate lists for each node to store only its adjacent nodes.
- Graph operations allow for adding/removing vertices or edges, checking for edge existence, and finding node successors.
- Essential graph traversal techniques and shortest path algorithms, focusing on Breadth-First Search (BFS) and Depth-First Search (DFS).
- BFS is a traversal method that uses a queue to explore each vertex layer-by-layer, visiting neighbours first.
- In contrast, DFS uses a stack to explore as far down a path as possible before backtracking. Both methods have applications in different problem-solving scenarios.



Terminal Questions

1. What formula calculates the number of edges in a complete graph with 'n' vertices?
2. What are the two primary methods of graph representation?
3. How is an adjacency matrix different from an adjacency list?
4. List three applications and operations of graphs and explain one.
5. Define graph traversal and list the two main traversal methods.
6. Describe the Depth-First Search (DFS) process with an example.



Answer Keys

Self-Assessment Questions	
Question No.	Answer
1	A
2	B
3	D
4	B
5	D
6	A
7	B
8	B
9	C
10	D



Activity

Activity Type: Offline

Duration: 1 hour

Imagine you're building a city navigation system where neighbourhoods are nodes and roads are edged with weights for distance and congestion. Choose the best graph representation (adjacency matrix or list) considering memory and efficiency. To find the shortest route, explain why you'd use specific algorithms like Dijkstra's or BFS for distance and other ways to avoid congestion. Finally, describe how you'd adapt the system if real-time traffic data were available, ensuring the graph updates efficiently and algorithms perform optimally for dynamic conditions.



Glossary

- **Vertex (Node):** A fundamental unit in a graph representing an entity or a position.
- **Adjacency Matrix:** A 2D array that records adjacency relationships between vertices.
- **Adjacency List:** A graph representation where each vertex has a list of its adjacent vertices.
- **Topological Sort:** An algorithm for ordering vertices in a directed acyclic graph.
- **Kruskal's Algorithm:** Kruskal's Algorithm is a popular algorithm in graph theory used to find the minimum spanning tree (MST) of a connected, undirected graph.



Bibliography

Textbooks

- Balagurusamy, E. (2017). *Data Structures*. McGraw Hill Education.
- Sedgewick, R., & Wayne, K. (2014). *Algorithms II*. Pearson Education.
- Sridhar, S. (2014). *Design and Analysis of Algorithms*. Oxford University Press.
- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (1989). *Introduction to Algorithms*. Prentice-Hall.



e-References

- **Graphs:** <https://stmarysguntur.com/wp-content/uploads/2019/07/UNIT-5.pdf>
- **Graph Data Structure:** Types, Uses, Examples, Algorithms: <https://www.wscubetech.com/resources/dsa/graph-data-structure>
- **Introduction to Graph Data Structure:** <https://www.geeksforgeeks.org/introduction-to-graphs-data-structure-and-algorithm-tutorials/>



Video Links

Video	Links
Graphs in Data Structures	https://www.youtube.com/watch?v=bvWVs0tJUOY
Graph Traversals	https://www.youtube.com/watch?v=pcKY4hjDrxk



Image Credits

- **Fig. 1:** Self-Made



Keywords

- Graph
- Breadth-First Search (BFS)
- Depth-First Search (DFS)
- Spanning Tree
- Minimum Spanning Tree
- Prim's Algorithm

MODULE 4

Graphs, Searching, Sorting, and Hashing

Unit 2

Searching and Sorting



■ Unit Table of Contents

Unit 4.2 Searching and Sorting

Aim	332
Instructional Objectives	332
Learning Outcomes	332
4.2.1 Searching and Searching Types	333
Self-Assessment Questions	338
4.2.2 Sorting and Sorting Types	339
Self-Assessment Questions	355
Summary	357
Terminal Questions	357
Answer Keys	358
Activity	358
Glossary	359
Bibliography	359
e-References	359
Video Links	360
Image Credits	360
Keywords	360



Aim

To familiarise students with a comprehensive interpretation of the fundamentals of searching and sorting algorithms.



Instructional Objectives

This unit is designed to:

- Describe the concept of searching in arrays and its significance in data retrieval
- Explain the linear search algorithm, including best and worst-case performance scenarios
- Discuss the different types of sorting algorithms



Learning Outcomes

At the end of the unit, the student is expected to:

- Examine the binary search algorithm in C to find elements in sorted arrays efficiently
- Classify sorting algorithms based on their characteristics
- Evaluate the performance of different sorting algorithms in terms of time complexity

4.2.1 Searching and Searching Types

Searching is the process of finding an element in each Array.

There are two types of search methods. They are:

1. Linear Search
2. Binary search

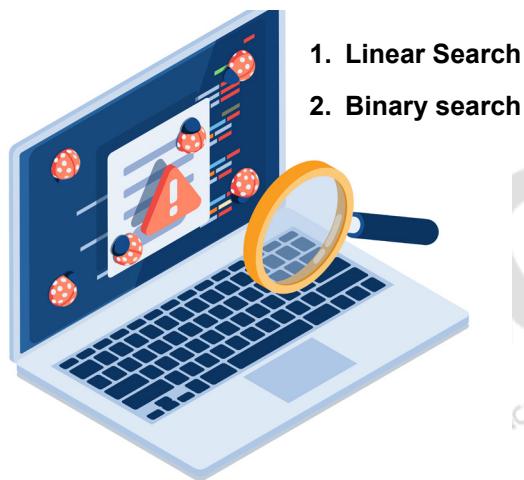


Fig. 1: Searching method types

1. Linear search

- Linear search is a process for finding the given element in an array in which the search elements complete the sequence from the starting element to the ending element.
- The search process will be completed if the search element is found successfully.
- The performance of this Search method depends on the element's position in an Array.
- If it is in the first position, the search process will be completed quickly, which is the best case.
- If the search elements are not found, even after searching, all the positions of the Array are called Worst-Case.

Algorithm

Step 1: Start

Step 2: Take a given Array with n elements ($a[],n$).

Step 3: Input Search element “X”

Step 4: for loop : $i=0; i < n; i++$

Step 5: Compare the search element

$X=a[i]$

Step 6: If step 5 is true,

Print “element is found” and go to stop.

Step 7: If step 5 is false, go to step 4.

Step 8: If step -4 is false for $i=n-1$,

Then print “element not found” and go to stop.

Step 9: Stop

Example Program:

```
#include <stdio.h>

int linearSearch(int a[], int n, int val) {
    for (int i = 0; i < n; i++) {
        if (a[i] == val)
            return i+1;
    }
    return -1;
}

int main() {
    int a[] = { 20, 35, 40, 16, 23, 22, 40}; // Array
    int val = 30; Key Element
    int n = sizeof(a) / sizeof(a[0]); // size of array
    int res = linearSearch(a, n, val); // Store result
    printf ("The elements of the array are – ");
    for (int i = 0; i < n; i++)
        printf("%d ", a[i]);
    printf("\nElement to be searched is – %d", val);
    if (res == -1)
```

```

printf("\nElement is not present in the array");
else
printf("\nElement is present at %d position of array", res);
return 0;
}
  
```

2. Binary search

Binary search is a process of searching an element by dividing the given array into two parts using a sorted array. The Search element may be found in one of the two parts.

The given sorted array will be divided into two parts using the formula $\text{mid}=\lfloor \frac{l+u}{2} \rfloor$.

Where,

l =lower boundary

Initially, it can be taken as $l=0$

U =upper boundary

Initial $u=n-1$

Let x be a search element

- if ($x==a[\text{mid}]$): Then print the search element found and stop the search process.
- if($x < a[\text{mid}]$): We can ignore all the greater elements than $a[\text{mid}]$. So move “ u ” to the $\text{mid}-1$ position.
- if($x > a[\text{mid}]$): We can ignore all the elements less than $a[\text{mid}]$ and move l to the $\text{mid}+1$ position.

Again, calculate $\text{mid}=\lfloor \frac{l+u}{2} \rfloor$ and continue the above process until the search process is completed.

Algorithm:

Step 1: Start

Step 2: Take a sorted array, with n elements ($a[], n$)

Step 3: Take $l=0, u=n-1, \text{flag}=0, \text{mid}$

Step 4: Input search element “ x ”

Step 5: Check $l > 0 \ \&\& \ u \leq n-1$

Step 6: If step 5 is true, calculate $\text{mid}=\lfloor \frac{l+u}{2} \rfloor$

Step 7: Compare $x=a[\text{mid}]$

Step 8: if step-7 is true calculate mid=l+u/2

Flag and goto step-12

Step 9: If step-7 is false

Check $x < a[mid]$

Step 10: If step-9 is true do $u = mid - 1$ and go to step 5

Step 11: If Step-9 is false do $l = mid + 1$ and go to step 5

Step 12: If step-5 is false check flag==1

Step 13: If step-12 is true

Print "search element is found"

Step 14: If step-12 is false

Print "search element is not found" and go to stop

Step 15: Stop

Program:

```
#include <stdio.h>
int main()
{
    int i, low, high, mid, n, key, array[100];
    printf("Enter number of elementsn");
    scanf("%d",&n);
    printf("Enter %d integersn", n);
    for(i = 0; i< n; i++)
        scanf("%d",&array[i]);
    printf("Enter value to findn");
    scanf("%d", &key);
    low = 0;
    high = n - 1;
    mid = (low+high)/2;
    while (low <= high) {
        if(array[mid] < key)
            low = mid + 1;
        else if (array[mid] == key) {
```

```
printf("%d found at location %d.n", key, mid+1);
break;
}
else
high = mid - 1;
mid = (low + high)/2;
}
if(low > high)
printf("Not found! %d isn't present in the list.n", key);
return 0;
}
```





Self-Assessment Questions

1. In Linear Search, the algorithm scans each element in sequence until it finds the element or reaches the _____.
 - a) Middle
 - b) Beginning
 - c) Upper bound
 - d) End

2. Binary Search can only be applied to an array that is _____.
 - a) Unsorted
 - b) Sorted
 - c) Empty
 - d) Fixed

3. Binary Search is an example of which type of algorithm?
 - a) Divide-and-conquer
 - b) Recursive
 - c) Greedy
 - d) Dynamic programming

4. In a linear search, the best-case scenario occurs when the target element is located at the _____ of the array.
 - a) Middle
 - b) End
 - c) Beginning
 - d) Sorted portion

4.2.2 Sorting and Types of Sorting

Sorting is a process of arranging the elements of an array into ascending or descending order.

The different types of sorting are as follows:

1. Bubble sort
2. Selection sort
3. Insertion sort
4. Quick sort
5. Merge sort
6. Heap sort

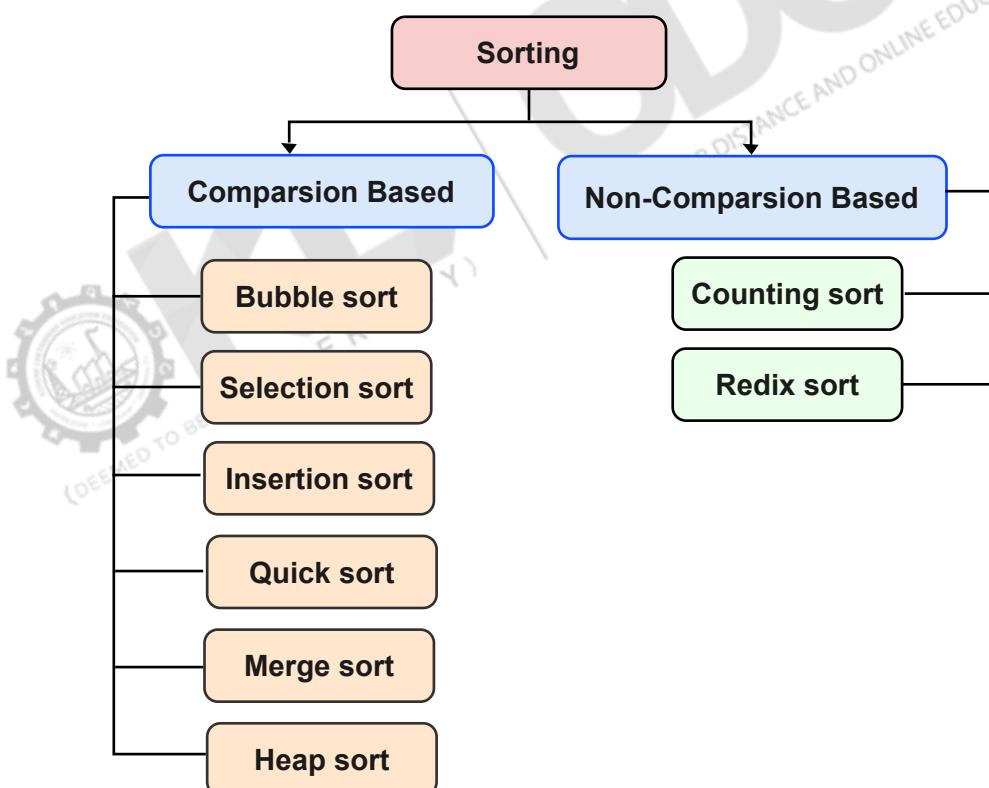


Fig. 2: Sorting techniques

1. Bubble sort:

Bubble sorting is one sorting technique. In this technique, the first element of an Array is compared with the second element of the array.

If the first element is greater than the second element [$a[0] > a[1]$], then both elements will be interchanged.

Now, the first element's position has changed. After that, we must compare the first position element with the third position element.

If the first position element is greater than the third position element, both elements are interchanged.

After that, the first position element will be compared with the fourth one. The process above will be continued up to the nth position element, which will be compared with the first position element.

This is called pass -1'

The above process will be continued again from and element to nth element and 3rd element to nth element and soon($n-1$)th element to nth element

The entire process will be done in different passes.

If the Array contains n elements, it takes $(n-1)$ passes to sort the array.

Note: The above-said process is for arranging the data elements in ascending order

Example

Given array 47,35,50,20,30

Pass-1:

47	35	35	20	20
35	47	47	47	47
50	50	50	50	50
20	20	20	35	35
30	30	30	30	30

Pass-2:

20	20	20	20	20
----	----	----	----	----

47	47	35	35	30
----	----	----	----	----

50	50	50	50	50
----	----	----	----	----

35	35	47	47	47
----	----	----	----	----

30	30	30	30	35
----	----	----	----	----

Pass-3:

20	20	20
----	----	----

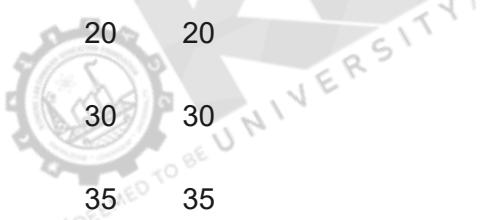
30	30	30
----	----	----

50	47	35
----	----	----

47	50	50
----	----	----

35	35	47
----	----	----

Pass-4:



50	47
----	----

47	50
----	----

Algorithm for Bubble sort:

Step 1: Start

Step 2: Take an Array A with n-element ($A[],n$).

Step 3: for loop1: $i=0$, check $i \leq n-1$.

Step 4: If step -3 is true

 for loop2: $j=i+1$, check $j < n$

Step 5: If step 4 is true

check $a[j] < a[i]$

Step 6: If step 5 is true

 Interchange the values of $a[j]$ and $a[i]$ and do $j=j+1$ then go to step- 4

Step 7: If step 5 is false

 do $j=j+1$ and go to step- 4

Step 8: If step 4 is false

 do $i=i+1$ and go to step- 3

Step 9: If step 3 is the false end for loop1 and print the sorting Array A

Step 10: Stop

Program:

```
//Program for Bubble Sort
#include<stdio.h>
//#include<conio.h>
void main()
{
    int i,j,temp,n,a[10], cnt,k;
    // clrscr();
    printf("Enter how many values \n");
    scanf("%d",&n);
    printf("Enter %d numbers \n");
    for(i=0; i<n; i++)
    {
        scanf("%d",&a[i]);
    }
```

```

printf("Before Sorting \n");
for(i=0; i<n; i++)
{
printf("%d \t",a[i]);
}
//Sorting Technique
for(i=1; i<n; i++)
{
cnt=0;
for(j=0; j<(n-i); j++)
{
if(a[j]>a[j+1])
{
temp=a[j];
a[j]=a[j+1];
a[j+1]=temp;
cnt++;
}
}
printf("After Pass%d \n",i);
for(k=0; k<n; k++)
{
printf("%d\t",a[k]);
}
if (cnt==0)
break;
}
printf("After Sorting \n");
for(i=0; i<n; i++)
{
printf("%d\t",a[i]);
}
}

```

2. Insertion Sort

Insertion sort is another technique used to sort the elements in ascending or descending order.

In this sort of, the given Array is taken as an unsorted array

Now, the given array can be divided into 2 arrays.

1. Sorted array and
2. Unsorted array

Initially, the sorted Array contains the first element of the given array, and the remaining elements are taken as unsorted array elements.

Now, the first element of the unsorted array will be taken and compared with the elements of the sorted Array.

Now, insert the element in a desired position in the sorted array. This means the element will compare with each element of the sorted array, find the desired position, and then insert it.

Algorithm for Insertion Sort

Step 1: Start

Step 2: Take an unsorted array with "n" elements [a[],n]

Step 3: for loop :i=0;i<n-1;i++

Step 4: if step -3 is true, j=i+1,

 Check while ($j > 0 \& \& a[j] < a[j-1]$)

Step 5: A step 4 is true

 Interchange the values of $a[j]$ & $a[j-1]$ and $j=j-1$, then goto step4.

Step 6: if step -4 is false, goto step 3.

Step 7: if step 3 is false,

 End for a loop.

Step 8: print sorted array

Step 9: Stop

Program:

```
#include <stdio.h>

void insertionSort(int arr[], int n) {
    int i, key, j;
    for (i = 1; i < n; i++) {
        key = arr[i];
        j = i - 1;

        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}

int main() {
    int arr[] = { 12, 11, 13, 5, 6 };
    int n = sizeof(arr) / sizeof(arr[0]);
    insertionSort(arr, n);
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");
    return 0;
}
```

3. Selection Sort

Selection sort is another sorting technique to sort the element of an unsorted array.

In this sorting technique, the least element of an array is identified, and then the positions are exchanged with the first element of an array.

Now, identify the smallest element from the second location of the array to the nth location of the array. Again, exchange the second-least-element position with the second-position element of the array.

Continue this process ($n-1$) times where n is the number of elements of the array.

Example:

Pass 1:	Find the location LOC of the smallest in the list of N elements
	A[1], A[2], ..., A[N], and then interchange A[LOC] and A[1]. Then A[1] is sorted.
Pass 2:	Find the location LOC of the smallest in the sublist of N -1 elements
	A[2], A[3], ..., A[N], and then interchange A[LOC] and A[2]. Then A[1], A[2] is sorted.
Pass 3:	Find the location LOC of the smallest in the sublist of N -2 elements
	A[3], A[4], ..., A[N], and then interchange A[LOC] and A[3]. Then A[1], A[2], A[3] is sorted.

Pass N -1: Find the location LOC of the smaller of the elements A[N – 1], A[N], and then interchange A[LOC] and A[N – 1]. Then A[1], A[2], ..., A[N] is sorted

Selection Sort for n = 5 items

Pass	A[1]	A[2]	A[3]	A[4]	A[5]
Pass 1, LOC =5	40	30	20	50	10
Pass 2, LOC=3	10	30	20	50	40
Pass 3, LOC = 3	10	20	30	50	40
Pass 4, LOC =5	10	20	30	50	40
Sorted	10	20	30	40	50

Algorithm for selection sort:

Step 1: Start

Step 2: Take an Array with elements (a[],n);

Step 3: for loop i=0, check i < n ; i++

Step 4: If step3 is true

Find the minimum elements from the Array

Step 5: Interchange a[i] with minimum elements and goto step-3

Step 6: Repeat Steps 4 and 5

(i.e., from i=0 to i=n-1)

Step 7: If step3 is false

Print the sorted array

Step 8: Stop.

Program:

```
#include <stdio.h>

void selection(int arr[], int n)
{
    int i, j, small;

    for (i = 0; i < n-1; i++) // One by one move boundary of unsorted subarray
    {
        small = i; //minimum element in unsorted array

        for (j = i+1; j < n; j++)
            if (arr[j] < arr[small])
                small = j;

        // Swap the minimum element with the first element
        int temp = arr[small];
        arr[small] = arr[i];
        arr[i] = temp;
    }

    void printArr(int a[], int n) /* function to print the array */
    {
        int i;
        for (i = 0; i < n; i++)
            printf("%d ", a[i]);
    }

    int main()
    {
        int a[] = { 12, 31, 25, 8, 32, 17 };
        int n = sizeof(a) / sizeof(a[0]);
        printf("Before sorting array elements are - \n");
        printArr(a, n);
        selection(a, n);
    }
}
```

```

printf("\nAfter sorting array elements are - \n");
printArr(a, n);
return 0;
}

```

4. Merge Sort or Divided Conquer

- Merge sorting is one of the important sorting techniques, also known as the divide and conquer algorithm.
- This technique will divide the unsorted array into 2 subgroups.
- The subarrays are again divided into two other arrays.
- Continue this process until we get a single element in the subarrays.
- Now, combine the single element sub-arrays to make two elements' subarrays in sorting order. Again, do the same thing for the two elements sub-arrays to make 4 elements sub-arrays in sorting order.
- Continue the above process until we get a sorted single Array with n elements, as shown below.

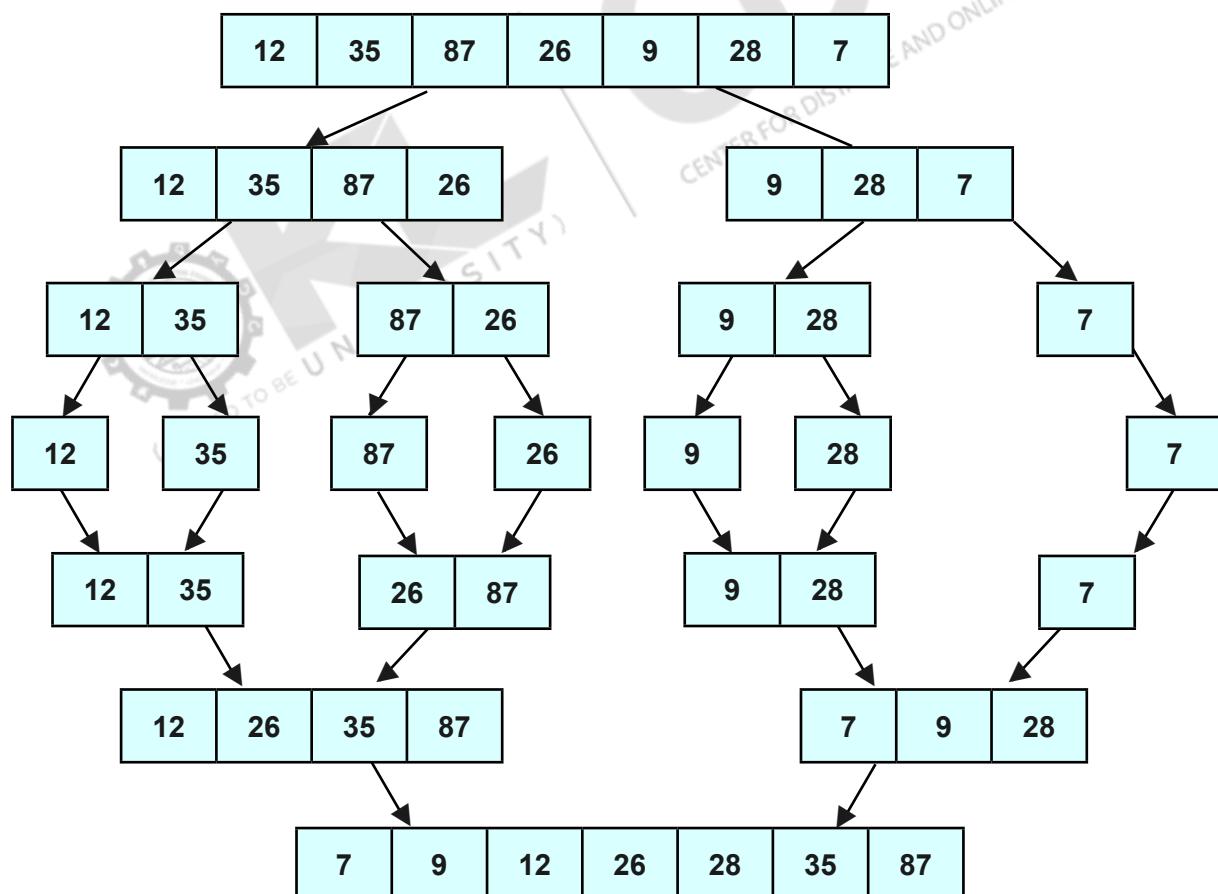


Fig. 3: Example of merge sort

Algorithm for merge Sort:

- Step 1:** Start
- Step 2:** Take an Array with n elements ($a[],n$).
- Step 3:** Divide the array into two sub-lists.
- Step 4:** Again, divide each sub-list into two other sub-lists.
- Step 5:** Repeat step 4 until the sub-lists contain only one element.
- Step 6:** Combine the sub-list and sort the elements in it
- Step 7:** Repeat step 6 until you make all sub-lists as a single array.
- Step 8:** Print the sorted Array
- Step 9:** Stop.

5. Quick sort:

Quick sort is a highly efficient sorting algorithm that partitions a given array into smaller arrays. It is designed based on the divide-and-conquer algorithm.

In this algorithm the three keys taken the major role.

1. Index
2. Current
3. Pivot

Generally, the pivot is taken as the last element of the array. Initially, we can take the index and current points as the first element of the array. Now, the current element will be compared with the pivot.

If the current element is less than the pivot, swap it with the index element and increment the index and current.

```
C<p
Swap the C with I
C++
I++
```

If the current element is greater than the pivot. Just increment the current position only.

```
C>p
C++
```

If the current element is equal to the pivot. Then, exchange the positions of the index and pivot. Now, the smaller elements of the pivot will be arranged on the left side of the pivot, and the elements greater than the pivot will be arranged on the right side of the pivot.

Now, the left side of the pivot elements should be treated as a sub-array, and the right-side elements of the pivot should be treated as another sub-array.

Now, continue the above process on sub-arrays. Then, finally, you will get sorted arrays.

Algorithm:

Step 1: Start

Step 2: Take a given array with “n” elements. ($a[], n$)

Step 3: Take pivot= $a[n-1]$

 Current=0, Index=0

Step 4: Compare current < pivot

Step 5: If step 4 is true,

 Exchange index value and current value and increment current and Index.

Step 6: If step 4 is false, check current > pivot.

Step 7: If step 6 is true, just increment current ($c++$)

 And go to step 4.

Step 8: Repeat steps 5, 6, 7 until the pivot is set into the correct position.

Step 9: Take all the right elements of the pivot and sub-array and take all the left elements of the pivot as another sub-array.

Step 10: Apply step 3 to step 8 on sub-arrays.

Step 11: Print the sorted array

Step 12: Stop

Program:

```
#include <stdio.h>
```

```
// Function to swap two elements
```

```
void swap(int* a, int* b) {
    int t = *a;
    *a = *b;
    *b = t;
```

```

}

int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = (low - 1);

    for (int j = low; j <= high - 1; j++) {
        if (arr[j] < pivot) {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }

    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

// Function to print the array
void printArray(int arr[], int size) {
    int i;
    for (i = 0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

int main() {
    int arr[] = { 12, 17, 6, 25, 1, 5 };
    int n = sizeof(arr) / sizeof(arr[0]);
    quickSort(arr, 0, n - 1);
}

```

```
printf("Sorted array: \n");
printArray(arr, n);
return 0;
}
```

6. Heap Sort

Heap Sort is a comparison-based sorting algorithm that uses a binary heap data structure. A binary heap is a complete binary tree with two types:

- **Max-Heap:** Each node's value is greater than or equal to its children's values in a max-heap. The largest value is at the root.
- **Min-Heap:** In a min-heap, each node's value is less than or equal to its children's values. The smallest value is at the root.

Heap Sort builds a maximum heap from the input array, repeatedly extracting the maximum element (the root of the heap) and moving it to the end of the array. It maintains the heap structure to sort the elements in place.

Steps in Heap Sort:

1. **Build the Max-Heap:** Convert the array into a max-heap, which arranges the largest element at the root.
2. **Extract the Maximum Element:** Swap the root element (the largest element) with the last element of the heap and reduce the heap size by one.
3. **Restore Heap Structure:** After each extraction, restore the max-heap property by “heapifying” the subtree.
4. Repeat the extraction and heapify process until the entire array is sorted.

Algorithm for Heap Sort:

Step 1: Start

Step 2: Take an unsorted array of 'n' elements ($a[]$, n).

Step 3: Build a max-heap from the array.

Step 4: Swap the root element (largest element) with the last element in the heap.

Step 5: Reduce the size of the heap by one, as the last element is in its correct position.

Step 6: Restore the max-heap structure for the remaining heap.

Step 7: Repeat steps 4–6 until the array is sorted.

Step 8: Print the sorted array.

Step 9: Stop.

Program for Heap Sort:

```
#include <stdio.h>

// Function to heapify a subtree rooted with node 'i' which is an index in arr[], n is size of heap
void heapify(int arr[], int n, int i) {
    int largest = i; // Initialize largest as root
    int left = 2 * i + 1; // left child
    int right = 2 * i + 2; // right child

    // If left child is larger than root
    if (left < n && arr[left] > arr[largest])
        largest = left;

    // If right child is larger than largest so far
    if (right < n && arr[right] > arr[largest])
        largest = right;

    // If largest is not root
    if (largest != i) {
        int temp = arr[i];
        arr[i] = arr[largest];
        arr[largest] = temp;

        // Recursively heapify the affected subtree
        heapify(arr, n, largest);
    }
}

// Main function to do heap sort
void heapSort(int arr[], int n) {
    // Build heap (rearrange array)
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);
}
```

```
// One by one extract an element from heap
for (int i = n - 1; i > 0; i--) {
    // Move current root to end
    int temp = arr[0];
    arr[0] = arr[i];
    arr[i] = temp;

    // call max heapify on the reduced heap
    heapify(arr, i, 0);
}

// Function to print the array
void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

// Main function
int main() {
    int arr[] = { 12, 11, 13, 5, 6, 7 };
    int n = sizeof(arr) / sizeof(arr[0]);

    heapSort(arr, n);

    printf("Sorted array: \n");
    printArray(arr, n);

    return 0;
}
```



Self-Assessment Questions

5. In _____ sort, the array is divided into a sorted and an unsorted section and elements from the unsorted section are inserted into their correct positions in the sorted section.
- a) Merge
 - b) Bubble
 - c) Insertion
 - d) Quick
6. The _____ sort algorithm repeatedly finds the smallest element from the unsorted part of the list and moves it to the sorted part.
- a) Selection
 - b) Bubble
 - c) Quick
 - d) Heap
7. The _____ sorting technique divides the array into two sub-lists, sorts each, and then merges them.
- a) Selection
 - b) Bubble
 - c) Merge
 - d) Insertion
8. Which of the following sorting techniques has a worst-case time complexity of $O(n^2)$?
- a) Merge Sort
 - b) Quick Sort
 - c) Bubble Sort
 - d) Heap Sort



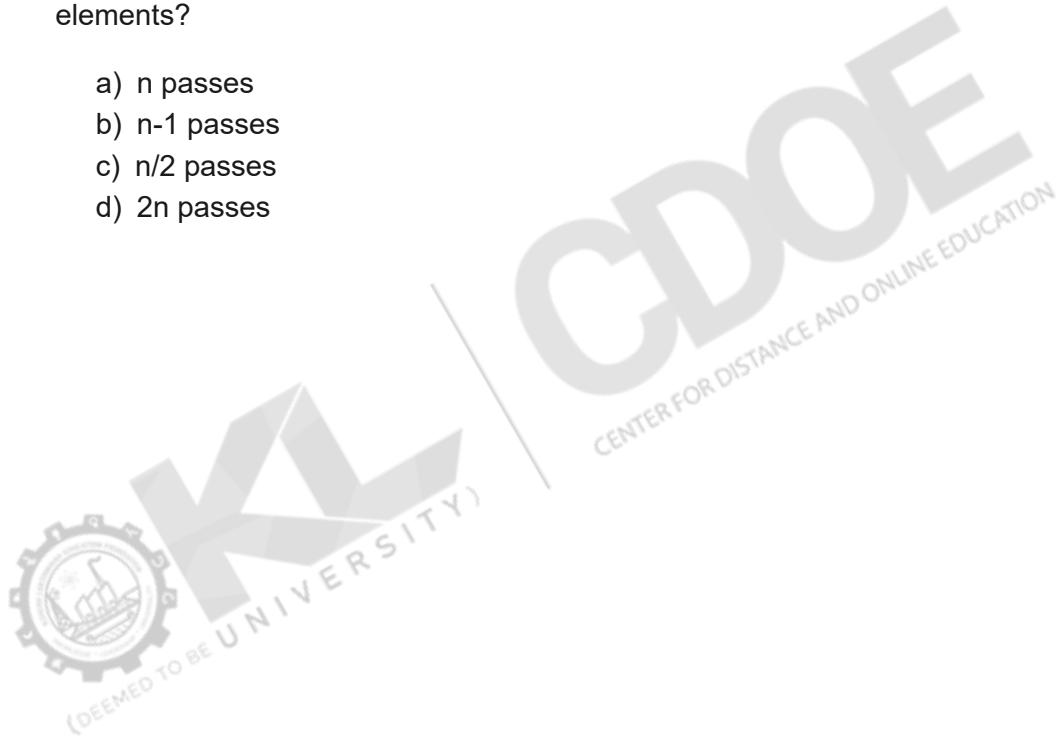
Self-Assessment Questions

9. Which of the following sorting algorithms is the most efficient for large datasets?

- a) Selection Sort
- b) Bubble Sort
- c) Insertion Sort
- d) Quick Sort

10. How many passes does Bubble Sort require to complete sorting an array with n elements?

- a) n passes
- b) $n-1$ passes
- c) $n/2$ passes
- d) $2n$ passes





Summary

- Searching is a fundamental operation to locate an array element. There are two main types of search methods: Linear Search and Binary Search.
- Linear Search sequentially checks each element from the beginning until it finds the element or reaches the end.
- Binary Search is a more efficient approach that applies only to sorted arrays.
- Sorting algorithms are essential tools in computer science for organising data into a specified order, typically ascending or descending.
- There are several common types of sorting algorithms, each with unique mechanisms and efficiencies.
- Bubble Sort iteratively compares and swaps adjacent elements, “bubbling” them into the correct order.
- In Selection Sort, the smallest element from the unsorted section is repeatedly moved to the sorted section.
- Insertion Sort builds a sorted list by placing each unsorted element into its correct position in the already sorted portion of the list.
- Merge Sort employs a divide-and-conquer strategy, recursively splitting an array into two halves, sorting each, and merging them back.
- Quick Sort also uses divide-and-conquer but selects a pivot to partition the data, sorting each partition separately.



Terminal Questions

1. Describe the Linear Search algorithm and its performance characteristics.
2. Explain the Binary Search algorithm and how it differs from Linear Search.
3. Define sorting and explain why it is essential in data processing.
4. Describe how the Bubble Sort algorithm works. What are its advantages and disadvantages?
5. Explain the steps involved in Insertion Sort and discuss a scenario where it is an efficient sorting choice.
6. Compare the time complexities of Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, and Quick Sort.



Answer Keys

Self-Assessment Questions	
Question No.	Answer
1	D
2	B
3	A
4	C
5	C
6	A
7	C
8	C
9	D
10	B



Activity

Activity Type: Offline

Duration: 1 hour

Assume you need to sort a list of employee records by their ID numbers. The list is nearly sorted, with only a few elements out of order. Discuss which sorting algorithm might be best for this situation and explain why certain algorithms might perform better with nearly sorted data.



Glossary

- **Divide-and-Conquer:** - A technique in which a problem is divided into smaller, more manageable sub-problems to simplify the solution.
- **Sorted Array:** A Sorted Array is an array in which the elements are arranged in a specific order, typically in ascending or descending order.
- **Unsorted Array:** An Unsorted Array is an array in which the elements are not arranged in any order.
- **Max-Heap:** A Max-Heap is a complete binary tree where each node's value is greater than or equal to the values of its children.
- **Min-Heap:** A Min-Heap is a complete binary tree where each node's value is less than or equal to the values of its children.



Bibliography

Textbooks

- Balagurusamy, E. (2017). *Data Structures*. McGraw Hill Education.
- Sedgewick, R., & Wayne, K. (2014). *Algorithms II*. Pearson Education.
- Sridhar, S. (2014). *Design and Analysis of Algorithms*. Oxford University Press.
- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (1989). *Introduction to Algorithms*. Prentice-Hall.



e-References

- **Searching Techniques:** <https://gcglhdh.org/downloads/e-Content/Learning-Material/Computer-Science/Search-Techniques-Data-Structure.pdf>
- **Searching and Sorting:** https://www.lkouniv.ac.in/site/writereaddata/siteContent/202003251324427324himanshu_Searching_Sorting.pdf



Video Links

Video	Links
Linear search vs Binary search	https://www.youtube.com/watch?v=sSYQ1H9-Vks
Sorting Algorithms	https://www.youtube.com/watch?v=rbbTd-gkajw



Image Credits

- **Fig. 1:** Self-Made
- **Fig. 2:** Self-Made
- **Fig. 3:** Self-Made



Keywords

- Searching
- Linear search
- Binary search
- Bubble sort
- Selection sort
- Insertion sort
- Quick sort
- Merge sort
- Heap sort

MODULE 4

Graphs, Searching, Sorting, and Hashing

Unit 3

Hashing



■ Unit Table of Contents

Unit 4.3 Hashing

Aim	363
Instructional Objectives	363
Learning Outcomes	363
4.3.1 Introduction	364
Self-Assessment Questions	370
4.3.2 Hash Function and Types of Hash Functions	371
Self-Assessment Questions	376
4.3.3 Collision Resolution Technique (CRT)	377
Self-Assessment Questions	381
4.3.4 Perfect Hashing	382
Self-Assessment Questions	385
Summary	386
Terminal Questions	386
Answer Keys	387
Activity	387
Glossary	388
Bibliography	388
e-References	388
Video Links	389
Image Credits	389
Keywords	389



Aim

To enable students to comprehensively interpret hashing concepts, hash functions, collision resolution techniques, and perfect hashing.



Instructional Objectives

This unit is designed to:

- Describe the concept of hash tables and their applications in computer science
- Explain the purpose and functionality of hash functions
- Define collision and explain its impact on hash table performance
- Discuss the two-level hashing structure: primary hash table and secondary hash tables



Learning Outcomes

At the end of the unit, the student is expected to:

- Summarise the concept of hashing and explain why it is essential for efficient data handling
- Evaluate various types of hash functions
- Demonstrate the use of open hashing with separate chaining
- Analyse the two-level hashing approach with primary and secondary hash tables to eliminate collisions

4.3.1 Introduction

In the C programming language, hashing is a method for transforming a significant amount of data into a smaller or fixed-size value known as a hash. This hash is created using a hash function, which maps input data to a concise output hash. Hashing facilitates efficient data searching, retrieval, and comparison, especially with extensive datasets. Hashing is widely applied in data structures like hash tables-arrays that organise data to allow rapid insertion, deletion, and retrieval. The hash function maps a key or data to an index within the hash table, the location for storing the data within the array.

Hashing offers several key benefits. It can reduce memory usage by compressing large data sets into smaller hash values. Additionally, hashing enhances algorithm performance by enabling faster data searching and retrieval. It can also support data integrity by identifying duplicate data and handling collisions, where two keys produce the same hash index.

The hashing process has three main steps: creating the hash function, generating the hash value, and storing data in the hash table. Creating a hash function involves designing an algorithm that consistently converts input data into a fixed-size hash value. This function should distribute data evenly within the hash table, minimising the chances of collisions. An effective hash function is typically simple, fast, and deterministic, producing the same output for identical input data. After creating the hash function, data can be passed through it to generate a hash value, which is then used as an index in the hash table to store the data.

When storing data in the hash table, the data is placed at the index indicated by the hash value. If a collision occurs where different keys map to the same index, the table may employ chaining, creating a linked list to hold all keys. Hashing in C can be implemented using division, multiplication, and folding methods. The division method determines the index by dividing the key by the table size and using the remainder. The multiplication method multiplies the key by a constant, using the fractional part of the result for indexing. In the folding method, the key is divided into parts and summed, and the result is used for indexing.

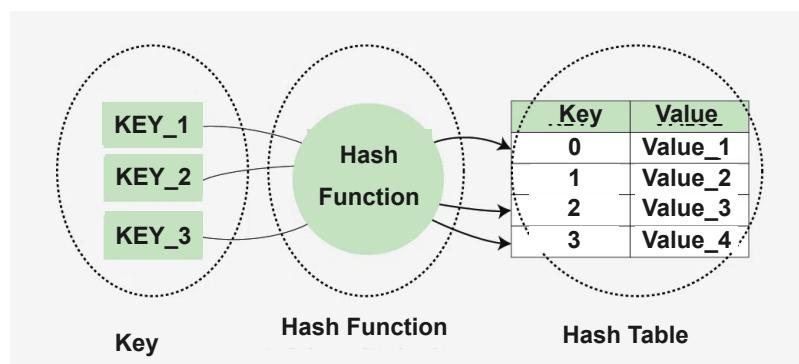


Fig. 1: Components of Hashing

Implementation of a hash table in C using arrays:

```
#include<stdio.h>
#define size 7
int array[size];
void init()
{
    int i;
    for(i = 0; i < size; i++)
        array[i] = -1;
}

void insert(int val)
{
    int key = val % size;

    if(array[key] == -1)
    {
        array[key] = val;
        printf("%d inserted at array[%d]\n", val, key);
    }
    else
    {
        printf("Collision : array[%d] has element %d already!\n", key, array[key]);
        printf("Unable to insert %d\n", val);
    }
}

void del(int val)
{
    int key = val % size;
    if(array[key] == val)
        array[key] = -1;
    else

```

```
printf("%d not present in the hash table\n",val);
}
```

```
void search(int val)
{
    int key = val % size;
    if(array[key] == val)
        printf("Search Found\n");
    else
        printf("Search Not Found\n");
}
```

```
void print()
{
    int i;
    for(i = 0; i< size; i++)
        printf("array[%d] = %d\n",i,array[i]);
}
```

```
int main()
{
    init();
    insert(10);
    insert(4);
    insert(2);
    insert(3);
```

```
    printf("Hash table\n");
    print();
    printf("\n");

    printf("Deleting value 10..\n");
    del(10);
```

```

printf("After the deletion hash table\n");
print();
printf("\n");

printf("Deleting value 5..\n");
del(5);
printf("After the deletion hash table\n");
print();
printf("\n");

printf("Searching value 4..\n");
search(4);
printf("Searching value 10..\n");
search(10);

return 0;
}

```

Output

10 inserted at array[3]
 4 inserted at array[4]
 2 inserted at array[2]
 Collision : array[3] has element 10 already!
 Unable to insert 3
 Hash table
 array[0] = -1
 array[1] = -1
 array[2] = 2
 array[3] = 10
 array[4] = 4
 array[5] = -1
 array[6] = -1

Deleting value 10..

After the deletion hash table

```
array[0] = -1  
array[1] = -1  
array[2] = 2  
array[3] = -1  
array[4] = 4  
array[5] = -1  
array[6] = -1
```

Deleting value 5..

5 not present in the hash table

After the deletion hash table

```
array[0] = -1  
array[1] = -1  
array[2] = 2  
array[3] = -1  
array[4] = 4  
array[5] = -1  
array[6] = -1
```

Searching value 4..

Search Found

Searching value 10..

Search Not Found

Hashing is frequently used in C programming for efficient data search and retrieval, especially for building data structures like hash tables or associative arrays. Here are some common uses, along with advantages and disadvantages of hashing in C:

Uses:

- Hashing is employed for efficient data lookup operations, such as finding a specific value in a large array or table.
- It is also used in constructing data structures like hash tables, which support constant-time lookup, insertion, and deletion.

Advantages:

- Hashing enables rapid data retrieval and search operations, making it highly valuable for managing large datasets where speed is concerned.
- It is relatively straightforward to implement in C and provides the basis for complex structures like hash tables or hash maps.
- Hashing can enhance data security, including password storage or encryption.

Disadvantages:

- Collisions may occur, leading to reduced efficiency and slower search operations.
- A well-designed hash function is essential to evenly distribute data in the table yet crafting such a function can be complex and time-intensive.
- Hashing can be memory-intensive, particularly for large hash tables or if collisions are frequent.
- In summary, hashing is a powerful technique for efficiently managing and searching large data collections. However, it has some limitations, such as potential collisions, the need for a carefully crafted hash function, and increased memory demands.



Self-Assessment Questions

1. Hashing transforms large data into a smaller or fixed-size value known as a _____.
 - a) Key
 - b) Hash
 - c) Index
 - d) Table

2. The division method of hashing finds the index by dividing the key by the table size and using the _____.
 - a) Quotient
 - b) Product
 - c) Remainder
 - d) Sum

3. Which of the following is NOT a step in the hashing process?
 - a) Creating a hash function
 - b) Generating a hash value
 - c) Storing data in the hash table
 - d) Sorting the hash table

4.3.2 Hash Function and Types of Hash Functions

In data structures, the hash function is a tool that maps data of arbitrary size into a fixed-size value. This function returns values such as a small integer (often called a hash value), hash codes, or hash sums. The hashing techniques in the data structure are very interesting, such as:

```
hash = hashfunc(key)
index = hash % array_size
```

Various hashing techniques add versatility to data structures, with requirements for a strong hash function including:

- **Ease of Computation:** The hash function should be simple and quick to compute.
- **Uniform Distribution:** It should distribute keys evenly across the hash table, preventing clustering.
- **Collision Avoidance:** Minimising collisions where two distinct elements are mapped to the same hash value is crucial.
- Another benefit of hashing is data integrity, as any modification in the data results in a completely different hash.

A good hash function possesses three main characteristics:

- **Collision Resistance:** It should minimise the likelihood of two inputs resulting in the same hash.
- **Hidden Property:** The function should conceal the original input details.
- **Puzzle Friendliness:** It should have properties that make it computationally challenging to reverse-engineer the hash.

Hash Table

In data structures, hashing uses hash tables to store key-value pairs, leveraging a hash function to produce an index. This index is then used to quickly execute insertion, updating, and retrieval operations. A hash table essentially acts like a “bucket” where data is stored in an array format, each with its index, making data retrieval fast if the index is known.

How Hashing Works in Data Structures:

The hashing process converts data, whether strings or numbers, into a small integer. Hash tables utilise the hashing function to retrieve items by assigning each element a unique key. The goal of hashing is to evenly distribute data across an array, creating a unique index for each item in the hash table.

In a hash table, data is stored as key-value pairs. The key is passed into the hashing function to generate an index uniquely identifying each value stored. This index keeps track of the value associated with that key, with the hash function returning a small integer output or hash value.

Consider a hash table with 30 cells where items are stored as key-value pairs to better interpret.

The values are: (3,21) (1,72) (40,36) (5,30) (11,44) (15,33) (18,12) (16,80) (38,99)

The hash table will look like the following:

Serial Number	Key	Hash	Array Index
1	3	$3\%30 = 3$	3
2	1	$1\%30 = 1$	1
3	40	$40\%30 = 10$	10
4	5	$5\%30 = 5$	5
5	11	$11\%30 = 11$	11
6	15	$15\%30 = 15$	15
7	18	$18\%30 = 18$	18
8	16	$16\%30 = 16$	16
9	38	$38\%30 = 8$	8

Table 1: Hash table example

Hashing involves taking data of any size and compressing it into a smaller “hash value,” which is used as an index in the hash table.

Types of Hash Functions

Here are the primary types of hash functions:

1. Division Method

The Division method is a simple and quick way to produce a hash value. The key value k is divided by M, and the remainder becomes the hash value.

Formula:

$$h(K) = k \bmod M$$

where,

- k is the key value
- M is the size of the hash table

Advantages:

- This approach is effective for most values of M.
- It requires only a single operation, making it very fast.

Disadvantages:

- Mapping consecutive keys to successive hash values can cause performance issues.
- Choosing an optimal M may require additional consideration.

Example:

$$k = 1987$$

$$M = 13h(1987) = 1987 \bmod 13$$

$$h(1987) = 4$$

2. Mid-Square Method

The Mid Square method squares the key and then selects the middle digits to create the hash value.

Formula:

$$h(K) = h(k \times k)$$

where,

- k is the key value

Advantage:

- This method is efficient because it uses most or all digits in the key, leading to a result not dominated by the high or low digits.

Disadvantages:

- For large keys, the square has many digits, complicating the process.
- There is a higher probability of collisions.

Example:

$k = 60$ Therefore, $k = k \times k$

$k = 60 \times 60$

$k = 3600$ Thus,

$h(60) = 60$

3. Folding Method

The Folding method breaks the key into equal parts and then sums these parts to obtain the hash value. The last part can have fewer digits if needed.

Formula:

$k = k_1, k_2, k_3, k_4, \dots, k_n$

$s = k_1 + k_2 + k_3 + k_4 + \dots + k_n$

$h(K) = s$

where,

- s is the sum of the parts of key k

Advantage:

- The key is divided evenly, resulting in a straightforward hash calculation.

Disadvantage:

- Excessive collisions can reduce efficiency.

Example:

$k = 12345$

$k_1 = 67; k_2 = 89; k_3 = 12$ Therefore, $s = k_1 + k_2 + k_3$

$s = 67 + 89 + 12$

$s = 168$

4. Multiplication Method

In the multiplication method, a constant A between 0 and 1 is chosen. This constant is multiplied with the key, and the resulting fractional part calculates the hash.

Formula:

$$h(K) = \text{floor}(M(kA \bmod 1))$$

where,

- M is the hash table size
- k is the key value
- A is the constant

Advantages:

- The method can use any constant between 0 and 1, though certain values work better for specific applications.

Disadvantages:

- This method works best when the table size is a power of two, as it simplifies index computation.

Example:

$$k = 5678$$

$$A = 0.6829$$

$$M = 200$$

Now, calculating the new value of $h(5678)$:

$$h(5678) = \text{floor}[200(5678 \times 0.6829 \bmod 1)]$$

$$h(5678) = \text{floor}[200(3881.5702 \bmod 1)]$$

$$h(5678) = \text{floor}[200(0.5702)]$$

$$h(5678) = \text{floor}[114.04]$$

$$h(5678) = 114$$

So, with the updated values, $h(5678)$ is 114.



Self-Assessment Questions

4. In a hash table, data is stored as _____ pairs.
- a) Key-value
 - b) Index-value
 - c) Data-value
 - d) Hash-value
5. The Multiplication method in hashing requires selecting a constant between _____.
- a) 0 and 2
 - b) 1 and 2
 - c) 0 and 1
 - d) -1 and 1
6. In the Folding method, the key is divided into _____, which are then summed.
- a) Squares
 - b) Equal parts
 - c) Constant values
 - d) None

4.3.3 Collision Resolution Technique (CRT)

A collision occurs when two keys are mapped to the same index in a hash table. This presents a challenge, as each index should ideally store only one entry. Various collision resolution techniques are used to handle these collisions and maintain hash table efficiency.

The process involves finding an alternate spot for the colliding key-value pair. Key collision resolution methods are:

1. Open Hashing (Separate Chaining)

- Chaining

2. Closed Hashing (Open Addressing)

- Linear Probing
- Quadratic Probing
- Double Hashing

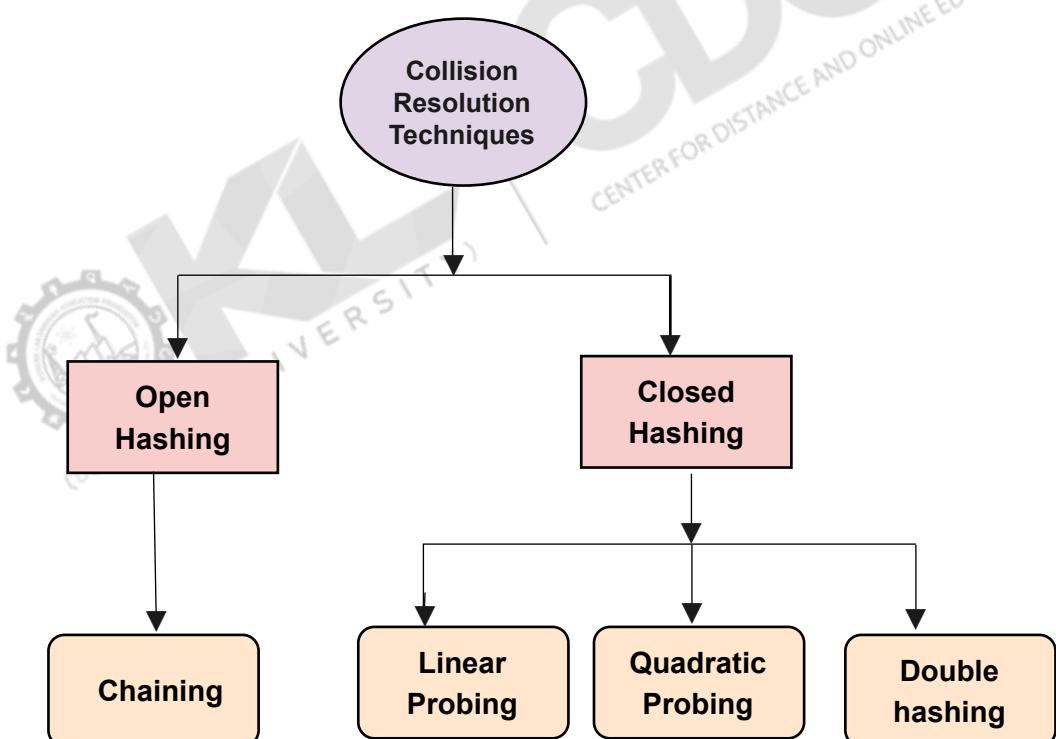


Fig. 2: Collision resolution methods

1. Open Hashing (Separate Chaining)

In Open Hashing, the Chaining method uses a linked list to store keys at an index where collisions occur. When a collision happens, the new key is added to a linked list at that index. This approach makes deletion straightforward due to the linked list structure; however, in the worst-case scenario, searching and deleting have a time complexity of $O(n)$, especially if many keys hash to the same index. Additionally, this method can be inefficient regarding space usage since key addresses are kept in the linked list rather than directly within the hash table.

2. Closed Hashing (Open Addressing)

Closed hashing, or open addressing, is another collision resolution method for hash tables. Unlike open hashing, which uses linked lists to manage collisions, closed hashing stores all elements within the hash table, eliminating the need for external structures.

In closed hashing, when a collision occurs (a hash slot is occupied), the algorithm searches for the next free slot directly in the table through a probing process until a space is found.

Here's how basic operations work in closed hashing:

- **Search:** To locate an element, calculate its hash value and probe through the table slots. If the desired element is found, retrieve it; if an empty slot appears during probing, the element is absent from the table.
- **Insert:** To add a new element, compute its hash value, then probe until an empty slot is available and place the element there.
- **Delete:** To delete an element, mark its slot as “deleted” or use a placeholder to signify it’s now vacant.

Linear Probing

When the computed index in the hash table is already occupied in linear probing, the method searches sequentially for the next available position. This approach probes linearly through the table until it finds an empty slot, making it one of the simplest methods to resolve collisions.

Linear Probing Example

Suppose you need to store several key-value pairs in a hash table of size 30. The given values are: (3,21), (1,72), (63,36), (5,30), (11,44), (15,33), (18,12), (16,80), and (46,99). If the hash function maps a key to an index that is already filled, linear probing will search for the next open index by moving sequentially, trying $(\text{hash}(n) + 1) \% T$, then $(\text{hash}(n) + 2) \% T$, and so on until an empty slot is found.

The hash table will look like the following:

Serial Number	Key	Hash	Array Index	Array Index after Linear Probing
1	3	$3 \% 30 = 3$	3	3
2	1	$1 \% 30 = 1$	1	1
3	63	$63 \% 30 = 3$	3	4
4	5	$5 \% 30 = 5$	5	5
5	11	$11 \% 30 = 11$	11	11
6	15	$15 \% 30 = 15$	15	15
7	18	$18 \% 30 = 18$	18	18
8	16	$16 \% 30 = 16$	16	16
9	46	$46 \% 30 = 8$	16	17

Table 2: Linear probing hash table

Quadratic Probing

Instead of probing the next slot linearly, the algorithm uses a quadratic function to determine the next probe position. The probing sequence follows a quadratic pattern until an empty slot is found.

The general formula for quadratic probing:

$$H(\text{key}, i) = (\text{Hash}(\text{key}) + c_1 i + c_2 i^2) \bmod \text{Table Size}$$

Quadratic probing may cause secondary clustering. Secondary clustering is another form of clustering in closed hashing that occurs when different keys produce the same probe sequence, leading to repeated collision patterns.

Double Hashing

Double hashing uses two hash functions to manage collisions. When the first hash function results in a collision, the second hash function generates an offset for the next available index.

The formula used is:

$$(\text{firstHash}(\text{key}) + i * \text{secondHash}(\text{key})) \% \text{sizeOfTable}$$

where i is the offset counter that increases until an open slot is found.

Example of Double Hashing Using two hash functions, h1 and h2, the steps to store a value are:

- First, check if $h_1(\text{key})$ is vacant. If it is, store the value there.
- If not, apply $h_2(\text{key})$ to calculate another index.
- Check $h_1(\text{key}) + h_2(\text{key})$; if it's available, place the value in that slot.
- If still full, continue with $h_1(\text{key}) + 2 \times h_2(\text{key})$, $h_1(\text{key}) + 3 \times h_2(\text{key})$, etc., until finding an empty index.

Double Hashing Example

Imagine you need to store some items inside a hash table of size 20. The values given are: (16, 8, 63, 9, 27, 37, 48, 5, 69, 34, 1).

$$h_1(n) = n \% 20$$

$$h_2(n) = n \% 13$$

$$n \ h(n, i) = (h_1(n) + ih_2(n)) \bmod 20$$

N	$h(n,i) = (h_1(n) + ih_2(n)) \% 20$
16	$i = 0, h(n,0) = 16$
8	$i = 0, h(n,0) = 8$
63	$i = 0, h(n,0) = 3$
9	$i = 0, h(n,0) = 9$
27	$i = 0, h(n,0) = 7$
37	$i = 0, h(n,0) = 17$
48	$i = 0, h(n,0) = 8$
	$i = 0, h(n,1) = 9$
	$i = 0, h(n,2) = 12$
5	$i = 0, h(n,0) = 5$
69	$i = 0, h(n,0) = 9$
	$i = 0, h(n,1) = 10$
34	$i = 0, h(n,0) = 14$
1	$i = 0, h(n,0) = 1$

Table 3: Example of double hashing



Self-Assessment Questions

7. In double hashing, if a collision occurs, a second _____ function generates an offset.
- a) Linear
 - b) Hash
 - c) Quadratic
 - d) Sequential
8. In Closed Hashing, all elements are stored _____ the hash table.
- a) Inside
 - b) Outside
 - c) At the end of
 - d) Before
9. Which of the following is NOT a Closed Hashing (Open Addressing) collision resolution method?
- a) Linear Probing
 - b) Quadratic Probing
 - c) Double Hashing
 - d) Separate Chaining

4.3.4 Perfect Hashing

Perfect hashing is explained as a model of hashing in which any set of n elements can be stored in a hash table of equal size and have lookups performed constantly. It was specifically invented and discussed by Fredman, Komlos and Szemerédi (1984) and has been nicknamed as “FKS Hashing”.

Perfect hashing is an advanced hashing technique that ensures no collisions occur. Each key is mapped to a unique index in the hash table. This makes it ideal for applications needing quick, constant-time data access, especially when the dataset is fixed (static perfect hashing).

Steps in Perfect Hashing

Perfect hashing is generally implemented in a two-level structure:

1. **First Level (Primary Hash Table):** A primary hash function distributes keys across buckets. If there are initial collisions, each bucket may contain multiple keys.
2. **Second Level (Secondary Hash Tables):** A secondary hash function creates a small, separate hash table for each bucket with multiple keys. This function carefully assigns each key a unique slot within the secondary hash table, ensuring no further collisions within that bucket.

Example of Perfect Hashing

Let's go through an example to clarify this process.

Suppose we have the following five keys: {31, 43, 56, 72, 89}. We'll map these keys into a hash table of size 5 using a two-level perfect hashing technique.

Step 1: Applying the Primary Hash Function

Let's use the primary hash function $h(k) = k \bmod 5$, which gives us initial buckets based on the remainders when the keys are divided by 5.

- $h(31)=31 \bmod 5=1$
- $h(43)=43 \bmod 5=3$
- $h(56)=56 \bmod 5=1$
- $h(72)=72 \bmod 5=2$
- $h(89)=89 \bmod 5=4$
- $h(89) 89 \bmod 5 = 4$

After this step, our primary hash table might look like this:

Bucket	Keys
0	None
1	31, 56
2	72
3	43
4	89

Since bucket 1 has multiple keys (31 and 56), we need to resolve this by creating a secondary hash table for that bucket.

Step 2: Applying the Secondary Hash Function

For bucket 1, we'll use a secondary hash function, say $h'(k) = k \bmod 2$, in a new secondary hash table with a size of 2 (for the two colliding keys). We need a secondary hash function that distributes 31 and 56 uniquely across the slots in this new table.

- $h'(31)=31 \bmod 2=1$
- $h'(56)=56 \bmod 2=0$

Now, we place each key in a unique slot in the secondary hash table:

Bucket 1 Secondary Table	Slot
31	1
56	0

Our primary hash table now points to individual secondary tables as needed, so we have unique slots for each key, eliminating collisions.

Summary of Operations in Perfect Hashing

- **Search:** To find a key like 31, the primary hash table points to bucket 1. From there, the secondary hash table (using $h'(31)$) will quickly locate 31 at its unique slot.
- **Insert/Delete:** New keys or deletions involve adjusting the secondary hash tables if needed, but perfect hashing assumes a mostly static data set, minimising frequent changes.

Applications of Perfect Hashing

Perfect hashing is highly effective in applications where a large set of static data requires fast lookups, such as:

- Symbol tables in compilers
- Databases with fixed or rarely updated records
- Caching systems for fast retrieval

Perfect hashing, especially static perfect hashing, allows for efficient lookups in $O(1)$ time and is highly predictable, making it ideal for systems with fixed or immutable datasets.





Self-Assessment Questions

10. Perfect Hashing is particularly useful for datasets that are _____.

- a) Dynamic
- b) Static
- c) Minimal
- d) Unsorted

11. A key advantage of Perfect Hashing is its ability to perform lookups in _____ time.

- a) $O(n)$
- b) $O(\log n)$
- c) $O(n^2)$
- d) $O(1)$

12. In Perfect Hashing, the primary hash table distributes keys into _____.

- a) Linked lists
- b) Buckets
- c) Trees
- d) Matrices



Summary

- In C programming, hashing is a technique to convert large datasets into smaller, fixed-size hash values, enabling efficient data storage, search, and retrieval.
- Hashing utilises hash functions to map data into concise hash values, which serve as indices in data structures like hash tables.
- The hash function distributes keys evenly across the table, minimising collisions, though techniques like chaining can manage collisions when they occur.
- Hash functions in data structures transform data of arbitrary size into a fixed-size hash value, allowing efficient data retrieval.
- A hash table that stores key-value pairs applies the hash function to place data at specific indices in an array, enabling quick insertion, updating, and retrieval.
- Types of hash functions include the Division, Mid-Square, Folding, and Multiplication methods, each with unique formulas and advantages for various applications.
- Collision resolution techniques in hashing are critical for managing situations where two or more keys map to the same index in a hash table.
- In Open Hashing, a linked list handles collisions, allowing multiple entries at a single index.
- In contrast, closed hashing stores all entries within the table and manages collisions through probing methods such as linear probing, Quadratic Probing, and Double Hashing.
- Perfect Hashing is an advanced hashing technique designed to prevent collisions by assigning each key to a unique index in the hash table.
- Developed by Fredman, Komlos, and Szemerédi in 1984, this method, also known as “FKS Hashing,” ensures constant-time lookups and is particularly efficient for static data sets.



Terminal Questions

1. Describe the three main steps involved in the hashing process.
2. Describe how a hash table uses a hash function to store data.
3. List and briefly describe four types of hash functions.
4. Explain the Open Hashing (Separate Chaining) technique and how it manages collisions.
5. Describe Closed Hashing (Open Addressing) and list its probing methods.
6. What is Perfect Hashing, and why is it beneficial for static datasets?



Answer Keys

Self-Assessment Questions	
Question No.	Answer
1	B
2	C
3	B
4	A
5	C
6	D
7	B
8	A
9	D
10	B
11	D
12	B



Activity

Activity Type: Offline

Duration: 1 hour

Imagine you are tasked with designing a hash function for a hash table that stores students' IDs (eight-digit numbers). Design a hash function and justify why it would effectively reduce collisions and evenly distribute keys in a hash table with 500 slots.



Glossary

- **Hash Value:** The hash function output, often used as an index in a hash table.
- **Secondary Clustering:** Clustering is caused when multiple keys follow the same probe sequence in hashing.
- **Key-Value Pair:** A set of two linked data items, one serving as an identifier (key) and the other as the associated data (value).
- **Static Data Set:** A dataset that does not change frequently, making it suitable for Perfect Hashing.
- **Symbol Table:** Compilers commonly store symbols and identifiers using data structures.



Bibliography

Textbooks

- Balagurusamy, E. (2017). *Data Structures*. McGraw Hill Education.
- Sedgewick, R., & Wayne, K. (2014). *Algorithms II*. Pearson Education.
- Sridhar, S. (2014). *Design and Analysis of Algorithms*. Oxford University Press.
- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (1989). *Introduction to Algorithms*. Prentice-Hall.



e-References

- **Hashing in Data Structure:** <https://www.naukri.com/code360/library/hashing-in-data-structure>
- **Hash Functions and Types of Hash functions:** <https://www.geeksforgeeks.org/hash-functions-and-list-types-of-hash-functions/>
- **Hashing and Collision Resolution Technique:** <https://apurva-komnak19.medium.com/hashing-and-collision-resolution-technique-254993baaff>
- **Perfect Hashing:** <https://www.kidsonthegenius.com/perfect-hashing/>



Video Links

Video	Links
Hashing in data structure	https://www.youtube.com/atch?v=W5q0xgxmRd8&list=PLxM5rzx4f4fwOPORqEZZhaaY5OG0WMZfF
Hash Functions	https://www.youtube.com/watch?v=GJ13Ks33GzU
Collision Resolution Techniques	https://www.youtube.com/watch?v=v63nhWwde_Q



Image Credits

- **Fig. 1:** Self-Made
- **Fig. 2:** Self-Made
- **Table 1:** Self-Made
- **Table 2:** Self-Made
- **Table 3:** Self-Made



Keywords

- Hashing
- Hash function
- Collision
- Collision resolution technique (CRT)
- Hash table
- Perfect hashing
- FKS hashing



DATA STRUCTURES AND ALGORITHMS



🌐 <https://www.kluniversity.in/cdoe/>
✉️ supportcdoe@kluniversity.in