DATA STRUCTURE IN C

M.Sc (COMPUTER SCIENCE)
SEMESTER-I, PAPER-I

Lesson Writers:

Dr. Neelima Guntupalli
Asst.Professor,
Dept. of Comp.Science,
Acharya Nagarjuna University,
Nagarjunanagar – 522 510

Dr. Vasantha Rudramalla
Faculty
Dept. CS &E
Acharya Nagarjuna University
Nagarjunanagar – 522 510

Mrs. Appikatla Pushpa Latha
Faculty, Dept. of CS&E
Acharya Nagarjuna University
Nagarjunanagar – 522 510

Dr.U. Surya Kameswari
Asst. Professor
Acharya Nagarjuna University
Nagarjunanagar – 522 510

Editor

Dr. K. Lavanya
Asst. Professor,
Dept. Of CS&E
Acharya Nagarjuna University
Nagarjunanagar – 522 510.

Director, I/c.
Prof. V. Venkateswarlu
M.A., M.P.S., M.S.W ., M.Phil., Ph.D.
Professor
Centre for Distance Education
Acharya Nagarjuna University
Nagarjuna Nagar 522 510

Ph: 0863-2346222, 2346208
0863- 2346259 (Study Material)
Website www.anucde.info
E-mail: anucdedirector@gmail.com

M.Sc Computer Science

First Edition : 2025

No. of Copies :

This book is exclusively prepared for the use of students of M.Sc (Computer Science), Centre for Distance Education, Acharya Nagarjuna University and this book is meant for limited circulation only.

Published by:

Director **I/c**
Prof. V. Venkateswarlu,
M.A., M.P.S., M.S.W . M.Phil., Ph.D.
Centre for Distance Education,
Acharya Nagarjuna University

Printed at:

FOREWORD

Since its establishment in 1976, Acharya Nagarjuna University has been forging ahead in the path of progress and dynamism, offering a variety of courses and research contributions. I am extremely happy that by gaining 'A+' grade from the NAAC in the year 2024, Acharya Nagarjuna University is offering educational opportunities at the UG, PG levels apart from research degrees to students from over 221 affiliated colleges spread over the two districts of Guntur and Prakasam.

The University has also started the Centre for Distance Education in 2003-04 with the aim of taking higher education to the door step of all the sectors of the society. The centre will be a great help to those who cannot join in colleges, those who cannot afford the exorbitant fees as regular students, and even to housewives desirous of pursuing higher studies. Acharya Nagarjuna University has started offering B.Sc., B.A., B.B.A., and B.Com courses at the Degree level and M.A., M.Com., M.Sc., M.B.A., and L.L.M., courses at the PG level from the academic year 2003-2004 onwards.

To facilitate easier understanding by students studying through the distance mode, these self-instruction materials have been prepared by eminent and experienced teachers. The lessons have been drafted with great care and expertise in the stipulated time by these teachers. Constructive ideas and scholarly suggestions are welcome from students and teachers involved respectively. Such ideas will be incorporated for the greater efficacy of this distance mode of education. For clarification of doubts and feedback, weekly classes and contact classes will be arranged at the UG and PG levels respectively.

It is my aim that students getting higher education through the Centre for Distance Education should improve their qualification, have better employment opportunities and in turn be part of country's progress. It is my fond desire that in the years to come, the Centre for Distance Education will go from strength to strength in the form of new courses and by catering to larger number of people. My congratulations to all the Directors, Academic Coordinators, Editors and Lesson-writers of the Centre who have helped in these endeavors.

Prof. K. Gangadhara Rao
M.Tech., Ph.D.,
Vice-Chancellor I/c
Acharya Nagarjuna University

M.Sc. Computer Science
Semester-I, Paper-I
101CP24 -Data Structure In C

Syllabus

UNIT-1

Arrays and Structures Arrays, Dynamically allocated arrays, Structures and Unions, polynomials. Stacks and Queues Stacks, Stacks using Dynamic Arrays, Queues, Circular queues using dynamic arrays, Evaluation of expressions, multiple stacks and queues.

UNIT-2

Linked List - Single Linked List and chains, Representing chains in C, Linked stacks and queues, polynomials, Polynomial representation, Adding polynomials, Additional list operations, Operations on chains, Operations for Circularly linked lists, Sparse Matrices, Sparse Matrix representation, Doubly Linked lists.

UNIT-3

Introduction Terminology, Representation of trees. Binary Trees- The abstract data type, Properties of binary trees, Binary tree representations. Binary tree traversals – In order traversal, Preorder traversal, Post order traversal. Threaded Binary trees Threads In order traversal of a threaded binary tree. Binary Search Trees - Definition, Searching a BST, Insertion into a BST, Deletion from a BST.

UNIT-4

Sorting-Motivation, Insertion sort, Quick sort, Merge sort, Heap sort, External sorting. Hashing - Introduction, Static hashing, Hash tables, hash functions, Overflow handling

UNIT-5

Graphs The graph abstract data type, Introduction, definitions, graph representations. Elementary graph operations -Depth First Search, Breadth First Search, Connected Components, Spanning trees, Biconnected Components. Minimum cost Spanning trees - Kruskals Algorithm, Prims algorithm. Shortest paths - Single source problem, all pairs shortest path.

Prescribed Book

Horowitz, Sahani, Anderson - Freed, "Fundamentals of Data

Structures in C" Chapters 2-8

Reference Book

1. D SAMANTA, "Classic Data Structures", -PHI

2. Balagurusamy, "C Programming and Data Structures", Third Edition, TMH (2008).

(101CP24)

M.SC DEGREE EXAMINATION, Model QP

Computer Science – First Semester

DATA STRUCTURES IN C

Time: 3hours Max. Marks: 70

Answer ONE Question from each unit 5 x 14-70 M

Unit-I

1. a) Define structure. Explain how to create a structure
   b) Write a program to push and pop elements into and from the stack

(OR)

2. a) Write the applications for queue. Write pseudo code for inserting and deleting elements from the queue.
   b) Write the advantages of Circular queue over Queue.

Unit-II

3. a) Explain linear chain with example.
   b) Write a program to multiply two polynomials

(OR)

4. a) Explain single linked list and its operations with an examples
   b) Develop an algorithm to insert an element into double linked list.

Unit-III

5. Explain about Trees as nonlinear data strcutures and explain binary search trees

(OR)

6. Explain various tree traversal techniques and explain Threaded binary trees

Unit-IV

7. Illustrate Quick sort through an example.

(OR)

8. a) What is Bubble sort ?
   b) Explain static hashing.

Unit-V

9. a) Write the ways to represent Graphs as a nonlinear data structure.
   b) Explain Kruskals Algorithm.

(OR)

10. a) Explain Breadth first search and Depth first search.
    b) Explain all pairs shortest path.

CONTENTS

LESSON - 1

ARRAYS

OBJECTIVES

The objectives of this lesson are to

- Understand arrays and structures as fundamental data structures in C programming.
- Learn how arrays and structures are used to efficiently store, access, and manage data.
- Gain insight into the key characteristics, advantages, and limitations of arrays.
- Comprehend the internal memory representation of arrays for efficient data access.
- Apply arrays to solve real-world programming problems, laying a foundation for advanced data structures.

STRUCTURE

1.1 INTRODUCTION

A data structure is a way of organizing and storing data in a computer so that it can be accessed and used efficiently. Think of data structures like tools in a toolbox: each tool has a specific purpose, and using the right tool makes tasks easier. In the same way, each data structure is designed to handle data in a way that makes it easier to perform certain operations, like searching for an item, sorting items in a specific order, or modifying elements in a dataset.

Data structures are crucial tools in programming, allowing us to organize and manage data efficiently. In C programming, data structures help in storing and accessing data in ways that make it easy to manipulate, search, and manage within a program. Two of the most fundamental and commonly used data structures in C are arrays and structures.

Consider some real-life examples to understand the importance of data structures:

- **Folders on a computer**: You may have folders for different types of documents, images, and videos, and within those folders, you might organize items by name or date. This structure helps you quickly find what you're looking for.
- **Library organization**: Libraries organize books by categories, authors, and titles, making it easy to locate a specific book.

Data structures work in a similar way by providing methods for storing, organizing, and retrieving data effectively.

Using an appropriate data structure can help solve problems more efficiently. The right data structure allows:

- **Faster access** to data (e.g., using a sorted list when you often need to find specific items).
- **Efficient memory use** by storing data compactly.
- **Better performance** by optimizing the time required for operations like insertion, deletion, and search.

1.2 TYPES OF DATA STRUCTURES

Data structures can be classified into two main categories: **primitive** and **non-primitive**. Each type of data structure serves a specific purpose and is used for different kinds of tasks. Below is a breakdown of the types of data structures:

1.2.1 Primitive Data Structures

These are the basic types of data structures that store single values and are directly supported by most programming languages.

- **Integer**: A whole number (e.g., 1, -45, 78).
- **Float**: A number with a fractional part (e.g., 3.14, -0.001).
- **Character**: A single alphabet or symbol (e.g., 'A', 'b').
- **Boolean**: A data type that can only store true or false.
- **String**: A sequence of characters (e.g., "Hello", "World").

1.2.2 Non-Primitive Data Structures

These data structures are more complex and can store multiple values. They are built using primitive data types.

❖ **Linear Data Structures**

In linear data structures, data elements are stored in a sequential manner.

- **Array**: A collection of elements of the same type, stored in contiguous memory locations.
- **Linked List**: A linear collection of elements (nodes) where each node points to the next node in the sequence.
- **Stack**: A collection of elements that follows the Last In First Out (LIFO) principle.
- **Queue**: A collection of elements that follows the First In First Out (FIFO) principle.

❖ **Non-Linear Data Structures**

In non-linear data structures, elements are stored in a hierarchical manner.

- **Tree**: A hierarchical structure with a root element and sub-elements (children). Examples include binary trees, AVL trees, and heap trees.
- **Graph**: A collection of nodes (vertices) connected by edges. Graphs can be directed or undirected, and they can represent networks, relationships, etc.

❖ **Hash-Based Data Structures**

❖ These are used for fast data retrieval.

- **Hash Table**: A data structure that stores key-value pairs. The key is hashed to an index, making retrieval efficient.

1.3 INTRODUCTION TO ARRAYS

An **array** is a collection of elements of the same type, stored in contiguous memory locations. Arrays are useful when you need to store multiple values of the same type and want easy, direct access to each element by an index.

1.3.1 Characteristics of Arrays:

- **Fixed Size**: Once an array is created, its size cannot be changed. If you create an array with 10 elements, you can store only 10 elements in it.
- **Same Data Type**: All elements in an array must be of the same data type (e.g., all integers, all characters).
- **Direct Access**: Each element in an array can be accessed directly using an index. The first element is at index 0, the second at index 1, and so on.

❖ **Example of an Array**

Imagine you want to store the test scores of 5 students. Instead of creating a separate variable for each score, you can use an array to store all the scores in one place.

```
int scores[5] = {85, 90, 78, 92, 88};
```

In this example:

- scores[0] is 85 (score of the first student).
- scores[1] is 90 (score of the second student).
- scores[2] is 78, and so on.

Consider a class of students whose ages we need to store and update. Here's an example in C:

```
int main( )
{
   int ages[3] = {15, 16, 17};  // Array to store ages of 3 students
   printf("First student age: %d\n", ages[0]);  // Output: 15
   ages[1] = 18;    // Update age of second student
   printf("Updated age of second student: %d\n", ages[1]);  // Output: 18
   return 0;
}
```

In this code:

- We define an array ages with 3 elements, storing ages of students.
- We access ages[0] to get the first student's age.
- We update ages[1] to set the second student's age to 18.

Arrays are a simple yet powerful way to store collections of data and access individual elements efficiently by index. However, if flexibility or diverse data types are needed, other data structures may be more suitable.

Using data structures effectively can greatly improve the efficiency and clarity of your programs, enabling faster data processing and more manageable code.

❖ **Accessing Array Elements**

To access or modify elements in an array, you use the index:

- Access: scores[2] will give you 78.
- Modify: You can update the score of the third student by assigning a new value: scores[2] = 80;.

Arrays are widely used because they allow quick access to elements, but they also have limitations:

- They can't grow or shrink after being defined.
- Inserting or deleting elements can be cumbersome, as it may require shifting elements.

1.3.2 Comparison of Arrays and Other Data Structures

Table 1: Differences between Arrays and other Data Structures

Feature	Array	Other Data Structures (e.g., Linked List)
Fixed Size	Yes, size is fixed after creation	No, size can grow or shrink dynamically
Same Data Type	Yes, all elements must be of the same type	Depends on structure
Access Time	Fast, direct access by index	Slower in some structures (e.g., linked lists require traversal)
Memory Efficiency	Uses continuous memory, can be efficient	May use more memory depending on structure

1.3.3 The Role of Arrays in C

An array is a collection of elements, all of the same type, stored in consecutive memory locations. Arrays are used when we need to store a list of items where each item is of the same type, such as a list of numbers, names, or temperatures.

1. **Uniformity**: Since all elements in an array are of the same data type (like int, float, or char), we can perform the same types of operations on all of them. For example, if we have an array of integers, we can easily calculate the sum of all elements or find the largest number.
2. **Contiguous Memory**: Arrays are stored in a contiguous block of memory, meaning that each element follows the previous one in memory. This arrangement allows quick and easy access to each element using its index (position) in the array. Accessing an element by its index is fast, making arrays efficient for tasks that involve frequently accessing or updating elements at known positions.
3. **Fixed Size**: In C, arrays have a fixed size defined at the time of declaration. This means that we need to know in advance how many elements we want to store in the array. Once created, an array cannot grow or shrink during program execution.

For example, an array can represent a sequence of student scores or daily temperatures for a month, where all items in the array are of the same data type. By accessing each item using

its index, we can perform various operations like calculating averages, finding maximums, or sorting.

1.3.4 Why Arrays and Structures are Fundamental

Both arrays and structures are fundamental data structures in C because they provide the building blocks for organizing and working with data in different ways. While arrays allow us to handle multiple values of the same type, structures let us group various types of related data. Together, they enable us to represent real-world data more effectively and efficiently, forming the basis for more complex data structures and programming techniques.

For instance, a database application might use arrays to store lists of records, where each record is represented by a structure. Alternatively, an application that processes text could use an array of structures to hold information about each word and its attributes, like position and frequency.

In summary:

1. **Arrays** are ideal for managing collections of homogeneous (same type) data elements.
2. **Structures** allow for managing collections of heterogeneous (different types) data elements under a single entity.

These two data structures, along with their unique characteristics, help programmers organize data in an organized and efficient way, making it easier to build powerful and maintainable programs. They lay the groundwork for learning and using more advanced data structures, such as linked lists, stacks, queues, and trees, which are built upon the principles of arrays and structures.

1.4. ARRAYS AS AN ABSTRACT DATA TYPE (ADT)

An **array** is one of the most commonly used data structures in programming. It allows for the storage of multiple elements of the same type in a contiguous block of memory. In C, arrays are useful for managing lists of data, such as a series of numbers, strings, or other types, that we need to process collectively.

Arrays are known as an **Abstract Data Type (ADT)** because they allow a logical way to manage data, independent of the physical details of how they are implemented in memory. When working with arrays, we think of them as a series of slots where we can store data in a structured way. This structure allows efficient access to elements, especially when we know the position of the element we want to retrieve or update.

Arrays have some specific characteristics that make them unique and useful for many types of data processing:

1. **Fixed Size**:
    - The size of an array is specified at the time of declaration and is fixed. This size determines the maximum number of elements the array can hold.

- Once an array is created with a specific size, this size cannot be changed during program execution. This feature helps the compiler allocate a fixed block of memory, simplifying memory management.
- For example, if we declare an array int numbers[5], this array can hold exactly 5 integers, and no more.

2. **Homogeneous Elements**:
   - All elements in an array must be of the same data type. This means an array of integers can only hold integer values, an array of floats can only hold floats, etc.
   - This characteristic helps ensure that all elements have the same memory size, making it possible to calculate and access their positions using an index.

3. **Indexed Access**:
   - Arrays use indices (positions) to access each element. In C, indexing starts from 0, meaning the first element is at position 0, the second at position 1, and so on.
   - Indexed access allows us to directly retrieve or modify elements using their position in the array, making operations like searching, sorting, and iterating very efficient.

In C programming, arrays are a crucial data structure used to store and manage collections of data. C provides several types of arrays, each serving unique purposes depending on how the data needs to be accessed and managed. Here's an in-depth look at the different types of arrays in C:

1.4.1. Single Dimensional Arrays

A one-dimensional array is the simplest form of an array, consisting of a single row of elements, all of which are of the same data type. It is essentially a linear list where elements are accessed using a single index, making it ideal for storing simple lists of data, such as scores, prices, or a sequence of numbers.

Declaration and Usage:

```
int numbers[5];  // Declaration of a one-dimensional array of integers
```

This code creates an array numbers with space for 5 integer elements, each accessed by an index, starting from 0 to 4.

Example:

```
int numbers[5] = {10, 20, 30, 40, 50}; // Initialization with values
printf("%d", numbers[2]);  // Outputs 30
```

Applications:

- Storing simple data collections, such as student scores, age groups, or item prices.
- Using a loop for traversal and accessing elements in a sequential manner.

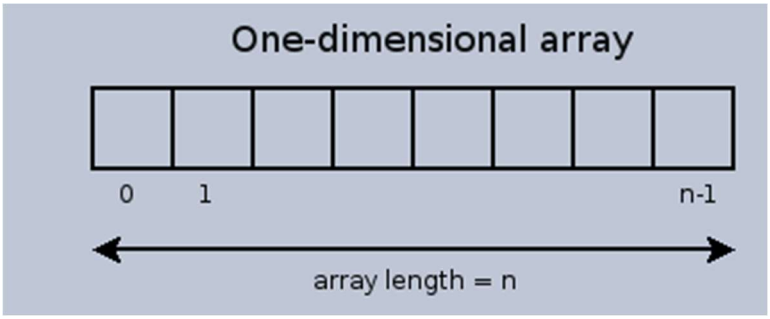


Fig 1.1 Single dimensional array

1.4.2. Two-Dimensional Arrays

A two-dimensional array in C represents data in a matrix or table form, where data is stored in rows and columns. Two-dimensional arrays are particularly useful for representing grids, tables, or matrices, where elements are accessed using two indices, one for the row and one for the column.

Declaration and Usage:

int matrix[3] [3]; // Declaration of a 3x3 two-dimensional array of integers

This code creates a 3x3 matrix that can store 9 integer elements in total.

Example:

int matrix[2] [3] = { {1, 2, 3}, {4, 5, 6} }; // Initializing a 2x3 array

printf("%d", matrix[1][2]); // Outputs 6

Applications:

- Representing matrices in mathematical computations.
- Storing tables of data, like the scores of multiple players across several games.
- Used in graphical applications to represent pixel data.

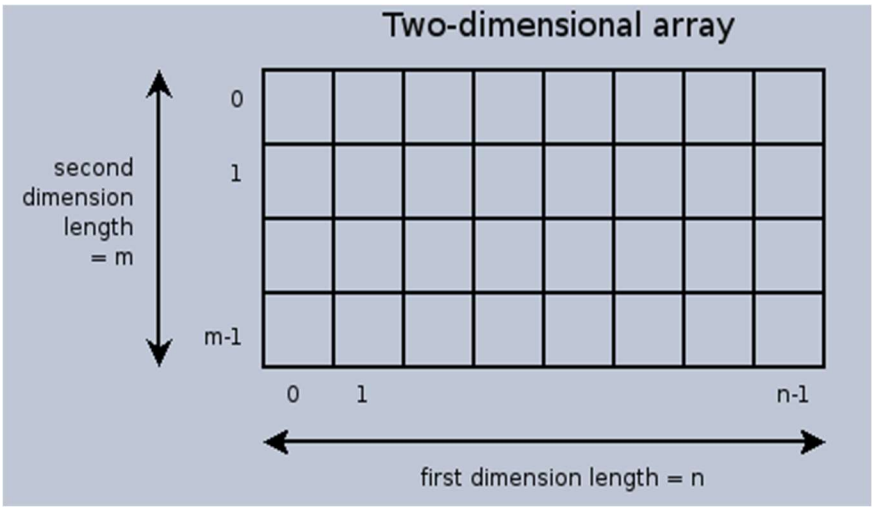


Fig 1.2 Two dimensional array

1.4.3. Multi-Dimensional Arrays

A multi-dimensional array extends beyond two dimensions, allowing for three or more dimensions to store complex data structures. In C, three-dimensional arrays are most common, but arrays can have any number of dimensions, though they become increasingly complex to manage and visualize.

Declaration and Usage:

int a[3] [3] [3]; // Declaration of a 3x3x3 three-dimensional array of integers

This array can store 27 elements (3 rows × 3 columns × 3 depth levels).

Example:

```
int a [2] [2] [2] = { {{1, 2}, {3, 4}}, {{5, 6}, {7, 8}}};
printf ("%d", a [1] [1] [0] ); // Outputs 7
```

Applications:

- Representing spatial data, such as a 3D model or a volume of data points.
- Used in scientific simulations to model data in three-dimensional space.
- Employed in games and graphics for manipulating multi-layered data.

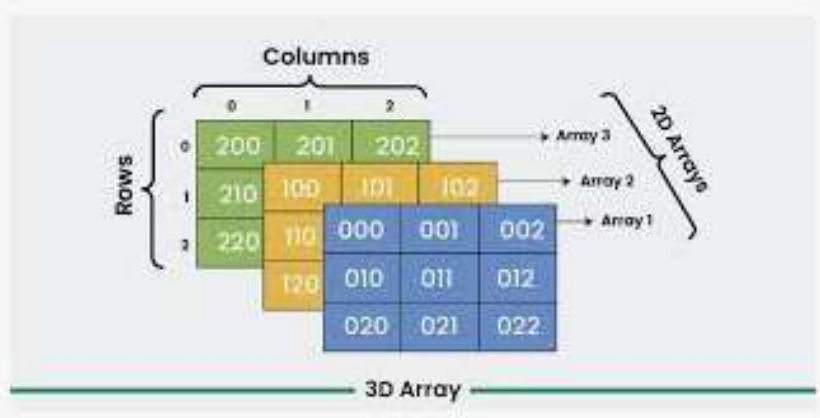


Fig 1.3 Three dimensional array

1.4.4. Dynamic Arrays

Unlike static arrays (one-dimensional, two-dimensional, and multi-dimensional), which have a fixed size determined at compile time, dynamic arrays allow for memory allocation at runtime, making it possible to resize the array as needed. In C, dynamic arrays are created using pointers and functions like malloc or calloc for memory allocation.

Declaration and Usage:

int *array = (int*) malloc (size * sizeof (int)); // Allocates memory dynamically

The above code allocates an array of integers of size size dynamically.

Example:

```
int *dynamicArray;
int size = 5;
dynamicArray = (int*) malloc (size * sizeof ( int ) );  // Allocate memory for 5 integers
for(int i = 0; i < size; i++)
{
   dynamicArray [i] = i + 1;  // Assign values to array
}
free (dynamicArray);  // Free the allocated memory
```

Applications:

- Useful in cases where the size of the data structure is unknown at compile time.
- Allows resizing the array when more elements need to be added, such as in dynamic lists.
- Commonly used in applications that require flexible memory management.

1.5 ARRAY OPERATIONS

Arrays support several fundamental operations, which help us work with data more effectively:

1. **Element Access**:
   - To access an element in an array, we use its index. For example, if we have an array int numbers[5] = {10, 20, 30, 40, 50};, we can access the first element with numbers[0], which gives 10, the second element with numbers[1], which gives 20, and so on.
   - Direct access by index allows quick retrieval or updating of any element in constant time, denoted as **O(1)** in algorithmic terms.
2. **Traversal**:
   - Traversing an array means accessing each element one by one, usually in a loop. Traversal is essential for operations that involve processing all elements, such as calculating the sum or finding the maximum value.
   - In C, we can use for or while loops to iterate through the array using each index in sequence.

3. **Insertion and Deletion**:

   - Inserting or deleting elements in an array is not straightforward, as arrays have a fixed size. If we want to insert a new element, we often need to shift other elements to make space, which can be time-consuming for large arrays.
   - Similarly, deleting an element requires shifting elements to fill the gap left by the deleted item.
   - Dynamic insertion and deletion are generally easier to handle in linked lists, but arrays provide faster access to any element.

1.6 MEMORY LAYOUT AND INDEX CALCULATION

When we declare an array in C, the compiler assigns a continuous block of memory for the array elements. For an array of integers, each element typically occupies 4 bytes of memory (depending on the system architecture).

For example, let's assume the base address of an integer array numbers is 1000, and each integer occupies 4 bytes:

- numbers [0] is located at memory address 1000.
- numbers[1] is at 1004 (1000 + 4).
- numbers[2] is at 1008 (1000 + 2*4), and so on.

The memory address of an element numbers[i] in an array can be calculated as:

Address of numbers[i] = Base Address + (i * Size of each element)

This calculation makes it easy for the compiler to access any element by index in constant time.

1.6.1 Visual Representation of Arrays

Here's a simple visual representation of an array with five integer elements:

In this example:

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

- numbers[0] holds 10.
- numbers[1] holds 20.
- numbers[2] holds 30, and so on.

The contiguous nature of the array means that if we know the memory address of numbers[0], we can calculate the address of any other element by simply adding the product of the index and the element's size in memory.

1.7 INTERNAL MEMORY REPRESENTATION OF ARRAYS

In C programming, arrays are a fundamental data structure that benefits from an efficient memory layout. Arrays are stored in contiguous blocks of memory, meaning that each element is positioned directly after the previous one. This contiguous storage method allows for direct, rapid access to any element based on its index and the array's starting location, or *base address*.

1.7.1 Contiguous Memory Storage: Advantages and Characteristics

The contiguous allocation of memory for arrays provides two main advantages:

1. **Efficient Element Access:** Accessing any element in an array is quick and takes constant time (O(1)) because the compiler can calculate the exact memory address of any element based on its index.

2. **Predictable Memory Structure:** Since each element in an array is stored in a sequential order, it minimizes memory fragmentation, making it simpler to manage memory space, particularly when dealing with large datasets.

For example, in a two-dimensional array, this contiguous structure allows the compiler to map elements in either a row-major or column-major order, depending on the system, ensuring that all elements are still stored in one continuous block of memory.

1.7.2 Memory Address Calculation

In C, each data type occupies a specific amount of memory (e.g., an int may take 4 bytes, and a float may take 4 bytes). The memory address of each element in an array can be calculated using the **base address** and the **index** of the element.

For an integer array a of size n, the memory address of an element a[i] can be calculated using this formula:

Address of a[i] = Base Address + (i * Size of int)

Here:

- **Base Address**: The starting memory location of the array.
- **i**: The index of the element we want to access.
- **Size of int**: The number of bytes occupied by each integer (typically 4 bytes on most systems).

This formula is essential for efficient access to elements in the array because it allows the program to jump directly to the correct location in memory using the index, without needing to search through each element sequentially.

❖ **Single dimensional Arrays**

When an array is created, the compiler allocates a continuous block of memory that can store all the elements of the array.

Example 1:
An integer array int a[3] = {10, 20, 30}; would be stored in memory as follows:

Index	0	1	2
Value	10	20	30

If the base address of the array a is 1000 and each integer occupies 4 bytes in memory, then:

- a[0] is stored at address 1000
- a[1] is stored at address 1004
- a[2] is stored at address 1008

This layout allows the compiler to quickly calculate the address of any element a[i] by using a formula, avoiding the need for repeated memory searches.

Example 2:
Consider an array int b[5] = {5, 10, 15, 20, 25}; with the following characteristics:

- **Base Address**: 2000
- **Element Size**: 4 bytes (since each int occupies 4 bytes)

Using our formula, we can calculate the memory address of each element:

Table 1.1 Memory Allocation of an array

Element	Index (i)	Memory Address
b[0]	0	2000 + (0 * 4) = 2000
b[1]	1	2000 + (1 * 4) = 2004
b[2]	2	2000 + (2 * 4) = 2008
b[3]	3	2000 + (3 * 4) = 2012
b[4]	4	2000 + (4 * 4) = 2016

This table shows how each element is stored at a unique memory location, calculated based on the base address and the index.

Visual Representation of Array Memory Layout

Below is a visual representation of the memory layout for b[5] in this example, with each box representing 4 bytes of memory:

Table 1.2 Memory Allocation of a single dimensional array

Address	2000	2004	2008	2012	2016
Element	b[0]	b[1]	b[2]	b[3]	b[4]
Value	5	10	15	20	25

Calculating Element Addresses

In C, the address of an element within an array can be derived using the formula:

Address of array[i] = Base Address + (i × Size of element)

This formula enables the compiler to locate any element in the array immediately, providing direct access to elements without needing to search sequentially. This approach is particularly beneficial when working with large arrays or performing frequent read/write operations, as it significantly reduces access time.

❖ Two dimensional Arrays

For two-dimensional arrays, such as a 2x3 integer array

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

stored in a row-major order with a base address of 3000:

1. **Row-Major Order Calculation**: Elements are stored row by row. To access an element at a[i][j], the address calculation expands as follows:

   Address of a [i] [j] = Base Address + ((i × Number of Columns + j) × Size of element)

2. **Memory Layout**: The layout in memory for a[2][3] with each integer occupying 4 bytes would be:

Table 1.3 Memory Allocation of a two dimensional array

Element	Row	Column	Memory Address
a[0][0]	0	0	3000
a[0][1]	0	1	3004
a[0][2]	0	2	3008
a[1][0]	1	0	3012
a[1][1]	1	1	3016
a[1][2]	1	2	3020

In programming, understanding how arrays are stored in memory is essential for efficient data manipulation. In C, arrays are stored in **contiguous memory locations**. This means that each element of the array is stored immediately after the previous element in memory. This layout allows for **fast, direct access** to any element of the array using its index and the array's base address (starting memory location).

1.8 ADVANTAGES OF CONTIGUOUS MEMORY LAYOUT

The contiguous memory layout provides several key advantages:

1. **Direct Access**: Knowing the base address and the size of each element allows direct access to any element using its index. This is highly efficient for operations that require accessing elements randomly or quickly.
2. **Predictable Memory Usage**: Since the entire array is stored contiguously, the total memory required for the array is predictable and easy to calculate (Size of Array = Size of Element * Number of Elements).
3. **Efficient Traversal**: Iterating over an array is efficient, as elements are stored consecutively in memory. The program doesn’t need to jump around in memory, which helps optimize the cache usage and speeds up access.

1.9 LIMITATIONS OF ARRAYS IN MEMORY

While arrays are highly efficient for direct access, they have some limitations due to their fixed size and contiguous storage:

1. **Fixed Size**:
    - Once an array is declared, its size cannot be changed. This limitation means that if you run out of space in an array, you can't expand it to add more elements.
    - In scenarios where the amount of data varies, such as dynamic lists or real-time data streams, arrays may be less suitable. Instead, **dynamic data structures** like linked lists or dynamically allocated arrays (using pointers and malloc in C) are preferred.

2. **Inefficient Insertion and Deletion**:
    - Inserting or deleting elements in an array can be inefficient, especially if it requires shifting elements. For example, inserting an element at the beginning of an array requires shifting all existing elements to the right, which can take time proportional to the array size.
    - This limitation makes arrays more suitable for scenarios where data is mostly static and does not need to be frequently modified.

3. **Memory Contiguity Requirement**:
    - Arrays require a contiguous block of memory large enough to hold all elements. On systems with limited memory or fragmented memory, it may be challenging to allocate large arrays, leading to memory allocation failures.

1.10 ALTERNATIVE SOLUTIONS TO ARRAY LIMITATIONS

Arrays, while fundamental and widely used in programming, have inherent limitations that can hinder their application in certain scenarios. These limitations include fixed size, inefficient insertion and deletion, and the inability to dynamically adapt to changing data requirements. To overcome these limitations, several alternative data structures and techniques can be employed:

1. Dynamic Arrays

Dynamic arrays, such as vectors in C++ or ArrayLists in Java, address the fixed-size limitation of standard arrays by automatically resizing when elements are added or removed. They provide:

- **Dynamic resizing:** The array grows or shrinks as needed.
- **Ease of use:** Simplified handling of variable-sized data.
- **Performance trade-offs:** Amortized constant time for appending elements but can involve overhead during resizing.

2. Linked Lists

A linked list is a collection of nodes where each node contains data and a reference to the next node. They offer:

- **Dynamic memory allocation:** No predefined size; memory is allocated as nodes are added.
- **Efficient insertion and deletion:** Operations do not require shifting elements, unlike arrays.
- **Drawbacks:** Random access is inefficient, as traversal is needed to reach a specific element.

3. Hash Tables

Hash tables store data in key-value pairs, enabling efficient access and management of elements. They provide:

- Constant time complexity for access, insertion, and deletion (on average).
- **Scalability:** They adapt dynamically as the data grows.
- **Limited order preservation:** Data is typically unordered.

1.11 SUMMARY OF INTERNAL MEMORY REPRESENTATION OF ARRAYS

Internal Memory Representation of Arrays can be summarized as

- **Contiguous Memory**: Arrays are stored in contiguous memory, which allows fast access to elements by calculating the address using the base address and index.
- **Fixed Size**: Arrays have a fixed size, defined at the time of declaration, which cannot be changed during program execution.
- **Efficient Access**: The contiguous layout enables efficient direct access to any element by index, ideal for scenarios requiring frequent data retrieval.
- **Memory Address Calculation**: The address of an element a[i] in an array a is calculated as Base Address + (i * Size of Element).

Arrays provide a simple and effective way to manage collections of data with known, fixed sizes, but may not be ideal for scenarios where data size needs to vary dynamically.

1.12 APPLICATIONS OF ARRAYS

The below applications illustrate how arrays can be used for simple data storage and quick access, making them handy for managing small collections of related data in a program

1.Storing a List of Numbers

- Arrays can be used to store a collection of numbers, such as student grades, ages, or exam scores.
- Example: int grades[5] = {90, 85, 78, 92, 88};

2. Storing a Collection of Characters (String)

- Arrays are commonly used to store sequences of characters, making up a string.
- Example: char name[6] = "Alice";

3. Storing Daily Temperatures

- Arrays can store daily temperatures or other time-based data, allowing for quick access to a specific day's temperature.
- Example: float temperatures[7] = {72.5, 74.2, 73.8, 70.0, 68.9, 71.5, 75.0};

4. Representing a Game Board (e.g., Tic-Tac-Toe)
   - A 2D array can represent a simple game board, such as a 3x3 grid for tic-tac-toe.
   - Example: char board[3][3] = {{'X', 'O', 'X'}, {'O', 'X', 'O'}, {' ', ' ', 'X'}};
5. Storing Lookup Tables
   - Arrays can store lookup values or simple mappings.
   - Example: int square[5] = {0, 1, 4, 9, 16}; // Stores squares of 0 to 4 for quick lookup

1.13 KEY TERMS

Array, Contiguous Memory, Abstract Data Type (ADT), Indexing, Fixed Size, Memory Address Calculation

1.14 SELF ASSESSMENT QUESTIONS

1. What is an array, and how is it structured in memory?
2. Explain the concept of contiguous memory in arrays.
3. How +does indexing work in an array?
4. Why is an array considered an Abstract Data Type (ADT)?
5. Describe the advantages and limitations of using arrays.
6. What formula is used to calculate the memory address of an element in an array?
7. Compare single-dimensional, two-dimensional, and multi-dimensional arrays.
8. How are dynamic arrays different from static arrays?

1.15 SUGGESTED READINGS

1. "Data Structures and Algorithm Analysis in C" by Mark Allen Weiss.
2. "The C Programming Language" by Brian W. Kernighan and Dennis M. Ritchie.
   "Algorithms in C" by Robert Sedgewick.

Dr.G.Neelima

Lesson - 2
Structures, Unions and Polynomial ADT

OBJECTIVES

The objectives of the lesson are outlined as follows

1. Understand the Concept of Structures in C Programming
2. Explore the Memory Layout and Characteristics of Structures and Unions
3. Gain Proficiency in Nested and Self-Referential Structures
4. Represent and Operate on Polynomials Using Arrays
5. Understand Sparse Matrices and Their Representations.

STRUCTURE

2.1 INTRODUCTION

A structure is a user-defined data type in C that groups variables of different types under a single name. This is useful for creating complex data types that represent real-world entities, such as Student, Book, or Employee.

2.1.1 The Role of Structures in C

While arrays are ideal for storing multiple items of the same type**,** structures allow us to group different types of data under a single entity. In many real-life scenarios, we deal with collections of related but different types of data. For instance, a student has a name (string), age (integer), and GPA (float), which are all related pieces of information but of different data types.

1. **Heterogeneous Data Storage**: Structures let us store different data types together. For example, in `struct Student`, we can store a name (character array), age (integer), and GPA (float), all within one entity.
2. **Logical Grouping**: Structures enable logical grouping of related data, making it easier to manage complex data, like employee records with fields for ID, name, and salary.
3. **Readability and Organization**: By using descriptive fields in structures, such as title, author, and year in `struct Book`, we enhance code readability and clarity, helping others quickly understand the data's purpose.

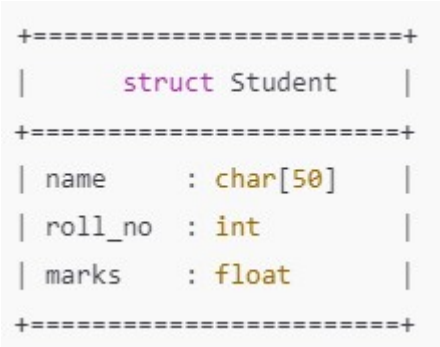
2.1.2 Characteristics of Structures

- **Heterogeneous Elements**: Structures can contain elements (fields) of different data types.
- **Logical Grouping**: Structures group related data, making code more organized and readable.
- **Access with Dot Operator**: Each field within a structure can be accessed using the dot (.) operator.

Example Structure Declaration

```
struct Student {
    char name[50];
    int roll_no;
```

```
float marks; };
```



Here, struct Student defines a structure with three fields: name, roll_no and marks.

2.1.3. Memory Layout of Structures

In C, structures allow us to group different types of data together under one name. For example, a struct Student might contain a student's name, age, and GPA, each represented by different data types (such as char, int, and float). Understanding how structures are laid out in memory helps us write efficient code, especially when dealing with large data sets.
When a structure is declared, the compiler arranges its fields in memory in the order they are listed. For example, if we define a structure like this:

```
struct Example {
    char c;
    int i;
    float f;  };
```

The compiler stores c (character), i (integer), and f (float) in consecutive memory locations. However, due to memory alignment rules, the exact memory layout may include padding (extra unused space) between fields to align each field to memory boundaries. This alignment improves performance on many processors by reducing the number of memory accesses needed to read or write data.

2.1.4 Memory Alignment and Padding

Memory alignment refers to the way data is arranged and accessed in memory to match the requirements of the system's processor. Different data types require different alignment boundaries:

- **Character (char)**: Often requires 1-byte alignment.
- **Integer (int)**: Often requires 4-byte alignment.
- **Float (float)**: Also commonly requires 4-byte alignment.

Padding is the extra memory added by the compiler between structure fields to meet alignment requirements. This padding may make the structure use more memory than the sum of its fields alone, but it ensures faster access by aligning fields to boundaries suitable for the system.

Example: Structure with Padding Let's illustrate this with the struct Example mentioned earlier. Assume the following alignment requirements:

- char requires 1 byte.
- int and float each require 4 bytes.

If struct Example is declared as:

```
struct Example {
   char c;   // 1 byte
   int i;    // 4 bytes
   float f;  // 4 bytes  };
```

In memory, c would start at the base address. However, to align i on a 4-byte boundary, the compiler would insert 3 bytes of padding after c. Similarly, f is naturally aligned and doesn't require padding after it.

The memory layout would look like this:

Table 1: Memory layout of Structure

Field	Bytes	Memory Layout	Address Offset
c	1	c	Base Address
Padding	3	Unused	+1 to +3
i	4	i	+4 to +7
f	4	f	+8 to +11

Total memory usage for struct Example = 12 bytes.

Without padding, the structure would only require 9 bytes (1 for c, 4 for i, and 4 for f). However, with padding, it uses 12 bytes. This extra memory helps ensure that the processor can access the int and float fields efficiently.

2.1.5 Significance of Padding

Memory alignment and padding are crucial for the following reasons:

1. **Efficient Access**: Certain processors perform best when data is aligned to specific boundaries. For example, a 4-byte int aligned to a 4-byte boundary allows the processor to access it in a single memory operation.
2. **Compatibility**: Alignment ensures that code performs consistently across different systems. Structures with proper alignment are more portable, working across processors with different alignment requirements.

Structure Padding Example with Visual Representation

Consider the following structure:

```
struct Example2 {
   char a;   // 1 byte
   char b;   // 1 byte
   int x;    // 4 bytes
   short y;  // 2 bytes };
```

Assuming:

- char requires 1 byte alignment.
- int requires 4 bytes alignment.
- short requires 2 bytes alignment.

The compiler would arrange and add padding as follows:

Table 2: Memory Layout of a Structure with Padding.

Field	Bytes	Memory Layout	Address Offset
a	1	a	Base Address
b	1	b	+1
Padding	2	Unused	+2 to +3
x	4	x	+4 to +7
y	2	y	+8 to +9
Padding	2	Unused	+10 to +11

Total memory usage for struct Example2 = 12 bytes.
Here, the compiler adds padding:

- 2 bytes after b to align x to a 4-byte boundary.
- 2 bytes after y to make the structure's size a multiple of the largest field alignment requirement (4 bytes).

2.2 NESTED STRUCTURES

In C programming, a nested structure (or structure within a structure) allows us to build more complex data types by combining different, related structures. This approach is particularly useful when representing hierarchical or composite data, where a single entity is made up of multiple smaller components. By nesting structures, we can organize data logically and make our code cleaner and easier to understand.

2.2.1 What is a Nested Structure?

A **nested structure** is a structure that contains another structure as one of its fields. This concept enables us to create more sophisticated data models by grouping multiple structures into a single unit. Instead of having multiple fields in one large structure, nesting allows us to logically group related fields, making the structure easier to understand and manage.

For example, consider a Student entity. A student has properties like name, age, and address. While name and age can be stored as a char array and an int, respectively, an address itself contains multiple details like city, state, and ZIP code. To organize these related details,

we can create an Address structure to hold the address data and then include it within the Student structure.

2.2.2 Usage of Nested Structures

Nested structures are useful for:

1. **Hierarchical Data**: They allow us to represent data that naturally has a hierarchical or grouped structure. For instance, a student's address has multiple components, each of which is part of a larger structure.
2. **Improving Code Organization**: By grouping related fields into separate structures, nested structures help improve code readability and organization.
3. **Encapsulation of Data**: Nested structures allow us to separate concerns by grouping data into logical units. This way, we can handle each part of the data separately or together as needed.

Consider a situation where we need to store information about employees in a company. Each employee might have a name, ID, and contact details. The contact details could include phone number, email, and address, each of which could itself be structured further. By nesting structures, we can manage these different parts without making a single large, complicated structure.

Example of Nested Structure Declaration

Here is a practical example using nested structures to represent a student and their address details.

1. Define the Address Structure:

```
struct Address
{
char city[50];
char state[20];
int zip;
};
```

In this example, struct Address contains three fields:

- city: a character array of 50 elements to hold the name of the city.
- state: a character array of 20 elements to hold the name of the state.
- zip: an integer to hold the ZIP code.

2. Define the Student Structure with Nested Address:

```
struct Student
 {
   char name[50];
    int age;
   struct Address address;
 };
```

In this example, struct Student contains:

- name: a character array to hold the student's name.
- age: an integer to store the student's age.
- address: a nested structure of type struct Address.

By including struct Address within struct Student, we effectively create a hierarchical data model where the address information is logically grouped under the student.

2.2.3 Memory Layout of Nested Structures

When we declare a nested structure, the memory layout follows the same principles as for regular structures, including alignment and padding rules. The nested structure's fields are stored as part of the outer structure, occupying contiguous memory within it.

For example, consider the memory layout for struct Student with the struct Address nested within it:

```
struct Address
{
   char city[50];
   char state[20];
   int zip;
};

struct Student
{
  char name[50];
   int age;
  struct Address address;
};
```

Assuming:

- char arrays (city and state) require 50 and 20 bytes, respectively.
- int requires 4 bytes.

The memory layout could look like this:

Table 3: Memory layout of nested Structures

Field	Size (bytes)	Address Offset
`name`	50	0 - 49
`age`	4	50 - 53
`address.city`	50	54 - 103
`address.state`	20	104 - 123
`address.zip`	4	124 - 127

2.3. UNIONS IN C

In C programming, a **union** is a user-defined data type that is similar to a structure but with one significant difference: all fields in a union share the same memory location. This means that only one field in a union can hold a value at any given time, as they all occupy the same memory space. The primary purpose of unions is to save memory when we need to store different types of data in the same memory location but only one at a time.

2.3.1 What is a Union?

A union is a data type that allows storing different types of data (such as int, float, char, etc.) in the same memory location. By sharing the memory between fields, unions provide a way to handle variables of different data types in an efficient manner.

Unions are particularly useful in applications where memory is limited, or we need to handle data of different types at different times. For example, if a program requires storing either an integer, a float, or a character but never all three simultaneously, a union allows these to share the same memory, thereby saving space.

2.3.2 Characteristics of Unions

1. **Single Shared Memory**:
   - All fields in a union occupy the same memory location. This shared memory space is the size of the largest member of the union.
   - If the union has fields of different sizes, the compiler allocates memory equal to the size of the largest field to ensure that any member can fit in the union.
2. **One Field at a Time**:
   - Since all fields share the same memory, only one field can hold a value at any given time. If a new value is assigned to a different field, it overwrites the previous value.
   - This means that accessing a field in a union will yield valid data only if it's the most recently assigned field.
3. **Memory Efficiency**:
   - By allowing multiple data types to share the same memory, unions save memory. This is particularly valuable in memory-constrained applications, such as embedded systems or low-level programming.

2.3.3 Memory Layout of a Union

The memory layout of a union is unique compared to a structure. In a structure, each field has its own memory space, so the total size of the structure is the sum of all fields plus any padding. In a union, however, all fields occupy the same memory location, so the total size of the union is only as large as its largest field.
For example, consider the following union:

```
union Data
 {
   int intValue;
   float floatValue;
   char charValue;
 };
```

In this union:

- intValue is an integer (typically 4 bytes).
- floatValue is a float (also typically 4 bytes).
- charValue is a character (1 byte).

Since the largest field (either intValue or floatValue) requires 4 bytes, the compiler will allocate 4 bytes for the entire union. All fields share this 4-byte memory location.

2.3.4 Visual Representation of a Union's Memory Layout

Here's how memory is allocated in union Data:

Table 4: Memory layout of Union

Field	Size	Memory Layout
`intValue`	4 bytes	Occupies entire union memory
`floatValue`	4 bytes	Shares same 4 bytes as `intValue`
`charValue`	1 byte	Shares the first byte of the 4 bytes

If intValue is assigned a value, it will use the full 4 bytes. If we then assign a value to charValue, it will overwrite only the first byte of those 4 bytes.

Let's look at an example where we declare a union and then assign values to its fields to see how data is stored.

```
union Data
   {
    int intValue;
    float floatValue;
    char charValue;
     };

int main( )
{
    union Data data;

    data.intValue = 10;
    printf("intValue: %d\n", data.intValue);

    data.floatValue = 3.14;
    printf("floatValue: %f\n", data.floatValue);
```

```
    data.charValue = 'A';
    printf("charValue: %c\n", data.charValue);

    return 0;
}
```

In this program:

1. data.intValue = 10: When we assign 10 to intValue, it occupies the shared 4 bytes.
2. data.floatValue = 3.14: Assigning 3.14 to floatValue overwrites the shared memory with the floating-point representation of 3.14.
3. data.charValue = 'A': Finally, assigning 'A' to charValue writes only to the first byte of the shared memory.

Each time we assign a new value, it overwrites the previous value because all fields share the same memory space.

2.3.6 Benefits of Using Unions

1. **Memory Efficiency**: Unions allow for efficient memory usage by reusing the same memory location for different data types, which is beneficial when memory is limited.
2. **Data Flexibility**: They provide a flexible way to handle data of multiple types, especially when handling data formats where each instance contains only one type at a time.

2.3.7 Limitations of Unions

While unions are powerful, they come with certain limitations:

1. **Single Active Field**: Only one field in a union can hold a valid value at any given time. Accessing another field after assigning a value to a different one can lead to undefined or incorrect data.
2. **Complexity in Usage**: Unions can make the code complex and harder to maintain, especially when managing which field currently holds a valid value.
3. **No Type Safety**: Since unions allow multiple types in the same memory, there is no guarantee of type safety. Developers must carefully manage which field is active to avoid unintended behavior.

2.4. SELF-REFERENTIAL STRUCTURES

A **self-referential structure** in C is a structure that contains a pointer to another instance of the same structure. This allows each structure instance to be linked with other instances, creating chains or networks of data. Self-referential structures are fundamental to creating **dynamic data structures**, such as linked lists, trees, and graphs, which allow for flexible data management where the data size can change dynamically.

2.4.1 Understanding Self-Referential Structures

In C, structures are typically defined to group related data. However, when we need a structure to link to others of its kind, we add a **pointer field** that points to another structure of the same type. This approach is commonly used in data structures where elements need to be dynamically linked, enabling the structure to grow or shrink as needed.

For example, in a **linked list**, each element (node) points to the next element, creating a chain. Similarly, in **binary trees** and **graphs**, nodes point to other nodes, creating branching structures or complex networks.

Self-referential structures make it possible to:

1. **Link Data Dynamically**: By pointing to other instances of the structure, we can create a flexible, expandable data structure.
2. **Organize Data Recursively**: Using pointers, we can organize data in a recursive manner, where each instance is related to others of the same type.

2.4.2 Characteristics of Self-Referential Structures

1. **Recursive Design**:
    - A self-referential structure's pointer field enables it to link to other instances of itself, allowing for recursive relationships.
    - By linking each instance to another, we can create a series or hierarchy of instances, such as in linked lists (where each node points to the next node) or trees (where each node can point to child nodes).
    - This recursive nature allows us to build data structures that grow or shrink dynamically, as each new element can be linked to others.
2. **Pointer Field**:
    - The self-referential structure includes a pointer field of the same type as the structure itself. This pointer field stores the memory address of another instance of the same structure, enabling it to reference other instances.
    - For example, a Node structure in a linked list might contain an int data field and a Node* next pointer, where next points to another Node.

Example: Self-Referential Structure for a Linked List Node

A linked list is a chain of elements called **nodes**, where each node contains data and a pointer to the next node. Here's an example of a self-referential structure for a node in a linked list.

```
struct Node {
   int data;          // Data field to store an integer
   struct Node* next;  // Pointer field to point to the next Node
};
```

In this example:

- data is an integer field that stores the node's data.
- next is a pointer to another Node instance. This pointer field allows each Node to link to the next node in the list, creating a chain.

2.4.3 Applications of Self-Referential Structures

Self-referential structures are essential for creating **dynamic, linked data structures** that can grow or shrink as needed. They are widely used in applications that require flexible and dynamic data storage.

Common applications include:

1. **Linked Lists**:
   - A linked list is a sequence of nodes, where each node points to the next node in the sequence.
   - Linked lists allow for efficient insertion and deletion of nodes without resizing or shifting data, making them ideal for applications where the data size changes frequently.
2. **Binary Trees**:
   - In a binary tree, each node points to two child nodes, creating a branching structure. Each node has two pointers, typically named left and right, pointing to other nodes in the tree.
   - Binary trees are used in applications like sorting, searching, and hierarchical data organization.
   -
3. **Graphs**:
   - Graphs consist of nodes connected by edges, and each node can have multiple pointers to other nodes.
   - Graphs are used in various applications, including social networks, route optimization, and network analysis.
4. **Stacks and Queues**:
   - Linked lists can also be used to implement stacks (LIFO) and queues (FIFO) dynamically, where nodes are added and removed as needed.

2.5 THE POLYNOMIAL ABSTRACT DATA TYPE (ADT)

A **polynomial** is a mathematical expression composed of terms that are combined by addition. Each term in a polynomial has:

1. A **coefficient**: A constant multiplier for the term.
2. A **variable**: Usually denoted as x.
3. An **exponent** (or power): The power to which the variable is raised.

For example, the polynomial:

$$4x^3 + 3x^2 - 5x + 7$$

has four terms: $4x^3$, $3x^2$, -5x and 7

- **Coefficient**: Each term has a coefficient (e.g., 4, 3, -5, and 7).
- **Exponent**: The power to which xxx is raised in each term (e.g., 3, 2, 1, and 0).

In programming, we often represent polynomials using arrays, where each index in the array corresponds to an exponent, and the value at that index is the coefficient of the term with that exponent.

2.5.1 Representation of Polynomials in Programming

We can represent a polynomial as an array where:

- **Index**: Represents the exponent of the variable x.
- **Value at Index**: Represents the coefficient of x raised to that exponent.

For example, consider the polynomial:

$$5x^4 + 2x^2 + 3$$

This polynomial has:

- A term with x4 and a coefficient of 5.
- A term with x2 and a coefficient of 2.
- A constant term (coefficient 3 for x0).

In an array, this polynomial could be represented as:

```
int poly[5] = {3, 0, 2, 0, 5};
```

Here:

- poly[0] = 3 represents the term 3x0.
- poly[2] = 2 represents the term 2x2.
- poly[4] = 5 represents the term 5x4.

2.5.2 Visual Representation of Polynomial in Array Format

For a clearer visual representation:

Exponent (index)	4	3	2	1	0
Coefficient	5	0	2	0	3

This array representation makes it easy to perform operations like addition, subtraction, and multiplication on polynomials by manipulating their coefficients.

2.5.3 Polynomial Operations Using Arrays

Representing polynomials as arrays allows for efficient computation of polynomial operations, such as addition and multiplication. These operations can be performed using element-wise operations on the arrays.

1. Addition of Polynomials

To add two polynomials, we can add their corresponding coefficients for each exponent. For example:

Let's say we have two polynomials:

1. P(x)=3x3+2x+4 represented as poly1[4] = {4, 2, 0, 3}
2. Q(x)=x3+x2+6 represented as poly2[4] = {6, 0, 1, 1}

The resulting polynomial R(x)=P(x)+Q(x) will have coefficients obtained by adding the corresponding elements of `poly1` and `poly2`:

```
int poly1[4] = {4, 2, 0, 3};
int poly2[4] = {6, 0, 1, 1};
int result[4];
for (int i = 0; i < 4; i++) {
   result[i] = poly1[i] + poly2[i];
}
```

This gives result[4] = {10, 2, 1, 4}, representing the polynomial:

$$4x^3 + x^2 + 2x + 10$$

2. Multiplication of Polynomials

Multiplying polynomials involves multiplying each term in the first polynomial by every term in the second polynomial and summing terms with the same exponent.

For example, let's multiply:

1. P(x)=x+2 , represented as poly1[2] = {2, 1}
2. Q(x)=x+1, represented as poly2[2] = {1, 1}

The resulting polynomial R(x) = P(x) × Q(x) = x2 + 3x + 2

This requires us to multiply each coefficient in `poly1` by each coefficient in `poly2`:

```
int poly1[2] = {2, 1};  // Represents P(x) = x + 2
int poly2[2] = {1, 1};  // Represents Q(x) = x + 1
int result[3] = {0};    // Result array for R(x)
for (int i = 0; i < 2; i++) {
   for (int j = 0; j < 2; j++) {
      result[i + j] += poly1[i] * poly2[j];
   }
}
```

The resulting array `result[3] = {2, 3, 1}` represents

$$R(x) = x^2 + 3x + 2$$

2.5.4 Advantages of Array Representation for Polynomials

1. **Efficient Operations**: Arrays allow for efficient element-wise operations, such as addition and multiplication.
2. **Space Management**: For polynomials with sparse terms (e.g., high exponents with zero coefficients), this array-based approach can be enhanced by using only non-zero terms, saving space.
3. **Ease of Implementation**: Representing polynomials as arrays simplifies code for polynomial operations, as each term's coefficient can be directly accessed by its index.

2.6 SPARSE MATRICES AND THEIR REPRESENTATION

In programming and scientific computing, a **sparse matrix** is a matrix in which most of the elements are zero. Sparse matrices are common in applications where data has large dimensions but only a few meaningful, non-zero values. Examples include matrices in graph theory (adjacency matrices), optimization problems, and scientific computations where data is typically scattered rather than dense.

2.6.1 What is a Sparse Matrix?

A matrix is considered **sparse** if the number of zero elements significantly outweighs the number of non-zero elements. For example, in a 5x5 matrix with only 4 non-zero values, it would be inefficient to store all 25 elements if only 4 hold meaningful data. Instead, we focus on storing only the **non-zero elements** and their positions, making sparse matrices an efficient solution in terms of memory and computation.

Consider this 5x5 matrix:

$$\begin{bmatrix} 0 & 0 & 0 & 5 & 0 \\ 0 & 0 & 8 & 0 & 0 \\ 0 & 0 & 0 & 0 & 3 \\ 0 & 6 & 0 & 0 & 0 \\ 9 & 0 & 0 & 0 & 0 \end{bmatrix}$$

In this example, only 5 elements are non-zero: 5, 8, 3, 6, and 9. Instead of storing all 25 elements, we store only these 5 elements and their positions.

2.6.2 Representation of Sparse Matrices

There are several ways to represent a sparse matrix efficiently. Two common methods include Coordinate List (COO) representation and Compressed Sparse Row (CSR) representation.

1. Coordinate List (COO) Representation

The Coordinate List (COO) representation stores each non-zero element along with its row and column indices. For the matrix above, we store the non-zero values and their coordinates in three arrays:

- **Row Array**: Stores the row index for each non-zero element.
- **Column Array**: Stores the column index for each non-zero element.
- **Value Array**: Stores the non-zero values themselves.

For the example matrix, the COO representation would look like this:

Table 5 : COO (Coordinate) Representation of a Sparse Matrix

Row Index	Column Index	Value
0	3	5
1	2	8
2	4	3
3	1	6
4	0	9

In this format:

- **Row Array**: {0, 1, 2, 3, 4}
- **Column Array**: {3, 2, 4, 1, 0}
- **Value Array**: {5, 8, 3, 6, 9}

The COO format is simple and widely used for sparse matrices with random access requirements. It allows for easy traversal of non-zero elements and is suitable for applications where non-zero elements are scattered across the matrix.

2. Compressed Sparse Row (CSR) Representation

Compressed Sparse Row (CSR) representation is another way to store sparse matrices efficiently, especially useful when the matrix is large and has most of its non-zero values clustered within certain rows.

The CSR format uses three arrays:

- **Values Array**: Stores all the non-zero elements in row-major order.
- **Column Index Array**: Stores the column indices of each non-zero element corresponding to the values in the Values Array.
- **Row Pointer Array**: Stores the starting index of each row in the Values Array.

For the same matrix:

$$\begin{bmatrix} 0 & 0 & 0 & 5 & 0 \\ 0 & 0 & 8 & 0 & 0 \\ 0 & 0 & 0 & 0 & 3 \\ 0 & 6 & 0 & 0 & 0 \\ 9 & 0 & 0 & 0 & 0 \end{bmatrix}$$

the CSR representation would look like this:

- **Values Array**: {5, 8, 3, 6, 9}
- **Column Index Array**: {3, 2, 4, 1, 0}
- **Row Pointer Array**: {0, 1, 2, 3, 4, 5}

Explanation:

- The **Values Array** stores the non-zero values in row order: 5, 8, 3, 6, 9.
- The **Column Index Array** specifies the column position of each non-zero value: 3, 2, 4, 1, 0.
- The **Row Pointer Array** contains the starting index in the Values Array for each row:
    - Row 0 starts at index 0.
    - Row 1 starts at index 1.
    - Row 2 starts at index 2, and so on.

CSR is efficient for matrix-vector multiplication and is widely used in scientific computing, where matrix rows tend to have clustered non-zero values.

2.6.3 Benefits of Sparse Matrix Representation

1. **Memory Efficiency**:
    - By only storing non-zero values and their positions, sparse matrix representations reduce memory consumption significantly, especially for large matrices with very few non-zero entries.
    - This efficiency makes sparse representations suitable for applications that deal with large datasets but with minimal significant data.
2. **Efficiency in Operations**:
    - Sparse matrix representations make operations like matrix multiplication, addition, and other transformations more efficient by focusing only on significant (non-zero) data.
    - Rather than processing every element, algorithms can skip zero elements, reducing computational load and improving speed.

2.6.4 Applications of Sparse Matrices

Sparse matrices are commonly used in areas where data is mostly empty or zero-valued. Some typical applications include:

1. **Scientific Computing**:
    - In fields such as physics, chemistry, and engineering, sparse matrices are used to represent large systems with a limited number of interactions, such as those found in finite element analysis and molecular modeling.

2. **Graph Representations**:
    - Adjacency matrices used to represent graphs are often sparse, especially when representing large but sparse graphs (graphs with many nodes but relatively few edges).
    - Sparse matrices provide an efficient way to store and manipulate graph data, especially in network analysis and social network applications.

3. **Image Processing**:
    - Sparse matrices can represent large, sparse images or grids, where only certain regions contain meaningful data.
    - In medical imaging or astronomical data analysis, sparse representations help store and process data with minimal memory overhead.

2.6.5 Limitations of Sparse Matrix Representations

While sparse matrices are efficient, they have certain limitations:

1. **Complexity of Access**:
    - Accessing elements in sparse matrix representations can be more complex than in dense matrices, especially if the matrix format (like CSR or COO) does not support random access directly.

2. **Increased Overhead for Small Matrices**:
    - Sparse representations are beneficial for large matrices with few non-zero elements. For smaller matrices, the overhead of managing index arrays can outweigh memory savings.

3. **Limited Operations**:
    - Not all mathematical operations are directly supported on sparse representations, so additional algorithms are often required to perform complex operations.

2.7 KEY TERMS

Coordinate List (COO) Representation, Compressed Sparse Row (CSR) Representation, Memory Alignment, Nested Structure, Padding, Polynomial ADT, Self-Referential Structure, Sparse Matrix, Structure, Union.

2.8 SELF ASSESSMENT QUESTIONS

1. What is the purpose of a structure in C programming?
2. How does a union differ from a structure in terms of memory allocation?
3. Explain the concept of a nested structure with an example.
4. What is a self-referential structure, and why is it useful?
5. Describe the Coordinate List (COO) representation of a sparse matrix.

2.9 SUGGESTED READINGS

1. "The C Programming Language" by Brian W. Kernighan and Dennis M. Ritchie
2. "Fundamentals of Data Structures in C" by Ellis Horowitz, Sartaj Sahni, and Susan Anderson-Freed
3. "Data Structures Using C" by Reema Thareja
4. "Data Structures and Algorithm Analysis in C" by Mark Allen Weiss

Dr.G.Neelima

LESSON - 3

Stacks

OBJECTIVES

The objectives of the chapter can be summarised as follows

1. Learn what a stack is and how it follows the Last-In, First-Out (LIFO) principle.
2. Discover practical uses of stacks, like managing function calls and undo operations.
3. Learn key stack operations like push, pop, peek, and check for empty or full conditions.
4. Understand how to create stacks using arrays and dynamic memory allocation.
5. Apply stacks in real-world scenarios, such as expression evaluation and algorithm backtracking.

Structure

3.1 INTRODUCTION

A stack is a linear data structure that follows the Last-In, First-Out (LIFO) principle. This means that the last element added to the stack is the first one to be removed. You can imagine a stack as a collection of elements arranged vertically, like a stack of plates; when a new plate is added, it goes on top, and when a plate is removed, it's also taken from the top. Stacks are often represented using an array or linked list. An array-based stack is more straightforward and requires a maximum size, while a linked-list stack allows for dynamic resizing. In stacks, all operations (such as inserting, removing, or accessing elements) occur only at one end, referred to as the top of the stack.

Stacks play a critical role in computer science for several reasons:

1. **Function Call Management**: In programming, particularly in recursion, stacks manage function calls. Every time a function is called, its return address, parameters, and local variables are pushed onto the stack. When the function completes, the data is popped off the stack to resume the previous state.
2. **Expression Evaluation**: Stacks are instrumental in evaluating mathematical expressions in postfix or prefix notation, especially when converting from infix to postfix.
3. **Backtracking**: Algorithms like Depth-First Search (DFS) in graphs use stacks to track nodes. Undo mechanisms in software (e.g., the "Undo" button) also rely on stacks to store recent operations, allowing reversal.
4. **Memory Management**: Stacks manage memory for temporary storage, which is automatically cleaned up in LIFO order, making it suitable for managing local variables and function calls.

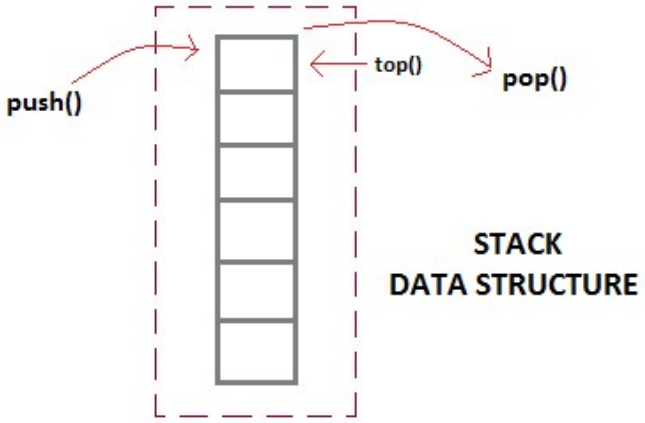


Fig 3.1 Represenation of Stack

3.2 FEATURES OF STACK

Stacks are specialized data structures that follow the Last-In, First-Out (LIFO) principle, meaning the last element added is the first to be removed. This structure is commonly used in situations where temporary storage is needed and where order matters, such as in function calls, expression evaluation, and algorithmic backtracking. The main features of the stack are

- A stack is an ordered collection of elements of the same data type, arranged in a specific sequence.
- It follows the Last-In, First-Out (LIFO) or First-In, Last-Out (FILO) principle, meaning the last element added is the first to be removed.
- The Push operation adds new elements to the stack, while the Pop operation removes the top element from the stack.
- The Top is a pointer or variable that references the topmost element in the stack. Both insertion and removal of elements occur only at this end.
- A stack is in an Overflow state when it reaches its maximum capacity (FULL), and in an Underflow state when it has no elements left (EMPTY).Operations on Stacks

3.3 OPERATIONS ON STACKS

Stacks are widely used in various applications, such as managing function calls, evaluating expressions, and backtracking algorithms. To effectively work with stacks, several basic operations are defined. The primary operations performed on stacks are:

1. **Push**: Adds an element to the top of the stack. If the stack is full, it results in a stack overflow condition.
2. **Pop**: Removes the element from the top of the stack. If the stack is empty, it results in a stack underflow condition.
3. **Peek (or Top)**: Retrieves the element at the top of the stack without removing it. This operation allows inspection of the top value without modifying the stack.
4. **isEmpty**: Checks whether the stack is empty. This is useful for preventing underflow errors before a pop operation.
5. **isFull**: Checks whether the stack is full, mainly in array-based stacks, to prevent overflow errors during a push operation.
6. **Traversal**: Displays all elements in the stack from top to bottom without altering the stack. This operation helps in examining the stack's contents.
7. **Search**: Searches for a specific element within the stack and returns its position relative to the top, or an indication if it's not present.

3.4 IMPLEMENTATION OF STACK OPERATIONS USING ARRAYS

Stack is a data structure which can be represented as an array. An array is meant to store an ordered list of elements. Using an array for representation of stack is the easiest technique to manage the data. Stack can be implemented without memory limit. But when the stacks are implemented using an array, size of the stack will be fixed. Stack can be implemented with the help of an array. In the below figure, the elements of the stack are

linearly organised in the stack starting from index of 0 to n-1 or 1 to n. The array subscripts of a stack may be from 0 to n-1 or from 1 to n.

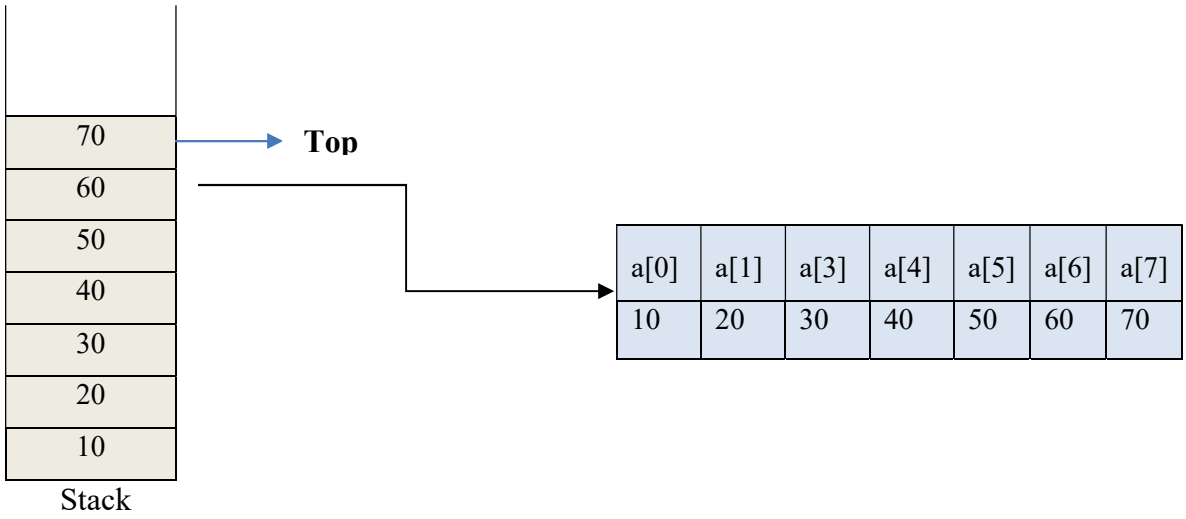


Fig 3.2 Array Represenation of Stack

3.4.1 Push Operation

The push operation adds an element to the top of the stack. In an array-based stack, we first check if there is space available (to prevent overflow). If space is available, we increment the `top` pointer and add the new element at this position. If the stack is full, we display a "Stack Overflow" message.

```
void push(int stack[], int *top, int value, int max_size) {
  if (*top == max_size - 1) {      // Check if the stack is full
   printf("Stack Overflow\n");
   } else {
   *top = *top + 1;              // Move top to the next position
   stack[*top] = value;          // Place value at the new top position
   printf("Pushed %d onto stack\n", value);    } }
```

The push function adds an element to the stack.

- First, it checks if the stack is full by comparing top with max_size - 1.
- If the stack is full, it prints "Stack Overflow" to indicate that no more elements can be added.
- If there's space, it increments `top` to the next position and places the new value at stack[top].
- Finally, it prints a message to confirm that the value was added to the stack.

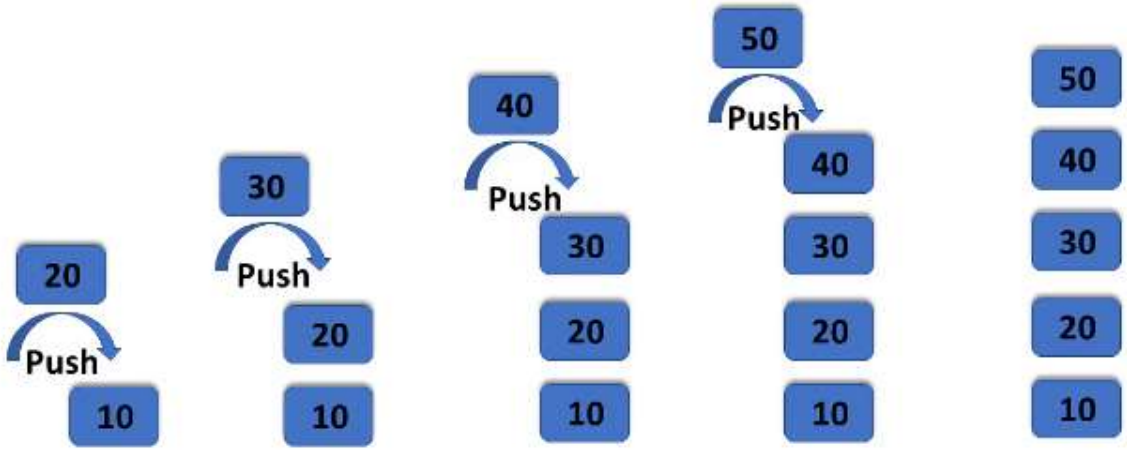


Fig 3.3 Push Operation

The above diagram illustrates a series of push operations on a stack, showing how elements are added one by one.

1. **Starting with an Empty Stack**: The stack begins empty, and the first element, `10`, is pushed onto it, becoming the bottom element.
2. **Adding `20`**: The next element, `20`, is pushed onto the stack, sitting on top of `10`.
3. **Adding `30`**: The element `30` is pushed on top, making it the new top of the stack, with `20` and `10` below it.
4. **Adding `40`**: The element `40` is pushed onto the stack, becoming the top element, while `30`, `20`, and `10` are below it in order.
5. **Adding `50`**: Finally, `50` is pushed onto the stack, sitting at the top, with `40`, `30`, `20`, and `10` below it.

Each "push" operation adds a new element to the top of the stack, following the Last-In, First-Out (LIFO) principle, where the most recently added element is always on top.

3.4.2 Pop Operation

The pop operation removes the element at the top of the stack. Before popping, we check if the stack is empty to avoid underflow. If it's not empty, we retrieve the value at `top`, decrement the `top` pointer, and return or display the removed element. If the stack is empty, we display a "Stack Underflow" message.

```
int pop(int stack[], int *top) {
   if (*top == -1) {              // Check if the stack is empty
      printf("Stack Underflow\n");
      return -1;                  // Return an error value
   } else {
```

```
int value = stack[*top];     // Retrieve the top element
*top = *top - 1;             // Decrement the top pointer
printf("Popped %d from stack\n", value);
return value;   } }
```

The pop function removes the top element from the stack.

- It first checks if the stack is empty by seeing if `top` is `-1`.
- If the stack is empty, it prints "Stack Underflow" to indicate there's nothing to remove and returns `-1` as an error value.
- If the stack has elements, it retrieves the value at `stack[top]`, decreases `top` by 1, and returns the removed value.
- It also prints a message confirming which value was removed from the stack.

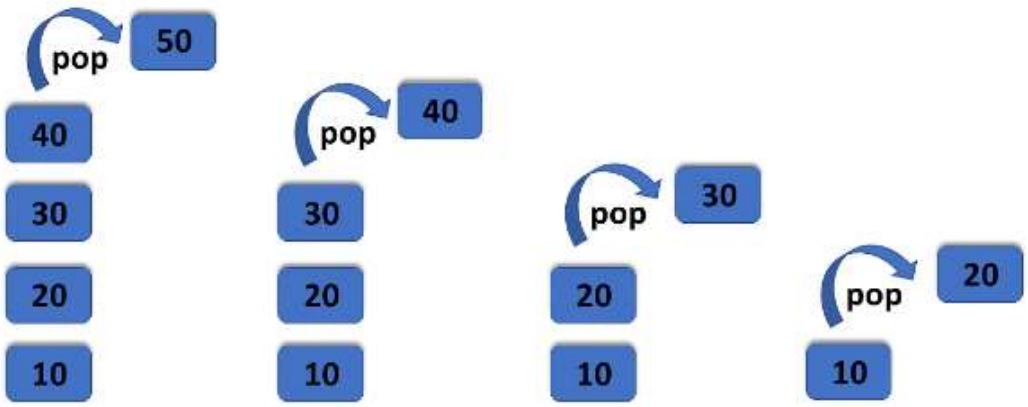


Fig 3.4 Pop Operation

Fig shows a series of **pop operations** on a stack, where elements are removed from the top one by one.

1. **Removing `50`**: The stack starts with `50` at the top. A pop operation removes `50`, leaving `40` as the new top element.
2. **Removing `40`**: The next pop operation removes `40`, making `30` the top element.
3. **Removing `30`**: Another pop operation removes `30`, leaving `20` at the top of the stack.
4. **Removing `20`**: The pop operation removes `20`, making `10` the last remaining element.
5. **Removing `10`**: Finally, `10` is removed from the stack, leaving it empty.

Each "pop" operation removes the top element, following the Last-In, First-Out (LIFO) principle, where the most recently added element is removed first.

3.4.3 Peek (or Top) Operation

The peek operation retrieves the element at the top of the stack without removing it. This operation is useful for accessing the current top element to check its value before performing other operations. If the stack is empty, we display a message indicating this.

```
int peek(int stack[], int top) {
   if (top == -1) {              // Check if the stack is empty
      printf("Stack is empty\n");
      return -1;                 // Return an error value
   } else {
      printf("Top element is %d\n", stack[top]);
      return stack[top];         // Return the top element
   } }
```

The peek function shows the top element without removing it.

- It checks if the stack is empty by seeing if `top` is `-1`.
- If the stack is empty, it prints a message saying "Stack is empty" and returns `-1` as an error value.
- If the stack has elements, it prints and returns the value at `stack[top]`, which is the current top of the stack.

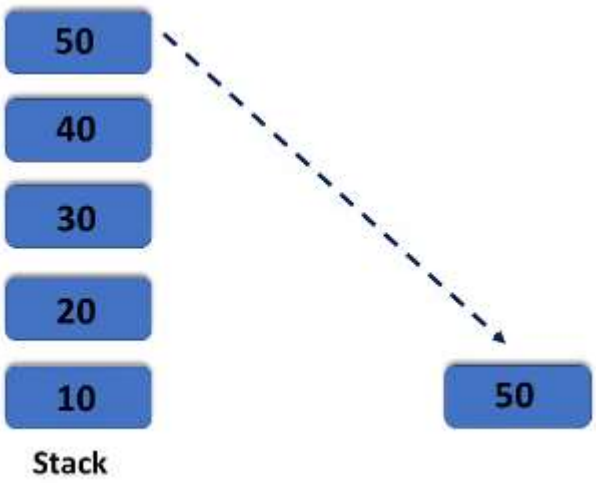


Fig 3.5 Peek Operation

The above figure illustrates the peek operation on a stack.

- The stack contains the elements `10`, `20`, `30`, `40`, and `50`, with `50` at the top.
- The peek operation retrieves the value of the top element, which is `50`, without removing it from the stack.
- The dashed line shows that `50` is being accessed but remains in the stack.

The peek operation allows you to view the top element (`50` in this case) without altering the stack's contents, following the Last-In, First-Out (LIFO) principle.

3.4.4 isEmpty Operation

The isEmpty operation checks if the stack has no elements. This is done by checking if `top` is equal to `-1`, which indicates an empty stack. This operation is often used as a safety check before performing `pop` or `peek` operations to prevent errors.

```
int isEmpty(int top)
{
  return top == -1;              // Returns 1 if stack is empty, 0 otherwise
}
```

The isEmpty function checks if the stack has any elements.

- It simply returns `1` (true) if top is -1, meaning the stack is empty.
- If `top` is not `-1`, it returns `0` (false), indicating the stack has elements.
- This function helps prevent errors by allowing us to check if the stack is empty before performing pop or peek operations.

3.4.5 isFull Operation

The isFull operation checks if the stack has reached its maximum capacity. In array-based stacks, we compare `top` with `max_size - 1`. If `top` is equal to this value, the stack is full, and no additional elements can be pushed.

```
int isFull(int top, int max_size)
{
  return top == max_size - 1;      // Returns 1 if stack is full, 0 otherwise
}
```

The isFull function checks if the stack has reached its maximum capacity.

- It returns `1` (true) if `top` equals `max_size - 1`, meaning the stack is full.
- Otherwise, it returns `0` (false), meaning there's still room for more elements.
- This function helps avoid overflow errors when trying to `push` elements onto a full stack.

3.4.6 Traversal Operation

The traversal operation displays all elements in the stack from the top to the bottom without altering the stack. This operation is helpful for checking the stack's contents. Starting from `top`, each element is printed until reaching the bottom of the stack.

```
void traverse(int stack[], int top)

{
   if (top == -1)
   {  printf("Stack is empty\n"); // Check if the stack is empty
    } else
      {
       printf("Stack elements: ");
       for (int i = top; i >= 0; i--) { // Print elements from top to bottom
          printf("%d ", stack[i]);
       }
       printf("\n");     } }
```

The traverse function displays all elements in the stack from top to bottom.

- It first checks if the stack is empty by seeing if `top` is `-1`.
- If the stack is empty, it prints "Stack is empty."
- If there are elements, it loops from `top` down to `0`, printing each element along the way.
- This allows us to see all elements in the stack without modifying it.

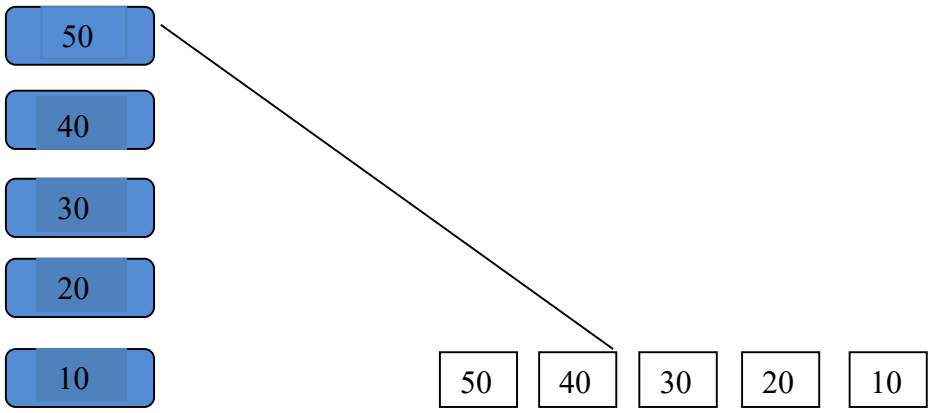


Fig 3.6 Stack Traversal

In the context of stack traversal, the above figure shows how each element in the stack can be accessed from the top to bottom.

- Starting with the top element (50), traversal involves visiting each element downwards.
- After 50, the traversal continues to 40, then 30, 20, and finally 10 at the bottom.

Traversal allows you to view all elements in the stack in order, from the top to the bottom, without altering the stack's structure. This process is useful for inspecting the entire content of the stack without performing any push or pop operations.

3.4.7 Search Operation

The search operation finds a specified element within the stack, returning its position relative to the top. We start from the top and move down, checking each element. If the element is found, its position is displayed. If not, a message is shown indicating the element isn't in the stack.

```
int search(int stack[], int top, int value)
{
   for (int i = top; i >= 0; i--) {    // Start from the top
      if (stack[i] == value) {       // Check if element matches
         printf("Found %d at position %d from top\n", value, top - i);
         return top - i;            // Return position from top
          }
  }
   printf("Element %d not found\n", value);
   return -1;                      // Return -1 if not found
 }
```

The search function looks for a specific element in the stack.

- It starts at top and moves down to `0`, checking each element to see if it matches the value being searched.
- If it finds the element, it prints the position of the element (relative to `top`) and returns this position.
- If it doesn't find the element, it prints "Element not found" and returns `-1` as an indication.
- This function is useful when we need to know if a certain value exists in the stack.

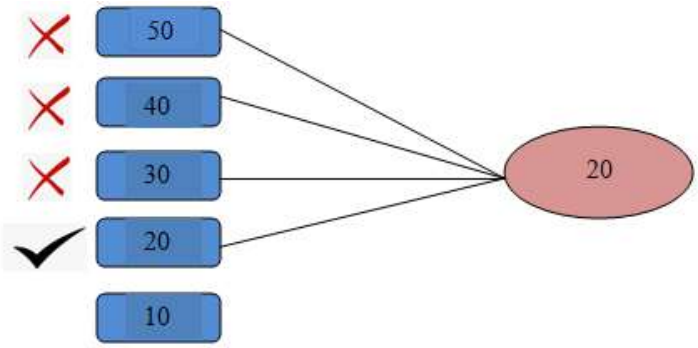


Fig.3.7 Search Operation

The above diagram illustrates the **process of searching for a specific element (`20`) within a stack**.

- The search begins at the top of the stack, examining each element sequentially.
- Each element is checked to see if it matches the target element (`20`).
- If an element does not match, the search moves down to the next element in the stack.
- The process continues until the target element (`20`) is found.
- Once the element is located, the search stops, and there is no need to check the remaining elements in the stack.

This approach demonstrates a typical stack search process, where elements are checked from the top downwards until the desired element is found or the entire stack is traversed.

3.5 Usage of Stack Functions in a Sample Program

In a stack-based program, the essential operations (push, pop, and peek) work together to manage elements in a Last-In, First-Out (LIFO) order. Here's an example that demonstrates the usage of these functions in a simple main program.

```
#define MAX_SIZE 100
int stack[MAX_SIZE];
int top = -1;
void push(int value) {
   if (top == MAX_SIZE - 1) {
      printf("Stack Overflow\n");  }
   else {
      stack[++top] = value;
      printf("Pushed %d onto stack\n", value);
       } }
int pop() {
   if (top == -1) {
      printf("Stack Underflow\n");
      return -1;  // Indicates error
   } else {
      return stack[top--];     } }
int peek() {
   if (top == -1) {
      printf("Stack is empty\n");
      return -1;  // Indicates error
     } else {
      return stack[top];    } }
int main() {
   push(10);  // Add 10 to the stack
   push(20);  // Add 20 to the stack
   printf("Top element is %d\n", peek());  // Check the top element
   pop();  // Remove the top element (20)
   pop();  // Remove the top element (10)
   pop();  // Attempt to pop from an empty stack (Underflow)
   return 0;
}.
```

Explanation:

1. **push(10)**: Adds the integer `10` to the stack.
2. **push(20)**: Adds `20` to the stack, making it the new top element.
3. **peek()**: Retrieves and prints the top element (`20`) without removing it.
4. **pop()**: Removes `20` from the stack.
5. **pop()**: Removes `10` from the stack, leaving it empty.
6. **pop()**: Attempts to remove an element from an empty stack, triggering an "Underflow" error.

This example shows how `push`, `pop`, and `peek` interact, demonstrating basic stack operations and handling stack overflow and underflow conditions.

3.6 STACK USING DYNAMIC ARRAYS IN C

Dynamic arrays offer greater flexibility compared to static arrays, especially when the stack size is unknown or may change frequently. By using dynamic memory allocation (such as malloc and realloc), we can create a stack that grows or shrinks based on usage, optimizing memory management.

3.6.1. Dynamic Allocation

- **malloc**: Used to allocate an initial memory block for the stack. For example, int* stack = (int*)malloc(initial_size * sizeof(int)); allocates memory for a stack with initial_size elements.
- **realloc**: Allows resizing the allocated memory block when the stack needs more or less space. For example,

  stack = (int*) realloc (stack, new_size * sizeof (int));

  expands or contracts the memory for the stack.

With dynamic allocation, the stack is not bound by a fixed size, making it more efficient for applications that handle variable amounts of data.

3.6.2 Push and Pop Operations

- **Push Operation**: Similar to a static stack, the push function adds an element to the stack. However, with a dynamically allocated stack, if the stack is full, realloc is used to increase its capacity.
- **Pop Operation**: Also similar to a static stack. However, with a dynamic stack, we can reduce the allocated memory if the number of elements falls below a certain threshold, preventing memory waste.

Using dynamic allocation allows the stack to expand as needed without risking overflow until the system memory is exhausted. It also helps in managing memory more effectively by releasing unused memory when elements are removed.

3.6.3 Initial Size and Resizing

- **Initial Size**: The stack starts with a small initial capacity (e.g., `10` elements). This keeps the memory usage low when the stack is empty or has only a few elements.
- **Doubling Capacity**: When the stack becomes full, realloc doubles its capacity to handle more elements. This exponential growth (doubling each time it's full) allows the stack to expand efficiently without needing frequent reallocations.
- **Shrinking Capacity**: To optimize memory usage, the stack can shrink when elements are removed. For instance, if the number of elements drops below half the current capacity, realloc can reduce the stack's size by half. This approach is particularly helpful when stacks are used for temporary data storage, as it prevents memory from being occupied by unused space.

3.7 APPLICATIONS OF STACKS

Some simple applications of stacks:

1. **Managing Function Calls**: When a function is called in a program, it is added to a stack. When the function finishes, it is removed from the stack, allowing the program to go back to the previous task. This is helpful for handling functions that call other functions, especially in recursion.
2. **Solving Math Expressions**: Stacks help in solving complex math expressions, especially when they are written in a form like `AB+` (postfix) or `+AB` (prefix). This helps computers understand and solve expressions step by step.
3. **Backtracking**: In puzzles or searches (like mazes or maps), stacks help keep track of choices. If a choice leads to a dead end, the stack allows you to go back to the previous choice and try a different path.
4. **Undo Feature**: In text editors and other programs, stacks are used to implement "undo." Each action (like typing a letter) is added to a stack. When you click "undo," the last action is removed, reversing it.
5. **Temporary Storage**: Stacks store temporary data, like variables used within a function. When the function finishes, this temporary data is automatically removed in the Last-In, First-Out (LIFO) order.

These examples show how stacks help organize tasks and data in many everyday applications.

3.8 KEY TERMS

Stack, Function Call Stack, Postfix Expression, Prefix Expression, Backtracking

3.9 SELF ASSESSMENT QUESTIONS

1. What is the Last-In, First-Out (LIFO) principle in stacks, and how does it work?
2. Describe the push and pop operations in a stack. What conditions can lead to overflow or underflow?
3. How is a stack implemented using arrays, and what are the limitations of this approach?
4. Give an example of a real-world application of stacks, such as function call management or backtracking.
5. What is the purpose of the peek operation, and how does it differ from the pop operation?

3.10 SUGGESTED READINGS

1. "Introduction to Algorithms" by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein – Explains the stack data structure and its use in algorithms.

2. "The Art of Computer Programming" by Donald Knuth – Provides an in-depth discussion on data structures, including stacks, and their mathematical applications.

3. "Data Structures Using C" by Reema Thareja – Introduces stacks, stack operations, and practical applications in C programming.

4. "Data Structures and Algorithms Made Easy" by Narasimha Karumanchi – A beginner-friendly guide to understanding stacks and other fundamental data structures.

Dr.G.Neelima

Lesson - 4

Queues

OBJECTIVES

The objectives of this lesson are

1. Understand what a queue is and how it follows the First-In-First-Out (FIFO) principle.
2. Learn about different types of queues, including linear, circular, priority, and double-ended queues.
3. Explore basic queue operations such as enqueue, dequeue, and checking if the queue is full or empty.
4. Understand how to implement queue operations using arrays.
5. Learn the practical applications of queues like task scheduling.

STRUCTURE

4.1 INTRODUCTION

Queues are essential data structures in computer science that operate on the First-In, First-Out (FIFO) principle, where elements are added at one end (rear) and removed from the other (front). They are crucial in scenarios requiring ordered processing, such as task scheduling, data streaming, and real-time data handling. This lesson examines the characteristics and applications of queues, their types (linear, circular, priority, and deque), and the fundamental operations—enqueue, dequeue, isEmpty, isFull, and traversal. The implementation of queues using arrays and the distinction between linear and circular queues are also discussed, with practical applications like Breadth-First Search (BFS) and task management systems highlighted.

4.2 QUEUES

A queue is a linear data structure that follows the First-In, First-Out (FIFO) principle. This means that the first element added to the queue is the first one to be removed. You can imagine a queue as a line of people waiting for service; the person at the front of the line is the first to be served, and new people join at the end.

Queues are often implemented using arrays or linked lists. An array-based queue is straightforward but requires a maximum size, while a linked-list queue allows dynamic resizing. In queues, elements are added at one end (the rear) and removed from the other end (the front), making it suitable for applications where the order of processing is crucial.

4.2.1 Role and Characteristics of Queues

Queues play a critical role in computer science for several reasons:

1. **Task Scheduling**: In operating systems, queues are used to manage tasks, where each task waits in line until its turn. Job scheduling and print spooling are examples where queues are essential.
2. **Data Streaming**: Queues are commonly used to manage streaming data. As data arrives, it's added to the rear, and as it's processed, it's removed from the front.
3. **Breadth-First Search (BFS)**: Queues are central to BFS algorithms, where nodes in a graph or tree are processed level by level. Each unvisited node is added to the queue and processed in FIFO order.
4. **Real-Time Data Processing**: In applications like real-time data analytics or messaging systems, queues handle data as it arrives and is processed in order, ensuring that the earliest data is handled first.

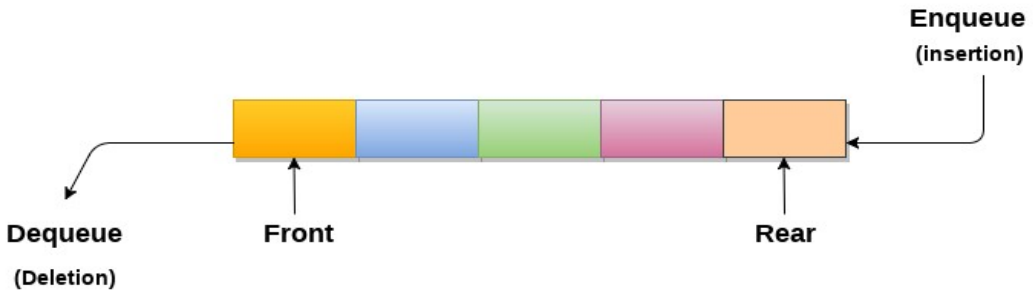


Fig 4.1 Representation of Queue

4.2.2 Features of Queue

Queues are specialized data structures that follow the First-In, First-Out (FIFO) principle, meaning the first element added is the first to be removed. This structure is commonly used when the order of processing is important, such as in scheduling, task management, and real-time data handling. The main features of a queue are:

- A queue is an ordered collection of elements of the same data type, arranged in a specific sequence.
- It follows the First-In, First-Out (FIFO) principle, meaning the first element added is the first to be removed.
- The Enqueue operation adds new elements to the queue, while the Dequeue operation removes elements from the front.
- The Front points to the first element in the queue, and the Rear points to the last element. Insertion occurs at the rear, and removal occurs at the front.
- A queue is in an Overflow state when it reaches its maximum capacity (FULL) and in an Underflow state when it has no elements left (EMPTY).

4.3 TYPES OF QUEUES

Queues can be implemented in different ways, each serving various needs:

1. **Linear Queue**: A simple queue with fixed-size, where elements are added at the rear and removed from the front. However, once full, it does not reuse empty spaces created by removed elements.
2. **Circular Queue**: A more efficient queue that reuses empty spaces by wrapping around to the beginning of the array.
3. **Priority Queue**: Elements are removed based on priority rather than FIFO order.
4. **Deque (Double-Ended Queue)**: Allows insertion and deletion from both ends.

4.4 Linear Queue and Circular Queue

Linear Queue and **Circular Queue** are two fundamental types of queue data structures that operate on the **First-In-First-Out (FIFO)** principle, meaning the first element added is the first one to be removed.

- A **Linear Queue** is the simplest form, where elements are added at the rear and removed from the front. It is implemented as a fixed-size array, so once the rear pointer reaches the end, no more elements can be added, even if there is space at the

beginning. This can lead to inefficiencies, as empty positions cannot be reused without resetting or shifting elements.

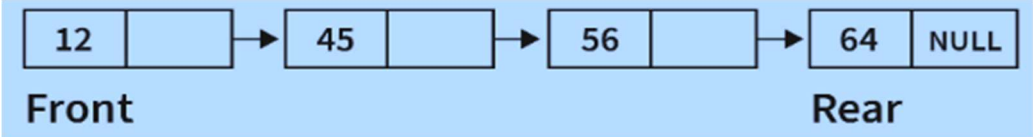


Fig 4.2. Linear Queue

- A **Circular Queue**, on the other hand, overcomes this limitation by allowing the rear and front pointers to "wrap around" the array when they reach the end. This wrap-around feature allows efficient use of all available positions in the array, making it suitable for applications where memory utilization is essential.

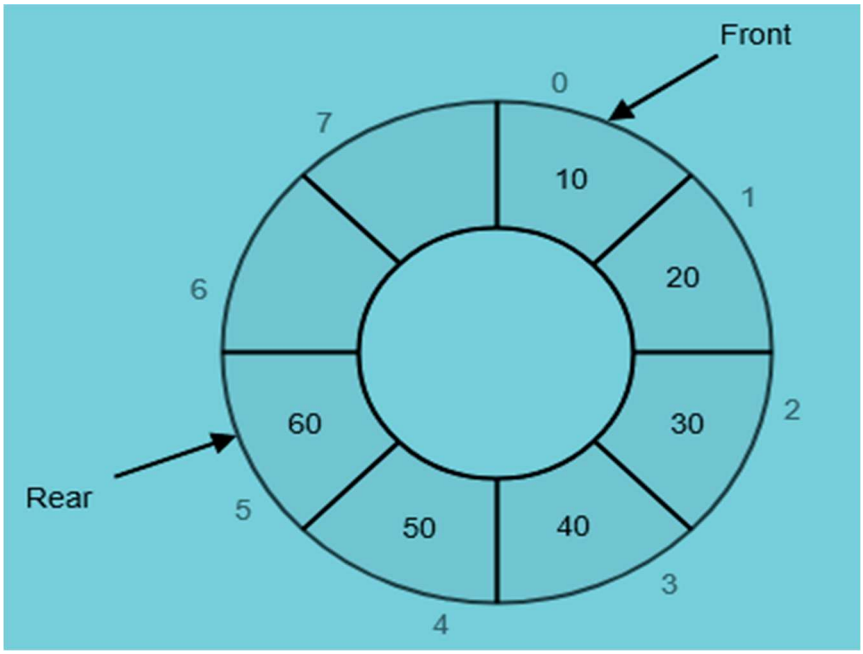


Fig 4.3. Circular Queue

Each type of queue is suited to different situations, with linear queues being simpler to implement and understand, while circular queues are more memory-efficient and versatile in continuous data processing.

4.5 Queue Operations

Queues are versatile data structures that operate on the First-In-First-Out (FIFO) principle, where elements are added at one end (rear) and removed from the other (front). Below is a detailed explanation of the key operations used in managing a queue.

1. CreateQueue

Purpose: Initializes an empty queue with a specified maximum size. This operation sets up the fundamental structure of the queue and prepares it for use.

Process:

1. A queue is initialized with a predefined maximum capacity to store a specific number of elements.
2. Two pointers or indices, front and rear, are used to manage the queue:
    - Front points to the first element in the queue (initially set to -1, indicating the queue is empty).
    - Rear points to the last added element (also set to -1 initially).
3. Memory is allocated for the queue, either statically (in a fixed-size array) or dynamically (using techniques like dynamic memory allocation).

2. IsEmptyQueue

Purpose: Checks whether the queue is empty and ensures that operations like deletion or retrieval are only performed when the queue contains elements.

Process:

1. The queue is considered empty if:
    - In a linear queue: `front == -1` or `front > rear`
    - In a circular queue: `front == -1` (when front and rear pointers are uninitialized or reset).
2. This operation is particularly useful before performing DeleteQueue or Front operations to avoid underflow errors (attempting to remove an element from an empty queue).

3. IsFullQueue

Purpose: Checks whether the queue has reached its maximum capacity, ensuring that new elements are only added when space is available.

Process:

1. The queue is considered full when:
    - In a linear queue: `rear == MAX_QUEUE_SIZE - 1` (rear pointer reaches the end of the array).
    - In a circular queue: `(rear + 1) % MAX_QUEUE_SIZE == front` (rear pointer loops back and coincides with the front pointer).
2. This operation is essential before performing the **AddQueue** operation to avoid overflow errors (attempting to add an element to a full queue).

4. AddQueue (Enqueue)

Purpose: Adds a new element to the rear of the queue. This operation ensures that elements are added in the correct sequence, maintaining the FIFO order.

Process:

1. **Check for Overflow:**
    - Before adding a new element, the queue is checked for full capacity using the **IsFullQueue** operation.
    - If the queue is full, an overflow error is raised, and the element cannot be added.
2. **Initialize Front (if required):**
    - If the queue is empty (indicated by front == -1), initialize the front pointer to 0.
3. **Add Element:**
    - Increment the rear pointer to point to the next available position in the queue.
    - Place the new element at the position indicated by the rear pointer.

5. DeleteQueue (Dequeue)

Purpose: Removes an element from the front of the queue. This operation follows the FIFO principle, ensuring that elements are removed in the same order they were added.

Process:

1. **Check for Underflow:**
    - Before removing an element, the queue is checked for emptiness using the **IsEmptyQueue** operation.
    - If the queue is empty, an underflow error is raised, and no element can be removed.
2. **Remove Element:**
    - Retrieve the element at the position indicated by the front pointer.
    - Increment the front pointer to point to the next element in the queue.
3. **Reset the Queue (if required):**
    - If the queue becomes empty after the operation (front > rear in a linear queue or front == rear + 1 in a circular queue), reset both pointers to -1.

These operations provide the basic foundation for effectively managing a queue in various applications, ensuring data is processed in the correct sequence without errors.

4.6 INSERT OPERATION(ENQUEUE) IN A LINEAR QUEUE

A **queue** can be implemented using arrays, where each element occupies a specific position within the array. In this implementation, two main operations are performed:

- **Insert**: Adding an element to the end of the queue.
- **Delete**: Removing an element from the front of the queue.

In an array-based linear queue, the Insert operation, also known as Enqueue, adds an element to the end of the queue. This is achieved by managing two pointers:

1. **Front** - indicating the start of the queue
2. **Rear** - indicating the end of the queue

The following conditions help define the state of the queue:

1. **Queue is empty:** front == -1 && rear == -1
2. **Queue has only one element:** front == rear && front != -1
3. **Queue is full:** rear == N - 1 (for a fixed-size array of size N).

The ENQUEUE operation adds an element to the queue at the position indicated by rear.

A one dimensional array Q[I.....N] can be used to represent a queue. Here, we need two variables namely Front and Rear. The Rear is used to insert an element into the queue. whereas front is used to delete an element from the queue.

Front	Rear	State of the Queue
0	0	Queue has only one Element
-1	-1	Queue is Empty(Underflow Condition)
0	N-1	Queue is FULL
0	N	Overflow Condition

Fig 4.4 States of the Queue

In the above diagram, the states of the queue depending on the values of front and rear were represented.

4.6.1 Steps for Insert(Enqueue) Operation

The Insert (Enqueue) operation adds a new element to the rear of the queue. This operation ensures that elements are added in the correct sequence, maintaining the First-In, First-Out (FIFO) order.
Below are the steps involved in the Enqueue operation:

1. **Check for Overflow**
    - Before adding an element, ensure that the queue is not full.
    - For a linear queue, this condition is checked as:
        - `rear == N - 1` (where `N` is the maximum size of the array).
    - If the queue is full, the operation is halted, and an overflow error is raised.
2. **Initialize Front Pointer**
    - If the queue is empty (`front == -1` and `rear == -1`), initialize the front pointer to `0`.
    - This step is required only for the first insertion, as the front pointer needs to point to the first element in the queue.
3. **Increment the Rear Pointer**
    - Move the rear pointer to the next available position in the array to indicate where the new element will be inserted:
        - For linear queues: Increment the rear pointer by 1 (rear++).
        - For circular queues: Update the rear pointer using the formula: rear = (rear + 1) % N (to wrap around the array).
4. **Insert the Element**
    - Place the new element at the position in the array indicated by the updated rear pointer.
    - This ensures that the new element is added to the end of the queue.
5. **Confirm Successful Insertion**
    - After the element is added, confirm that the insertion was successful by verifying the updated state of the queue. Optionally, display a success message or the updated queue structure.

Implementation

```
void enqueue_linear(int queue[], int *front, int *rear, int max_size, int item)
 {
  if (*rear == max_size - 1) {  // Check if queue is full
     printf("Queue is Full!\n");
  } else {
     if (*front == -1) {  // First insertion
        *front = 0;
```

```
    }
    *rear = *rear + 1;
    queue[*rear] = item;
}
```

Example of Insert Operation

Let's say we have an array A of size 5 and we want to insert elements 10, 20, 30, 40, 50 in sequence. For this implementation array of size 5 is taken.

Initial State:

The queue is empty, so front = -1 and rear = -1.

Array a: [_, _, _, _, _]
Front = -1, Rear = -1

Step 1: Insert 10

- Since the queue is empty (front = -1), set front = 0.
- Increment rear to 0.
- Place 10 at A[0].

Array: [10, _, _, _, _]
Front = 0, Rear = 0

Step 2: Insert 20

- Increment rear to 1.
- Place 20 at A[1].

Array: [10, 20, _, _, _]
Front = 0, Rear = 1

Step 3: Insert 30

- Increment rear to 2.
- Place 30 at A[2].

Array: [10, 20, 30, _, _]
Front = 0, Rear = 2

Step 4: Insert 40

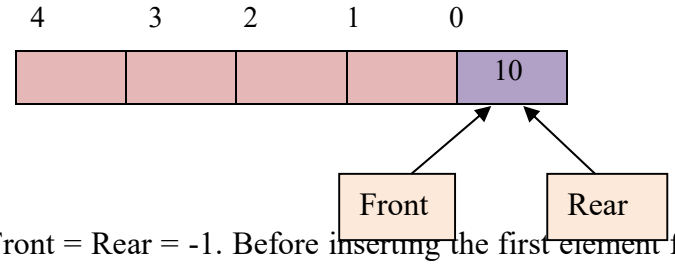
- Increment rear to 3.
- Place 40 at A[3].

Array: [10, 20, 30, 40, _]
Front = 0, Rear = 3

Step 5: Insert 50

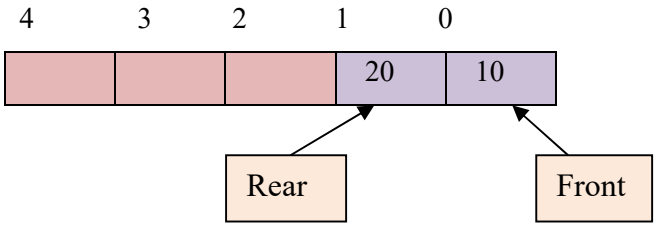
- Increment rear to 4.
- Place 50 at A[4].

Visual Representation of the above example is

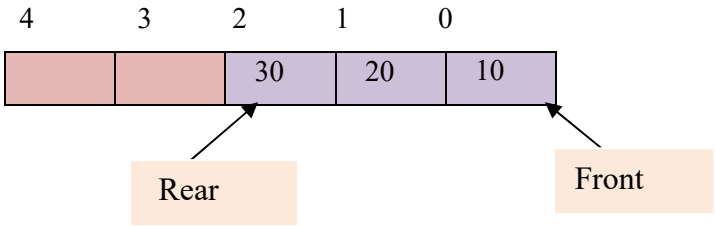


Initially, Front = Rear = -1. Before inserting the first element front and rear values should be initialised to 0.

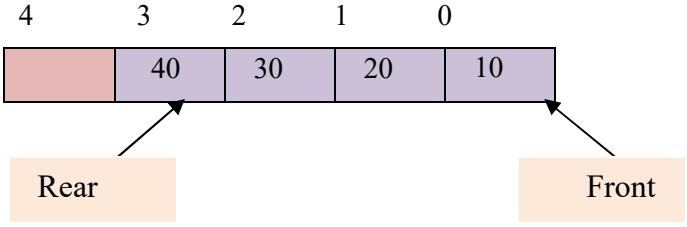
Front = Rear = 0, A[Rear] = 10



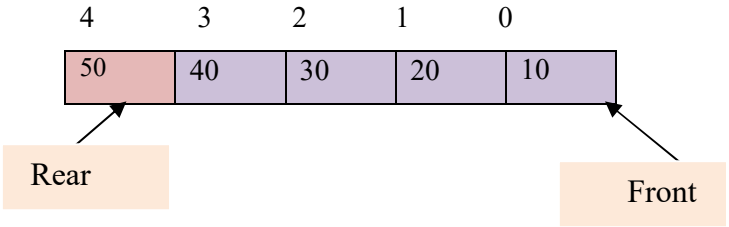
Front = 0 , Rear = 1, A[Rear] = 20



Front = 0 , Rear = 2, A[Rear] = 30



Front = 0 , Rear = 2, A[Rear] = 30



At this point, the queue is full since rear has reached N - 1.

4.7 DELETE OPERATION(DEQUEUE) IN A LINEAR QUEUE

The Delete (Dequeue) operation removes an element from the front of the queue. It follows the First-In, First-Out (FIFO) principle, ensuring that the element which has been in the queue the longest is removed first.

Below are the detailed steps involved in the Dequeue operation:

1**. Check for Underflow**

- Before attempting to delete an element, ensure that the queue is not empty.
- In a linear queue, the condition for an empty queue is:
  - front == -1 (queue has no elements).
  - Or, front > rear (all elements have been dequeued).
- If the queue is empty, the operation is halted, and an underflow error is raised, indicating that there are no elements to delete.

2. **Retrieve the Element**

- Access the element at the position indicated by the front pointer in the array. This is the element to be removed from the queue.
- The retrieved element can be used or displayed for confirmation before deletion.

3. Increment the Front Pointer

- After retrieving the element, move the front pointer to the next position in the array to indicate the new front of the queue:
  - front++ (increment the front pointer by 1).
- This step effectively removes the element from the queue, as it will no longer be accessible.

4. **Check for Queue Reset**

- If the front pointer surpasses the rear pointer (i.e., front > rear), it means the queue has become empty after the deletion.
- In this case, reset both pointers to `-1` to indicate that the queue is empty.

4.7.1 Steps for Delete Operation

From the above example, Consider the queue in its current state:

Array: [10, 20, 30, 40, 50]
Front = 0, Rear = 4

Step 1: Delete Element 10

- Retrieve 10 from A[0].
- Increment front to 1.

Array: [_, 20, 30, 40, 50]
Front = 1, Rear = 4

Step 2: Delete Element 20

- Retrieve 20 from A[1].
- Increment front to 2.

Array: [_, _, 30, 40, 50]
Front = 2, Rear = 4

Step 3: Delete Element 30

- Retrieve 30 from A[2].
- Increment front to 3.

Array: [_, _, _, 40, 50]
Front = 3, Rear = 4

Step 4: Delete Element 40

- Retrieve 40 from A[3].
- Increment front to 4.

Array: [_, _, _, _, 50]
Front = 4, Rear = 4

Step 5: Delete Element 50

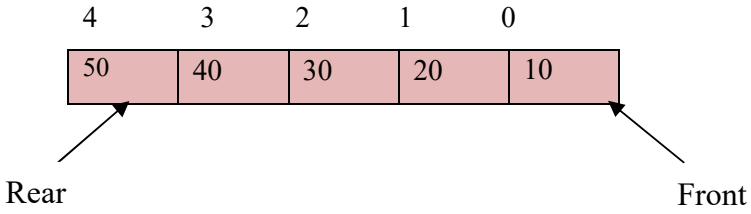
- Retrieve 50 from A[4].
- Increment front to 5, making front > rear.
- Since front has surpassed rear, reset front and rear to -1 to indicate the queue is empty.

Implementation

```
int dequeue_linear(int queue[], int *front, int *rear) {
```

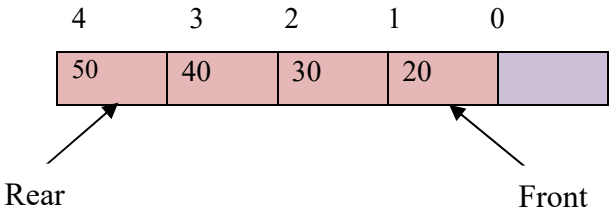
```
if (*front == -1 || *front > *rear) {  // Check if queue is empty
    printf("Queue is Empty!\n");
    return -1;
} else {
    int item = queue[*front];
    *front = *front + 1;
    return item;     } }
```

Visual Representation of the above example

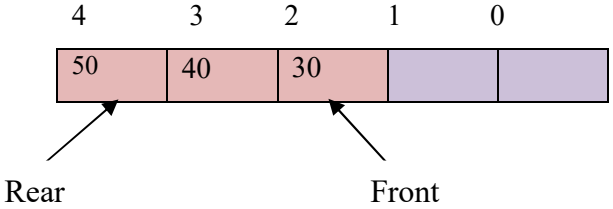


If suppose delete( ) operation has to be performed on the queue, then the first element that was inserted into the queue would be deleted first. So it can be said that the element at the front end of the queue would be deleted.

After performing one delete operation the elements in the queue are



If another delete operation is performed, the element pointed by the front variable will be deleted. Then the queue structure would be like



Whenever delete operation is performed, elements are deleted from the front end of the queue. These operations allow the queue to maintain a First-In-First-Out (FIFO**)** structure, with the front element always removed first.

4.8 INSERT OPERATION(ENQUEUE) IN A CIRCULAR QUEUE

n a **circular queue**, the **Enqueue** operation adds a new element to the rear of the queue. Unlike a linear queue, where the rear pointer cannot wrap around, a circular queue overcomes the limitation of unused spaces by treating the array as circular. This means the rear pointer wraps to the beginning of the array when it reaches the end.

Below are the steps for performing the **Enqueue** operation in a circular queue:

Steps for Enqueue Operation in a Circular Queue

1. **Check for Overflow**

- Before adding a new element, check if the queue is full.
- In a circular queue, the condition for a full queue is:
  - (rear + 1) % N == front
    (where N is the maximum size of the array, and % ensures wrapping around).
- If this condition is true, the operation is halted, and an **overflow error** is raised, indicating that the queue has no space to add more elements.

2. **Initialize Front Pointer (if needed)**

- If the queue is empty (front == -1 and rear == -1), initialize the **front pointer** to 0 before proceeding.
- This step is only performed for the first insertion, as the **front** pointer needs to point to the first element of the queue.

3. **Update the Rear Pointer**

- Increment the **rear pointer** to the next position in a circular manner using the formula:
  - rear = (rear + 1) % N
- This ensures that the **rear pointer** wraps back to the beginning of the array when it reaches the end.

4. **Insert the Element**

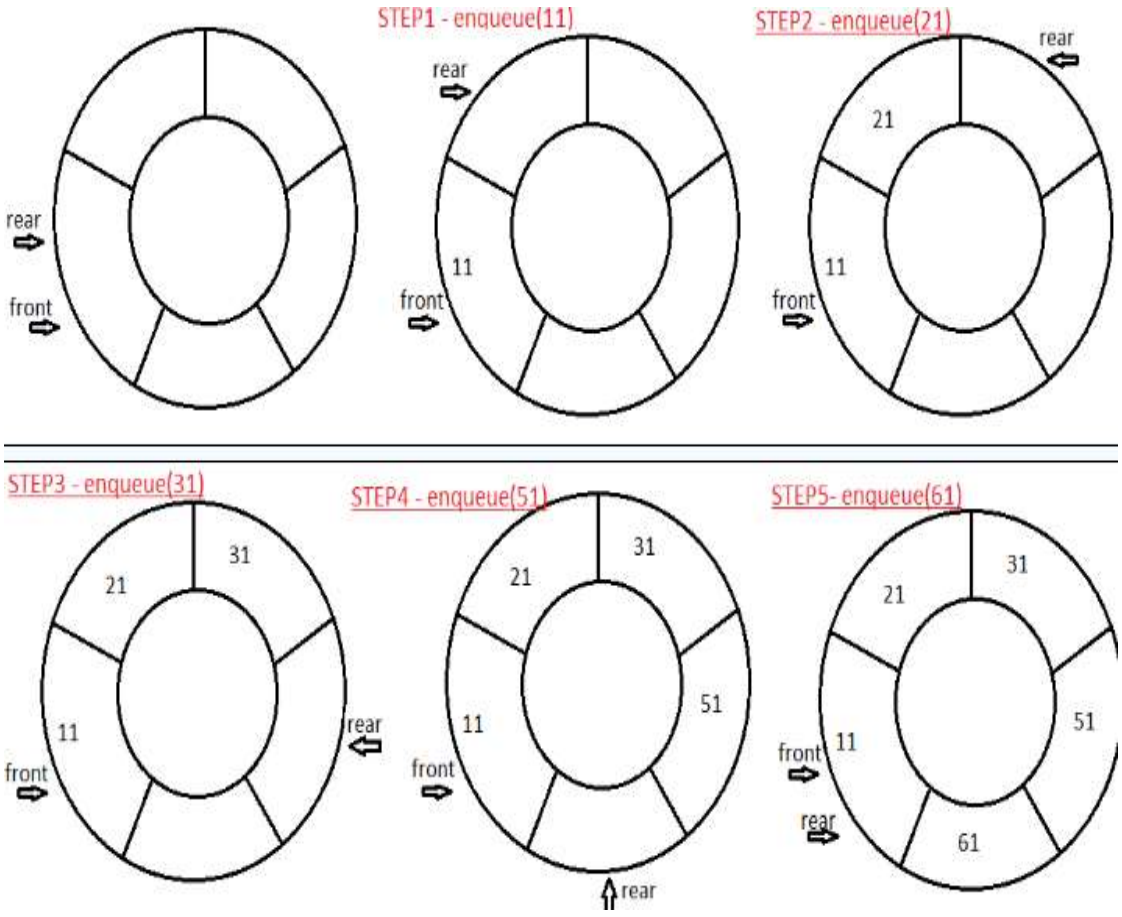
- Place the new element at the position indicated by the **rear pointer** in the array.
- This step ensures that the new element is added to the end of the queue.

5. **Confirm Successful Insertion**

- After adding the element, confirm that the operation was successful by verifying the updated state of the queue or displaying a success message.

```
void enqueue_circular(int queue[], int *front, int *rear, int max_size, int item) {
    if ((*rear + 1) % max_size == *front) {  // Check if queue is full
        printf("Queue is Full!\n");
    } else {
        if (*front == -1) {  // First insertion
            *front = 0;
        }
        *rear = (*rear + 1) % max_size;
        queue[*rear] = item;
    }
}
```

Example



The above figure is explained below

Initial State

- The circular queue is empty, as indicated by the absence of any elements.
- Both the front and rear pointers are not yet defined (usually -1), or in this case, are pointing outside the structure, ready for the first element to be added.

Step 1: Enqueue(11)

- Action: The element 11 is added to the queue.
- Front Pointer: Since the queue was empty, the front pointer is initialized to 0.
- Rear Pointer: The rear pointer is also set to 0, as this is the first element.
- Queue State: [11, _, _, _, _]
  (Front = 0, Rear = 0).

Step 2: Enqueue(21)

- The element 21 is added to the queue.
- Rear Pointer: The rear pointer is incremented to the next position:

  (rear + 1) % N = (0 + 1) % 5 = 1.

- Queue State: [11, 21, _, _, _]
  (Front = 0, Rear = 1).

Step 3: Enqueue(31)

- The element 31 is added to the queue.
- Rear Pointer: The rear pointer is incremented: (rear + 1) % N = (1 + 1) % 5 = 2.
- Queue State: [11, 21, 31, _, _]
  (Front = 0, Rear = 2).

Step 4: Enqueue(51)

- The element 51 is added to the queue.
- Rear Pointer: The rear pointer is incremented: (rear + 1) % N = (2 + 1) % 5 = 3.
- Queue State: [11, 21, 31, 51, _]
  (Front = 0, Rear = 3).

Step 5: Enqueue(61)

- The element 61 is added to the queue.
- Rear Pointer: The rear pointer is incremented: (rear + 1) % N = (3 + 1) % 5 = 4.
- Queue State: [11, 21, 31, 51, 61]
  (Front = 0, Rear = 4).

4.9 DELETE OPERATION (DEQUEUE) IN A CIRCULAR QUEUE

The Delete (Dequeue) operation in a circular queue removes an element from the front of the queue. It follows the First-In, First-Out (FIFO) principle, ensuring that the element that has been in the queue the longest is removed first. The circular nature of the queue allows efficient memory utilization, as the front pointer wraps around when it reaches the end of the queue.

Below are the detailed steps involved in the Dequeue operation in a circular queue:
Steps for Dequeue Operation in a Circular Queue

1. **Check for Underflow**

- Before attempting to delete an element, check whether the queue is empty.
- The conditions for an empty queue in a circular queue are:
  - front == -1 (no elements in the queue).
- If the queue is empty, the operation is halted, and an underflow error is raised.

2. **Retrieve the Element**

- Access the element at the position indicated by the front pointer in the array. This is the element to be removed from the queue.
- Optionally, you can display or store the retrieved element for confirmation.

3. **Update the Front Pointer**

- Increment the front pointer to the next position in a circular manner:
  - front = (front + 1) % N (where N is the size of the array).
- This ensures that the front pointer wraps around to the beginning of the array when it reaches the end.

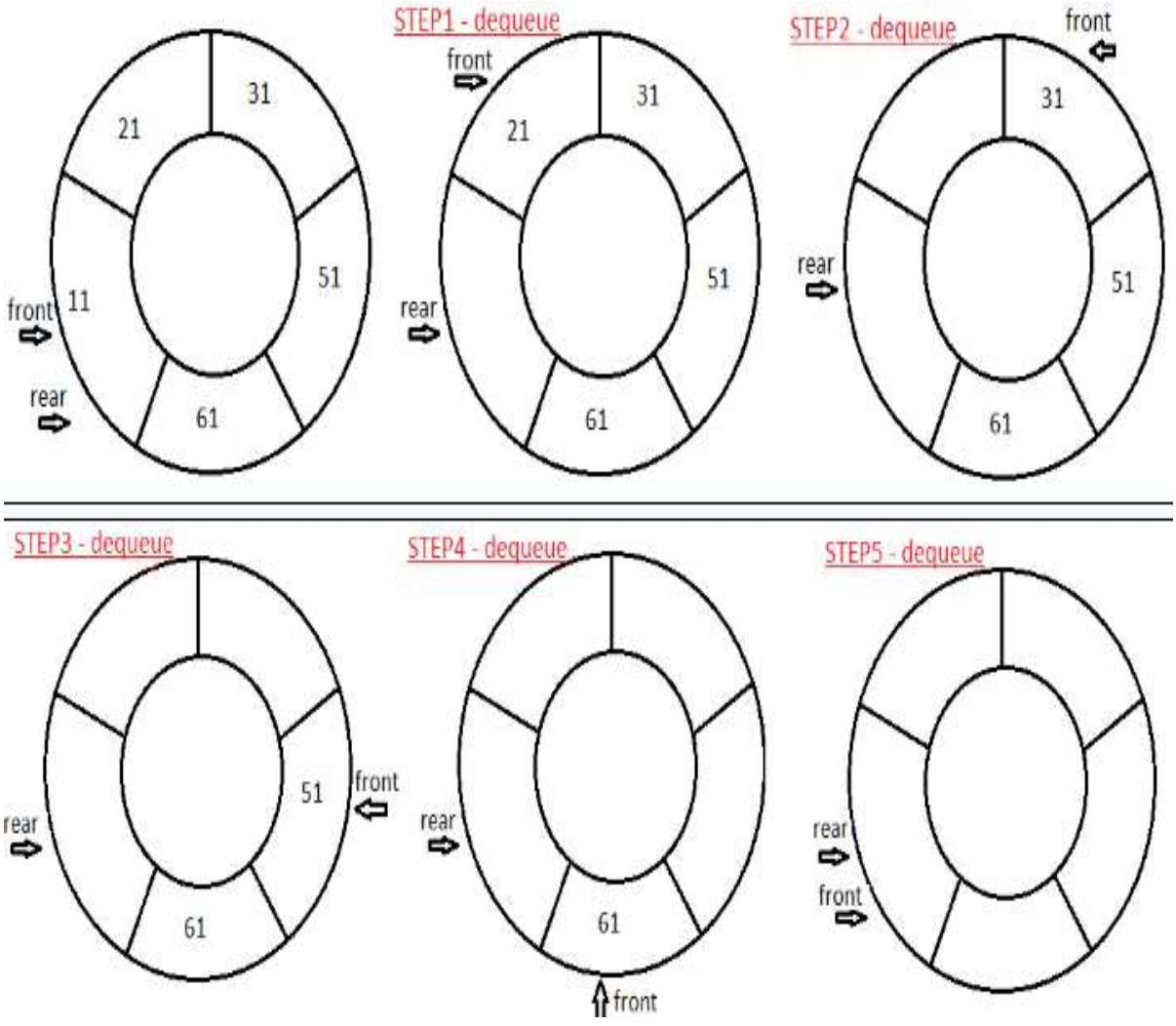
4. **Check for Queue Reset**

- If the front pointer surpasses the rear pointer after the operation, it indicates that the queue has become empty.
- Reset both the front and rear pointers to -1 to indicate that the queue is empty.

```
int dequeue_circular(int queue[], int *front, int *rear, int max_size) {
   if (*front == -1) {  // Check if queue is empty
      printf("Queue is Empty!\n");
      return -1;
   } else {
      int item = queue[*front];
      if (*front == *rear) {  // Queue becomes empty after deletion
         *front = *rear = -1;
```

```
} else {
    *front = (*front + 1) % max_size;
}
return item;    } }
```

Example



Step-by-Step Explanation:

1. **Initial State (Before Step 1):**
   - The queue contains the elements: 11, 21, 31, 51, 61.
   - Front is at the first position (pointing to 11).
   - Rear is at the last position (pointing to 61).

Step 1 - Dequeue:

- The element at the front (11**)** is removed.
- The `Front` pointer moves to the next position, pointing to `21`.

Step 2 - Dequeue:

- The element at the front (21) is removed.
- The `Front` pointer moves to the next position, pointing to `31`.

Step 3 - Dequeue:

- The element at the front (31) is removed.
- The `Front` pointer moves to the next position, pointing to `51`.

Step 4 - Dequeue:

- The element at the front (51) is removed.
- The `Front` pointer moves to the next position, pointing to `61`.

Step 5 - Dequeue:

- The element at the front (61**)** is removed.
- Now the queue becomes empty.
- Both the `Front` and `Rear` pointers reset, indicating the queue is empty.

4.10 Queue Operations: IsEmpty and IsFull

The IsEmpty and IsFull operations are fundamental for checking the status of both linear and circular queues. Below is a breakdown of these operations for both types.

1. Linear Queue

IsEmpty Operation

- A linear queue is considered empty when there are no elements left to dequeue.
- If the front pointer is greater than the rear pointer, the queue is empty.

Pseudocode:

```
if front > rear:
    return True
else:
    return False
```

IsFull Operation

- A linear queue is full when the rear pointer reaches the maximum capacity (end of the queue array).
- If the rear pointer is equal to the maximum size - 1, the queue is full.

```
IsFull:
    if rear == MAX_SIZE - 1:
        return True
    else:
        return False
```

2. Circular Queue

In a circular queue, the rear pointer wraps around to the start of the queue when the array's end is reached, ensuring efficient use of space. Thus, the conditions for IsEmpty and IsFull differ.

IsEmpty Operation

- A circular queue is considered empty when the front and rear pointers are at the same position, and the queue contains no elements.
- If front == -1, or front == rear + 1 after dequeuing, the queue is empty.

```
    if front == -1:
        return True
    else:
        return False
```

IsFull Operation

- A circular queue is full when the next position of rear (calculated using modulo) is equal to the front pointer.
- If (rear + 1) % MAX_SIZE == front, the queue is full.

```
IsFull:
    if (rear + 1) % MAX_SIZE == front:
        return True
    else:
        return False
```

4.11 KEY TERMS

Queue, Enqueue, Dequeue, Overflow, Underflow, Linear Queue, Circular Queue, Priority Queue, Front and Rear Pointers

4.12 SELF ASSESSMENT QUESTIONS

1. Explain the First-In-First-Out (FIFO) principle and its relevance to queues.
2. Describe the difference between linear and circular queues. How does a circular queue overcome the limitations of a linear queue?

3. What is the role of the front and rear pointers in a queue's array implementation?
4. Discuss the conditions of overflow and underflow in queue operations, with examples.
5. Implement a queue using an array and demonstrate the enqueue and dequeue operations in code.

4.13 Suggested Readings

1. "Fundamentals of Data Structures in C" by Ellis Horowitz, Sartaj Sahni, and Susan Anderson-Freed
2. "Introduction to Algorithms" by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein
3. "Data Structures Using C" by Reema Thareja
4. "Data Structures and Algorithms Made Easy" by Narasimha Karumanchi

Dr. G. Neelima

Lesson - 5
Introduction to Linked Lists

OBJECTIVE

The objective of this lesson is to

1. Provide a comprehensive understanding of linked lists, a fundamental data structure in computer science.
2. Explain the concept, structure, and types of linked lists, including singly linked, doubly linked, circular, and doubly circular linked lists.
3. Explore the significance of linked lists and their representation in memory.
4. Demonstrate efficient management of dynamic memory through linked list operations.
5. Compare linked lists with arrays in terms of memory layout, access efficiency, insertion and deletion performance, and use cases to aid in selecting the most suitable data structure for specific applications.

STRUCTURE

5.1 INTRODUCTION

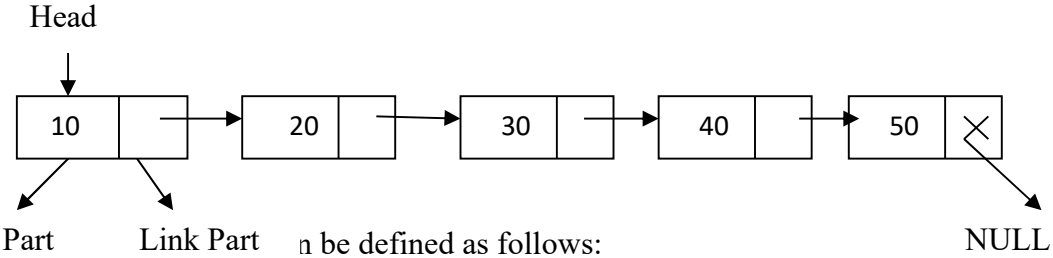
A queue is a linear data structure that operates on the First-In-First-Out (FIFO) principle, meaning that the first element added to the queue will be the first to be removed. This structure is essential for managing ordered sequences of elements, making it ideal for situations where tasks need to be processed in the exact order they arrive, such as in scheduling, resource management, and data transfer applications. A queue typically supports two primary operations: enqueue, which adds an element to the end, and dequeue, which removes an element from the front. Variants of queues, such as circular queues, priority queues, and double-ended queues (deques), offer flexibility for various scenarios requiring efficient and structured data handling. As a fundamental concept in data structures, queues play a crucial role in algorithms and system designs, particularly in real-time and multi-user environments where orderly processing is critical.

5.2 REPRESENTATION OF LINKED LISTS

Linked lists are one of the fundamental data structures in programming, commonly used to organize data in a flexible, dynamic way. Unlike arrays, linked lists allow for efficient insertion and deletion of elements at various positions without requiring reallocation or reorganization of the entire data structure.

Linked lists are represented as a series of nodes, where each node contains two essential parts:

1. **Data Part**: This part stores the actual data of the node, which could be of any data type (integer, character, structure, etc.).
2. **Pointer Part**: This part holds the address of the next node in the list, creating a chain-like structure that links nodes together.



A n be defined as follows:

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

```
} Node;
```

In this structure:

- data holds the information in the node.
- next is a pointer to the next node in the list.

The linked list typically has a **head pointer** that points to the first node in the list. The head pointer is essential as it serves as the starting point for traversing the list. If the list is empty, the head pointer is set to NULL.

5.3 TYPES OF LINKED LISTS

Linked lists are a collection of nodes arranged in various configurations to facilitate different use cases. Here's a detailed overview of the four primary types of linked lists:

- Singly Linked List
- Doubly Linked List
- Circular Linked List
- Circular Doubly Linked List

5.3.1. Singly Linked List

A **Singly Linked List** is the simplest form of linked list, consisting of nodes where each node has:

- **Data**: Stores the node's value or information.
- **Next Pointer**: Points to the next node in the list.

In a singly linked list, nodes are connected in a single direction, from the head node to the last node, where the last node's next pointer is NULL, indicating the end of the list.

Example Structure in C:

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

Characteristics

- **Traversal**: Singly linked lists support forward traversal only, from the head to the last node.
- **Insertion/Deletion**: Easy to insert or delete nodes at the beginning or end but requires traversal to modify nodes in the middle.
- **Memory**: Memory-efficient as each node only requires one pointer (to the next node).

Example Diagram of a Singly Linked List

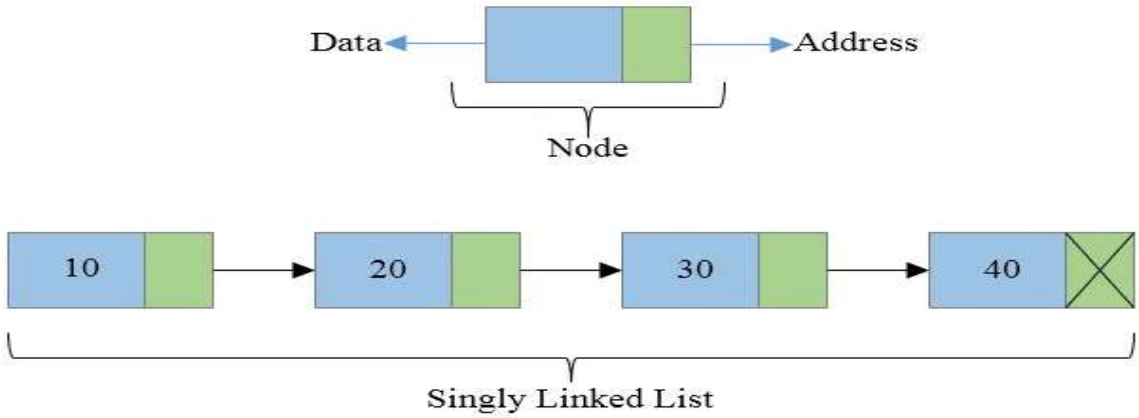


Figure 5.1 Single Linked List representation

In this example:

- The **head** points to the first node, which holds 10.
- Each node's next pointer links to the following node.
- The last node's next pointer is NULL, marking the end.

Advantages:

- Simple structure with minimal memory overhead.
- Efficient for applications needing single-direction traversal.

Disadvantages:

- No backward traversal.
- Accessing an element in the middle requires traversing from the head.

5.3.2. Doubly Linked List

A **Doubly Linked List** extends the singly linked list by adding an additional pointer to each node:

- **Data**: Stores the node's information or value.
- **Next Pointer**: Points to the next node in the sequence.
- **Previous Pointer**: Points to the previous node in the sequence.

This setup allows traversal in both forward and backward directions, making it useful for applications where reverse navigation is required.

Example Structure in C:

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

Characteristics

- **Bidirectional Traversal**: Allows navigation in both forward and reverse directions, enabling more flexible operations.
- **Efficient Deletion**: Nodes can be easily removed without needing to traverse from the head, as each node has a pointer to its predecessor.
- **Memory Usage**: Requires more memory than singly linked lists, as each node contains an additional pointer to the previous node.

Example Diagram of a Doubly Linked List

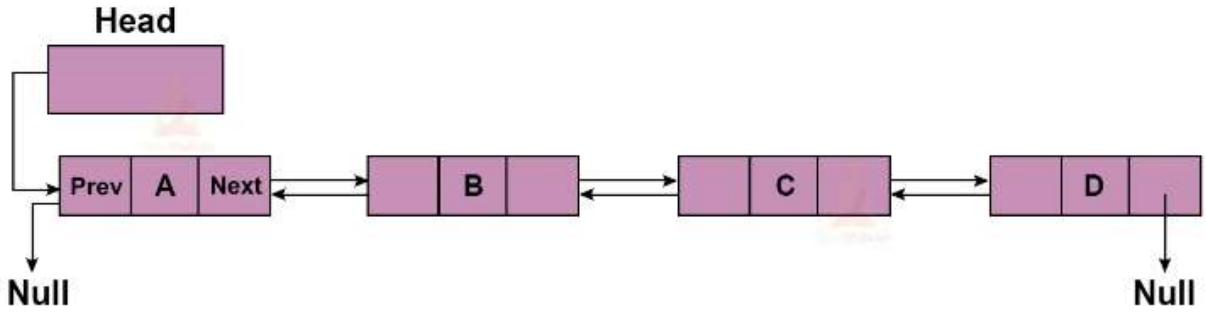


Figure 5.2 Double Linked List representation

In this example:

- Each node has a **prev** pointer pointing to the previous node and a **next** pointer pointing to the next node.
- The first node's prev pointer is NULL, indicating it has no predecessor.
- The last node's next pointer is NULL, indicating the end of the list.

Advantages:

- Allows traversal in both directions.
- Easier to delete nodes as each node has a link to its previous node.

Disadvantages:

- Higher memory usage due to the additional prev pointer in each node.
- More complex to implement than singly linked lists.

5.3.3 Circular Linked List

In a **Circular Linked List**, the last node's next pointer points back to the first node, creating a loop structure. Circular linked lists can be singly or doubly linked.

Singly Circular Linked List

A **Singly Circular Linked List** is a singly linked list in which the last node points back to the head, forming a circle. This type of list allows continuous traversal from any node without reaching a NULL terminator.

Example Diagram of a Singly Circular Linked List:

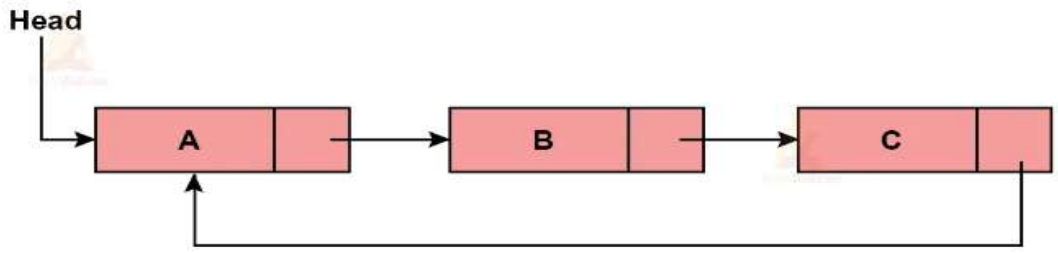


Figure 5.3 Circular Linked List representation

In this example:

- The **head** points to the first node.
- The last node's next pointer links back to the head, creating a circular structure.

Characteristics:

- **Continuous Traversal**: The list can be traversed in a loop without needing to stop at the end.
- **Efficient Cyclic Operations**: Useful for applications that require repeated cycles, such as round-robin scheduling.

Advantages:

- Continuous traversal, with no defined end.
- Simplifies algorithms for cyclic tasks.

Disadvantages:

- More complex to manage than a standard singly linked list, as the last node must always point to the head.

5.3.4 Doubly Circular Linked List

A **Doubly Circular Linked List** combines the properties of a doubly linked list and a circular list, where each node has both prev and next pointers, and the list forms a circle.

Example Diagram of a Doubly Circular Linked List:

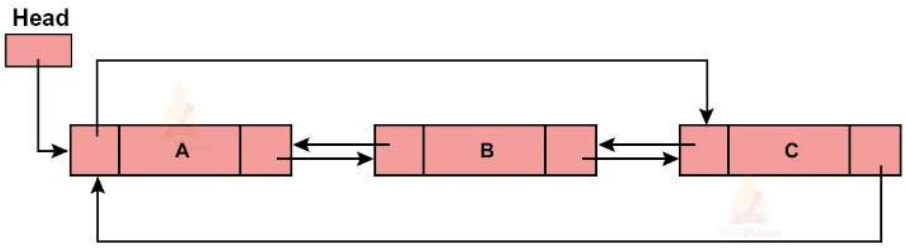


Figure 5.4 Doubly Circular Linked List representation

In this example:

- Each node's next pointer links to the next node, and its prev pointer links to the previous node.
- The last node's next pointer points to the head, and the head's prev pointer points to the last node, completing the circle.

Characteristics:

- **Bidirectional Cyclic Traversal**: Allows traversal from any node in both forward and backward directions.
- **Applications**: Suitable for applications requiring both circular and bidirectional navigation, like playlist management or task scheduling.

Advantages:

- Allows bidirectional and cyclic traversal.
- Efficient for applications that need continuous, two-way traversal.

Disadvantages:

- Highest memory usage among all linked list types.
- Most complex to implement due to maintaining both prev and next pointers in a circular structure.

Summary Table

Table 5.1 Comparison of different linked lists

Type of Linked List	Structure	Advantages	Disadvantages
Singly Linked List	Each node has one pointer to the next node.	Simple, memory-efficient	Single-direction traversal
Doubly Linked List	Each node has pointers to both the next and previous nodes.	Bidirectional traversal	Higher memory usage due to additional pointers
Singly Circular Linked List	Last node points back to the head, forming a loop.	Continuous traversal without an end	More complex to manage
Doubly Circular Linked List	Nodes have pointers to both next and previous nodes in a circular structure.	Bidirectional and cyclic traversal	High memory usage, complex implementation

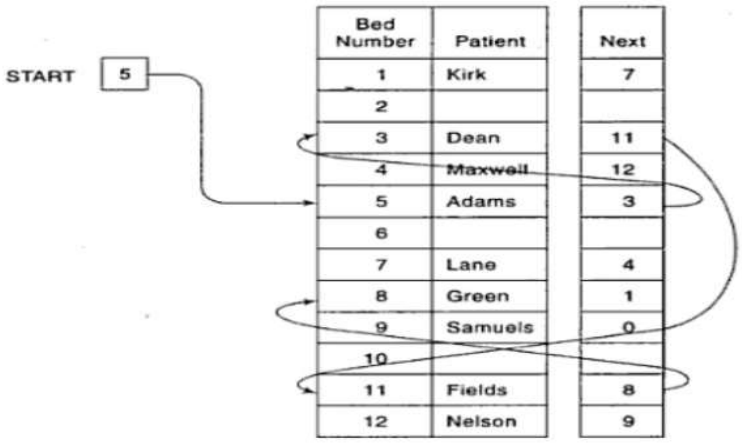
5.4 SIGNIFICANCE OF LINKED LISTS

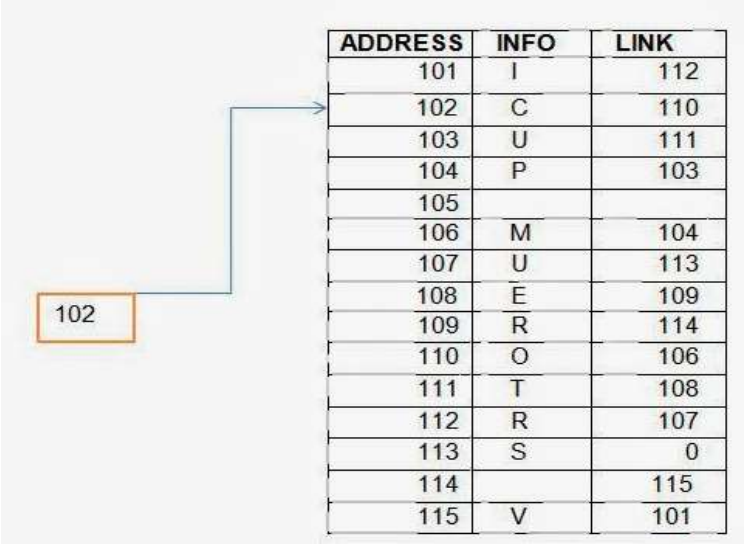
Linked lists offer several advantages over other data structures like arrays, particularly in situations where dynamic data handling and efficient insertion or deletion are required. Here are some key reasons why linked lists are significant:

- **Dynamic Size**: Unlike arrays, which require a predefined size, linked lists can grow or shrink dynamically by allocating or deallocating memory as needed. This flexibility makes them highly suitable for applications where the size of the dataset is not known beforehand.
- **Efficient Insertion and Deletion**: Adding or removing elements in a linked list is efficient, especially when compared to arrays. In an array, insertion or deletion requires shifting elements, leading to a time complexity of O(n). However, in a linked list, insertion or deletion at a particular position requires only adjusting pointers, resulting in a time complexity of O(1) if the position is known.
- **Memory Efficiency**: Since linked lists use only as much memory as needed, they are generally more memory-efficient than arrays, which reserve a contiguous block of memory. Linked lists allocate memory for each node individually, making them suitable for memory-constrained applications.
- **Useful for Implementing Other Data Structures**: Linked lists serve as the basis for implementing other data structures like stacks, queues, hash tables, and adjacency lists for graphs. For instance, in a stack implemented with a linked list, nodes can be added or removed from the top in constant time.

5.5. REPRESENTATION OF LINKED LISTS IN MEMORY

The memory representation of linked lists differs significantly from arrays. Unlike arrays, which store elements in contiguous blocks of memory, linked lists use dynamic memory allocation, where each node is allocated separately. This allows linked lists to grow and shrink dynamically but also requires careful management of memory. In the below diagram, detailed breakdowns of memory allocation, pointer-based structure, memory layout visualization, and memory management are explained.





ADDRESS	INFO	LINK
101	I	112
102	C	110
103	U	111
104	P	103
105		
106	M	104
107	U	113
108	E	109
109	R	114
110	O	106
111	T	108
112	R	107
113	S	0
114		115
115	V	101

Figure 5.5 Memory representation of Linked Lists

5.5.1 Memory Allocation for Nodes

Each node in a linked list is dynamically allocated, which means that memory is assigned to each node separately, rather than all at once in a contiguous block. In C, functions like malloc and calloc are used for this purpose, which allocate memory in the heap rather than the stack. The size of each node is based on the data type and the pointer field(s) it contains.

For example, consider the following code for creating a new node in a linked list:

```
Node* newNode = (Node*) malloc(sizeof(Node));
newNode->data = 10;
newNode->next = NULL;
```

Here's what each line does:

1. malloc(sizeof(Node)): Allocates memory for a new node of type Node. The sizeof(Node) calculates the amount of memory needed for the node's data and pointer fields combined.This allocation happens on the heap, so the memory persists until it is explicitly freed, unlike stack memory that is automatically freed after a function call ends.
2. newNode->data = 10;: Assigns the integer value 10 to the data field of the node. This is an example of a data assignment, but the data field can hold any type or structure of data depending on the application.
3. newNode->next = NULL;: Sets the next pointer to NULL, indicating that this node is currently the last node in the list. The next pointer will later be updated if another node is added after this one.

Advantages of Dynamic Allocation:

- **Flexibility**: Nodes can be added or removed at any time, allowing the list to grow or shrink as needed.

- **Efficient Use of Memory**: Memory is allocated only for nodes that are currently in use, avoiding the problem of wasted memory that can occur in arrays with unused slots.

Disadvantages of Dynamic Allocation:

- **Memory Fragmentation**: Because nodes are not stored in contiguous memory locations, memory fragmentation can occur, which may lead to inefficient use of memory in the long run.
- **Performance Overhead**: Dynamic memory allocation and deallocation are slower compared to stack memory allocation, which can impact performance if nodes are frequently added or removed.

5.5.2 Pointer-based Structure

The unique aspect of linked lists is that each node contains a pointer to the next node in the sequence, creating a "chain" of nodes connected by pointers. This pointer-based structure allows linked lists to be dynamic and flexible but also requires careful handling to maintain the connections between nodes.

Each time a new node is added to the list, the pointers need to be updated to maintain the structure. Let's explore how this works with some examples.

❖ **Adding a Node to the Beginning of a Singly Linked List**

When adding a new node at the beginning of a singly linked list:

1. **Set the New Node's Next Pointer**: The new node's next pointer is set to point to the current head of the list, effectively inserting it before the first element.
2. **Update the Head Pointer**: The head pointer is then updated to point to the new node, making it the new starting point of the list.

Example Code:

```
Node* newNode = (Node*) malloc(sizeof(Node));
newNode->data = 10;
newNode->next = head; // Step 1: Link new node to the current head
head = newNode;       // Step 2: Update head to point to the new node
```

❖ **Adding a Node to the End of a Singly Linked List**

When adding a node to the end of the list:

1. **Traverse to the Last Node**: Start from the head and follow each node's next pointer until reaching the last node (where next is NULL).
2. **Update the Last Node's Pointer**: Set the last node's next pointer to the new node, linking it to the end of the list.
3. **Set the New Node's Next Pointer to NULL**: Set the new node's next pointer to NULL, as it is now the last node.

Example Code:

```
Node * newNode = (Node*) malloc ( sizeof ( Node ) );
newNode->data = 30;
newNode->next = NULL; // Set new node's next to NULL

// Traverse to the end of the list
Node* temp = head;
while (temp->next != NULL)
{
   temp = temp->next;
}

// Link the new node at the end
temp->next = newNode;
```

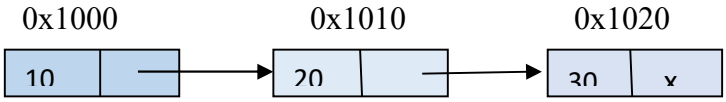
The pointer-based structure is what makes linked lists flexible and dynamic. However, it also requires careful pointer manipulation to ensure the structure remains intact, especially during insertions and deletions.

5.5.3 Visualization of Memory Layout

The non-contiguous memory layout of linked lists allows each node to be located at different memory addresses, which are connected through pointers. Here's an example visualization to illustrate this concept.

Consider a simple linked list with three nodes containing data values 10, 20, and 30. Suppose the nodes are stored at non-contiguous memory addresses.

Linked List Representation



Memory Layout and Pointers

Suppose the memory addresses of each node are as follows:

- **Node 1**: Data = 10, Address = 0x1000
- **Node 2**: Data = 20, Address = 0x1010
- **Node 3**: Data = 30, Address = 0x1020

The nodes are linked by their pointers as follows:

1. **Head** points to 0x1000, which is the address of Node 1.
2. **Node 1 (0x1000)**:
    - data = 10
    - next = 0x1010 (address of Node 2)
3. **Node 2 (0x1010)**:
    - data = 20
    - next = 0x1020 (address of Node 3)

4. **Node 3 (0x1020)**:
    - data = 30
    - next = NULL (indicating the end of the list)

Since each node's memory is distinct, they are connected by pointers rather than being stored contiguously in memory, as would be the case in an array.

Advantages of This Layout

- **Flexible Memory Use**: Nodes can be located anywhere in memory, allowing linked lists to utilize memory more flexibly.
- **Efficient Insertions/Deletions**: Pointers can be updated to add or remove nodes without shifting elements, making insertion and deletion efficient.

5.6 MEMORY MANAGEMENT IN LINKED LISTS

Because linked lists use dynamic memory allocation, memory management is crucial to avoid memory leaks. Each node in a linked list is allocated independently, which means that when a node is removed, its memory must be freed manually. Failure to free memory for removed nodes can lead to memory leaks, where memory is consumed but not released back to the system.

5.6.1 Freeing Nodes Individually

To delete all nodes in a linked list, each node must be freed one by one. The process generally involves:

1. **Storing the Head in a Temporary Pointer**: This allows you to move through the list without losing the reference to the rest of the list.
2. **Updating the Head Pointer**: Move the head pointer to the next node.
3. **Freeing the Previous Node**: Use the free function to deallocate the memory of the node that was just removed from the list.

Example Code for Memory Cleanup:

```
Node* temp;
while (head != NULL) {
   temp = head;       // Store the current head node
   head = head->next; // Move head to the next node
   free(temp);        // Free the memory of the current node
}
```

In this code:

- **Each Node is Freed Individually**: The loop iterates over each node, freeing its memory after moving to the next node in the list.
- **Prevents Memory Leaks**: By freeing each node individually, we ensure that all dynamically allocated memory is returned to the system.

5.6.2 Importance of Memory Management

- **Avoiding Memory Leaks**: Linked lists can consume a lot of memory if nodes are added frequently without proper deallocation of removed nodes.
- **Efficient Memory Use**: Properly freeing memory helps in efficiently managing the system's memory resources, which is especially important in applications that run for extended periods or handle large datasets.

Linked lists are represented in memory as a series of nodes that are dynamically allocated. Each node contains data and a pointer (or pointers) linking it to other nodes, forming a chain. This dynamic structure allows for flexible memory usage but requires careful management to ensure efficient memory allocation and deallocation.

- **Memory Allocation**: Each node is allocated memory separately, which makes resizing flexible but also introduces fragmentation.
- **Pointer-based Structure

5.7 COMPARISON OF LINKED LISTS AND ARRAYS IN MEMORY REPRESENTATION

Arrays and linked lists are both fundamental data structures used in programming to store and manage collections of data. Each has unique properties that make it suitable for specific tasks, and understanding the differences between them can help in selecting the right data structure for various applications. Below is a comprehensive analysis of their differences across key aspects:

5.7.1 Memory Layout

- **Arrays**: Arrays store elements in a contiguous block of memory. This means all elements are stored one after another in a single, continuous memory space. For instance, if an array is declared to hold five integers, the memory allocated will consist of five consecutive integer-sized blocks.
    - **Implication**: Accessing an element by its index in an array is extremely fast, as the position of each element can be directly calculated using the starting address and the index. This characteristic provides constant time, O(1), access to elements by index.
- **Linked Lists**: Linked lists, on the other hand, do not use contiguous memory. Instead, each element, known as a node, is stored separately, and each node contains a pointer to the next (and sometimes the previous) node in the list. Thus, nodes can be scattered throughout memory and do not need to be consecutive.
    - **Implication**: Since nodes are linked by pointers, accessing an element at a particular position requires traversal from the head (or beginning) of the list, making access time proportional to the position in the list (linear time, O(n).

5.7.2 Size Flexibility

- **Arrays**: Arrays have a fixed size, meaning the number of elements they can hold is defined at the time of creation and cannot be changed. For example, in most languages, declaring an array int arr[10] allocates space for exactly 10 integers, and this size cannot be adjusted later.

- **Implication**: The fixed size of arrays can lead to either inefficient use of memory if the allocated size is more than needed or frequent reallocations if the size is underestimated.

- **Linked Lists**: Linked lists are dynamic in nature. They can grow or shrink in size as nodes are added or removed. Each node is allocated memory separately using dynamic memory allocation (e.g., malloc in C).
  - **Implication**: Linked lists provide flexibility for applications where the number of elements is unknown or may vary frequently. Memory usage is efficient as it is allocated only as needed, but this dynamic allocation can have an overhead cost in terms of memory and processing.

5.7.3. Insertion and Deletion Efficiency

- **Arrays**: Inserting or deleting elements in an array can be inefficient, especially if the operation is not at the end. To insert or delete an element in the middle of an array, all subsequent elements must be shifted to make space or fill the gap.
  - **Implication**: Inserting or deleting elements, especially in large arrays, can be costly, with a time complexity of O(n) for these operations in the worst case (when elements are added or removed at the beginning).
- **Linked Lists**: Insertion and deletion operations in linked lists are more efficient, particularly when adding or removing nodes at the beginning or end of the list. This is because each node is linked by pointers, so adding or removing a node only requires updating pointers, not shifting elements.
  - **Implication**: Inserting or deleting a node in a linked list has a time complexity of O(1) if the location is known, making linked lists ideal for applications where frequent insertions or deletions are needed.

5.7.4 Access Time and Random Access

- **Arrays**: Arrays support random access, meaning that any element can be accessed directly using its index in constant time, O(1). This is possible due to the contiguous memory layout, which allows direct calculation of an element's memory address.
  - **Implication**: Arrays are preferable when frequent access to elements by index is needed, such as in cases where the elements must be accessed quickly and in a non-sequential order.
- **Linked Lists**: Linked lists do not support random access. To access a particular element, one must start from the head node and traverse the list until reaching the desired position, resulting in a time complexity of O(n).
  - **Implication**: Linked lists are less efficient for accessing elements by position, making them less suitable for scenarios where fast access to elements by index is required.

5.7.5 Memory Overhead

- **Arrays**: Memory overhead for arrays is minimal. Since all elements are stored in a contiguous memory block, there is no extra storage required for linking elements. The only overhead might be unused space if the allocated array size is larger than the number of elements stored.
  - **Implication**: Arrays are memory-efficient for storing data with a known, fixed size, as they do not require additional memory for pointers.

- **Linked Lists**: Linked lists have a higher memory overhead due to the need for storing pointers in each node. For a singly linked list, each node has one extra pointer for the next node; in a doubly linked list, each node has two extra pointers (one for the next node and one for the previous node).
    - **Implication**: The pointer storage requirement in linked lists results in additional memory overhead, which can be significant, particularly for large lists or lists with small data elements.

5.7.6 Resizing and Memory Allocation

- **Arrays**: Arrays require contiguous memory for all elements, which means resizing can be a challenge. If an array needs to be expanded, a new, larger block of memory must be allocated, and all elements must be copied to this new block. This process can be time-consuming, especially if it needs to be done frequently.
    - **Implication**: Arrays are less flexible for resizing, and resizing may not be feasible in memory-constrained environments due to the requirement for contiguous memory.
- **Linked Lists**: Linked lists do not require resizing because each node is allocated separately. As the list grows or shrinks, memory is allocated or deallocated for each node individually.
    - **Implication**: Linked lists are more adaptable to applications where data is frequently added or removed, and they are better suited to environments with limited contiguous memory availability.

5.7.7 Use Cases and Applications

- **Arrays**: Due to their fast access time and fixed size, arrays are suitable for scenarios where the data size is known and remains constant, or when quick access by index is required. Common applications include:
    - Storing matrices and tables in applications like image processing.
    - Implementing static data collections, such as lookup tables.
    - Using arrays in environments with memory constraints where dynamic allocation is not desired.
- **Linked Lists**: Linked lists are better suited for scenarios where the number of elements is dynamic, or where frequent insertion and deletion of elements are required. Typical applications include:

- Implementing stacks and queues in situations where the collection size may vary.
- Dynamically managing data in applications like playlist management, where items are frequently added or removed.
- Storing sparse data or implementing graph adjacency lists.

Table 5.2.Comparision of Arrays and Linked Lists

Aspect	Arrays	Linked Lists
Memory Layout	Contiguous	Non-contiguous (nodes allocated separately)
Size	Fixed at declaration	Dynamic (grows/shrinks as needed)
Insertion/Deletion	Costly (requires shifting elements)	Efficient (only pointers are adjusted)
Random Access	Yes, constant time $O(1)$	No, requires traversal $O(n)$
Memory Overhead	Minimal (only data storage)	Higher (additional pointers in each node)
Resizing	Difficult (requires contiguous space)	Easy (each node allocated separately)
Applications	Fixed-size data, fast access by index	Dynamic-size data, frequent insertion/deletion

Linked lists are a powerful and flexible data structure, especially valuable when dynamic data management is necessary and memory efficiency in terms of allocated size is a priority. However, they require careful memory management, particularly for operations involving large lists or frequent deletions.

5.8 KEY TERMS

Linked List, Node, Head, Tail, Singly Linked List, Doubly Linked List, Circular Linked List, Memory Fragmentation, Dynamic Memory Allocation

5.9 SELF ASSESSMENT QUESTIONS

1. What is a linked list, and how does it differ from an array?
2. Explain the structure of a node in a linked list.
3. Describe the process of adding a new node at the beginning of a singly linked list.
4. How does a doubly linked list differ from a singly linked list?
5. What are the advantages of using linked lists over arrays in terms of memory management?

5.10 Suggested Readings

1. Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2013). Data Structures and Algorithms in C++. Wiley.
2. Weiss, M. A. (2014). Data Structures and Algorithm Analysis in C. Pearson.
3. Sedgewick, R., & Wayne, K. (2011). Algorithms (4th Edition). Addison-Wesley.
4. Sahni, S. (2005). Data Structures, Algorithms, and Applications in C++. Silicon Press.
5. Horowitz, E., & Sahni, S. (1983). Fundamentals of Data Structures in C. Computer Science Press.

Dr. Vasantha Rudramalla

LESSON - 6

Operations on Linked Lists

OBJECTIVES

The objectives of the lesson are

1. Learn how to represent polynomials using singly and circularly linked lists, including the structure and purpose of each node.
2. Gain insights into adding, deleting, and erasing polynomials dynamically using linked list structures.
3. Explore linked list-based representation of sparse matrices for efficient storage and memory management.
4. Utilize advanced techniques like the avail list for efficient node reuse and dynamic memory allocation.
5. Study key operations such as inverting, concatenating, and inserting nodes in singly and circularly linked lists, and understand their real-world applications.

STRUCTURE

6.1 INTRODUCTION

This lesson delves into the dynamic representation and manipulation of polynomials and sparse matrices using linked lists, an essential data structure in programming. Starting with the fundamentals of polynomial representation using singly linked lists, it explores advanced operations such as addition, deletion, and memory optimization using avail lists. The lesson also covers circular linked lists, highlighting their advantages for continuous traversal, and sparse matrix representations, which save memory and computation time. Through examples and code implementation, learners will grasp the efficiency and flexibility linked lists bring to polynomial and sparse matrix operations, fostering a deeper understanding of memory-efficient algorithms.

6.2 REPRESENTING POLYNOMIALS USING SINGLY LINKED LISTS

In programming, one of the powerful uses of linked lists is for manipulating symbolic polynomials. Here's how we can represent and work with polynomials using singly linked lists in a detailed, easy-to-understand way.

6.2.1 What is a Polynomial?

A polynomial is an expression made up of terms. Each term has:

- A **coefficient** (a numerical factor).
- A **variable** (like x).
- An **exponent** (a power to which the variable is raised).

For example, in the polynomial $3x^4+2x^2+1$, we have three terms:

- $3x^4+2x^2+1$ (with coefficient 3 and exponent 4),
- $2x^2$(with coefficient 2 and exponent 2),
- 1 (with coefficient 1 and exponent 0).

6.2.2 Using Linked Lists to Represent Polynomials

To represent a polynomial, we can use a singly linked list where each node in the list represents a term in the polynomial. This method is flexible and allows efficient memory use.By structuring each term as a node with fields for the coefficient, exponent, and next term pointer, we can:

- Perform polynomial operations dynamically.
- Simplify adding and removing terms for operations like addition, multiplication, and evaluation.

6.2.3 Structure of a Polynomial Node

Each node in the linked list will contain:

1. **Coefficient (coef)**: Stores the numerical factor of the term.
2. **Exponent (exp)**: Stores the exponent of the term.
3. **Link (link)**: Points to the next term in the polynomial.

In C, we can represent a polynomial term (or node) like this:

```
typedef struct poly_node *poly_pointer;
typedef struct poly_node
{
   int coef;          // Coefficient of the term
   int exp;          // Exponent of the term
   poly_pointer link;  // Link to the next term
};
```

Here, poly_pointer is a pointer to a poly_node. Each node is connected to the next node in the list, forming a chain that represents the entire polynomial.

6.2.4 Adding a Node to a Polynomial List

To add a term to the end of a polynomial (or any queue), we can use the following add function:

```
void insert(list_pointer *ptr,
list_pointer node)
{
   list_pointer temp;
  temp =
(list_pointer)malloc(sizeof(list_node));
  if (IS_FULL(temp))
{
      fprintf(stderr, "The memory is
full\n");
      exit(1);
   }
   temp->data = 50;
   if (*ptr) {
```

```
        temp->link = node->link;
        node->link = temp;
    }
    else {
        temp->link = NULL;
        *ptr = temp;
    }
}
```

Here is how the function works:

1. A new node (temp) is allocated memory.
2. If memory allocation fails, an error message is displayed, and the program exits.
3. The new term (node) is then linked to the rear of the list:
    - If the list is empty (front is NULL), front points to this new node.
    - Otherwise, the current rear's link points to temptation.
4. Finally, rear is updated to point to the new node, making it the last element.

6.2.5 Removing a Node from the Front of a Polynomial List

The function for deletion and its working is defined as follows

```
void delete(poly_pointer *ptr)
{
    if (*ptr == NULL) {
        fprintf(stderr, "List is empty\n");
        exit(1);
    }
    poly_pointer temp = *ptr;
    *ptr = (*ptr)->link;
    free(temp);
}
```

1. Check if the list is empty; if so, display an error message and exit.
2. Store the item from the front node, advance the front pointer to the next node, and free the old front node.
3. Return the item stored in the removed node.

6.3 ADDING POLYNOMIALS USING SINGLY LINKED LISTS

Adding two polynomials using singly linked lists is a step-by-step process that allows us to handle polynomials of any size and degree. We start by traversing both polynomials term by term and creating a new polynomial as the result. The process of addition has the following steps.

6.3.1 Start at the Beginning of Both Polynomials

Each polynomial is represented by a linked list where each node contains:

- coef (the coefficient of the term)
- exp (the exponent of the term)
- link (a pointer to the next term)

For representation, we have to store the data about that polynomial. That data can be stored either in an array or a linked list. So, we have already seen array representation. Now we will see how to represent the data related to polynomials. If we observe the below polynomial, each term is having its coefficient and exponent.

If we know the value of x then the exponent of x can be raised and then the coefficient can be multiplied. So, in this way, we can get the answer for one term. Likewise, we can evaluate all the terms and get the result. It is sufficient to store the coefficient and exponent of each term. We can represent these terms in the form of a linked list. So, we can define each term as a node and we will have a set of nodes for a single polynomial. So let us define a node for a single term.

Here is the node that is having a coefficient, an exponent, and a pointer (next) to the next node. Let us define the structure for this.

```
Struct Node
{ int coefficient;
 int exponent;
struct node *next; }
```

This structure has a coefficient and exponent of type int and a pointer to the next node of type Node*. If the coefficient is in decimal, then you can take float also. Now one more thing you can observe, this is the node of the linked list and it is having 3 members. So, the linked list that we have studied was taking only one value but now we are using a linked list. Based on the requirements a node can have any number of data members. Now, let us represent the polynomial as a linked list.

Below figure is an example of a polynomial representation.

P(x) = 4x3 + 9x2 + 6x + 7

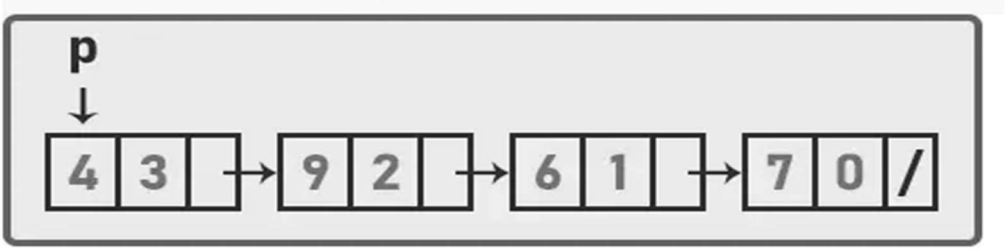


Fig 6.1. Polynomial Representation

6.3.2 Compare Terms Based on Exponents

As we traverse the polynomials, we compare the exponents of the current terms from each polynomial:

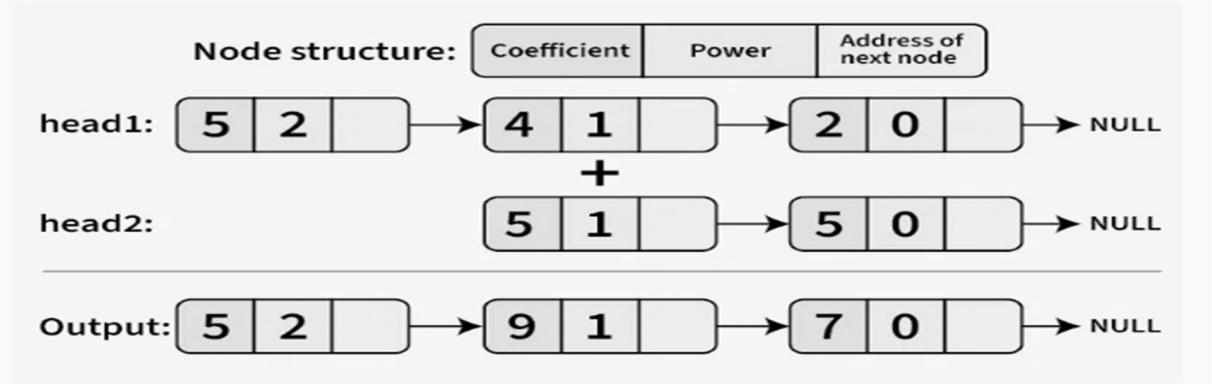


Fig 6.2. Polynomial Addition

This diagram shows the process of adding two polynomials using linked list representation. Each node in the linked list represents a term in the polynomial with three fields:

- **Coefficient**: The numerical multiplier of the term.
- **Power** (or exponent): The power to which the variable xxx is raised.
- **Address of next node**: A pointer to the next term in the polynomial.

The structure of the polynomials is represented as

1. **Polynomial 1 (head1)**:

   - $5x^2$: Coefficient = 5, Power = 2
   - $4x^1$: Coefficient = 4, Power = 1
   - $2x^0$: Coefficient = 2, Power = 0

   This polynomial is represented by a linked list with three nodes, where each node stores one term of the polynomial. The NULL pointer indicates the end of the list.

2. **Polynomial 2 (head2)**:

   - $5x^1$: Coefficient = 5, Power = 1
   - $5x^0$: Coefficient = 5, Power = 0

   This polynomial also has a linked list representation with two nodes.

6.3.3 Process of Addition

To add these polynomials, we compare the exponents of each term in both linked lists and combine terms with matching exponents by adding their coefficients. Here's how the addition proceeds:

❖ **Compare terms with highest powers**:

- The first term in head1 is $5x^2$, and the first term in head2 is $5x^{1.}$
- Since the exponent 2 (from head1) is greater than 1 (from head2), we add $5x^2$directly to the output polynomial.

- **Move to the next term in head1 and compare again**:

  - Now we compare $4x^1$ (from head1) with $5x^1$ (from head2).
  - Since both terms have the same exponent (1), we add their coefficients: 4+5=9.
  - So, we create a new term $9x^1$ and add it to the output polynomial.

- **Move to the next term in both lists**:

  - Now we compare $2x^0$ (from head1) with $5x^0$ (from head2).
  - Both terms have the same exponent (0), so we add their coefficients: 2+5=7.
  - We create a new term $7x^0$ and add it to the output polynomial.

6.3.4. Resulting Polynomial

- The resulting polynomial, represented by the Output linked list, is: $5x^2 + 9x^1 + 7x$

This final output is also represented as a linked list where each term is a node, and the NULL pointer at the end indicates the end of the polynomial.

This diagram demonstrates how polynomials can be added term-by-term by traversing the linked lists, matching exponents, and combining coefficients where necessary. This approach is efficient for dynamically managing polynomials with various numbers of terms and exponents.

6.4 ERASING POLYNOMIALS

In polynomial operations using linked lists, memory management becomes essential, especially when dealing with temporary polynomials. Since linked lists use dynamic memory allocation, each term in a polynomial is stored in a separate node. When a polynomial is no longer needed (e.g., a temporary result), we can "erase" it to free up memory for other uses.

In the example, suppose a user wants to:

1. Read in three polynomials: a(x), b(x), and d(x).
2. Compute a result polynomial e(x) using the expression:

$$e(x) = a(x) * b(x) + d(x)$$

This operation involves:

- **Multiplying** polynomials a(x) and b(x) to create a temporary result, temp.
- **Adding** the result of the multiplication (temp) to d(x) to produce e(x).

Once the operation is complete, the temporary polynomial temp is no longer needed. To conserve memory, it's a good idea to erase temp by freeing its nodes, allowing the memory to be reused.

```
void erase(poly_pointer *poly)
 {
```

```
    poly_pointer temp;
       while (*poly != NULL)
        {
       temp = *poly;
       *poly = (*poly)->link;
       free(temp);  // Free the current node
         }
}
```

The erase function will traverse each node in the linked list representing the polynomial and free the memory allocated for each node. This helps in managing memory efficiently, especially in cases where we work with multiple temporary polynomials.

6.5 Polynomials as Circularly Linked Lists

The above diagram shows a circular linked list representation of a polynomial. In a circular linked list, the last node points back to the first node, forming a circular structure. Each node in the list represents a term in the polynomial, containing:

- A **coefficient** field (e.g., 3, 2, 1).
- An **exponent** field (e.g., 14, 8, 0).
- A **link** to the next node in the list.

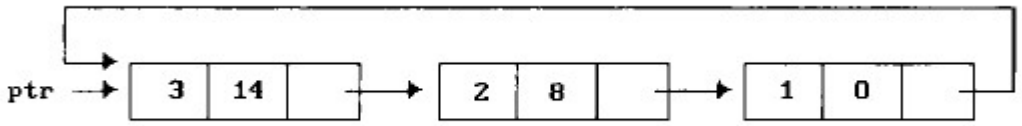


Fig 6.3. Circular Representation of ptr = 3x14 + 2x8 + 1

In this specific example:

1. The polynomial represented here is $3x^{14} + 2x^{8} + 1$
2. The ptr pointer points to the first node in the circular linked list.

Structure of Each Node in the Diagram:

1. **First Node**:
    - Coefficient: 3
    - Exponent: 14
    - Link: Points to the second node.
2. **Second Node**:
    - Coefficient: 2
    - Exponent: 8
    - Link: Points to the third node.

3. **Third Node**:

    - Coefficient: 1
    - Exponent: 0
    - Link: Points back to the first node, making the list circular.

This circular structure is useful for situations where continuous traversal of the list is needed, as it allows the list to loop back to the beginning without requiring a separate check for the end.

6.5.1 Managing Memory with an Available Space List (Avail List)

In circular linked lists, memory management is crucial for efficient operations, especially when nodes are frequently added and removed. To optimize this process, an available space list (avail list) is utilized. This technique reuses freed nodes instead of allocating new memory every time, reducing overhead and improving performance. Below is an explanation of the key concepts involved in managing memory with an avail list:

1. **Available Space List (avail list)**

    - Instead of calling malloc every time a new node is needed, a list of "freed" nodes (called the avail list) is maintained.
    - When a node is freed (e.g., during an erase operation), it is added to this avail list rather than being completely removed from memory.
    - When a new node is required, the program first checks if there are any nodes available in the avail list:

        - If available nodes exist, one is reused.
        - If the list is empty, malloc is used to create a new node.

2. **avail Pointer**

    - avail is a pointer to the first node in the avail list.
    - Initially, avail is set to NULL, indicating that no freed nodes are available.
    - Over time, as nodes are freed, they are added to the avail list, so avail points to the start of this chain of reusable nodes.

3. **get_node Function**

The get_node function provides a node for use. It works as follows:

- If avail is not NULL (meaning there are freed nodes available), it retrieves a node from the avail list, sets avail to point to the next node in the list, and returns the retrieved node.
- If avail is NULL (meaning no freed nodes are available), it allocates a new node using malloc.
- If memory allocation fails, it prints an error message and exits the program.

Here is the code for get_node:

```
poly_pointer get_node(void)
 {
   poly_pointer node;
   if (avail) {
      node = avail;
      avail = avail->link;
   } else {
      node = (poly_pointer)malloc(sizeof(poly_node));
      if (node == NULL) {
         fprintf(stderr, "The memory is full\n");
         exit(1);
      }
   }
   return node;  }
```

4. ret_node Function

The ret_node function returns a node to the avail list instead of freeing it completely. This makes it available for reuse.

Here's how ret_node works:

- It takes a node pointer (ptr) that needs to be freed.
- Sets ptr->link to point to the current start of the avail list (avail).
- Updates avail to point to ptr, effectively adding this node to the avail list.

```
void ret_node(poly_pointer ptr)
{
   ptr->link = avail;
   avail = ptr;
}
```

6.6 Representing Polynomials with a Head Node

1. **Zero Polynomial**:
   - The zero polynomial contains only the head node.
   - The coef and expon fields in this node are irrelevant.

Zero polynomial

2. **Non-Zero Polynomial**:
   - For example, the polynomial $a(x) = x^2 + 2x + 1$ would have nodes for each term, plus the head node at the start.

By adding a head node, we ensure that each polynomial (whether zero or non-zero) has a consistent structure, making it easier to handle in algorithms.

Using an avail list and head nodes in circular linked lists offers an efficient and manageable way to handle polynomial operations:

- **Avail List**: Recycles nodes to reduce memory allocation overhead.
- **get_node and ret_node**: Functions to manage the avail list and simplify node allocation and deallocation.
- **Head Node**: Provides a uniform structure for polynomials, avoiding special cases for zero polynomials.

This approach optimizes memory usage and improves the efficiency of polynomial operations by making node management more streamlined and reducing dependency on malloc and free for each node.

Summary of Polynomial Representation

Polynomial operations benefit greatly from using linked list structures, particularly singly and circularly linked lists. These structures support dynamic memory allocation, efficient traversal, and easy manipulation of terms. Using linked lists for polynomials also aids in memory management by employing an available space list. This list holds nodes that are freed and can be reused in future operations. By implementing these techniques, we achieve optimal time complexity for polynomial addition and other operations, ensuring that the code performs efficiently within the constraints of available memory

6.7 ADDITIONAL LIST OPERATIONS

Operations on lists, particularly linked lists, form a cornerstone of efficient data manipulation in computer science. These operations allow for dynamic memory allocation, streamlined data management, and enhanced performance in various applications. Whether dealing with singly linked lists or circular linked lists, the ability to add, remove, reverse, and concatenate nodes provides flexibility in handling complex data structures.

By implementing utility functions like get_node and ret_node, memory usage is optimized through dynamic node reuse. Advanced techniques such as reversing a list or concatenating multiple lists further enable developers to modify and manage data structures with minimal overhead. Circular linked lists offer additional advantages, including seamless traversal and efficient insertion operations at any point in the structure.

This section explores these fundamental operations, highlighting their algorithms, practical implementations, and efficiency considerations to demonstrate how linked lists can be effectively utilized for real-world data management and processing tasks.

6.7.1 Operations on Chains

When working with singly linked lists, it is often necessary to implement various utility functions to manipulate and manage these lists efficiently. Two important functions are:

1. **get_node and ret_node:** These manage memory allocation and deallocation by interacting with the available space list (avail list). Instead of repeatedly calling malloc and free, they retrieve and return nodes, improving memory efficiency.
2. **Inverting (Reversing) a Singly Linked List:** This function reverses the order of nodes in a singly linked list "in place" using three pointers:

   - previous: Tracks the previous node.
   - current: Tracks the node being processed.
   - next: Temporarily stores the next node in the original list.

   By iterating through the list and adjusting these pointers, the links between nodes are reversed without requiring additional memory. The operation is efficient with a time complexity of O(length), where length is the number of nodes in the list.

   Testing this function with:

   - An empty list ensures no errors occur when the list has no nodes.
   - A list with one node validates that the function correctly handles a trivial case.
   - A list with two or more nodes demonstrates the functionality of the loop and the correctness of the reversed structure.

3. **Concatenating Two Linked Lists:** This function joins two singly linked lists, ptr1 and ptr2, by linking the last node of ptr1 to the first node of ptr2. The operation has a time complexity of O(length of ptr1), as it requires traversing ptr1 to locate its last node. This approach is efficient because it does not require additional memory, and the concatenated list remains part of ptr1.

   Another variation of this function can concatenate two lists without altering ptr1, which is useful when the original list must remain unchanged.

6.7.2 Operations for Circular Linked Lists

Efficiently managing and manipulating linked lists, both singly and circular, requires key operations such as reversing, concatenating, and inserting nodes. The operations listed below highlight techniques for handling these tasks effectively in various scenarios.

❖ **Inverting (Reversing) a Linked List**

Reversing a linked list is a common and useful operation. This function, known as invert, reverses the order of nodes in the list "in place" (without using additional memory), by re-linking the nodes in reverse order.

To reverse the list in place, three pointers are used:

- **previous**: Keeps track of the previous node.
- **current**: Points to the current node being processed.
- **next**: Stores the next node, allowing us to re-link the current node without losing track of the rest of the list.

The process is as follows:

1. Set previous to NULL (since the new end of the list should point to NULL).
2. Set current to the head of the list.

3. While current is not NULL:
    - Set next to current->link (store the next node).
    - Update current->link to point to previous (reverse the link).
    - Move previous to current and current to next.
4. When current is NULL, previous points to the new head of the reversed list.

The time complexity of the invert function is O(length)**,** where length is the number of nodes in the list. This makes the function efficient for lists of any size.

Testing invert with Examples:

1. **Empty list**: The function should return NULL (no change).
2. **Single-node list**: The function should return the list unchanged, as there's nothing to reverse.
3. **Two-node list**: The function should swap the two nodes, effectively reversing the list.

❖ **Concatenating Two Lists**

Another useful function is concatenation, where two lists are joined together to form a single list. Suppose we have two lists, ptr1 and ptr2:

- The function finds the end of ptr1 and updates its last node to point to the head of ptr2, linking the two lists together.
- This operation does not require additional storage, as ptr1 now contains the concatenated list.

The complexity of this function is O(length of list ptr1)**,** as it involves traversing ptr1 to reach its last node.

The functions described here are essential for managing singly linked lists:

- **get_node and ret_node**: Manage memory efficiently by reusing nodes.
- **Inverting a list**: Reverses the list in place using three pointers, with a time complexity of O(length).
- **Concatenating two lists**: Links the end of one list to the start of another, with a time complexity of O(length of the first list).

These functions are foundational for working with linked lists and offer efficient ways to manipulate list structures in various applications.

❖ **Insertion in Circular Linked List**

We can insert a new node in three common places:

1. At the beginning
2. At the end
3. After a specific node

1. Inserting at the Front

When inserting at the front of a circular linked list:

- We create a new node with the data we want to insert.
- If the list is empty (no nodes), we simply set this new node to point to itself, forming a single-node circular structure, and make it the head of the list.
- If the list is not empty, we locate the last node (the node that points back to the head).
- We then adjust the pointers:
    - The new node's next pointer is set to the current head, making it the first node in the list.
    - The last node's next pointer is updated to point to this new node, keeping the list circular.
- Finally, we update the head pointer to the new node, making it the new front of the list.

This operation ensures that the new node is added at the beginning, while maintaining the circular structure of the list.

2. Inserting at the End

To insert a new node at the end of a circular linked list:

1. We create a new node with the data we want to add.
2. If the list is empty, we make this new node point to itself and set it as the head.
3. If the list is not empty, we locate the last node by traversing the list (the last node points back to the head).
4. Once we have the last node, we update the pointers:
    - The last node's next pointer is set to point to the new node.
    - The new node's next pointer is set to the head, maintaining the circular link back to the start of the list.
5. The head remains the same since we're only adding a node at the end.

This operation allows us to append nodes to the list without disturbing the circular connection.

❖ **Counting the Number of Nodes**

To count the nodes in a circular linked list:

1. We start from the head and begin a count at 1 (assuming the head exists).
2. We then traverse the list, moving from node to node, and incrementing the count for each node encountered.
3. We stop the traversal once we reach the head again, which indicates we've gone around the entire circle.
4. If the list is empty, the count is 0.

This method gives us an accurate count of all the nodes in the circular linked list.

6.8 EQUIVALENCE RELATIONS

Definition and Properties:

- An equivalence relation is defined by three properties: reflexivity, symmetry, and transitivity.
- Reflexive: Every element is related to itself (e.g., x = x).
- Symmetric: If x is related to y, then y is also related to x.
- Transitive: If x is related to y and y is related to z, then x is related to z.

Example: "Equal to" Relation ===

The equal to relation = is an equivalence relation because:

1. **Reflexivity**: For any x, x=x.
2. **Symmetry**: If x=y, then y=x.
3. **Transitivity**: If x=y and y=z then x=z.

Thus, the relation = satisfies all three properties of equivalence relations.

6.8.1 Equivalence Classes:

- An equivalence relation can be used to partition a set into equivalence classes.
- For a set S, if two elements x and y in S are equivalent, they belong to the same equivalence class.
- Example: If polygons in a VLSI design overlap and are electrically equivalent, they form distinct equivalence classes.

6.8.2 Application in VLSI Design:

Equivalence relations are applied in VLSI circuit design to group electrically equivalent polygons, forming a critical part of the design process. Overlapping polygons grouped into equivalence classes are tested for correctness in mask design.

Algorithm to Determine Equivalence Classes:

The algorithm involves two phases:

Phase 1: **Collecting Equivalence Pairs**

- In this phase, we gather pairs of elements that define the equivalence relation. Each pair represents two elements that are related (i.e., they satisfy the equivalence criteria such as overlap in VLSI design).
- We store these pairs in a way that allows us to efficiently retrieve and group related elements later. Arrays or linked lists are commonly used here to keep track of these relationships

Phase 2: **Forming Equivalence Classes**

- In this phase, we process the collected pairs to identify complete equivalence classes.
- Starting with each element, we find all other elements it is related to, directly or indirectly (using reflexivity, symmetry, and transitivity).
- As elements are processed, they are grouped into equivalence classes—each class containing all elements equivalent to each other.
- The result is a set of disjoint classes, where every element in each class is related under the equivalence relation, making it easy to verify or manipulate these groups in applications like VLSI design.

These phases work together to systematically and efficiently partition the elements into distinct, non-overlapping equivalence classes

6.9 SPARSE MATRICES

Representing a sparse matrix with a conventional 2D array often leads to wasted memory since we end up storing many zeros. Instead, the sparse matrix representation in an array format only keeps track of non-zero values, stored as "triplets." Each triplet includes three components:

- **Row** : The row number where the non-zero element is located.
- **Col**: The column number where the non-zero element is located.
- **Value**: The actual value of the non-zero element.

This approach uses only three fields, thus saving space compared to a standard array.

6.9.1 Linked List Representation of Sparse Matrix

In a linked list representation, each non-zero element is represented as a node in a linked list. This format includes four fields for each node:

- **Row**: The row number where the non-zero element is located.
- **Column**: The column number where the non-zero element is located.
- **Value**: The non-zero element value itself.
- **Next Node Pointer**: A pointer to the next node, which contains the next non-zero element.

The linked list representation is beneficial for matrices that undergo frequent changes (such as adding or removing elements), as linked lists provide dynamic storage without the need for reallocation as array sizes change.

Example of Linked List Representation

Consider a 4x4 sparse matrix:

$$\begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 3 & 0 & 0 & 0 \\ 0 & 4 & 0 & 2 \end{bmatrix}$$

The non-zero elements in the above matrix can be stored in a linked list format as follows:

Each node points to the next non-zero element, with the last node pointing to NULL to signify the end of the list. In this example, we store only non-zero values, avoiding memory waste, and each node dynamically links to the next, allowing easy addition or deletion of elements.

6.9.2 Advantages of Linked Representation

1. **Dynamic Size Management:**
   - Linked lists can grow or shrink dynamically, accommodating changes in the number of nonzero terms.
2. **Efficient Storage:**
   - Reduces memory usage by storing only nonzero terms and their positions.
3. **Optimized Traversal:**
   - Dual-linking allows efficient row and column traversal, especially useful for operations like matrix multiplication.
4. **Better Performance for Large Matrices:**
   - Asymptotic complexity is better than storing matrices in a 2D array, which requires **O(rows × columns)** space and time for many operations.

6.9.3 Implementation in C

The linked list representation for sparse matrices uses unions and structured memory allocations. Key operations include:

1. **Matrix Construction (mread):**

   - The matrix is constructed by reading input values (rows, columns, values) in order.
   - Each nonzero term is inserted into the corresponding row and column lists.
   - Efficiency:

     - Setting up head nodes takes O(max(rows, columns))**.**
     - Adding nonzero terms takes O(num-terms).
     - Overall complexity: O(max(rows, columns) + num-terms**)**.

2. **Matrix Display (mwrite):**

   - The matrix is displayed row by row.
   - For each row, all nonzero terms are traversed and printed.
   - Efficiency:

     - Outer loop: num-rows iterations.
     - Inner loop: Iterates over nonzero terms in each row.

- Overall complexity: O(num-rows + num-terms)**.**

3. **Matrix Deletion (merase):**
   - Deletes all nodes of the matrix, freeing allocated memory:
     - Frees entry nodes and row head nodes.
     - Finally, frees the remaining head nodes.
   - Efficiency:
     - Erasing entry nodes: O(num-rows + num-terms)**.**
     - Erasing head nodes: O(num-rows + num-cols)**.**
     - Overall complexity: O(num-rows + num-cols + num-terms)**.**

Efficiency Analysis

- Matrix Construction (mread): Faster than creating a 2D array, with a complexity of O(max(rows, columns) + num-terms)**.**
- Matrix Display (mwrite): Proportional to the number of rows and nonzero terms, complexity O(num-rows + num-terms)**.**
- Matrix Deletion (merase): Frees memory efficiently, complexity O (num-rows + num-cols + num-terms**)**.

By using linked lists, we achieve efficient memory management and improved computational performance, especially for sparse matrices with varying nonzero terms. This method is particularly effective in scenarios requiring frequent dynamic updates or operations on large matrices.

6.9.4 Summary of Benefits of Sparse Matrices

Using sparse matrices, especially in applications with large datasets that include many zeros, is advantageous due to:

- **Efficient Storage**: Sparse matrices reduce memory usage by storing only essential non-zero elements.
- **Optimized Computation**: By focusing only on non-zero values, sparse matrices save time during computational operations like searching and traversing.
- **Flexibility with Linked List Representation**: Linked lists allow efficient insertion and deletion of elements, particularly beneficial for applications that require frequent updates.

Overall, sparse matrix representations are valuable in fields requiring large-scale data management with predominantly zero values, such as scientific computing, engineering simulations, and data mining applications.

6.10 KEY TERMS

Inversion, Concatenation, Circular List, Insertion, Counting, Nodes, get_node, ret_node

6.11 SELF ASSESSMENT QUESTIONS

1. Define a **polynomial** and list its components.
2. Describe how a **singly linked list** is used to represent a polynomial. What information does each node contain?
3. Explain the process of **adding two polynomials** using singly linked lists. How are exponents and coefficients managed during this operation?
4. What are the advantages of using a **circularly linked list** over a singly linked list for polynomial representation?
5. Describe the purpose and implementation of an **avail list** in managing linked list memory allocation.

6.12 SUGGESTED READINGS

1. **"Introduction to Algorithms" by Cormen, Leiserson, Rivest, and Stein**

   A foundational text that discusses linked lists, their applications, and memory management techniques.

2. **"Data Structures and Algorithm Analysis in C" by Mark Allen Weiss**

   Covers linked lists and their use in dynamic data representations, including polynomials.

Dr. Vasantha Rudramalla

LESSON - 7

Evaluation of Expressions

OBJECTIVE

The objectives of the lesson are:

1. Understand the concept of expression evaluation and its significance in programming.
2. Explore different types of expressions (infix, prefix, and postfix) and their unique structures.
3. Examine operator precedence, associativity, and the role of parentheses in controlling evaluation order.
4. Learn efficient postfix expression evaluation using stacks and its advantages in compiler design.
5. Implement algorithms for postfix evaluation and infix-to-postfix conversion in C programming.

STRUCTURE

7.1 INTRODUCTION

In programming, the evaluation of expressions is the process of computing a result from a combination of values, variables, and operators. Expressions are fundamental in programming, acting like formulas that the computer interprets and calculates. For example, in mathematics, you might encounter an expression like a + b × c, similarly, in programming, an expression could look like `result = (a + b) * c`. Here, `a`, `b`, and `c` represent variables, while `+` and `*` are operators that dictate the operations to perform. Understanding how these operations are prioritized is essential, as different operators have different precedence levels, determining the sequence in which they are evaluated. Additionally, associativity rules help decide the order of evaluation for operators of the same precedence level. With the right understanding of operator precedence, associativity, and expression structures (infix, prefix, and postfix), programmers can write and evaluate expressions accurately and efficiently, optimizing performance and minimizing errors in code execution.

7.2 OPERATOR PRECEDENCE

Each operator in C (like +, -, *, /) has a priority level, known as precedence. The precedence tells the computer which operators to evaluate first when they appear together in an expression. Operators with higher precedence are evaluated before operators with lower precedence.

For example, in the expression a + b * c, multiplication (*) has higher precedence than addition (+). So, the computer will calculate b * c first and then add a to the result.

This is how a + b * c is evaluated:

1. First, b * c is calculated.
2. Then, a is added to the result of b * c.

The above context is represented using the below table.

Table 7.1. Example of Operator Precedence

Expression: `a + b * c`

Values: `a = 2`, `b = 3`, `c = 4`

Step	Calculation	Result
Step 1: `b * c`	`3 * 4`	12
Step 2: `a + (b * c)`	`2 + 12`	14
Final Result		14

If we wanted a + b to be calculated first, we could use parentheses to change the order as (a + b) * c;

Table 7.2. Example with Parentheses to Control Order

Expression: `(a + b) * c`

Values: `a = 2`, `b = 3`, `c = 4`

Step	Calculation	Result
Step 1: `a + b`	`2 + 3`	5
Step 2: `(a + b) * c`	`5 * 4`	20
Final Result		20

7.3 ASSOCIATIVITY

Associativity determines the order in which operators of the same precedence level are evaluated. Operators can be left-associative or right-associative.

1. **Left-Associative**: Operators are evaluated from left to right. Most binary operators (like +, -, *, /) are left-associative. This means if you have an expression like a - b + c, it's evaluated from left to right:
   - a - b is calculated first.
   - Then, the result is added to c.
2. **Right-Associative**: Operators are evaluated from right to left. The assignment operator (=) is an example of a right-associative operator. In an expression like a = b = c, the computer assigns c to b first, and then b to a.

7.4 USING PARENTHESES TO CONTROL EVALUATION ORDER

In programming, parentheses are used to explicitly define the order in which parts of an expression should be evaluated. By default, operators have their own precedence levels, dictating the order of operations in an expression. However, when we use parentheses, they override these default precedence rules, ensuring that the enclosed operations are evaluated first, regardless of operator precedence.

For example, consider the expression `a + b * c`. By precedence rules, multiplication has a higher priority than addition, so `b * c` would be calculated first, followed by adding `a`. To change this order and perform addition first, we can use parentheses: `(a + b) * c`. In this modified expression, `(a + b)` is calculated first, and the result is then multiplied by `c`.

Using parentheses is essential for clarity and correctness in complex expressions, as it avoids ambiguity, especially when multiple operators are involved. This makes parentheses an invaluable tool for controlling evaluation order and ensuring the intended result.

7.5 TYPES OF EXPRESSIONS AND THEIR DIFFERENCES

In programming, we encounter different types of expressions, and the way they're written affects how they're evaluated. The three main types of expressions are Infix, Prefix and Postfix. Each type has a unique structure, which changes how the operators and operands are arranged. Here's a detailed look at each type:

7.5.1 Infix Expression

Infix expressions are the most common in everyday mathematics and programming languages like C. In an infix expression, the operator is placed between the operands. For example, in the expression a + b, a and b are operands, and + is the operator.

Example: a + b * c

- Evaluation Steps:
  - b * c is evaluated first (as * has higher precedence than +).
  - Then, a is added to the result of b * c.

Characteristics:

- **Easy for Humans**: Infix is straightforward to read because it's similar to how we write math expressions.
- **Requires Precedence Rules**: To evaluate infix expressions, the computer needs rules for operator precedence and associativity to decide the order of operations.
- **Parentheses Control Order**: Parentheses can change the order in which operations are performed.

Example with Parentheses:

If a = 2, b = 3, and c = 4:

With parentheses: (a + b) * c; // (2 + 3) * 4 = 5 * 4 = 20

Without parentheses: a + b * c would give a different result:

Result = a + b * c; // 2 + (3 * 4) = 2 + 12 = 14

Table 7.3. Infix Expression Evaluation (No Parentheses vs. Parentheses)

Expression Type	Expression	Calculation	Result
Infix (no parentheses)	a + b * c	2 + (3 * 4)	14
Infix with parentheses	(a + b) * c	(2 + 3) * 4	20

7.5.2 Prefix Expression (Polish Notation)

In prefix expressions, also known as **Polish notation**, the operator appears before the operands. This type of expression removes the need for parentheses because the order of operations is unambiguously determined by the placement of the operators.

Example:

- Prefix: + a * b c
- Equivalent Infix: a + (b * c)

Step-by-Step Evaluation:

1. In the expression + a * b c, the * operator applies to b and c first.
2. Then, the + operator combines a with the result of b * c.

Characteristics:

- **Less Readable for Humans**: It's not intuitive since we don't normally place operators first in expressions.
- **No Need for Parentheses**: The position of each operator and operand defines the order of operations.
- **Common in Specific Calculators**: Certain calculators and computer systems use prefix notation because of its unambiguous structure.

Step-by-Step Example: Given a = 2, b = 3, and c = 4:

- Prefix expression: + 2 * 3 4
- Steps:
    1. Calculate * 3 4 = 3 * 4 = 12
    2. Apply + 2 12 = 2 + 12 = 14
- Final result: 14

Table 7.4. Prefix Expression (Polish Notation)

Step	Calculation	Result
Step 1: `* b c`	`3 * 4`	12
Step 2: `+ a (b * c)`	`2 + 12`	14
Final Result		14

7.5.3 Postfix Expression (Reverse Polish Notation)

In postfix expressions, also known as **Reverse Polish Notation (RPN)**, the operator comes after the operands. Postfix expressions are easier for computers to evaluate because they don't require precedence rules or parentheses.

Example:

- Postfix: a b c * +
- Equivalent Infix: a + (b * c)

Evaluation Process:

1. In postfix, operands are processed first, then the operator.
2. For the expression a b c * +, b and c are multiplied first.
3. The result is then added to a.

Characteristics:

- **Efficient for Computers**: Postfix expressions are processed easily using stacks, making them ideal for compilers and calculators.
- **No Parentheses Needed**: The structure of the expression dictates the order.
- **Common in Compilers**: Postfix notation is commonly used in compilers because it streamlines the evaluation process.

Example Calculation: Given a = 2, b = 3, and c = 4:

- Postfix expression: 2 3 4 * +
- Steps:
    1. Multiply 3 and 4 to get 12.
    2. Add 2 to 12 to get 14.
- Final result: 14

Table 7.5. Postfix Expression (Reverse Polish Notation)

Expression: `a b c * +`

Values: `a = 2`, `b = 3`, `c = 4`

Step	Calculation	Result
Step 1: `b * c`	`3 * 4`	12
Step 2: `a + (b * c)`	`2 + 12`	14
Final Result		**14**

The differences between the three types of expressions are depicted in the below table.

Table 7.6. Summary of Differences

Type	Structure	Example	Readability	Evaluation Complexity
Infix	Operator between operands	`a + b`	Most readable	Requires precedence rules
Prefix	Operator before operands	`+ a b`	Less readable	Fixed order, no precedence needed
Postfix	Operator after operands	`a b +`	Unfamiliar	Easy for computer evaluation

In summary, infix notation is most familiar to us, but postfix is the most efficient for evaluation because it avoids ambiguity, making it preferred by compilers.

7.6 EVALUATING POSTFIX EXPRESSIONS

Now that we know what a postfix expression is, let's dive into how to evaluate it step-by-step. Postfix expressions are particularly convenient for evaluation using a **stack data structure**. A stack allows us to temporarily store operands (numbers) until we encounter an operator, at which point we can retrieve the needed operands from the stack to perform the calculation.

7.6.1 Steps for Evaluating a Postfix Expression

Evaluating a postfix expression is straightforward with a stack, following these steps:

1. **Process Each Element**: Start by reading each element in the expression from left to right. Each element can be an operand (number) or an operator (+, -, *, /).
2. **Push Operands onto the Stack**: Whenever you encounter an operand, push it onto the stack. This will store the operand for future operations.
3. **Pop and Evaluate for Operators**: When you reach an operator, pop the appropriate number of operands from the stack. For a binary operator (like +, -, *, /), pop the top two operands, apply the operator, and push the result back onto the stack.
4. **Final Result**: Once you reach the end of the expression, the final result will be the only value left on the stack. This value represents the evaluated result of the entire postfix expression.

Example 1 : Evaluating 6 2 / 3 - 4 2 * +

Let's evaluate the postfix expression 6 2 / 3 - 4 2 * + step-by-step. We'll use a stack to keep track of operands and intermediate results. Here's a breakdown of each action as we process the expression from left to right.

1. Initialize the Stack

Begin with an empty stack.

2. Process Each Token

Each element in 6 2 / 3 - 4 2 * + is traversed and evaluated using the stack.

1. **Token 6**:
   - 6 is an operand, so we push it onto the stack.
   - **Stack**: [6]
2. **Token 2**:
   - 2 is also an operand, so we push it onto the stack.
   - **Stack**: [6, 2]
3. **Token /**:
   - / is an operator. We need to pop the top two operands from the stack (6 and 2), divide them, and push the result back onto the stack.
   - Calculation: 6 / 2 = 3
   - **Stack**: [3]
4. **Token 3**:
   - 3 is an operand, so we push it onto the stack.
   - **Stack**: [3, 3]
5. **Token -**:
   - - is an operator. We pop the top two values (3 and 3), subtract the second operand from the first, and push the result.
   - Calculation: 3 - 3 = 0
   - **Stack**: [0]
6. **Token 4**:
   - 4 is an operand, so we push it onto the stack.
   - **Stack**: [0, 4]
7. **Token 2**:
   - 2 is also an operand, so we push it onto the stack.
   - **Stack**: [0, 4, 2]

8. **Token *:**
    - * is an operator. We pop the top two values (4 and 2), multiply them, and push the result.
    - Calculation: 4 * 2 = 8
    - **Stack**: [0, 8]
9. **Token +:**
    - + is an operator. We pop the top two values (0 and 8), add them, and push the result.
    - Calculation: 0 + 8 = 8
    - **Stack**: [8]

After processing all tokens in the expression, we have a single value on the stack, which is the result of the entire expression.

Final Stack: [8]

So, the final result of evaluating the expression 6 2 / 3 - 4 2 * + is 8.

Table 7.6: Step-by-Step Postfix Evaluation

Expression: `6 2 / 3 - 4 2 * +`

Step	Token	Operation/Action	Stack
1	`6`	Push `6`	`[6]`
2	`2`	Push `2`	`[6, 2]`
3	`/`	Pop `6` and `2`, calculate `6 / 2 = 3`, push `3`	`[3]`
4	`3`	Push `3`	`[3, 3]`
5	`-`	Pop `3` and `3`, calculate `3 - 3 = 0`, push `0`	`[0]`
6	`4`	Push `4`	`[0, 4]`
7	`2`	Push `2`	`[0, 4, 2]`
8	`*`	Pop `4` and `2`, calculate `4 * 2 = 8`, push `8`	`[0, 8]`
9	`+`	Pop `0` and `8`, calculate `0 + 8 = 8`, push `8`	`[8]`
Final Result			`[8]`

The explanation for the above table is given below

1. Start with an Empty Stack

- **Stack**: `[]`

  `(empty)`

2. Push 6

- **Stack**: `[6]`

```
|  6  |
------
```

3. Push 2

- **Stack**: `[6, 2]`

```
| 6 |
-----
| 2 |
-----
```

4. After / (Divide 6 by 2)

- **Operation**: `6 / 2 = 3`
- **Stack**: `[3]`

```
| 3 |
-----
```

5. Push 3

- **Stack**: `[3, 3]`

```
|  3  |
------
|  3  |
------
```

6. After - (Subtract 3 from 3)

- **Operation**: `3 - 3 = 0`
- **Stack**: `[0]`

```
|  0  |
------
```

7. Push 4

- **Stack**: `[0, 4]`

```
|  0  |
------
|  4  |
------
```

8. Push 2

- **Stack**: `[0, 4, 2]`

```
|  0  |
------
|  4  |
```

```
------
|  2  |
------
```

9. After * (Multiply 4 by 2)

- **Operation**: `4 * 2 = 8`
- **Stack**: `[0, 8]`

```
|  0  |
------
|  8  |
------
```

10. After + (Add 0 and 8)

- **Operation**: `0 + 8 = 8`
- **Final Stack**: `[8]`

```
|  8  |
------
```

Another Example for the evaluation of Postfix Expression

Example 2 : Evaluating 5 1 2 + 4 * + 3 -

Consider evaluating a different postfix expression: 5 1 2 + 4 * + 3 -

1. **Initialize an Empty Stack**

   Start with an empty stack.

2. **Read and Process Each Token**

Table 7.6: Step-by-Step Postfix Evaluation of Example 2

Token	Action	Stack
`5`	Push `5`	`[5]`
`1`	Push `1`	`[5, 1]`
`2`	Push `2`	`[5, 1, 2]`
`+`	Pop `1` and `2`, calculate `1 + 2 = 3`, push `3`	`[5, 3]`
`4`	Push `4`	`[5, 3, 4]`
`*`	Pop `3` and `4`, calculate `3 * 4 = 12`, push `12`	`[5, 12]`
`+`	Pop `5` and `12`, calculate `5 + 12 = 17`, push `17`	`[17]`
`3`	Push `3`	`[17, 3]`
`-`	Pop `17` and `3`, calculate `17 - 3 = 14`, push `14`	`[14]`

Final Result: After processing all tokens, the stack contains a single value [14], which is the result of the postfix expression 5 1 2 + 4 * + 3 -.

Explanation of the Steps

1. **Push Operands onto the Stack**: Each time we encounter an operand (5, 1, 2, 4, and 3), it is pushed onto the stack, as shown in the first few steps.
2. **Applying Operators Using Stack Values**:
   - When we encounter the + operator after 1 and 2, we pop these two values from the stack, calculate 1 + 2 = 3, and push the result back onto the stack.
   - This pattern continues for each operator in the expression. For example, when we encounter *, we pop 3 and 4, multiply them to get 12, and push 12 onto the stack.
3. **Final Operation**: The last operation - pops the values 17 and 3, subtracting 3 from 17 to get 14, which is pushed back onto the stack as the final result.

7.7. Why Postfix Evaluation is Efficient

Using a stack for postfix evaluation is efficient because:

- **Single Pass Evaluation**: We only need to process the expression once from left to right.
- **No Need for Precedence or Parentheses**: Postfix notation eliminates the need for these rules, simplifying the evaluation process.
- **Widely Used in Compilers**: Compilers use postfix notation (RPN) internally, as it makes generating and executing machine code straightforward and efficient.

Now that we know what a postfix expression is, let's explore how to evaluate it step-by-step. Postfix expressions are particularly easy to evaluate using a stack data structure, which allows us to store and retrieve operands as we process each operator in the expression.

7.8 IMPLEMENTING POSTFIX EVALUATION IN C

In postfix evaluation, a stack is used to store operands until an operator is encountered. When an operator is encountered, the two most recent operands are popped from the stack, the operation is applied, and the result is pushed back onto the stack. This method allows us to evaluate expressions without worrying about operator precedence or parentheses.

7.8.1 Push Function

The push function adds a value to the top of the stack. This function is essential for storing operands and intermediate results in postfix evaluation.

```
int stack[MAX_STACK_SIZE];
int top = -1;
// Function to push a value onto the stack
void push(int value)
{
   if (top < MAX_STACK_SIZE - 1)
   {  // Check if there's room in the stack
      stack[++top] = value;        // Increment top and place value
   } else {
      printf("Stack overflow\n");  // Error if stack is full
   } }
```

Explanation:

- The condition top < MAX_STACK_SIZE - 1 ensures there's enough space in the stack to add a new item.
- stack[++top] = value increments the top index by 1 and stores value at the new top position.
- If the stack is full, an overflow message is printed.

7.8.2 Pop Function

The pop function removes and returns the top element of the stack. It ensures that there are elements in the stack before attempting to pop an item.

```
int pop() {
   if (top >= 0) {            // Ensure the stack is not empty
      return stack[top--];     // Return top element, then decrement top
   } else {
      printf("Stack underflow\n");  // Error if stack is empty
      return -1;               // Placeholder for error
   }
}
```

Explanation:

- The condition top >= 0 checks that the stack has elements, ensuring it is not empty before attempting to pop.
- stack[top--] returns the top element of the stack and then decreases the top index by 1, effectively removing the top item.
- If the stack is empty, an underflow message is printed, and -1 is returned as an error indicator.

7.9 EVALUATION OF POSTFIX FUNCTION

The evaluatePostfix function processes each character in the postfix expression:

1. If a character is a digit, it is converted to an integer and pushed onto the stack.
2. If a character is an operator, it pops the two most recent operands from the stack, applies the operator, and pushes the result back onto the stack.

C code for the evaluation of postfix function is as follows

```
int evaluatePostfix(char *expr) {
   int i;
   for (i = 0; expr[i] != '\0'; i++) {
      if (isdigit(expr[i])) {          // Check if character is a digit
         push(expr[i] - '0');          // Convert character to integer
      } else {                         // Character is an operator
         int operand2 = pop();         // Pop the second operand
         int operand1 = pop();         // Pop the first operand
         switch (expr[i]) {
```

```
            case '+': push(operand1 + operand2); break;
            case '-': push(operand1 - operand2); break;
            case '*': push(operand1 * operand2); break;
            case '/':
               if (operand2 != 0) {    // Avoid division by zero
                  push(operand1 / operand2);
               } else {
                  printf("Error: Division by zero\n");
                  return -1;
               }
               break;
            default: printf("Invalid operator\n"); break;
         }  }   }
      return pop();  // Final result }
```

7.9.1 Explanation of Key Components:

1. **Digit Check with isdigit(expr[i]):** This function checks if the character expr[i] is a digit. If it is, it's converted from a character to an integer (expr[i] - '0') and pushed onto the stack.
2. **Operator Handling with switch**: If expr[i] is an operator (+, -, *, /), the function pops the top two values from the stack, performs the corresponding operation, and pushes the result back onto the stack.
    - **Division by Zero**: A check is included to ensure that division by zero doesn't occur. If operand2 is zero during division, an error message is printed.
    - **Default Case**: A default case is added to handle any unexpected or invalid operators.
3. **Final Result**: After processing all characters in the expression, the function returns the last remaining value on the stack, which is the evaluated result.

7.10 INFIX TO POSTFIX CONVERSION

Infix and postfix are two different ways to write mathematical expressions. Infix is the standard mathematical notation where operators are placed between operands (e.g., A+B), while postfix (also called Reverse Polish Notation or RPN) places operators after operands (e.g., AB+).

To convert an infix expression (e.g., A+B) to a postfix expression (e.g., AB+), we need to follow a systematic approach. This is generally done using a stack data structure to handle operator precedence and parentheses.

Steps for Converting Infix to Postfix:

1. Initialize a stack for operators and an empty list for the postfix expression.
2. Process the infix expression from left to right:
    - If the token is an operand (i.e., a number or variable), add it directly to the postfix expression.
    - If the token is an operator (e.g., +, -, *, /):

        - While the stack is not empty and the operator at the top of the stack has greater or equal precedence than the current operator, pop operators from the stack and append them to the postfix expression.
        - Push the current operator onto the stack.
    - If the token is a left parenthesis (`(`), push it onto the stack.
    - If the token is a right parenthesis (`)`), pop from the stack and add operators to the postfix expression until a left parenthesis is encountered on the stack (which is then discarded).
3. After processing the entire infix expression, pop any remaining operators from the stack and append them to the postfix expression.

Precedence and Associativity of Operators:

- **Precedence** determines the order in which operators are applied. For example, multiplication (`*`) has higher precedence than addition (`+`).
- **Associativity** determines how operators of the same precedence are grouped. Most operators (like `+`, `-`, `*`, `/`) are left-associative, meaning they are evaluated from left to right. However, exponentiation (`^`) is right-associative, meaning it is evaluated from right to left.

Example Conversion:

Conversion of the infix expression A + B ∗ (C − D) to postfix.

1. **Infix Expression**: A + B ∗ (C − D)
2. **Initialize**: Stack = `[]`, Postfix = `[]`
    - **Process `A`**: Operand, so add it to postfix.
      Postfix = `[A]`
    - **Process `+`**: Operator, so push it onto the stack.
      Stack = `[+]`
    - **Process `B`**: Operand, so add it to postfix.
      Postfix = `[A, B]`
    - **Process `*`**: Operator. `*` has higher precedence than `+`, so push it onto the stack.
      Stack = `[+, *]`
    - **Process `(`**: Left parenthesis, so push it onto the stack.
      Stack = `[+, *, (]`
    - **Process `C`**: Operand, so add it to postfix.
      Postfix = `[A, B, C]`
    - **Process `-`**: Operator, so push it onto the stack.
      Stack = `[+, *, (, -]`
    - **Process `D`**: Operand, so add it to postfix.
      Postfix = `[A, B, C, D]`
    - **Process `)`**: Right parenthesis. Pop operators until `(` is encountered. Pop - and add it to postfix, then discard the left parenthesis.
      Postfix = [A, B, C, D, -]
      Stack = [+, *]
3. End of Expression: Pop all remaining operators from the stack and add them to the postfix expression.
   Pop * and add it to postfix.

Pop + and add it to postfix.
Postfix = [A, B, C, D, -, *, +]

Final Postfix Expression:

A B C D - * +

- **Infix**: Operators are placed between operands.
- **Postfix**: Operators are placed after operands, and no parentheses are needed for grouping.
- The conversion uses a **stack** to manage operator precedence and parentheses

7.11 KEY TERMS

Expression, Operator Precedence, Associativity, Infix Expression, Prefix Expression (Polish Notation), Postfix Expression (Reverse Polish Notation), Stack, Push, Pop, Postfix Evaluation

7.12 REVIEW QUESTIONS

1. What is an expression in programming, and how is it evaluated?
2. Explain operator precedence and give an example of how it affects expression evaluation.
3. Define associativity and distinguish between left-associative and right-associative operators with examples.
4. Describe the difference between infix, prefix, and postfix expressions.
5. Why is postfix notation often preferred in compiler design?
6. What role does a stack play in evaluating postfix expressions?

7.13 SUGGESTED READINGS

1. **"Data Structures and Program Design in C"** by Robert L. Kruse and Bruce P. Leung.
2. **"The C Programming Language"** by Brian W. Kernighan and Dennis M. Ritchie.
3. **"Introduction to Algorithms"** by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein
4. **"Compilers: Principles, Techniques, and Tools"** by Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman

Dr.Vasantha Rudramalla

Lesson - 8

Sorting Algorithms

OBJECTIVE

At the end of this chapter, students will be able to understand:

- The fundamental concepts and purposes of searching and sorting in data structures.
- The key differences between basic and optimized search techniques, such as sequential and binary search, and when to apply each.
- Various sorting algorithms, including Insertion Sort, Quick Sort, Merge Sort, Heap Sort, and Radix Sort, along with their specific use cases, efficiencies, and limitations.
- The concept of internal and external sorting, and the scenarios that necessitate each approach.

Structure

8.1 INTRODUCTION

Sorting is a fundamental operation in computer science, essential for organizing data to improve efficiency in retrieval, processing, and management. By arranging elements in a specific order, sorting enhances the performance of other operations like searching, merging, and indexing. This lesson explores a variety of sorting techniques, including comparison-based algorithms such as Quick Sort and Merge Sort, as well as non-comparison-based methods like Radix Sort, each suited to different scenarios based on their time and space complexities. It also distinguishes between internal sorting, for datasets that fit in memory, and external sorting, designed for managing large datasets stored on external devices. Through an understanding of sorting algorithms, their efficiencies, and their applications, learners will develop the skills necessary to solve real-world problems where data organization and optimization are critical.

8.1.1 Searching and List Verification

Searching and verification are critical processes in data management and programming. Efficiently locating and verifying data can significantly impact the performance of software applications. Searching and List Verification introduces core search algorithms, sequential and binary search, along with techniques for verifying list content. Sequential search is useful in general cases, especially with unsorted data, while binary search requires ordered data and is much faster for larger datasets. Lastly, list verification confirms the accuracy of data across two lists, a key operation in systems where data consistency is crucial.

For context, searching methods aim to locate specific elements within data structures, while verification involves ensuring data integrity by comparing multiple lists or datasets. Both tasks serve as foundational roles across fields, from databases to data processing algorithms. When dealing with vast data sets, these methods allow for efficient data handling and manipulation, contributing to overall system reliability and performance.

Algorithms for searching generally fall into two categories: brute-force and optimized searches. Brute-force approaches, like sequential search, check each element in a list individually, making them simpler but less efficient. Optimized searches, like binary search, leverage sorted data to significantly reduce search times by "dividing and conquering" the list, yielding logarithmic time complexity. List verification, on the other hand, can involve various techniques depending on the data's order and structure, either using straightforward element-by-element comparisons or more complex checks for sorted data.

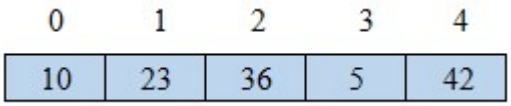
The remainder of this section delves into these search methods and verification techniques, detailing how they work, their applications, and the advantages and disadvantages they present.

8.1.2 Sequential Search

Sequential search, also known as linear search, is the most basic search algorithm. In sequential search, each element of the list is checked one by one until the desired item is found or the list ends. This algorithm is straightforward and easy to implement, making it suitable for small, unsorted lists. However, its inefficiency for larger datasets or ordered lists limits its use in such cases.

The process of a sequential search begins by taking a target item, often called the "search key," and comparing it to the first item in the list. If there is a match, the search ends. If not, the algorithm proceeds to the next element in the list, continuing until it either finds the target item or reaches the end. The worst-case scenario in a sequential search occurs when the target item is either the last item in the list or is absent entirely, necessitating n comparisons for a list of n elements. The time complexity of sequential search is, therefore, $O(n)$ in both the average and worst cases.

For example, consider the list [10, 23, 36, 5, 42] and a target value of 36. Sequential search will proceed as follows:



1. Compare 36 with 10 – no match.
2. Compare 36 with 23 – no match.
3. Compare 36 with 36 – match found.

This search method is highly adaptable to dynamic datasets, where new data may frequently be added or removed. Unlike binary search, sequential search requires no ordering, which means it can be applied to any dataset without additional preprocessing. However, for larger datasets, the linear time complexity can be prohibitively slow, which is why it is typically reserved for shorter or less frequently searched lists.

In C, sequential search can be implemented as a simple loop function. Here's a sample function:

```
int sequentialSearch(int arr[], int size, int target) {
    for (int i = 0; i < size; i++) {
        if (arr[i] == target) {
            return i;  // Return index if found
        }
    }
    return -1;  // Return -1 if not found
}
```

While sequential search is fundamental, its inefficiency becomes clear as data scales, prompting the use of more optimized methods like binary search for large, ordered datasets.

8.1.3 Binary Search

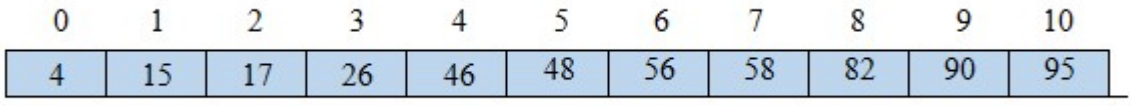
Binary search is an efficient algorithm that leverages ordered data to reduce the time required to find an item. Binary search repeatedly divides the search interval in half, determining which half contains the target item. This "divide and conquer" approach allows binary search to achieve logarithmic time complexity, $O(\log n)$, making it highly efficient for large, sorted datasets.

Binary search starts by comparing the target item with the middle element of the list:

1. If the middle element matches the target, the search is complete.
2. If the middle element is greater than the target, the algorithm narrows its search to the left half of the list.
3. If the middle element is less than the target, it focuses on the right half.

This process repeats until the item is found or the list can no longer be divided. The requirement that the list be ordered is both a strength and a limitation; binary search is among the fastest search methods but can only be applied to sorted data. The need for sorting means that if a list is unsorted, binary search becomes inefficient as it must first sort the data before performing the search.

For instance, given a sorted list [4, 15, 17, 26, 30, 46, 48, 56, 58, 82, 90, 95] and a target 58, binary search operates as follows:



0	1	2	3	4	5	6	7	8	9	10
4	15	17	26	46	48	56	58	82	90	95

1. The initial middle element, 30, is less than 58, so the search narrows to the right half.
2. The new middle element, 56, is also less than 58, further narrowing the search.
3. The final middle element is 58, a match.

In C, binary search can be implemented with a recursive or iterative function. Here's a simple example of the iterative method:

```
int binarySearch(int arr[], int size, int target) {
    int left = 0, right = size - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (arr[mid] == target) return mid;  // Match found
        if (arr[mid] < target) left = mid + 1;
        else right = mid - 1;
    }
    return -1;  // Target not found
}
```

Binary search offers significant efficiency gains over sequential search, particularly as data sizes increase. However, it requires that data be pre-sorted, which can add a time complexity cost if the list is dynamic.

8.1.4 List Verification

List verification is a technique used to confirm the consistency of two lists by ensuring they contain the same elements. This process is crucial in data management, especially in applications where data integrity is paramount, such as accounting, database management, and system integration.

The goal of list verification is to identify any discrepancies between two lists, such as missing, extra, or mismatched entries. This process is often applied in contexts where data accuracy must be maintained across multiple records or datasets. Examples include reconciling employee records with payroll data, verifying transactions in financial records, or checking data consistency in backups.

There are several approaches to list verification:

1. **Direct Comparison**: For unordered lists, each element of one list is checked against all elements of the other. This brute-force approach has a time complexity of $O(n * m)$, where n and m are the lengths of the lists.
2. **Sorted Comparison**: For sorted lists, verification is more efficient. Sorting allows for a single pass through both lists, reducing complexity to $O(n \log n + m \log m + n + m)$.

In an example application, the IRS may verify that reported incomes match by comparing records from employers and employees. Discrepancies reveal either missing entries or data mismatches, highlighting errors or potential fraud.

In C, list verification can involve nested loops for unordered comparisons, or it may use sorting algorithms to streamline the process. Here is an example function for unordered list verification:

```
int verifyLists(int list1[], int size1, int list2[], int size2)
{
   for (int i = 0; i < size1; i++) {
      int found = 0;
      for (int j = 0; j < size2; j++) {
         if (list1[i] == list2[j]) {
            found = 1;
        break; }  }
      if (!found) return 0;  // Lists are not identical
else  return 1;  // Lists are identical}
```

While simple, this method may be inefficient for large lists. Sorting both lists and comparing them directly can reduce verification time. List verification is an essential tool in data integrity, supporting consistent, reliable data across applications and records.

8.2 SORTING ALGORITHMS

Sorting algorithms are fundamental techniques in computer science that arrange elements in a specific order, usually ascending or descending. These algorithms are crucial in data processing and retrieval tasks, as sorted data enables faster searching, better organization, and more efficient data handling. Sorting also serves as a steppingstone for other complex algorithms, such as those for searching and optimization.

The primary goal of a sorting algorithm is to reorganize a given array or list of items into a desired order. Each sorting method has unique characteristics and is suited to different types of data and performance requirements. Some key aspects that define sorting algorithms include:

- **Time Complexity**: This measures the speed of an algorithm, usually in terms of best, average, and worst-case scenarios. Commonly, time complexity is expressed in Big-O notation, such as $O(n^2)$.or O(nlogn).
- **Space Complexity**: This refers to the amount of additional memory the algorithm requires. Some algorithms, like Merge Sort, require extra space, while others, such as Heap Sort, sort the data in place.
- **Stability**: A stable sort maintains the relative order of records with equal keys. For example, in a list of students with the same grade, a stable sort would keep them in the same order they originally appeared.
- **Adaptability**: Some algorithms perform more efficiently on data that is already partially sorted, making them suitable for dynamic or nearly sorted datasets.

Sorting algorithms can be broadly categorized as **comparison-based** and **non-comparison-based**.

1. **Comparison-Based Sorting**: These algorithms use comparisons between elements to determine their order. Examples include Quick Sort, Merge Sort, Heap Sort, and Bubble Sort. The theoretical lower bound for comparison-based algorithms is O(nlogn), meaning no comparison-based algorithm can perform better than this for large datasets.
2. **Non-Comparison-Based Sorting**: These algorithms sort data without directly comparing elements. They often rely on the specific properties of the data, such as its range or length. Examples include Radix Sort and Counting Sort. These methods can achieve linear time complexity, O(n), but are typically only applicable to integers or fixed-size data types.

The choice of a sorting algorithm depends on several factors, such as the size of the dataset, memory limitations, and the need for stability. For instance, Quick Sort is generally efficient and has an average time complexity of O(nlogn), making it suitable for large datasets. Merge Sort, which is stable and operates in O(nlogn) time, is often preferred when stability is required, especially in linked lists or database sorting. Meanwhile, Insertion Sort, with a time complexity of $O(n^2)$ is ideal for small or nearly sorted data.

Sorting algorithms provide the foundation for data processing, making them a core component in fields like computer science, data analysis, and software engineering.

8.3 INSERTION SORT

Insertion Sort is a straightforward, intuitive sorting technique. It works similarly to the way people sort playing cards in their hands: elements are picked one by one and placed in the correct position relative to those already sorted. This sorting method is efficient for small datasets or nearly sorted data but can be inefficient for large, randomly ordered datasets, with a worst-case time complexity of $O(n^2)$.

Steps in Insertion Sort:

1. Begin with the second element (as the first element is trivially sorted).
2. Compare the current element with its predecessors.
3. Shift each predecessor that is greater than the current element one position to the right.

4. Insert the current element in the correct position.
5. Move to the next element and repeat until the end of the array.

Example:

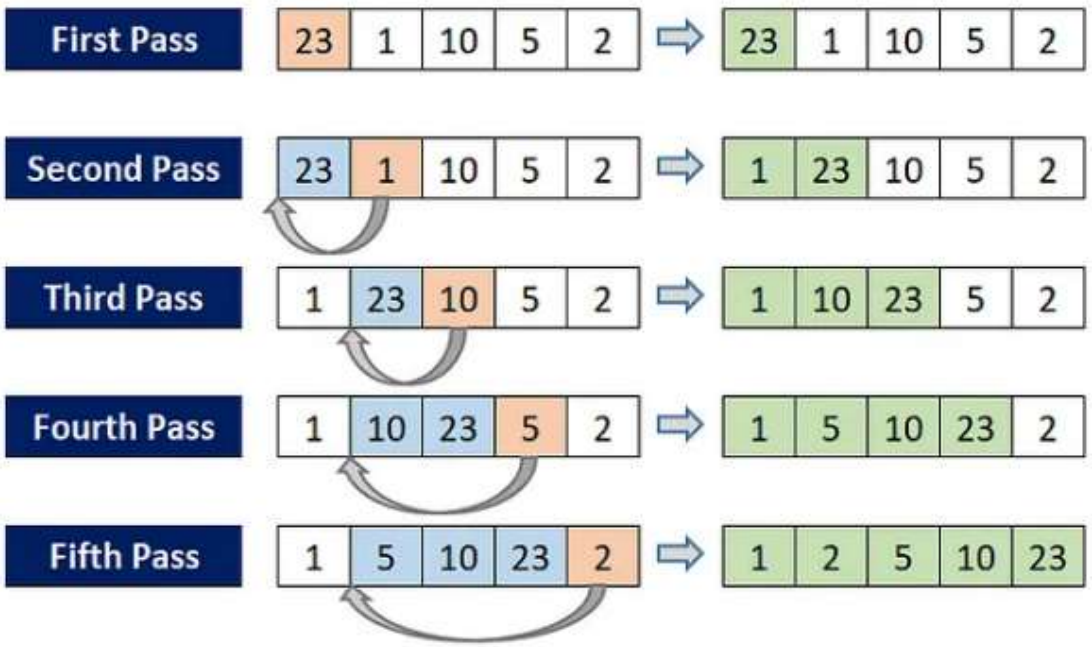


Fig 8.1. Iterations of Insertion Sort

Step-by-step explanation of how the insertion sort works for the array `[23, 1, 10, 5, 2]`

1. **First Pass (Insert 1)**:
   - Start with 23 (it's already in place since it's the first element).
   - Look at 1. Since 1 is less than 23, move 23 one position to the right and place 1 at the beginning.
   - Result after this pass: [1, 23, 10, 5, 2].
2. **Second Pass (Insert 10)**:
   - Now, look at 10. Compare 10 with 23.
   - Since 10 is less than 23, move 23 one position to the right.
   - Place 10 in the position where 23 was.
   - Result after this pass: [1, 10, 23, 5, 2].
3. **Third Pass (Insert 5)**:
   - Look at 5 next. Compare 5 with 23 and 10.
   - Since 5 is less than both, move 23 and 10 one position to the right.
   - Place 5 where 10 was.
   - Result after this pass: [1, 5, 10, 23, 2].
4. **Fourth Pass (Insert 2)**:
   - Finally, look at 2. Compare 2 with 23, 10, and 5.
   - Since 2 is less than all of them, move 23, 10, and 5 one position to the right.
   - Place 2 at the start of the array.
   - Result after this pass: [1, 2, 5, 10, 23].

The array is now sorted.

8.4 QUICK SORT

Quick Sort is a highly efficient, divide-and-conquer sorting algorithm developed by C.A.R. Hoare. The basic idea is to select a "pivot" element from the array, partition the remaining elements around this pivot, and recursively sort the subarrays on either side of the pivot. Quick Sort has an average-case time complexity of O(nlogn), although the worst case is $O(n^2)$ if the pivot elements are poorly chosen.

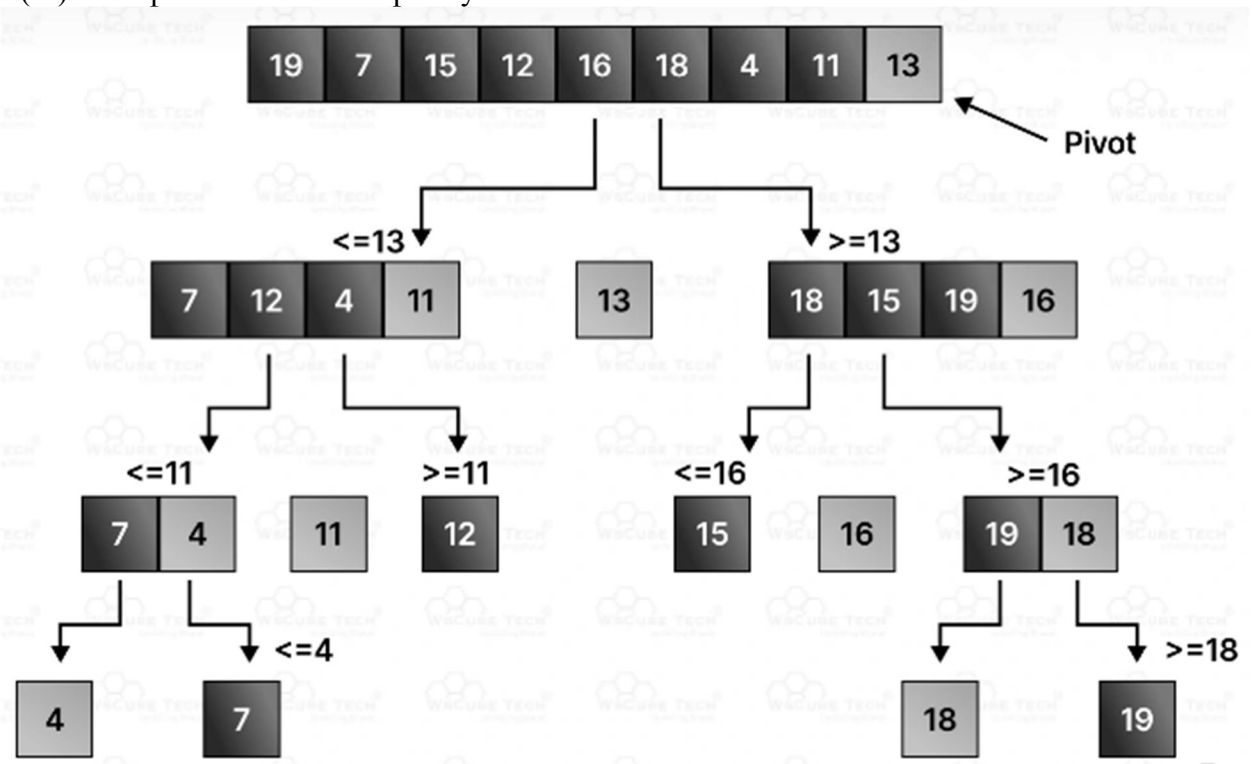


Fig 8.2. Iterations of Quick Sort

1. **Choose a Pivot**:
   - Start with the array [19, 7, 15, 12, 16, 4, 11, 13].
   - Select 13 as the pivot.
2. **Partition the Array**:
   - Divide the array into two groups:
     - Left side (values less than or equal to 13): [7, 12, 4, 11]
     - Right side (values greater than or equal to 13): [18, 15, 19, 16]
   - Place 13 in its correct sorted position.
3. **Repeat for Each Subarray**:
   - **Left Subarray [7, 12, 4, 11]**:
     - Choose 11 as the pivot.
     - Split it into two groups:

        - Left of 11 (values <= 11): [7, 4]
        - Right of 11 (values >= 11): [12]
    - Place 11 in its correct sorted position.
- **Further divide [7, 4]**:
    - Choose 4 as the pivot.
    - Sort it to get [4, 7].
- **Right Subarray [18, 15, 19, 16]**:
    - Choose 16 as the pivot.
    - Split it into two groups:
        - Left of 16 (values <= 16): [15]
        - Right of 16 (values >= 16): [19, 18]
    - Place 16 in its correct sorted position.
- **Further divide [19, 18]**:
    - Choose 18 as the pivot.
    - Sort it to get [18, 19].

4. **Combine All Sorted Subarrays**:
    - After each subarray is sorted, combine them to get the fully sorted array.

This process sorts the array step-by-step by dividing it around pivot values, arranging elements smaller than the pivot on one side and larger ones on the other.

8.5 OPTIMAL SORTING TIME

Optimal Sorting Time refers to the best achievable performance for any comparison-based sorting algorithm. According to the theory of comparison sorting, no algorithm can do better than O(nlog n)O in the worst case. This limitation is derived from a decision tree model of sorting, where each comparison corresponds to a node and each path from the root to a leaf represents a permutation of the list.

In this model:

- Every possible permutation of elements represents a unique leaf in the decision tree.
- To guarantee that every permutation can be reached, the tree must have at least n! leaves.
- Therefore, the height of the decision tree (representing comparisons) is O(nlogn)

This theoretical bound is crucial for understanding the efficiency of sorting algorithms like Merge Sort and Quick Sort, which achieve this optimal time.

8.6 MERGE SORT

Merge Sort is another divide-and-conquer sorting algorithm that splits the array into halves, sorts each half, and merges the sorted halves. Merge Sort is stable, meaning that it maintains the relative order of equal elements, and has a time complexity of O(nlogn). However, its space complexity is O(n) due to the need for auxiliary storage.

8.6.1 Merging

Merging is the process of combining two sorted arrays into one. The merge function compares the smallest elements of both arrays, moving the smaller element into the result array, until all elements are merged.

8.6.2 Iterative Merge Sort

Iterative Merge Sort, unlike the recursive method, uses an iterative approach with successive merging of subarrays until the entire array is sorted.

8.6.3 Recursive Merge Sort

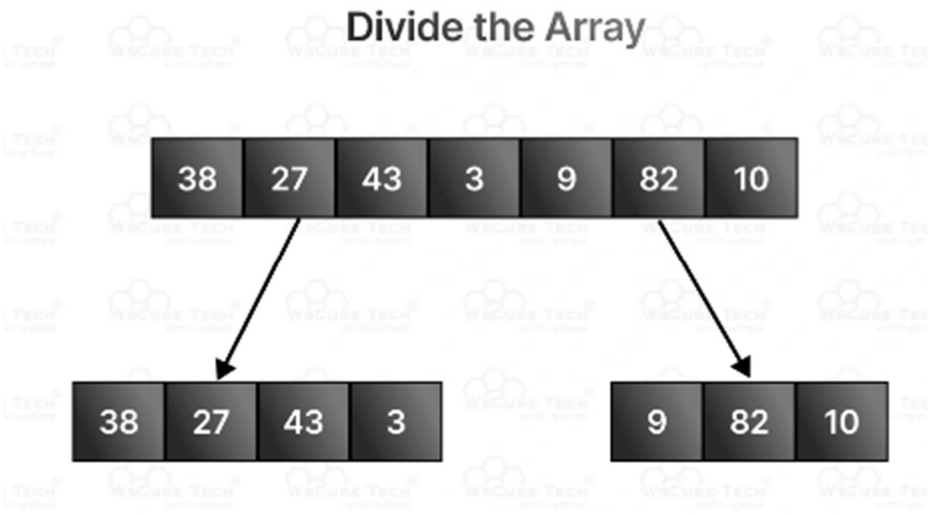
Recursive Merge Sort repeatedly splits the array until single-element subarrays are reached, then merges them recursively to produce the sorted array.

Example of Merge Sort

In the below example, the array [38, 27, 43, 3, 9, 82, 10] is sorted using merge sort.

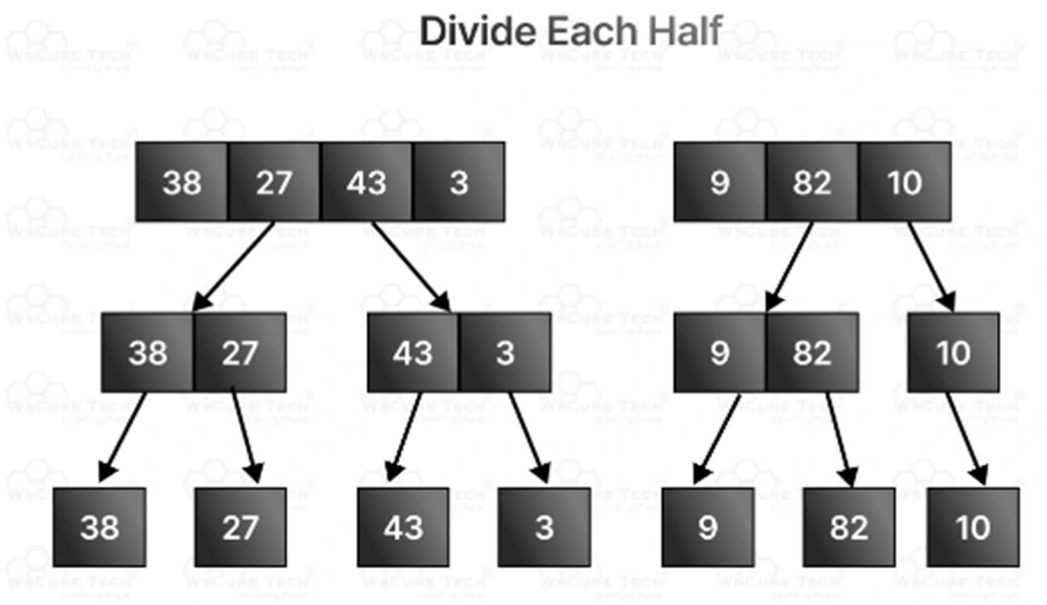
1. Divide the Array

The array is divided into two halves.



2. Divide Each Half

Continue dividing each half into smaller subarrays until each subarray has only one element.



3. **Merge Individual Pairs of Subarrays**

- Combine [38] and [27] to form [27, 38].
- Combine [43] and [3] to form [3, 43].
- Combine [9] and [82] to form [9, 82].

Result after this step: [27, 38], [3, 43], [9, 82], [10].

Merge Sorted Subarrays

- Merge [27, 38] with [3, 43] to get [3, 27, 38, 43].
- Merge [9, 82] with [10] to get [9, 10, 82].

Final merged result: [3, 27, 38, 43], [9, 10, 82].

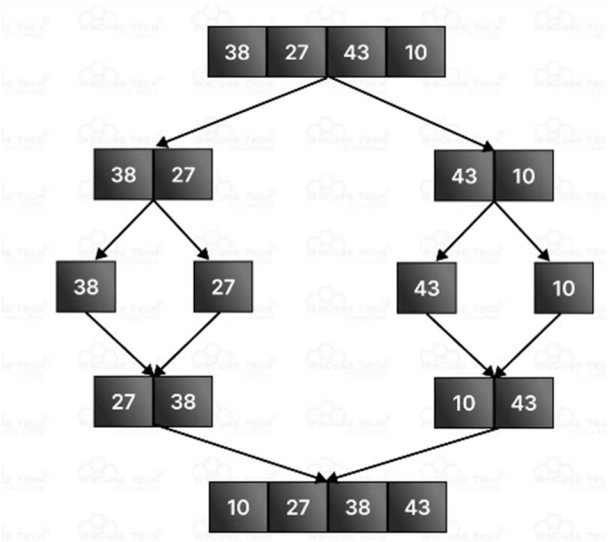


Fig 8.1. Iterations of Merge Sort

5. Merge the Final Two Halves

Merge [3, 27, 38, 43] and [9, 10, 82]

- Compare 3 (left) and 9 (right). Since 3 is smaller, add 3 to the new array.
- Compare 27 (left) and 9 (right). Since 9 is smaller, add 9 to the new array.
- Compare 27 (left) and 10 (right). Since 10 is smaller, add 10 to the new array.
- Compare 27 (left) and 82 (right). Since 27 is smaller, add 27 to the new array.
- Compare 38 (left) and 82 (right). Since 38 is smaller, add 38 to the new array.
- Compare 43 (left) and 82 (right). Since 43 is smaller, add 43 to the new array.
- Finally, add the remaining 82 to the new array

Final Sorted Array : [3, 9, 10, 27, 38, 43, 82]

The array is now fully sorted: [3, 9, 10, 27, 38, 43, 82].

8.7 HEAP SORT

Heap Sort is based on the heap data structure, specifically a max-heap or min-heap. It builds a max-heap from the input data, repeatedly removes the largest element from the heap, and places it at the end of the sorted array. The algorithm has a time complexity of O(nlogn), as each insertion and deletion operation in a heap is O(log n).

Heap sort is a way to sort a list of items, like numbers, in order. It uses a special tree structure called a heap. A heap is a kind of binary tree where each parent node is greater than or equal to its child nodes. This helps in easily finding the largest or smallest item.

Here's how it works:

- **Build a Heap:** First, we arrange the list of numbers into a heap. This makes sure the largest number is at the top of the heap.
- **Remove the Top:** Then, we remove the top (the largest number) and place it at the end of the list.
- **Rebuild the Heap:** After removing the top, we rebuild the heap with the remaining numbers.
- **Repeat:** We keep repeating the process of removing the top and rebuilding the heap until all numbers are sorted.

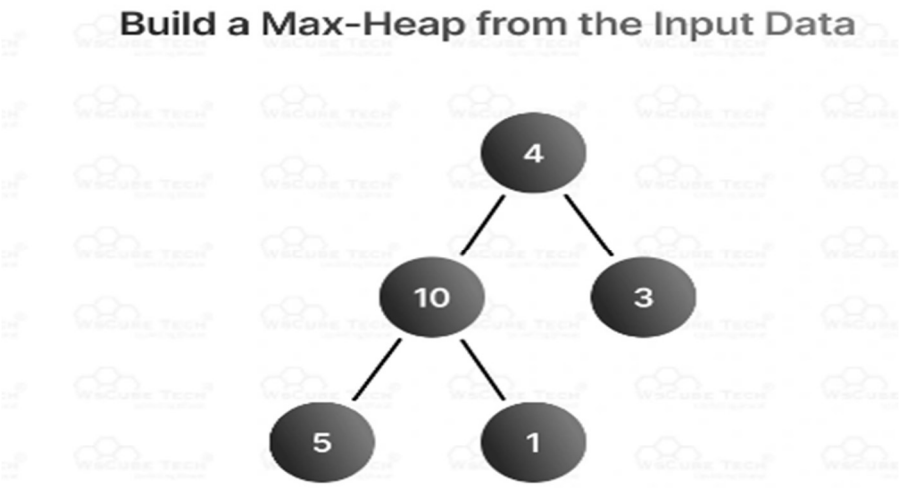
By repeatedly moving the largest number to the end of the list and restructuring the heap, we end up with a sorted list. Heap sort is efficient and works well for large lists.

How Heap Sort Works?

1. Build a Max-Heap from the Input Data

Convert the array into a heap, where the largest value is at the root of the heap.

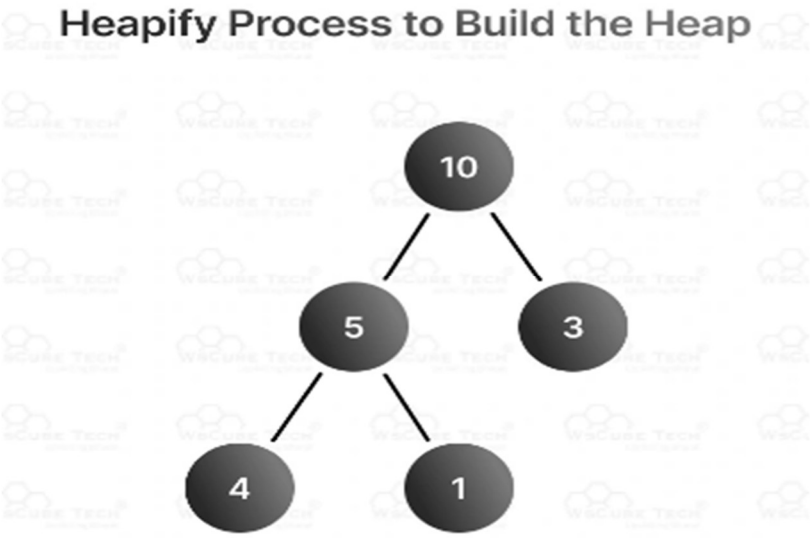
Initial array: [4, 10, 3, 5, 1]



2. Heapify Process to Build the Heap

Adjust the tree structure to ensure the heap property is maintained, where every parent node is larger than its child nodes.

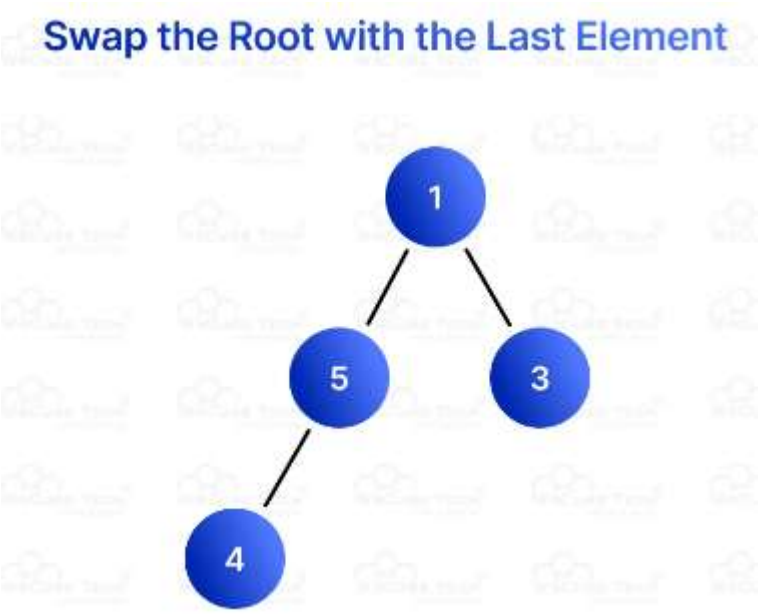
After heapifying:



3. Swap the Root with the Last Element

Move the largest element (root) to the end of the array and reduce the heap size by one.

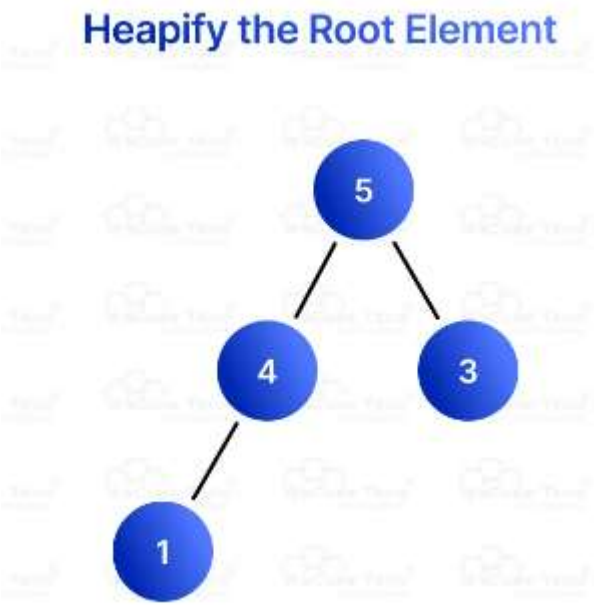
Swap 10 with 1:



4. Heapify the Root Element:

Restore the heap property by heapifying the root element.

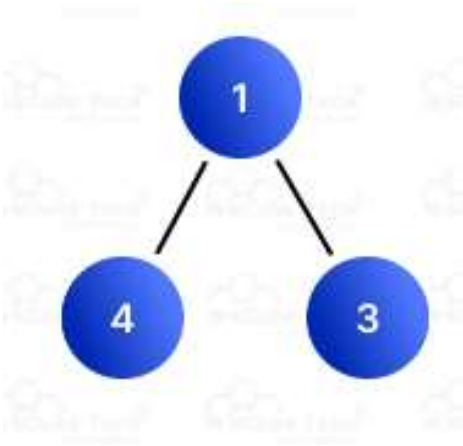
After heapifying:



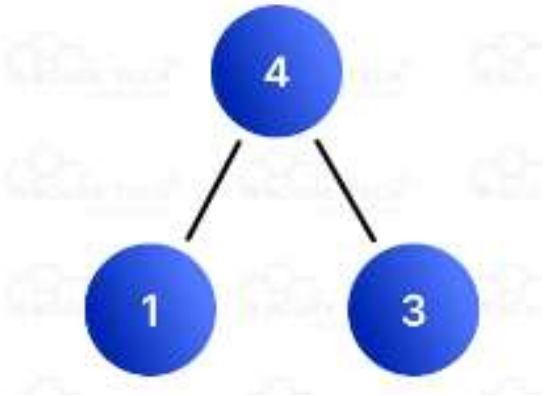
5. Repeat the Process

Continue swapping the root with the last element of the heap and heapifying until the entire array is sorted.

Swap 5 with 1:



After heapifying:



Swap 4 with 3:

Array representation: [3, 1, 4, 5, 10]

After heapifying:

Array representation: [3, 1, 4, 5, 10]

Swap 3 with 1:

Array representation: [1, 3, 4, 5, 10]

After heapifying:

Array representation: [1, 3, 4, 5, 10]

Now the array is sorted: [1, 3, 4, 5, 10].

8.8 RADIX SORT

Radix Sort is a non-comparative sorting algorithm that sorts elements digit by digit, starting from the least significant to the most significant digit. It is particularly efficient for sorting integers and strings with a fixed length.

Steps in Radix Sort:

1. Start with the least significant digit.
2. Group numbers into "buckets" based on the current digit.
3. Concatenate the buckets in order.
4. Move to the next significant digit and repeat.

Example:

Given [170, 45, 75, 90, 802, 24, 2, 66]:

1. Sort by the least significant digit to get [170, 90, 802, 2, 24, 45, 75, 66].
2. Sort by the next significant digit to get [802, 2, 24, 45, 66, 170, 75, 90].
3. Continue until all digits are processed, yielding [2, 24, 45, 66, 75, 90, 170, 802].

Radix Sort works best when the number of digits or character length is fixed, as it operates in O(d×(n+b))O(d \times (n + b))O(d×(n+b)), where ddd is the number of digits and bbb is the base.

8.9 LIST AND TABLE SORTS

List and Table Sorts are specialized sorting methods that are applied to data organized in lists or table structures. These sorting methods optimize the data structure properties to achieve efficient sorting, especially when dealing with linked lists or database-like table formats.

List Sorts:

- **Merge Sort on Linked Lists**: Because linked lists lack random access, merge sort is ideal as it splits and merges in place.
- **Insertion Sort for Nearly Sorted Lists**: For linked lists that are nearly sorted, insertion sort can be efficient because it only requires a scan from the head to place elements in order.

Table Sorts:

In databases or tables, sorting by specific columns, or keys, can be done using specialized techniques that allow efficient, stable sorting with minimal movement of data. Table sorts are widely used in database management for indexing and query optimization.

Each sorting technique has its unique strengths and ideal scenarios, with detailed algorithms, code examples, and illustrations provided

8.10 SUMMARY OF INTERNAL SORTING

Internal sorting techniques are applied when the entire dataset can be loaded into the computer's main memory. This type of sorting is commonly used for smaller datasets or in systems with ample memory. Internal sorting algorithms vary in complexity and performance, each offering advantages based on the data characteristics and requirements.

1. **Insertion Sort**:
   - **How It Works**: Insertion Sort builds the sorted array of one item at a time. Starting from the second element, each new element is compared to the ones before it and placed in its correct position. This shifting operation continues until all elements are in order.
   - **Performance**: Insertion Sort has a worst-case and average time complexity of $O(n^2)$. However, it performs well for small datasets and is particularly efficient for nearly sorted data due to its adaptability. Its best-case performance is O(n) when the data is already sorted.
   - **Example**:
     - Given the array [5, 4, 3, 2, 1]:
       1. Insert 4 before 5, resulting in [4, 5, 3, 2, 1].
       2. Insert 3 before 4, resulting in [3, 4, 5, 2, 1].
       3. Insert 2 before 3, resulting in [2, 3, 4, 5, 1].
       4. Insert 1 before 2, resulting in [1, 2, 3, 4, 5].
2. **Quick Sort**:
   - **How It Works**: Quick Sort is a divide-and-conquer algorithm that works by selecting a "pivot" element. It then partitions the array such that all elements less than the pivot are on its left and all elements greater than the pivot are on its right. This partitioning process continues recursively for the subarrays on either side of the pivot.
   - **Performance**: Quick Sort has an average time complexity of O(nlog n)O(n \log n)O(nlogn) but can degrade to $O(n^2)$ if poor pivot selection results in highly unbalanced partitions (e.g., if the array is already sorted in the worst pivot choice scenario). By using randomized or median-of-three pivot selection strategies, the chances of encountering the worst-case scenario are minimized.
   - **Example**:
     Given the array [26, 5, 37, 1, 61, 11, 59, 15, 48, 19]:
       1. Select 26 as the pivot.
       2. Rearrange elements to get [5, 1, 11, 15, 19, 26, 48, 37, 59, 61].
       3. Recursively apply Quick Sort to the subarrays [5, 1, 11, 15, 19] and [48, 37, 59, 61].
3. **Merge Sort**:
   - **How It Works**: Merge Sort is a stable, divide-and-conquer algorithm that divides the array into two halves, recursively sorts each half, and then merges them back together in a sorted manner. This process is repeated until the array is fully sorted.
   - **Performance**: Merge Sort's time complexity is O(nlog n)O(n \log n)O(nlogn) for all cases, making it reliable for large datasets. However, it requires O(n)O(n)O(n) additional memory to store temporary arrays during merging, which may be a drawback when memory usage is a concern.
   - **Example**:
     - For the array [38, 27, 43, 3, 9, 82, 10]:
       1. Split into [38, 27, 43] and [3, 9, 82, 10].
       2. Recursively split until each part has one element: [38], [27], [43], [3], [9], [82], [10].
       3. Merge pairs: [27, 38, 43] and [3, 9, 10, 82].
       4. Merge the final sorted arrays: [3, 9, 10, 27, 38, 43, 82].
       5.

4. **Heap Sort**:
    - **How It Works**: Heap Sort uses a binary heap data structure (usually a max heap) to sort an array in place. The largest element is repeatedly moved to the end of the heap, then removed from the heap, and the process continues until all elements are sorted.
    - **Performance**: Heap Sort has a time complexity of O(nlog n)O(n \log n)O(nlogn) and requires constant space for sorting in place, which makes it memory efficient. However, Heap Sort is not a stable sort, meaning it does not maintain the order of duplicate elements.
    - **Example**:
        - Given [26, 5, 37, 1, 61, 11, 59, 15, 48, 19]:
            1. Build a max heap.
            2. Swap the largest element 61 with the last element.
            3. Reduce the heap size by one and perform a heapify operation to maintain the max-heap property.
            4. Repeat until the array is sorted.

8.11 EXTERNAL SORTING

External sorting algorithms are used for data that cannot fit entirely in the main memory and must be stored on external devices, such as disks. Since accessing external memory is slow compared to accessing RAM, external sorting algorithms aim to minimize the number of I/O operations, which dominate the sorting time.

8.11.1 Introduction to External Sorting

External sorting is required when the data size exceeds main memory capacity, and the data resides in external storage. The primary goal is to reduce the number of times data must be read from and written to the external device, as these operations are time-consuming due to seek time (the time to locate data on the disk), rotational latency (waiting for the disk to spin to the correct position), and transmission time (the time to transfer data). The External Merge Sort is the most common external sorting method, where data is split into smaller, sorted chunks (runs) in memory and then merged in a sequence of passes.

8.11.2 K-way Merging

K-way merging extends 2-way merging by merging multiple sorted runs in one pass, which reduces the number of passes required to fully merge the data. This technique is beneficial in external sorting as it minimizes disk I/O operations.

- **Concept**: In a 2-way merging, two sorted arrays are combined into one sorted array in a single pass. With K-way merging, we merge K sorted arrays (or runs) simultaneously, requiring a single pass to process all K runs.
- **Example**:
    - Suppose we have 8 sorted runs: [1, 4, 7], [2, 5, 8], [3, 6, 9], and so on. A 4-way merger could combine these into one larger sorted run in fewer passes.
    - By using 4-way merging, 16 runs can be merged in only two passes, rather than four passes required for 2-way merging, which drastically reduces the I/O time.

- **Advantages**: By selecting a suitable K based on memory availability, K-way merging reduces the number of required passes, leading to faster overall sorting of large datasets.

8.11.3 Buffer Handling for Parallel Operation and Run Generation

Efficient buffer management is critical in external sorting, as it helps prevent delays associated with data loading and writing.

- **Buffer Usage**: For merging K runs, each run requires an input buffer. Additionally, an output buffer stores the merged data temporarily before it is written to the disk.
- **Double Buffering**: To improve efficiency, double buffering is employed. While one output buffer is being written to disk, the other buffer can continue receiving data from the merge process.
- **Example**: In a 4-way merge with double buffering, four input buffers would be used to read data from each run, while two output buffers are used to allow continuous merging and disk writing. This setup minimizes idle times as data is always available for processing.
- **Parallelization**: Using multiple buffers for input/output operations enables parallel data handling, allowing one buffer to load while another is processed. This reduces the waiting time between reading and writing operations, significantly enhancing the sorting speed in an external sorting environment.

8.11.4 Optimal Merging of Runs

Optimal merging involves constructing a merge tree that minimizes the total merging time, particularly when run sizes vary significantly. In external sorting, this is essential for balancing CPU and I/O costs.

- **Uneven Runs**: If runs are of uneven lengths, merging them without an optimized strategy can result in unnecessary merging passes, increasing the total merge time.
- **Optimal Strategy**: The Huffman coding algorithm provides a method for optimal merging by constructing a merge tree where runs closest to the root are of smaller sizes, minimizing their merge depth.
- **Example**:
    - Consider runs of lengths 2, 4, 5, and 15. Merging in a random order may result in inefficient merge operations. Using Huffman's algorithm ensures that smaller runs are merged early, minimizing the number of passes and reducing the merging cost.
- **Benefits**: Optimal merging ensures that runs are combined in the least number of passes, reducing the amount of data read/write operations. This is essential for large datasets in external storage, where minimizing I/O is key to performance.

8.12 KEY TERMS

Sorting, Searching, Sequential Search, Binary Search, List Verification, Sorting Algorithms, Insertion Sort, Quick Sort, Merge Sort, Heap Sort, Radix Sort.

8.13 SELF ASSESSMENT QUESTIONS

1. What are the differences between comparison-based and non-comparison-based sorting algorithms? Provide examples of each.
2. Explain the steps involved in performing a Quick Sort. What is its time complexity in the best and worst cases?
3. Describe the importance of stability in sorting algorithms. Which sorting algorithms covered are stable?
4. Define K-way merging and explain how it improves the efficiency of external sorting.
5. What are the advantages and limitations of Radix Sort compared to other sorting algorithms?

8.14 SUGGESTIVE READINGS

1. "Introduction to Algorithms" by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein – A comprehensive guide to algorithms, including sorting techniques and their complexities.
2. "Data Structures and Algorithm Analysis in C" by Mark Allen Weiss – Covers the implementation of sorting algorithms in C with detailed explanations.
3. "The Art of Computer Programming: Sorting and Searching" by Donald E. Knuth – A classic text focused on sorting and searching algorithms.
4. "Algorithms Unlocked" by Thomas H. Cormen – A beginner-friendly introduction to algorithms, including sorting and searching.
5. "Data Structures and Algorithms in Java" by Robert Lafore – Offers practical insights into sorting algorithms with Java examples.

Dr.Vasantha Rudramalla

Lesson – 9

Hashing Techniques

Objectives

The objectives of the lesson are:

- Understand the concept of hashing, hash functions, and the importance of collision resolution techniques in managing data efficiently.
- Explore different types of hashing methods (static and dynamic) and their real-world applications.
- Learn the significance of collision resolution strategies, such as chaining, open addressing, and dynamic hashing.
- Analyze the advantages, challenges, and trade-offs of various hashing techniques and their effectiveness in data management.

Structure

9.1 INTRODUCTION

Hashing is a technique used to map large datasets to specific locations in memory (usually in a data structure called a hash table) using a mathematical function called a hash function. This mapping transforms keys (such as names, identifiers, or numbers) into an index or "hash value" that corresponds to a specific slot in the hash table, where the data is stored. Hashing is widely used in scenarios requiring fast access, insertions, and deletions, such as managing symbol tables in compilers, caching, and database indexing.

The main advantages of hashing are its efficiency and speed. By using a hash function, hashing allows for direct access to data with a time complexity close to O(1), unlike traditional search algorithms, which may require O(log n) or O(n) time.

9.2 KEY CONCEPTS IN HASHING

1. **Hash Function**: A hash function takes an input (or "key") and produces a fixed-size string of bytes (or "hash value"). A good hash function distributes keys evenly across the hash table to minimize the number of collisions.
2. **Hash Table**: The hash table is an array-like data structure where the data is stored. The position of each element in the table is determined by the hash function.
3. **Collisions**: A collision occurs when two keys map to the same index in the hash table. Effective hashing minimizes collisions, but they are inevitable in fixed-size tables. Techniques such as **chaining** (where each slot points to a linked list of

elements) or **open addressing** (where alternative slots are found) are used to handle collisions.

9.3 DIFFERENCE BETWEEN HASHING AND TRADITIONAL SEARCH ALGORITHMS

Hashing and traditional search algorithms (such as linear search, binary search, and tree-based search) both aim to locate data, but they operate quite differently:

1. **Time Complexity**:
   - **Hashing**: Ideally provides constant time O(1) access for search, insertion, and deletion, as the hash function directly calculates the position of the data.
   - **Search Algorithms**: Time complexity varies based on the type of algorithm:
     - **Linear Search**: O(n) in unsorted lists.
     - **Binary Search**: O(log n) in sorted lists.
     - **Binary Search Tree**: Average O(log n), but can be O(n) in unbalanced trees.
2. **Data Structure Requirements**:
   - **Hashing**: Requires a hash table, and the performance depends on the quality of the hash function and the handling of collisions.
   - **Search Algorithms**: Can operate on various data structures, like arrays (for linear or binary search) or trees (for tree-based searches), without needing a separate hash function or collision handling.
3. **Use Case Scenarios**:
   - **Hashing**: Ideal for applications where the data size is known in advance and quick access is essential (e.g., caching, symbol tables, databases).
   - **Search Algorithms**: More flexible for dynamic datasets where data may not fit into a fixed-size table or where ordered structures (like binary trees) are needed for range queries or sorting.
4. **Memory Efficiency**:
   - **Hashing**: Can be less memory-efficient, especially when handling collisions through chaining or open addressing.
   - **Search Algorithms**: Memory usage varies, but algorithms like binary search in an array or search trees do not require additional structures like linked lists within hash table slots.

In summary, hashing is best for constant-time data access in stable datasets, while traditional search algorithms are versatile for dynamically structured data and scenarios requiring sorted or sequential access. Both have unique strengths, and choosing between them depends on the dataset size, access requirements, and memory constraints.

9.4 STATIC HASHING

Static hashing is a technique used to store and retrieve data efficiently by mapping each key to a specific location in a fixed-size hash table using a **hash function**. This method is ideal when the dataset size is fixed or changes infrequently. Each key is mapped to a bucket within the hash table, and if two keys hash to the same bucket (a **collision**), an overflow handling technique is used to resolve it.

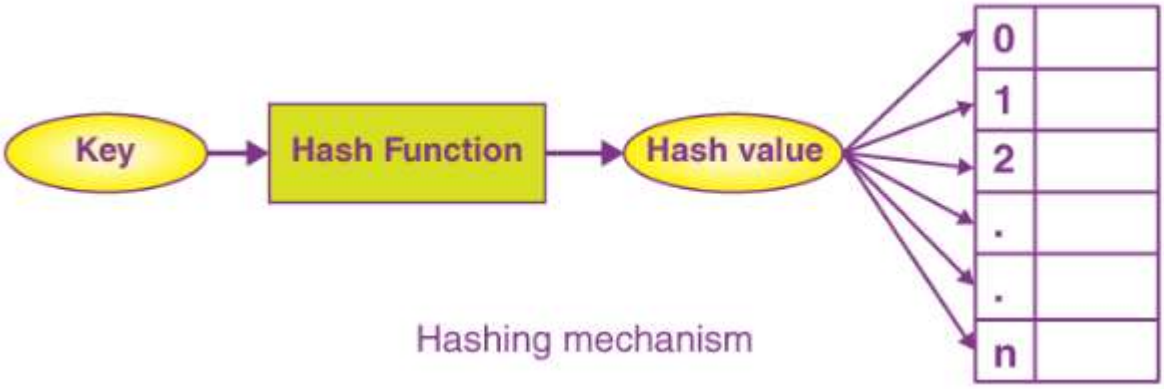


Fig 9.1. Process of Hashing

The above diagram illustrates the basic hashing mechanism, which involves the following steps:

1. **Key**:
   - The input data (or key) is provided to the hashing mechanism. This key could be any data, such as a number, string, or any other value that needs to be mapped to a specific location in a table.
2. **Hash Table**:
   - The hash table is represented as an array with slots (0, 1, 2, ..., n). Each index in the table corresponds to a potential storage location for the key-value pair. The hash value generated by the hash function maps the key to one of these slots.
3. **Hash Function**:
   - The key is processed by a hash function, which is a mathematical or logical algorithm designed to transform the key into a numerical value. This function ensures that the output (hash value) is within a certain range, usually the size of the hash table.
4. **Hash Value**:
   - The output of the hash function is the hash value. This value determines the index (or position) where the key-value pair will be stored in the hash table.
5. **Purpose**:
   - The hashing mechanism ensures efficient storage and retrieval of data. By converting the key into a hash value, the system can quickly locate the corresponding slot in the hash table, minimizing search time.

This process is fundamental to data structures like hash tables, which are widely used in scenarios requiring fast access, such as databases, caches, and indexing systems.

9.4.1 Hash Tables

A hash table is a data structure with a fixed number of buckets or slots where data entries are stored. Each bucket has a unique index, calculated using a hash function applied to the key. The hash function distributes entries across the table to allow quick access.

A hash table stores data in an array format, where each data item is associated with a unique key. The hash function processes this key to generate an index (or hash) that

represents a specific location in the array where the data will be stored. This design allows for efficient data retrieval by directly accessing the index associated with a key.

❖ **Key Components of a Hash Table**

1. **Array**: The primary storage of the hash table, where each position (or "bucket") in the array can store one or more entries.
2. **Keys and Values**: Each piece of data stored in a hash table has a key (often unique) that identifies it and a value (the data associated with the key).
3. **Hash Function**: A function that takes the key as input and computes an index in the array where the corresponding value should be stored.
4. **Collision Handling Mechanisms**: Since different keys may produce the same index (a collision), hash tables need a way to handle these collisions, such as chaining or open addressing.

❖ **How Hash Tables Work**

1. **Hashing**: The process begins by applying a hash function to the key. For example, if the key is a string (e.g., "name"), the hash function converts it into an integer that serves as an index in the array.

   For example there is a simple hash function for which

   hash(key)=ASCII sum of characters % table size

   For the key "Alice" and a table size of 10, the hash function could compute:

   hash ("Alice") = (65 + 108 + 105 + 99 + 101) % 10 = 478%10 = 8

   This means "Alice" would be stored at index 8 in the hash table.
2. **Insertion**: Once the hash function generates an index, the hash table places the value at that index. If the location is already occupied (collision), the hash table uses a collision resolution method to determine the next available position.
3. **Searching**: To retrieve a value, the hash function recomputes the index using the key. The hash table then accesses the data directly at that index, allowing for a time complexity of O(1) for most operations.

❖ **Visual Representation of a Hash Table**

Consider a hash table with a size of 10, and suppose we are storing the following key-value pairs:

Table 9.1. Hash Table

Key	Value
Alice	Engineer
Bob	Doctor
Carol	Teacher
Dave	Designer
Eve	Architect

Using the hash function:

hash (key) = ASCII sum of characters % 10

Here's a step-by-step illustration of how each entry would be added:

Step 1: Calculating Hash Values

1. **Alice**: ASCII sum = 478, 478 % 10 = 8
2. **Bob**: ASCII sum = 275, 275 % 10 = 5
3. **Carol**: ASCII sum = 489, 489 % 10 = 9
4. **Dave**: ASCII sum = 309, 309 % 10 = 9 (collision with Carol)
5. **Eve**: ASCII sum = 312, 312 % 10 = 2

Step 2: Inserting Data and Handling Collisions

Table 9.1.Collisions in Hash Table

Index	Value
0	
1	
2	Eve: Architect
3	
4	
5	Bob: Doctor
6	
7	
8	Alice: Engineer
9	Carol: Teacher, Dave: Designer (handled by chaining)

In this example, Carol and Dave both hash to index 9, resulting in a collision. Using chaining, both entries are stored at index 9 in a linked list format.

Example of a Hash Table:

Consider a hash table with 10 buckets as below, labeled from 0 to 9, and the following keys: 15, 25, 35, 45.

Bucket	Keys
Bucket 0	
Bucket 1	
Bucket 2	
Bucket 3	
Bucket 4	
Bucket 5	15, 25, 35, 45
Bucket 6	
Bucket 7	
Bucket 8	
Bucket 9	

Explanation:

1. Buckets 0 through 9 represent the slots in the hash table.
2. Keys 15, 25, 35, and 45 all hash to bucket 5 using the hash function h(x)=x mod 10, resulting in collisions in bucket 5.
3. Hash Function**:** Assume the hash function is h(x)=x mod 10
4. Bucket Assignments:
   - For key 15: 15mod 10=515 \mod 10 = 515mod10=5 → Place 15 in bucket 5.
   - For key 25: 25mod 10=525 \mod 10 = 525mod10=5 → Collision in bucket 5.
   - For key 35: 35mod 10=535 \mod 10 = 535mod10=5 → Another collision.
   - For key 45: 45mod 10=545 \mod 10 = 545mod10=5 → Another collision.

9.4.2 Hashing Functions

A **hash function** is a mathematical algorithm that transforms a given input (known as a "key") into an index, often called a "hash code" or "hash value." This index then corresponds to a specific location (or "bucket") in a hash table where the data associated with that key will be stored. The purpose of a hash function is to provide fast data access by mapping keys to locations in a predictable, repeatable manner.

The effectiveness of a hash function is measured by its ability to distribute keys uniformly across the hash table, minimizing the number of collisions (where multiple keys map to the same location). A good hash function ensures that each bucket in the table is equally likely to be used, leading to efficient data retrieval and storage.

❖ **Properties of a Good Hash Function**

1. **Deterministic**: A hash function must always produce the same output for the same input. This property ensures that keys are consistently mapped to the same location in the hash table, allowing reliable data retrieval.
2. **Uniform Distribution**: A good hash function spreads keys evenly across the hash table to prevent clustering. This minimizes the number of collisions, which can otherwise slow down data retrieval.
3. **Efficiency**: The hash function should be computationally efficient to ensure fast performance, especially when dealing with large datasets. Ideally, it should compute the hash value in constant time, O(1)O(1)O(1).

4. **Minimizing Collisions**: While collisions are inevitable in finite-sized hash tables, a good hash function reduces their occurrence by distributing keys evenly. When collisions do occur, efficient collision handling techniques, such as chaining or open addressing, manage them.
5. **Low Sensitivity to Key Patterns**: The function should handle diverse key patterns well, avoiding any biases in key distribution. For example, if keys follow a particular sequence or pattern, a good hash function should still distribute them evenly across the hash table.

❖ **Types of Hash Functions**

Different hash functions are used depending on the nature of the data and application requirements. Here are several common types of hash functions:

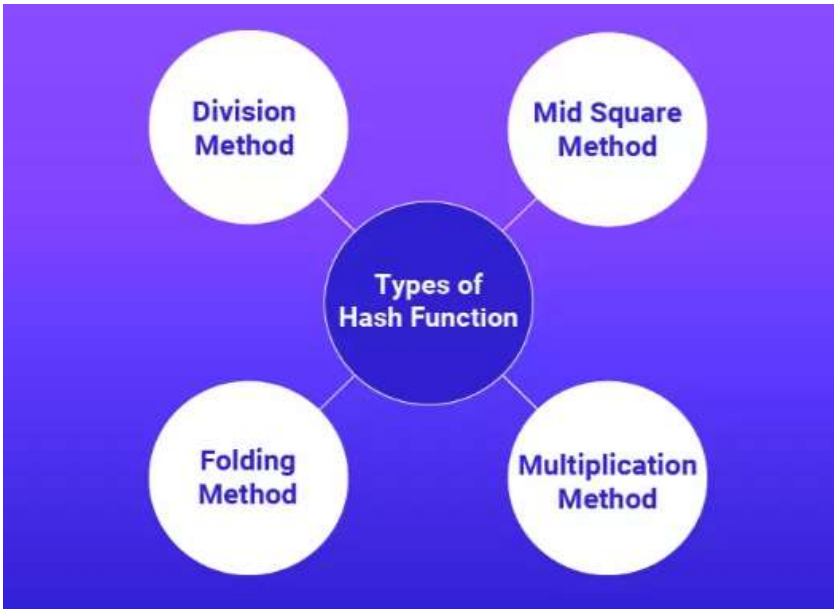


Fig 9.2. Hash Functions

1.Division (Modulo) Method:

- **Description**: The division method calculates the hash value by dividing the key by the size of the hash table and taking the remainder. The formula is:

  $$h(x) = x \bmod M$$

  where x is the key and M is the number of buckets (often chosen as a prime number to ensure better distribution).

  Example: For a hash table with M=10M buckets, the key 25 would hash as:

  $$h(25)=25 \bmod 10 = 5$$ so 25 would be placed in bucket 5..

- **Considerations**: Choosing M as a prime number helps avoid patterns in key distributions and improves even spreading across buckets. For example, if all keys are multiples of a specific number and M is a factor of that number, clustering may occur.

2. Mid-Square Method:

- **Description**: In this method, the key is squared, and the middle digits of the result are taken as the hash value. This technique is effective because squaring the key spreads the digits out, helping with uniform distribution.
- **Example**: If the key is 56, squaring it gives $56^2 = 3136$. Extracting the middle two digits, 13, we could use bucket 13 (or reduce it further if the table has fewer than 100 buckets).
- **Considerations**: This method is useful when keys have a similar pattern or are close in value, as squaring spreads the values and avoids clustering.

3.Folding Method:

- **Description**: The folding method splits the key into equal parts (often based on digits), and then these parts are added or XORed together to form the hash value.

  E**xample**: For a key 123456, divide it into parts 123 and 456, then add: 123+456=579 Taking 579 mod M (assuming M is the number of buckets), we can place the entry in the resulting bucket.
- **Considerations**: Folding is particularly effective for large keys, like account numbers, as it simplifies them into smaller values suitable for hashing.

4. Multiplicative Method:

- **Description**: This method multiplies the key by a constant A (where $0<A<1$), extracts the fractional part, and then scales it to fit within the table size. The formula is:

  $$h(x)=\lfloor M\cdot(x\cdot A \bmod 1)\rfloor$$

  where M is the table size, and A is often chosen as an irrational number like

  $$A=. (\sqrt{5}-1)/2.$$

- **Example**: For a key 123 and A=0.618033, compute

  $$h(123)= 10\cdot(123\cdot 0.618033 \bmod 1)$$

- **Considerations**: This method is less sensitive to patterns in keys and can produce a good distribution, although it requires more computation.

9.4.3 Choosing an Appropriate Hash Function

Selecting the right hash function depends on the dataset and its characteristics:

1. **Uniformity of Key Distribution**: For evenly distributed data (e.g., random numbers), simple hash functions like the division method work well. For datasets with clustered or patterned keys (e.g., sequential numbers), the mid-square or multiplicative methods may provide better results.

2. **Efficiency Needs**: Some hash functions are computationally simpler (e.g., division method) and are preferred for performance-critical applications, while more complex methods may be used for high-stakes data integrity where even distribution is crucial.
3. **Memory Constraints**: When memory is limited, a hash function with minimal overhead, such as the division method, is advantageous.

9.4.4 Evaluating Hash Function Effectiveness

The effectiveness of a hash function is measured by how well it minimizes collisions and distributes keys evenly. Testing a hash function involves running it on a sample dataset and analyzing the distribution of entries in the hash table. Metrics include:

1. **Load Factor**: The load factor, α=nM\alpha = \frac{n}{M}α=Mn, where nnn is the number of keys and MMM is the number of buckets, indicates how full the table is. Higher load factors increase the likelihood of collisions, impacting performance. Effective hash functions maintain an even spread across the table even at moderate load factors.
2. **Collision Rate**: By testing the number of collisions for a given set of keys, one can evaluate how well the hash function distributes keys. A lower collision rate indicates a more effective hash function.
3. **Performance in Different Scenarios**: Some hash functions perform well in specific scenarios. For example, mid-square is useful when keys are sequential, while double hashing is preferred for reducing clustering in open addressing.
4. **Empirical Testing**: Empirical testing on representative datasets can help evaluate the hash function's performance. Hash functions should be tested on both typical and edge-case data to ensure they perform well under various conditions.

9.4.5 Practical Examples of Hash Function Usage

1. **Database Indexing**: Hash functions in database indexing provide quick access to rows in large datasets, where each entry's primary key is mapped to a hash table location for efficient retrieval.
2. **Symbol Tables in Compilers**: Compilers use hash functions to manage symbol tables, where variable names are hashed to specific locations, allowing quick lookup of variable attributes during code compilation.
3. **Data Caching**: In caching mechanisms, hash functions determine the memory location for cached data, enabling quick retrieval of frequently accessed information.
4. **Load Balancing in Networking**: Hash functions can distribute incoming requests evenly across servers in load balancing systems, reducing the chance of overloading any single server.

In summary, hash functions are fundamental to the performance and efficiency of hash tables and are chosen based on the nature of the dataset, desired performance characteristics, and constraints of the application. A well-chosen hash function ensures fast access times, reduced memory usage, and an overall more efficient system for data management.

9.4.6 Collision Handling

In static hashing, collisions occur when multiple keys hash to the same bucket. Overflow handling techniques resolve these conflicts to ensure that all data can be stored in the table.

Collision Handling Techniques

1. **Chaining**: Uses a linked list for each bucket. When a collision occurs, the new entry is added to the linked list in that bucket.
    - **Example**: In a hash table where 15, 25, and 35 all hash to bucket 5, chaining would store these in a linked list within bucket 5, like 15 -> 25 -> 35.

    Diagram: A hash table with a linked list in bucket 5 containing entries 15, 25, and 35 demonstrates how chaining handles collisions.

2. **Open Addressing**: Searches for alternative slots in the table when a collision occurs. Common methods include:
    - **Linear Probing**: Checks the next bucket sequentially until an empty one is found.
        - **Example**: For key 25 colliding at bucket 5, linear probing places it in the next open bucket, say bucket 6.
    - **Quadratic Probing**: Checks in a quadratic sequence to reduce clustering.
    - **Double Hashing**: Uses a second hash function to calculate the step size for probing.

    Diagram: Show a hash table where linear probing resolves a collision by placing an entry in the next available bucket. Double hashing could be shown by illustrating how a secondary function provides a step size to locate an alternative bucket.

3. **Separate Overflow Area**: Stores overflowed entries in a separate memory region, keeping the main table uncluttered.
    - **Example**: If bucket 5 is full, additional entries for bucket 5 (like 35) are stored in the overflow area rather than the main table.

    Diagram: A hash table with a designated overflow area beside it would help illustrate how entries that cannot be stored directly in the main table are managed separately.

Collision resolution techniques are methods used in hash tables to handle cases where two or more keys produce the same index (or hash value). This situation, called a collision, is common in hash tables, especially when the number of keys exceeds the table's capacity or if the hash function generates the same index for different keys. Effective collision resolution is crucial to maintain the performance of a hash table, allowing it to achieve optimal search, insertion, and deletion times.

9.5 COLLISION RESOLUTION TECHNIQUES

Collisions occur due to the limitation of hash functions, where multiple keys can sometimes hash to the same index. For instance, if a hash table has only 10 slots (indices 0-9), and the hash function maps keys based on the remainder of division by 10, then the keys 15 and 25 will both hash to index 5, creating a collision.

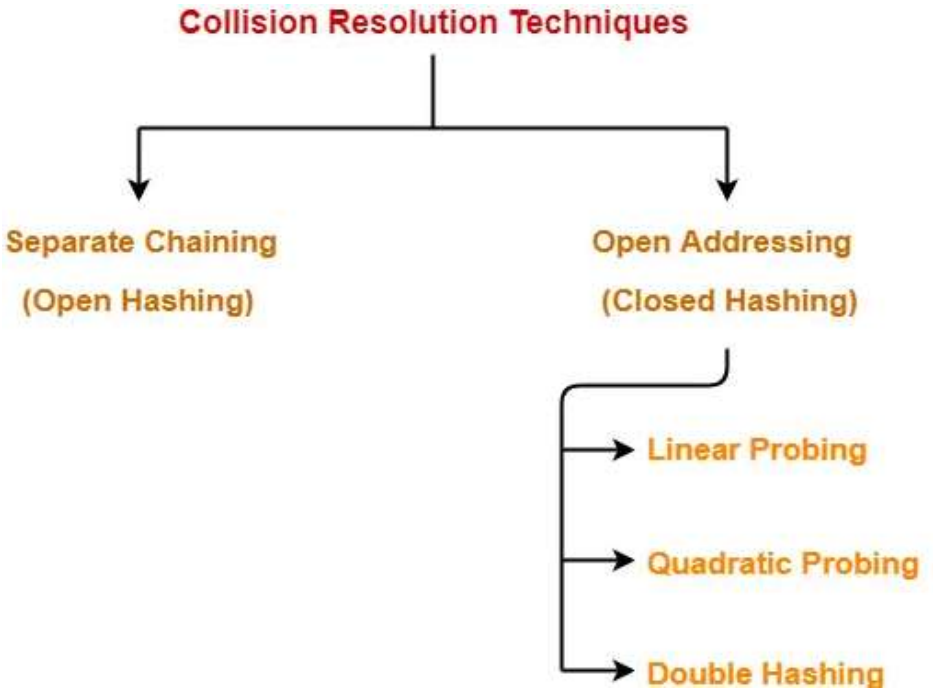


Fig 9.3. Collision Resolution Techniques

The two main types of collision resolution techniques that are represented in the above diagram. They are:

1. **Open Addressing**
2. **Chaining**

Each method has various sub-techniques and offers different trade-offs in terms of time complexity, memory usage, and ease of implementation.

9.5.1 Open Addressing

In open addressing, if a collision occurs, the hash table itself is searched for the next available slot. This method avoids using extra data structures (like linked lists in chaining) and keeps all entries within the hash table array itself. The primary open addressing methods include:

a. Linear Probing

With linear probing, when a collision occurs, the algorithm searches sequentially (linearly) through the table, starting from the original hash position, to find the next available slot.

- **How It Works**:
  - If the hashed index iii is occupied, linear probing checks i+1i+1i+1, i+2i+2i+2, and so on, wrapping around to the start of the array if necessary.
  - This process continues until an empty slot is found.
- **Example**:
  - Suppose we have a hash table with 10 slots, and a hash function that maps keys based on the remainder modulo 10.
  - We attempt to insert keys 10, 20, and 30, all of which hash to index 0.
  - With linear probing:
    - Key 10 goes to index 0.
    - Key 20 encounters a collision at index 0 and moves to index 1.
    - Key 30 encounters collisions at indexes 0 and 1 and is placed at index 2.

- **Diagram of Linear Probing**:

Index:	0	1	2	3	4	5	6	7	8	9
Values:	10	20	30	-	-	-	-	-	-	-

- **Pros**: Simple to implement and doesn't require extra data structures.
- **Cons**: Can lead to clustering, where a group of occupied slots forms, increasing search time for new slots.

b. Quadratic Probing

Quadratic probing reduces clustering by checking the next available slot in a non-linear (quadratic) sequence. Instead of moving one slot at a time, it moves by increasing intervals (e.g., 1, 4, 9, 16…)

- **How It Works**:

If the hashed index iii is occupied, quadratic probing checks $i + 1^2$, $i + 2^2$, $i + 3^2$,... and so forth, wrapping around if necessary.

- **Example**:

For keys 10, 20, and 30:

  - Key 10 is placed at index 0.
  - Key 20 encounters a collision at index 0, so it moves to $0+1^2=1$
  - Key 30 encounters collisions at indexes 0 and 1, so it moves to $0+2^2= 4$

Index:	0	1	2	3	4	5	6	7	8	9
Values:	10	20	-	-	30	-	-	-	-	-

- **Pros**: Reduces primary clustering.
- **Cons**: May still experience secondary clustering, where certain sequences of probes lead to repeatedly used positions.

c. Double Hashing

Double hashing uses a secondary hash function to determine the interval between probes. This technique further reduces clustering by making the step size variable based on the key.

- **How It Works**:
  - If the initial index iii is occupied, double hashing uses a second hash function to determine the step size (e.g., if the second hash function gives 3, the next position will be i+3, then i+6 etc.).
- **Example**:
  - Hash function 1 (primary): hash1(key)=key%10
  - Hash function 2 (secondary): hash2(key)= key%7
  - For key 10, place it at index 0.
  - For key 20, if index 0 is occupied, use the second hash function to move by 4 slots (next slot at index 4).

Double Hashing can be represented as below

Index:	0	1	2	3	4	5	6	7	8	9
Values:	10	-	-	-	20	-	-	-	-	-

- **Pros**: Effectively eliminates clustering.
- **Cons**: Requires careful selection of hash functions to ensure effective distribution.

9.5.2 Chaining or Open hashing

Chaining involves storing multiple entries at the same index using a secondary data structure, typically a linked list. When multiple keys hash to the same index, they are linked together in a list at that index. Chaining is represented in Fig 4.

❖ **How It Works**

- Each index in the hash table points to a linked list or another dynamic data structure.
- When a collision occurs, the new entry is simply appended to the linked list at the index.
- For retrieval, the algorithm searches the linked list at the hashed index to find the correct entry.

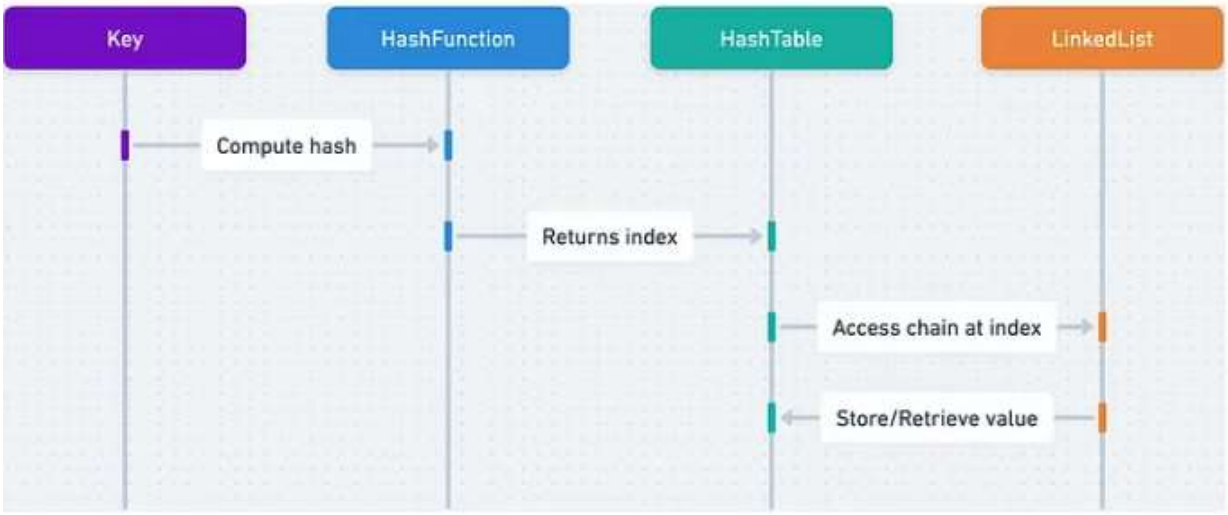


Fig 9.4. Chaining using Linked List

❖ **Example of Chaining**

- Suppose we insert keys 15, 25, and 35 into a hash table of size 10 with a hash function that computes key % 10.
- All three keys hash to index 5, creating a collision.
- Using chaining, all three entries are stored in a linked list at index 5.

❖ **Types of Chaining Structures**

1. **Linked List**: The simplest form, where each entry at a given index points to the next in a singly or doubly linked list.
2. **Binary Search Tree (BST)**: For faster searching within a chain, each index could use a BST rather than a list, allowing O(log n)O(\log n)O(logn) search time within chains.
3. **Dynamic Array**: Some implementations may use a dynamic array instead of a linked list, potentially improving access times when the chain is small.

❖ **Pros and Cons of Chaining**

- **Pros**:
  - Efficiently handles collisions by allowing multiple entries at each index.
  - Avoids clustering issues that can occur in open addressing.
  - Simple to implement and works well even when the load factor (entries per slot) is high.
- **Cons**:
  - Extra memory is needed for pointers or linked list nodes, increasing overhead.
  - Search times can degrade if chains become long, especially if the hash function distributes keys unevenly.

Table 9.3. Collision Resolution Techniques

Collision Resolution Technique	Description	Pros	Cons
Linear Probing	Sequentially search for the next available slot.	Simple, low memory overhead	Clustering, potential for long probe sequences
Quadratic Probing	Checks slots based on a quadratic sequence.	Reduces clustering, improves distribution	Secondary clustering, more complex probe calculation
Double Hashing	Uses a second hash function to calculate probe intervals.	Minimizes clustering, good distribution	Requires careful selection of hash functions
Chaining	Uses linked lists at each index to store multiple entries.	Flexible with high load factors, avoids clustering	Higher memory overhead, slower access in long chains

All the collision resolution techniques that are discussed in the above sections are summarized in the above table.

9.6 REAL-WORLD APPLICATION EXAMPLE

In a library database, each book could be uniquely identified by an ISBN number. Using a hash table, the ISBN serves as the key, and information like title, author, and location is stored as the value.

- **Hash Function**: Computes a hash based on the ISBN number to find an index in the hash table.
- **Collision Resolution**:
  - **Chaining**: Multiple books with similar ISBN prefixes (e.g., all published by the same publisher) may hash to the same index. Chaining stores these books in a linked list, allowing easy retrieval.
  - **Open Addressing**: Double hashing could be used to spread out books with similar ISBNs across different slots in the table.

9.7 THEORETICAL EVALUATION OF OVERFLOW TECHNIQUES

The effectiveness of overflow handling techniques is evaluated based on load factor (the ratio of entries to table size) and the average number of probes required for retrieval or insertion. Each technique has trade-offs:

1. **Chaining**:
   - **Average Comparisons**: The number of comparisons in chaining grows linearly with load factor. For a successful search, the expected time complexity is proportional to $1+\alpha/2$, where α \ alphaα is the load factor.

- **Strengths and Weaknesses**: Chaining performs well in dense tables and allows flexibility with growing lists, though longer lists in each bucket may slow down retrieval.

Diagram: A graph showing the increase in search comparisons with load factor for chaining, illustrating how retrieval becomes more costly as the load factor grows.

2. **Open Addressing**:
   - **Linear Probing**: Can suffer from clustering, increasing search time as the table fills up. Average search time increases significantly at high load factors.
   - **Quadratic Probing and Double Hashing**: Reduce clustering effects and improve performance but add complexity.

Diagram: A comparative line graph could illustrate how linear probing, quadratic probing, and double hashing each handle search times as the load factor increases.

3. **Separate Overflow Area**:
   - **Effectiveness**: This approach keeps the main table clear of overcrowded buckets, maintaining optimal search times for non-overflowed entries. However, retrieving overflowed entries may require additional processing.

Diagram: A schematic showing the main table and overflow area, with arrows indicating retrieval paths for entries stored in overflow.

Static hashing is a highly effective technique for quick data retrieval when data size is predictable. By carefully selecting hash functions and overflow handling techniques, static hashing can maintain efficient performance and manage collisions. Chaining is generally well-suited for dynamic or high-load applications, while open addressing can offer efficient in-place storage for static datasets. Each method's effectiveness varies based on the application's needs, the load factor, and memory constraints. The summary of all the above discussed topics is represented in the below diagram.

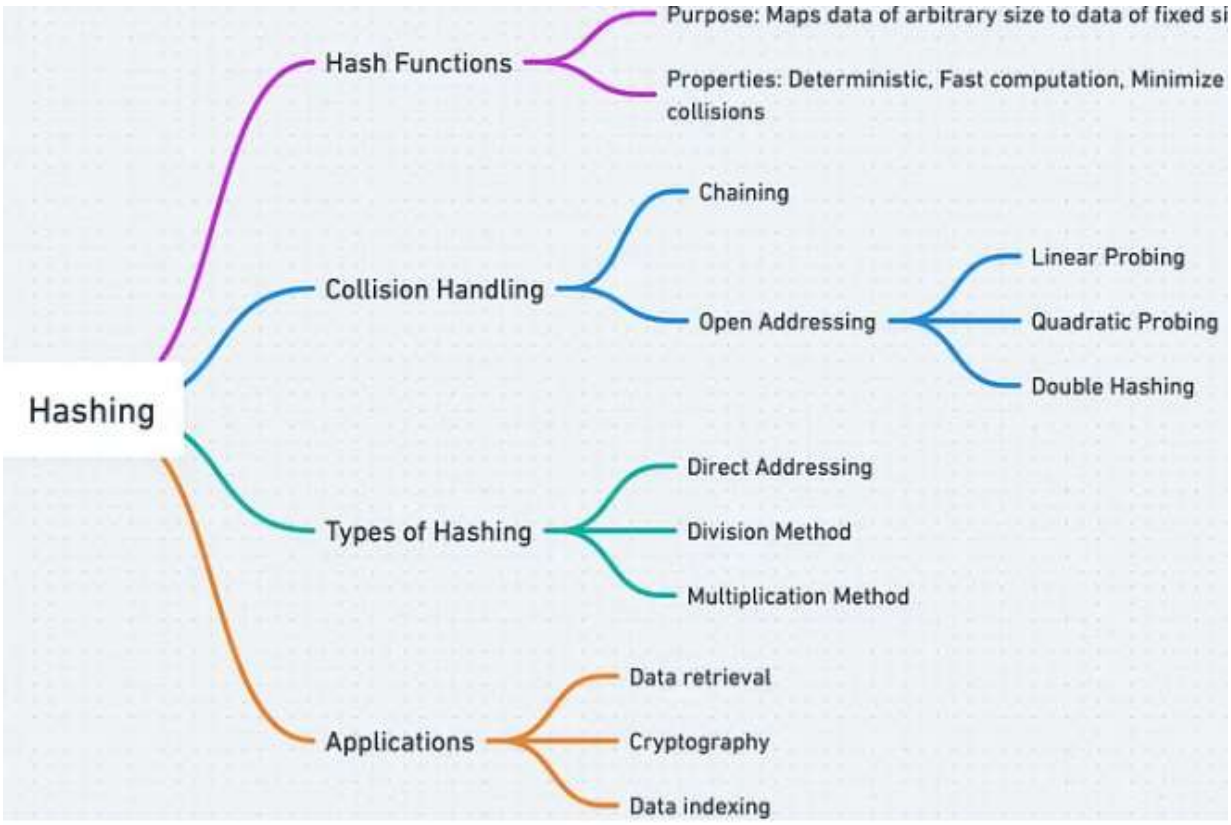


Fig 9.5. Flow diagram of Hashing

9.8 DYNAMIC HASHING

Dynamic hashing is a technique used to manage hash tables that can expand or shrink depending on the data load. Unlike static hashing, where the size of the hash table is fixed and overflow handling (like chaining or open addressing) is needed, dynamic hashing allows the table structure itself to grow. This makes it especially efficient for applications dealing with large, unpredictable datasets.

Dynamic hashing solves many problems of static hashing by:

- **Reducing collisions** through resizing as the number of elements increases.
- **Handling data overflow** by automatically adjusting the structure instead of relying on chaining or overflow areas.
- **Efficiently using memory**, as the table only expands when necessary, leading to better space utilization.

9.8.1 Key Operations:

1. **Insertion**: The system checks if a collision occurs when a new item is added. If so, a new hash structure (such as an additional bucket) is allocated, and items are redistributed as needed.
2. **Search**: Similar to static hashing, a hash function maps a search key to a bucket. However, with dynamic hashing, the structure may point to different buckets based on the current size of the hash table.

3. **Deletion**: After removing an item, dynamic hashing checks if the hash table can shrink to free up unused space.

9.8.2 Dynamic Hashing Using Directories

In directory-based dynamic hashing, a directory acts as an intermediary layer between hash values and actual data storage buckets. This directory is essentially a table that maps hashed keys to specific storage locations. This structure allows efficient bucket splitting and management when resizing is required.

How It Works:

- **Directory as an Index**: Each entry in the directory points to a specific bucket. The directory's size can dynamically grow, allowing the system to handle more elements without drastically increasing the number of collisions.
- **Bucket Splitting**: When a bucket overflows, the directory grows, often by doubling its size. Each entry is redistributed according to the new hash function that accounts for the expanded directory. This minimizes rehashing.
- **Expansion Control**: To control expansion, a *depth* parameter may be used, which adjusts the directory size based on how many buckets exist and their current load.

Example of Process:

1. **Initial Setup**: Start with a directory of size 2 (e.g., 00, 01) and two buckets. Each bucket can handle a certain number of records.
2. **Insertions**: If an insertion leads to an overflow in a bucket, the directory expands to accommodate this.
3. **Resizing**: The directory doubles in size (00, 01, 10, 11) when buckets need to be split, redistributing the existing keys accordingly.

Advantages:

- **Scalability**: The directory allows the hash table to grow smoothly, avoiding frequent collisions and making lookups efficient.
- **Memory Efficiency**: Memory is allocated as needed, making it more efficient compared to a static hash table with a high load factor.

Limitations:

- **Memory Overhead**: The directory itself requires memory, which can be a concern for extremely large datasets.

9.8.3 Analysis of Directory-Based Dynamic Hashing

Analyzing directory-based dynamic hashing involves understanding the impact on performance, memory usage, and overall efficiency. Here are some key points for analysis:

Space Complexity

- **Directory Size**: As the hash table grows, the directory size also increases. This could lead to more memory usage compared to directory-less dynamic hashing but provides flexibility and efficient handling of collisions.
- **Bucket Space**: Each bucket only expands when necessary, optimizing space and reducing memory wastage.

Time Complexity

- **Lookup**: On average, lookups remain efficient, generally taking O(1)O(1)O(1) time due to the direct mapping in the directory. However, when resizing is necessary, the system temporarily slows down.
- **Insertion and Deletion**: Typically fast but may slow during bucket splitting or resizing. Resizing is optimized to be infrequent, minimizing performance impacts.

Efficiency in Real-World Applications

- **Handling Dynamic Data**: For applications with unpredictable data sizes, directory-based dynamic hashing is highly efficient. It grows to meet demand without major redesign.
- **Collisions**: The directory helps manage and reduce collisions compared to static hashing or directory-less hashing.

Diagrammatic Representation

1. **Initial State**: Start with a small directory pointing to a couple of buckets.
2. **Growth Phase**: When a bucket reaches capacity, the directory expands, potentially redistributing existing data to balance load.
3. **Expanded State**: Show a larger directory structure with multiple buckets, each efficiently managing a subset of data items.

9.8.4 Directory-less Dynamic Hashing

Directory-less dynamic hashing removes the intermediate directory, relying directly on a scalable hash table. Without a directory, the hash table itself handles bucket allocation and resizing directly.

❖ **Structure and Functionality**

- **Direct Mapping**: Every key is hashed to a specific bucket without any directory pointers. This makes the data structure leaner but requires more careful handling of collisions and resizing.
- **Bucket Splitting**: When a bucket becomes full, the hash table expands by adding new buckets. Records are then redistributed according to an updated hash function to balance the load.

❖ **Advantages**

- **Reduced Memory Overhead**: No additional memory is required for a directory, leading to lower overall space requirements.
- **Simplicity**: The structure is simpler without a directory, making it easier to manage and implement.

❖ **Challenges**

- **Frequent Resizing**: Without a directory to manage bucket pointers, resizing may involve extensive rehashing and data redistribution.
- **Potential Collisions**: Higher collision rates may occur if the hash function isn't adaptable enough to handle varying data loads.

❖ **Use Cases**

Directory-less hashing is suitable for applications where memory overhead needs to be minimized. However, for very large and dynamic datasets, it may struggle with performance due to increased rehashing.

❖ **Example of Process**

1. **Hash Function Application**: Each key is hashed directly to a specific bucket.
2. **Bucket Full**: If a bucket fills up, the hash table expands by adding more buckets and redistributing keys.
3. **Rehashing**: After every expansion, a new hash function is applied to minimize collisions and balance the data load across buckets.

❖ **Comparisons**

- **Directory-less Versus Directory-Based Hashing**: While directory-based hashing requires additional memory, it provides greater control and efficiency in managing load distribution. Directory-less hashing, however, is more compact but may face performance issues with frequent rehashing under heavy loads.

9.9 KEY TERMS

Hashing, Hash Function, Hash Table, Collisions, Open Addressing, Chaining, Linear Probing, Quadratic Probing, Double Hashing, Dynamic Hashing, Directory-Based Hashing, Directory-Less Hashing, Load Factor.

9.10 SELF-ASSESSMENT QUESTIONS

1. What is hashing, and how does it help in efficient data management?
2. Explain the difference between static and dynamic hashing.
3. What are collision resolution techniques, and how does chaining differ from open addressing?
4. How does the load factor influence the performance of hash tables?
5. Compare directory-based and directory-less dynamic hashing. What are their advantages and challenges?

9.11 SUGGESTIVE READINGS

1. "Data Structures and Algorithm Analysis in C" by Mark Allen Weiss.
2. "Introduction to Algorithms" by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein.
3. "The Art of Computer Programming, Volume 3: Sorting and Searching" by Donald E. Knuth.
4. "Hashing Techniques and Applications" by M. Ramakrishna and Dhiraj Kamble.
5. "Data Structures Using C" by Aaron M. Tenenbaum, Yedidyah Langsam, and Moshe J. Augenstein.

Mrs. Appikatla Pushpa Latha

Lesson - 10
Binary Trees

OBJECTIVES

The objectives of the lesson are:

1. To understand the structure, characteristics, and applications of binary trees.
2. To learn different types of binary trees, such as complete, full, balanced, and skewed binary trees, along with their properties.
3. To explore array and linked list representations of binary trees and their respective advantages and limitations.
4. To develop a clear understanding of traversal, insertion, deletion, and indexing operations in binary trees.

STRUCTURE

10.1 INTRODUCTION

In computer science, a tree is a fundamental data structure used to represent hierarchical relationships. Unlike linear structures such as arrays or linked lists, trees organize data in a branching, non-linear form. Trees consist of nodes connected by edges, with each node potentially having multiple children, forming a natural parent-child hierarchy. This unique structure makes trees ideal for representing hierarchical data such as file systems, organizational charts, family trees, and decision-making processes.

10.2 KEY CHARACTERISTICS OF TREES

1. **Root Node**: The topmost node in a tree is called the root. It serves as the starting point, and every other node in the tree is a descendant of this root node.
2. **Nodes and Edges**: Each element in a tree is called a node, and connections between nodes are called **edges**. Nodes can store data and references to other nodes (children), and edges define relationships between parent and child nodes.
3. **Parent-Child Relationships**: Each node can have zero or more child nodes, and every node (except the root) has one parent. This parent-child relationship naturally forms the hierarchical structure of a tree.
4. **Leaf Nodes**: Nodes with no children are called leaves or leaf nodes. They represent the endpoints of the tree.
5. **Height and Depth**:
   - Height of a tree is the longest path from the root to any leaf.
   - Depth of a node is the distance from the root to that node.
6. **Subtrees**: Any node in a tree, along with all its descendants, can be considered a subtree. Trees are recursive structures, where each subtree is itself a smaller tree.

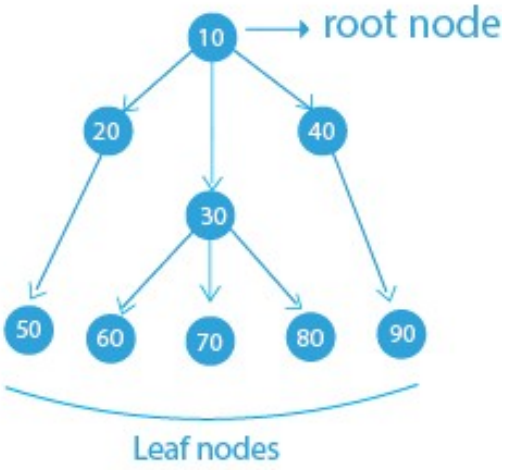


Fig 10.1. Structure of Tree data structure

10.3 TYPES OF TREES

1. **Binary Trees**: A tree where each node has at most two children (left and right). Binary trees are widely used in computer science for their simplicity and efficiency.
2. **Binary Search Trees (BSTs)**: A binary tree where each node's left subtree contains values less than the node, and the right subtree contains values greater than the node. BSTs support efficient search, insertion, and deletion operations.
3. **Balanced Trees**: Trees like AVL trees or Red-Black trees that maintain a balanced structure to keep operations efficient.
4. **Heaps**: A special type of binary tree used to implement priority queues. In a max heap, each parent node is greater than its children; in a min heap, each parent is less than its children.
5. **B-Trees and B+ Trees**: Self-balancing trees used in databases and file systems to manage large amounts of data by keeping operations fast even with disk-based storage.

10.4 INTRODUCTION TO BINARY TREES

Binary trees are a foundational concept in computer science, extensively used in algorithms, data structures, and various applications where hierarchical data organization is required. Their structure, consisting of nodes with at most two child nodes, is both simple and efficient, providing a balance between flexibility and performance. Binary trees allow efficient searching, insertion, deletion, and traversal, making them ideal for use in databases, operating systems, and even AI decision-making processes.

10.4.1 Binary Tree

A binary tree is a tree data structure in which each node has at most two children, commonly referred to as the left and right child. The "binary" in binary trees indicates the presence of only two children per node, as opposed to general trees, where nodes can have multiple children.

Binary trees have an ordered structure, meaning that each node has a defined "left" and "right" position. This ordered arrangement of nodes is a defining characteristic of binary trees, enabling efficient operations such as searching and sorting.

A binary tree begins with a **root node** at the top, and every node in the tree has a unique position. Below the root node, each child node recursively forms the "subtree" of its parent, with each subtree itself being a binary tree. This recursive nature allows for simple yet powerful manipulation and traversal algorithms.

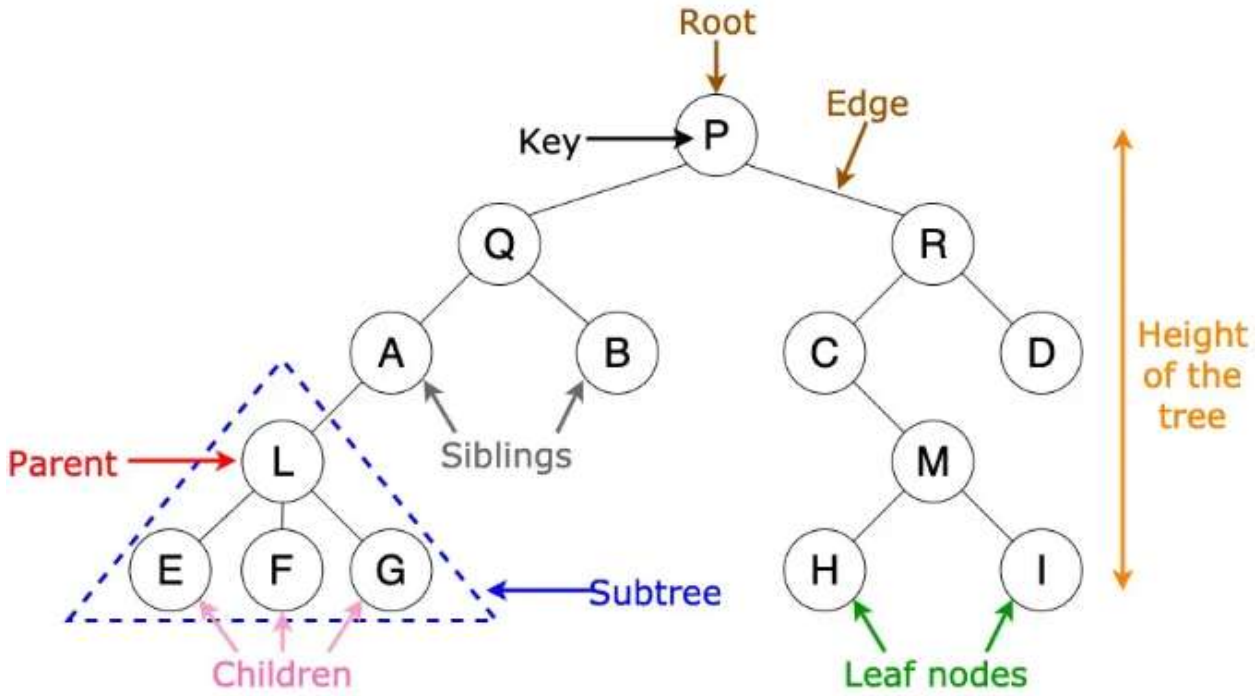


Fig 10.2. Structure of a Binary Tree

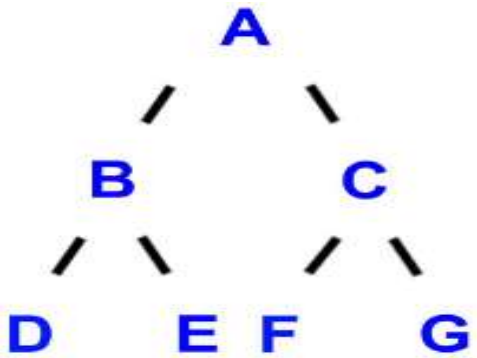
10.4.2 Types of Binary Trees

Binary trees can be classified into several specialized types based on their structural properties, such as balance and fullness. These types are optimized for specific use cases and operations. They are discussed below

1. Full Binary Tree

A full binary tree, also known as a proper binary tree, is a type of binary tree in which every node has either zero or two children. This structure maximizes the number of nodes and minimizes the tree's height, which is beneficial for certain applications that require balanced data retrieval.

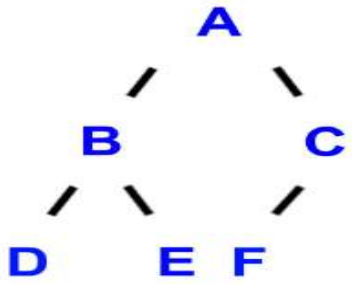
Representation:



2. Complete Binary Tree

In a complete binary tree, all levels are fully filled except possibly the last level, which is filled from left to right. This type is ideal for use in heaps, where completeness ensures efficient memory storage and allows for operations based on the structure of the array.

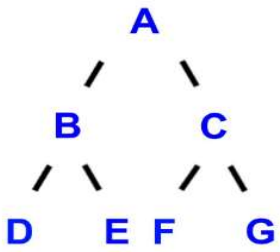
Representation:



3. Perfect Binary Tree

A perfect binary tree is a full binary tree where all interior nodes have exactly two children, and all leaf nodes are at the same level. Perfect binary trees are often used in balanced search tree structures, where the even distribution of nodes ensures optimal operation times.

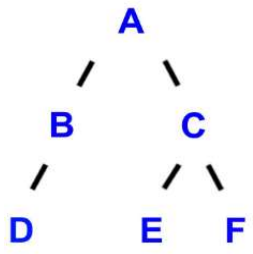
Representation:



4.Balanced Binary Tree

A balanced binary tree is a tree where the height of the left and right subtrees of any node differ by no more than one. This balance allows the tree to maintain efficient operations even as nodes are inserted and deleted.

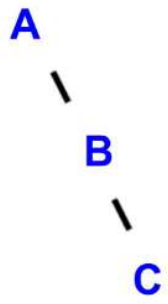
Representation:



5. Skewed Binary Tree

In a skewed binary tree, each node has only one child, either left or right, creating a linear-like structure. This type of binary tree occurs when data is inserted in an ordered manner, leading to unbalanced trees with performance characteristics similar to linked lists.

Representation (Right-skewed):



6. Degenerate Tree

A degenerate tree is an extreme case of a skewed tree, where each parent node has only one child. This structure results in a tree that functions as a linked list, losing the advantages of a balanced binary tree and causing slower performance for certain operations.

10.4.3 Properties of Binary Trees

Binary trees possess several key properties that are essential for understanding their structure, efficiency, and the computational complexity of operations like insertion, deletion, and traversal. Here's an in-depth look at some of these fundamental properties:

1. Maximum Nodes in a Binary Tree

For a binary tree of depth k (where k is the number of edges from the root to the deepest node), the maximum number of nodes is given by:

Maximum Nodes=$2^{k+1}-1$

This property implies that as the depth of the binary tree increases, the potential number of nodes grows exponentially. This exponential growth is due to each level potentially having twice as many nodes as the level above it.

Example:

- For k=2, the maximum number of nodes is $2^2 + 1 - 1 = 7$.
- For k=3, the maximum number of nodes is $2^3 + 1 - 1 = 15$.

This property is particularly useful when dealing with complete binary trees, where every level is fully populated with nodes. In such cases, we can easily determine the maximum number of nodes based on the depth alone.

2. Minimum Height of a Binary Tree

The height (or depth) of a binary tree with n nodes is at least:

$$\text{Minimum Height} = \log_2(n+1) - 1$$

In a balanced binary tree, the height is close to this minimum value. A balanced tree is structured such that the difference in height between the left and right subtrees of any node is minimal, ideally close to zero. This property is crucial because the height of a binary tree directly affects the efficiency of operations:

- Search, insertion, and deletion operations in a balanced binary tree run in O(log n)time, as the minimum height reduces the path length to reach nodes.

Example:

- For a binary tree with n=15 nodes, the minimum height is $\log_2(15+1) - 1 = 3$.

A balanced binary tree will generally approach this minimum height, ensuring optimal performance in terms of access and modification operations.

3. Node Degree and Leaf Nodes

In a binary tree, each node can have 0, 1, or 2 children:

- Nodes with 0 children are called leaf nodes.
- Nodes with 1 or 2 children are known as internal nodes.

The degree of a node refers to the number of children it has:

- Degree 0: Leaf nodes, with no children.
- Degree 1 or 2: Internal nodes, with one or two children.

An interesting and useful property of binary trees is the relationship between leaf nodes and nodes with two children (degree-2 nodes):

Number of Leaf Nodes = Number of Degree - 2 Nodes + 1

This property holds for any binary tree, regardless of its shape or balance. It stems from the fact that each additional degree-2 node increases the potential number of leaf nodes by one. This relationship is particularly useful for calculating the structural composition of a binary tree and estimating the number of leaf nodes based on internal node count.

Example:

- Suppose a binary tree has 5 nodes with 2 children (degree-2 nodes). By this property, it must have 5 + 1 = 6 leaf nodes.

Table 10.1 . Properties of a binary tree

Property	Formula / Description	Importance
Maximum Nodes	$2^{k+1} - 1$ for depth k	Determines the maximum capacity of a binary tree of a given depth.
Minimum Height	$\log_2(n+1) - 1$ for n nodes	Optimizes access and operation speeds, especially in balanced trees.
Node Degree and Leaves	Leaf Nodes = Degree-2 Nodes + 1	Useful for understanding the composition of leaf and internal nodes in the tree structure.

10.5 ARRAY REPRESENTATION OF BINARY TREES

The array representation of binary trees is a method of storing a binary tree's nodes in an array, which is particularly useful for complete binary trees. This representation uses a simple indexing scheme to represent parent-child relationships, eliminating the need for explicit pointers to the left and right children. This approach is efficient for certain types of binary trees but has limitations in flexibility compared to a linked representation. The array representation of a given binary tree is represented in the below diagram.

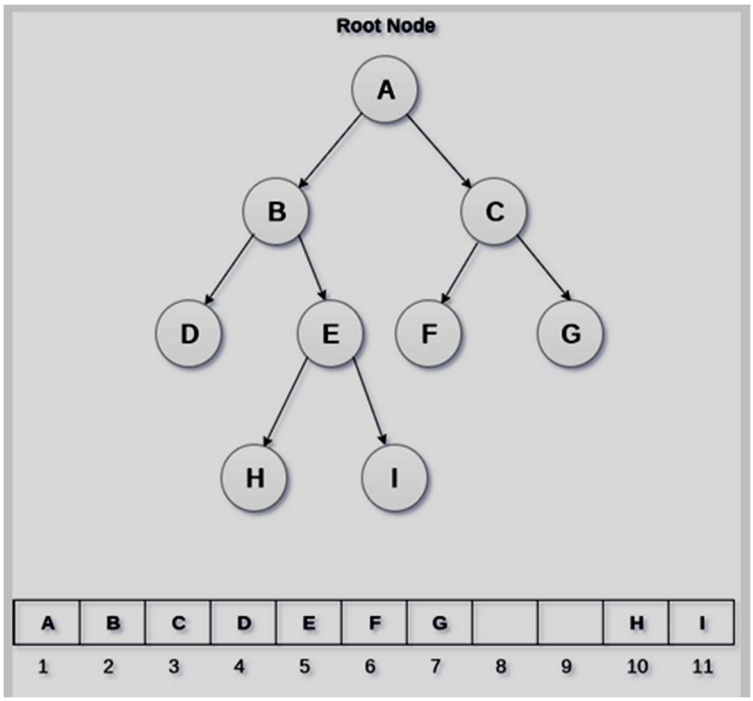


Fig 10.3. Array Representation

❖ Key Characteristics of Array Representation

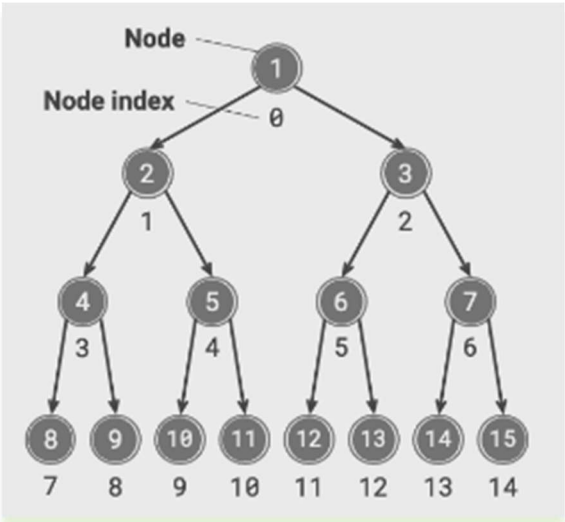
The array representation of binary trees is a method where nodes are stored in level order within an array, leveraging mathematical relationships between indices for efficient

navigation. This approach is most efficient for complete binary trees, ensuring minimal space usage while maintaining direct access to parent and child nodes

1. **Level-Order Storage**

    - In an array representation, nodes are stored in level order—that is, from top to bottom, left to right. Each node is placed in the array based on its position in the tree, making it straightforward to access nodes without using pointers.
    - For a binary tree of depth k, the maximum number of nodes is $2^{k+1}-1$, which determines the maximum required array size for a complete tree of depth k.

2. **Index-Based Relationships**
    - Parent and child relationships are derived from simple mathematical relationships based on indices, enabling efficient access and navigation within the tree. Given a node at index iii:
        - Left Child: Located at index 2i + 1
        - Right Child: Located at index 2i+2
        - Parent: Located at index i−1 / 2 (for i > 0)
    - These formulas are derived from the arrangement of nodes in a complete binary tree and allow efficient determination of a node's children and parent without explicit pointers.
3. **Efficiency for Complete Trees**
    - The array representation is ideal for complete binary trees, where all levels except possibly the last are fully populated, and nodes are added from left to right. In a complete binary tree, this array-based structure uses minimal space, as nodes are packed without gaps, and the above relationships apply seamlessly.
    - For an incomplete or unbalanced binary tree, this representation can result in wasted space in the array, as positions for missing nodes still consume memory. As a result, the array representation is rarely used for unbalanced binary trees.

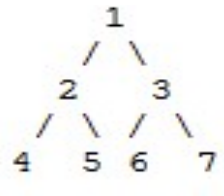


Left child node Index : 2i + 1
Right child node index: 2i + 2

Fig 10.4. Level-order traversal of the binary tree

❖ Example of Array Representation

Consider the following complete binary tree:



The level-order array representation of this tree is:
Array: [1, 2, 3, 4, 5, 6, 7]

Using the index-based relationships:

- **Node 1** at index 0 has left child at 2 × 0 + 1 = 1 (value 2) and right child at 2 × 0 + 2 = 2 (value 3).
  - **Node 2** at index 1 has left child at 2 × 1 + 1 = 3 (value 4) and right child at 2 × 1 + 2 = 4 (value 5).
  - **Node 3** at index 2 has left child at 2 × 2 + 1 = 5 (value 6) and right child at 2 × 2 + 2 = 6 (value 7).

❖ Advantages of Array Representation

1. Direct Access:
   - Nodes can be accessed directly by their index, which can improve access times, especially when working with a complete binary tree.
   - This makes operations like searching for children or parents straightforward without needing complex pointers.
2. Memory Efficiency in Complete Trees:
   - Since each node is packed tightly in the array, there is minimal memory overhead for complete trees, making the array representation highly space-efficient in such cases.
3. Simplified Operations:
   - The relationships between indices simplify certain operations, such as finding the depth of a node, since depth can be inferred from the index.

❖ Limitations of Array Representation

1. Wasted Space for Sparse Trees:
   - For incomplete or sparse trees, this representation may lead to significant wasted space. Gaps in the tree, such as missing nodes on one side, still require allocated positions in the array, resulting in unused array slots.

2. Limited Flexibility:
    - In dynamic scenarios where the binary tree grows or shrinks unpredictably, the array representation lacks flexibility. Inserting nodes or deleting nodes from arbitrary positions is difficult and inefficient since it may require shifting elements in the array or re-indexing nodes.
3. Fixed Size Requirement:
    - The array's size may need to be predetermined based on the expected height of the tree. If the tree grows beyond this size, resizing the array (copying to a larger one) can be costly.

The array representation is most commonly used for complete binary trees and data structures that inherently follow a complete binary tree structure, such as heaps (e.g., binary heaps used in priority queues). In a heap, nodes are continuously added or removed at the last level, aligning perfectly with the array representation's properties.

10.6 LINKED REPRESENTATION OF BINARY TREES

The linked representation of binary trees is one of the most flexible and widely used ways to implement binary trees. In this representation, each node in the binary tree is created as a structure that holds the data for that node and pointers to its left and right children. This approach is particularly useful for dynamic binary trees where the size of the tree can change frequently, as it allows easy insertion and deletion of nodes.

❖ Key Features of Linked Representation

1. **Node Structure**:
   Each node in the binary tree is represented by a structure with three components:
    - **Data**: Stores the value or information held by the node.
    - **Left Pointer**: Points to the left child node of the current node or NULL if no left child exists.
    - **Right Pointer**: Points to the right child node of the current node or NULL if no right child exists.
2. **Root Node**:
   The binary tree starts with a root node, which serves as the entry point to traverse the entire tree. Each node recursively connects to its left and right children, forming the hierarchical structure.
3. **Flexibility**:
   The linked representation allows:
    - Dynamic memory allocation, enabling trees to grow or shrink as needed.
    - Easy insertion and deletion of nodes without rearranging the entire structure, as required in arrays.
4. **Null Pointers**:
   If a node has no left or right child, the respective pointer is set to NULL, ensuring proper termination of branches during traversal.

❖ Advantages of Linked Representation

1. **Memory Efficiency**: It uses memory only for existing nodes, making it suitable for sparse and unbalanced trees.

2. **Ease of Modification**: Nodes can be easily added or removed without affecting the overall structure.
3. **Traversal Flexibility**: The direct pointers to child nodes make tree traversal algorithms (in-order, pre-order, post-order) straightforward and efficient.

In the linked representation:

- Each node is represented as a structure with three parts:
    1. **Data**: Stores the value or information held by the node.
    2. **Left Pointer**: Points to the left child node of the current node.
    3. **Right Pointer**: Points to the right child node of the current node.
- The binary tree starts with a root node, and every node can recursively have its own left and right children.

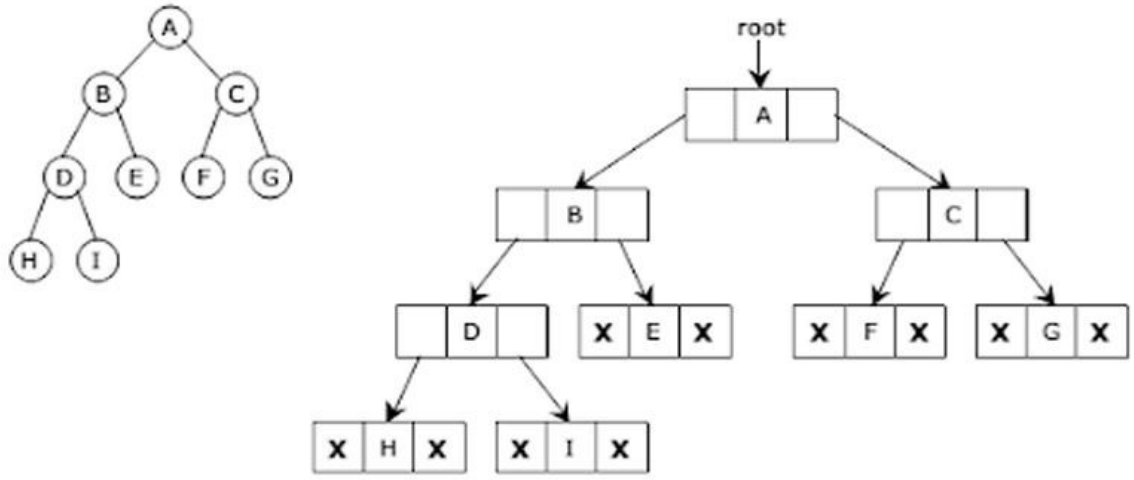


Fig 10.5. Linked List Representation of binary tree

If a node has no left or right child, the respective pointer is set to NULL. Figure 4 illustrates a binary tree and its linked list representation.

1. **Left Side (Tree Structure)**:
    - Represents the hierarchy of the binary tree with root `A`.
    - Each node has up to two children: left and right.
    - Leaf nodes (`E`, `F`, `G`, `H`, `I`) have no children.
2. **Right Side (Linked Representation)**:
    - Each node has three parts: **data**, **left pointer**, and **right pointer**.
    - Pointers point to the left and right children, or NULL (X) if no child exists.
    - Example: A points to B (left) and C (right), while E points to NULL for both.

This representation efficiently stores binary trees in memory, allowing dynamic addition and deletion of nodes. This linked structure allows easy traversal, insertion, and deletion of nodes, as each node points directly to its children.

❖ Structure of a Binary Tree Node in C

In C, a binary tree node can be defined as follows:

```
struct TreeNode {
   int data;              // Data or value of the node
   struct TreeNode* left;     // Pointer to the left child
   struct TreeNode* right;    // Pointer to the right child
};
```

10.7 ARRAY VS LINKED LIST REPRESENTATION OF BINARY TREES

Binary trees can be represented using either arrays or linked lists, each method having its own advantages and disadvantages. Here's a detailed comparison of the two approaches:

Table 10.2. Differences between array and linked list

Aspect	Array Representation	Linked List Representation
Storage Structure	Uses a contiguous block of memory	Uses nodes with pointers to left and right children
Indexing	Accessed via index (position in the array)	Accessed via pointers
Memory Efficiency	Fixed size, may have unused space for sparse trees	Dynamic size, allocates memory as needed
Ease of Traversal	Direct index calculation for parent/child positions	Requires traversal through pointers
Insertion/Deletion	Difficult due to fixed positions in array	Easier with dynamic node manipulation

The Differences Between Array Representation and Linked List Representation of Binary Trees can be summarized as

- **Storage Structure**:
  - In an array representation, the binary tree is stored in a continuous block of memory, like a long list.
  - In a linked list representation, each element (node) has pointers that link it to its left and right child nodes.
- **Indexing**:
  - In an array, you access elements by their position or index in the array.
  - In a linked list, you access elements by following pointers from one node to another.
- **Memory Efficiency**:
  - An array has a fixed size and might have empty spaces, especially for trees that aren't completely filled.

- A linked list only uses memory for nodes that exist, so it's more efficient for storing trees with a lot of missing nodes.

- **Ease of Traversal**:

  - In an array, you can quickly calculate the position of child or parent nodes based on simple formulas.
  - In a linked list, you have to follow pointers to move from one node to the next, which takes more steps.

- **Insertion/Deletion**:

  - In an **array**, adding or removing nodes is tricky because positions are fixed.
  - In a **linked list**, adding or removing nodes is simpler since you can just adjust the pointers.

10.8 KEY TERMS

Binary tree, Full binary tree, Complete binary tree, Perfect binary tree, Balanced binary tree, Skewed binary tree, Degenerate tree, Level-order traversal.

10.9 SELF-ASSESSMENT QUESTIONS

1. What is the main difference between a binary tree and a general tree?
2. How is the height of a binary tree calculated, and why is it significant?
3. Describe the differences between a complete binary tree and a perfect binary tree.
4. What are the advantages of using linked list representation over array representation for binary trees?
5. Explain the formulas used to determine the parent and child nodes in an array representation of a binary tree.

10.10 SUGGESTIVE READINGS

1. "Introduction to Algorithms" by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein.
2. "Data Structures and Algorithm Analysis in C" by Mark Allen Weiss.
3. "The Art of Computer Programming, Volume 1: Fundamental Algorithms" by Donald E. Knuth.
4. "Data Structures Using C" by Aaron M. Tenenbaum, Yedidyah Langsam, and Moshe J. Augenstein.
5. "Algorithms, 4th Edition" by Robert Sedgewick and Kevin Wayne.
6.

Mrs. Appikatla Pushpa Latha

Lesson - 11

Binary Tree Traversals and Threaded Binary Trees

OBJECTIVES

The objectives of the lesson are:

1. To understand the fundamentals of binary tree traversal techniques, including depth-first and breadth-first traversal.
2. To learn the algorithms and implementation of preorder, inorder, and postorder traversal.
3. To explore threaded binary trees and their advantages over traditional binary trees.
4. To identify real-world applications of binary tree traversals in data processing and hierarchical data representation.

STRUCTURE

11.1 INTRODUCTION

Binary tree traversal is a fundamental concept in computer science that involves visiting all nodes of a binary tree in a structured order to access, manipulate, or retrieve data. In a binary tree, each node can have at most two children (left and right), creating a hierarchical structure with each node connected to its children in a specific arrangement. Traversing a binary tree means systematically visiting each node, often to perform operations like searching, sorting, or modifying data in the tree.

11.2 BINARY TREE TRAVERSAL

Binary tree traversal methods enable access to each node in an organized way, ensuring that no node is skipped or revisited unnecessarily. These traversals are foundational for algorithms in search engines, databases, compilers, and other hierarchical data structures. By choosing the appropriate traversal method, specific outcomes can be achieved, such as processing nodes in a particular order, maintaining sorted data, or visualizing tree structures effectively.

11.3 TYPES OF BINARY TREE TRAVERSALS

1. **Depth-First Traversal (DFT)**
    - Depth-first traversal explores each branch of the tree to its deepest node before backtracking. In binary trees, DFT can be further categorized into three common approaches: inorder, preorder, and postorder traversal, each following a specific node visit sequence.
    - **General Process:** DFT is usually implemented with recursion, making it straightforward for hierarchical structures. Alternatively, it can be implemented using a stack to keep track of visited nodes.
    - **Applications:** Depth-first traversal is ideal for tasks that need complete exploration of paths, such as evaluating expressions (in expression trees), converting trees into other data structures, and performing complex searches in nodes with hierarchical relationships.
2. **Breadth-First Traversal (BFT):**
    - Also known as level-order traversal, breadth-first traversal visits all nodes at each level of the tree before moving to the next level. Nodes are accessed layer by layer from top to bottom and left to right at each level.
    - **Implementation:** BFT requires a queue to keep track of nodes at each level. Starting from the root, nodes are enqueued and dequeued level by level, with their children added to the queue for subsequent levels.
    - **Applications:** This traversal is useful in applications where the proximity of nodes to the root is essential, such as finding the shortest path, executing hierarchical commands, and generating visual tree representations.

Tree traversal is a core concept in data structures, specifically in trees, where it represents the process of visiting each node in a structured sequence. Unlike linear data structures such as linked lists, queues, and stacks, trees offer multiple ways to traverse their nodes, enabling various applications and benefits.

In binary trees, three primary **Depth-First Traversal (DFT)** techniques are commonly used. They are

1. **Preorder Traversal**
2. **Inorder Traversal**
3. **Postorder Traversal**

These traversal techniques differ in the order of node visits and serve different purposes. Let's explore each type of traversal in detail, including examples, algorithms, and practical applications.

11.3.1 Preorder Traversal

Preorder traversal is a method where each node is visited in the following sequence: root, left subtree, right subtree. This "root-left-right" sequence means that the traversal begins with the root node, proceeds to the left subtree, and finally moves to the right subtree. The name "preorder" indicates that the root node is processed first, followed by the subtrees.

❖ **Steps**

- Visit the root node.
- Traverse the left subtree recursively.
- Traverse the right subtree recursively.

❖ **Function in C**

```
void preorderTraversal(struct TreeNode* root)
{
   if (root != NULL)
{
      printf("%d ", root->data);       // Visit the root node
      preorderTraversal(root->left);   // Traverse the left subtree
      preorderTraversal(root->right);  // Traverse the right subtree
   }
}
```

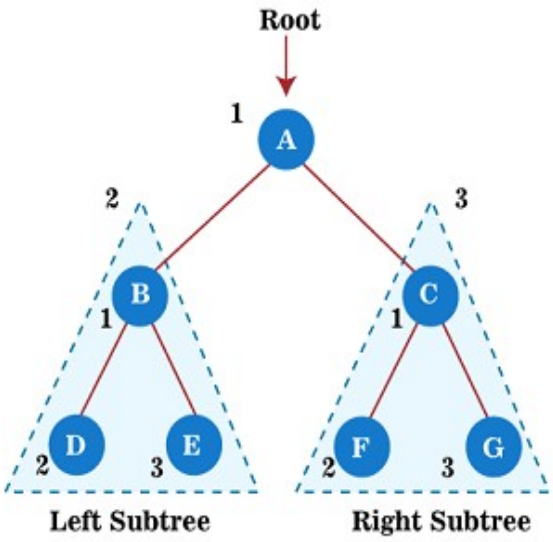


Fig 11.1. Preorder Traversal

In the provided binary tree, preorder traversal follows the order root, left subtree and right subtree. This means we visit the root node first, then recursively visit the left subtree, and finally, the right subtree.

The step-by-step preorder traversal of the above tree is discussed below.

❖ **Step-by-Step Preorder Traversal**

1. **Start with the Root Node (A):**

   In preorder traversal, we visit the root first. So, we start by visiting **A** and add it to our output.

2. **Left Subtree of A (Rooted at B):**

   - After visiting **A**, we move to its **left subtree** (rooted at B).
   - In the subtree rooted at B, we again follow the root-left-right order. So, we visit **B** next and add it to the output.

3. **Left Child of B (Node D):**

   - After visiting **B**, we move to its left child **D**.
   - **D** has no children, so we visit **D** directly and add it to our output.

4. **Right Child of B (Node E):**

   - After finishing with **D**, we go back to **B** and move to its right child **E**.
   - **E** has no children, so we visit **E** directly and add it to the output.
   - At this point, we have completed the traversal of the left subtree of **A**.

5. **Right Subtree of A (Rooted at C):**

   - After completing the left subtree, we return to **A** and proceed to its **right subtree**, which is rooted at **C**.
   - In this subtree, we follow the same root-left-right order. We start by visiting **C** and add it to our output.

6. **Left Child of C (Node F):**

   - After visiting **C**, we move to its left child **F**.
   - **F** has no children, so we visit **F** directly and add it to the output.

7. **Right Child of C (Node G):**

   - After completing the left child of **C**, we move to its right child **G**.
   - **G** has no children, so we visit **G** directly and add it to the output.

Now all nodes have been visited, and the traversal is complete.

❖ **Preorder Traversal Output**

The sequence of nodes visited in preorder traversal for this tree is:

A → B → D → E → C → F → G

❖ **Summary of the Steps in Order**

1. Visit **A** (root node)
2. Visit **B** (root of the left subtree of A)
3. Visit **D** (left child of B)
4. Visit **E** (right child of B)
5. Visit **C** (root of the right subtree of A)
6. Visit **F** (left child of C)
7. Visit **G** (right child of C)

In preorder traversal, we visit each root node before its subtrees, which allows us to process nodes in a top-down order. This traversal is particularly useful for applications like copying a tree or evaluating prefix expressions in expression trees.

❖ **Applications of Preorder Traversal**

- Tree Duplication: Preorder traversal is useful for creating a copy of the tree, as it captures each node's structure from top to bottom.
- Prefix Expressions: In expression trees, preorder traversal provides a prefix notation (Polish notation), where operators appear before their operands.

11.3.2 Inorder Traversal

Inorder traversal processes each node in a binary tree following a "left-root-right" sequence. This means that the traversal begins with the leftmost subtree, visits the root node, and then processes the right subtree. For binary search trees (BSTs), inorder traversal provides the nodes in a sorted order, making it particularly valuable for retrieving ordered data.

❖ **Steps**

1. Traverse the left subtree recursively.
2. Visit the root node.
3. Traverse the right subtree recursively.

❖ **Funtion in C**

```
void inorderTraversal(struct TreeNode* root)
 {
   if (root != NULL) {
      inorderTraversal(root->left);    // Traverse the left subtree
      printf("%d ", root->data);       // Visit the root node
      inorderTraversal(root->right);   // Traverse the right subtree
   }
  }
```

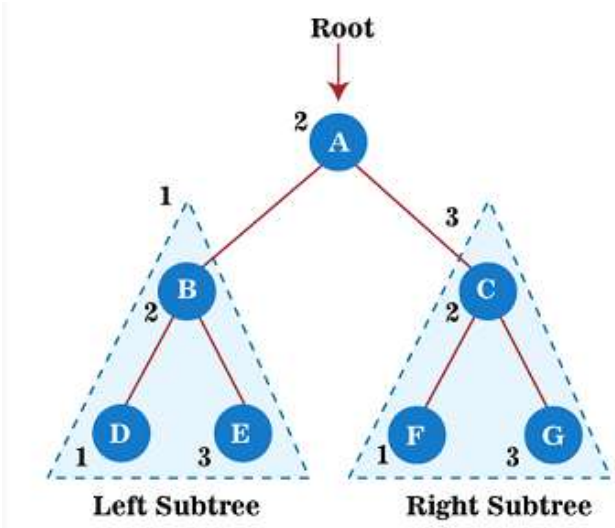


Fig 11.2. Inorder Traversal

In the provided binary tree image, inorder traversal means visiting each node in the order left subtree, root, right subtree. For each subtree in this traversal, we first process the left child, then the node itself, and finally the right child. In a binary search tree (BST), this traversal yields nodes in ascending order.

❖ **Step-by-Step Inorder Traversal**

The step-by-step inorder traversal of the above tree is discussed below

1. **Start with the Root (A):**
   - In inorder traversal, we begin with the left subtree before visiting the root. So, we first move to the left subtree of A (subtree rooted at B).
2. **Left Subtree of A (Rooted at B):**
   - In the subtree rooted at B, we again follow the left-root-right order. We first move to the left child of B, which is D.
3. **Node D:**
   - D has no children, so we visit D directly and add it to our output.
   - After visiting D, we go back to B.
4. **Visit Node B:**
   - Now that we've processed the left child (D), we visit B itself and add it to the output.
5. **Right Child of B (Node E):**
   - Next, we move to the right child of B, which is E.
   - E has no children, so we visit E directly and add it to the output.
   - Having completed the left subtree of A, we now return to A.
6. **Visit Root Node A:**
   - After completing the traversal of the left subtree, we visit the root node A and add it to our output.
7. **Right Subtree of A (Rooted at C):**
   - Now we move to the right subtree of A, which is rooted at C.
   - In this subtree, we first go to the left child of C, which is F.
8. **Node F:**
   - F has no children, so we visit F directly and add it to our output.
   - After visiting F, we return to C.

9. **Visit Node C:**

    - Now that we have processed the left child of C, we visit C itself and add it to the output.

10. **Right Child of C (Node G):**

    - Finally, we move to the right child of C, which is G.
    - G has no children, so we visit G directly and add it to the output.

❖ **Inorder Traversal Output:** D → B → E → A → F → C → G

❖ **Summary of the Steps in Order:**

1. Visit **D** (left child of B)
2. Visit **B** (root of left subtree)
3. Visit **E** (right child of B)
4. Visit **A** (root node)
5. Visit **F** (left child of C)
6. Visit **C** (root of right subtree)
7. Visit **G** (right child of C)

This output sequence follows the inorder traversal (left, root, right) for each node and subtree, providing a sorted sequence of nodes if this were a BST.

❖ **Applications of Inorder Traversal:**

- Sorted Data Retrieval: In BSTs, inorder traversal outputs the nodes in ascending order, which is essential for sorting and ordered data processing.
- Expression Viewing: In expression trees, this traversal gives a conventional (infix) view of expressions.

11.3.3 Postorder Traversal

Postorder traversal follows a "left-right-root" pattern, where each node is processed only after both of its subtrees have been visited. This traversal approach is valuable in applications that require the root node to be processed last, such as when deleting nodes in a tree.

❖ **Steps:**

1. Traverse the left subtree recursively.
2. Traverse the right subtree recursively.
3. Visit the root node.

❖ **Funtion in C**

```
void postorderTraversal(struct TreeNode* root)
{
   if (root != NULL)
{
```

```
    postorderTraversal(root->left);   // Traverse the left subtree
    postorderTraversal(root->right);  // Traverse the right subtree
    printf("%d ", root->data);        // Visit the root node
}
}
```

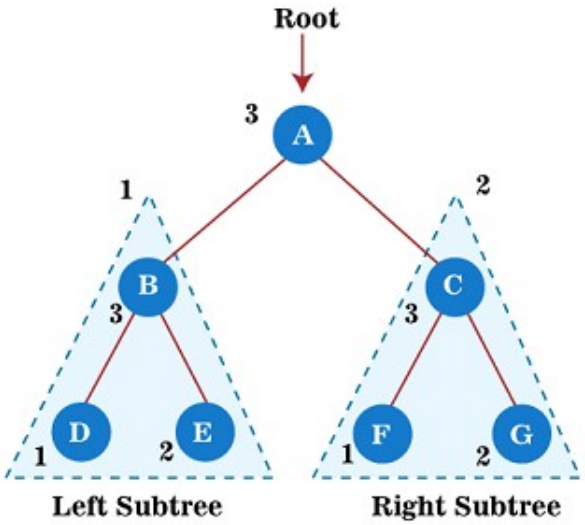


Fig 11.3. Postorder Traversal

In the provided binary tree, **postorder traversal** follows the order **left subtree, right subtree, root**. This means we first recursively visit the left subtree, then the right subtree, and finally, the root node. Each node is processed after both of its subtrees, which is why it's called "postorder."

❖ Step-by-Step Inorder Traversal

The step-by-step inorder traversal of the above tree is discussed below

1. **Start with the Root Node (A):**

   In postorder traversal, we begin with the left subtree first, so we move to the **left subtree of A** (rooted at B) and postpone visiting **A** itself until both subtrees are fully traversed.

2. **Left Subtree of A (Rooted at B):**
   - In the subtree rooted at **B**, we again follow the left-right-root order. So, we first move to **B's left child, D**.
3. **Node D:**
   - **D** has no children, so we visit **D** directly and add it to our output.
   - After finishing with **D**, we return to **B**.
4. **Right Child of B (Node E):**
   - Now that we have visited **D**, we move to the **right child of B**, which is **E**.

- **E** has no children, so we visit **E** directly and add it to our output.

5. **Visit Node B:**
   - After finishing both the left child (D) and the right child (E) of **B**, we visit **B** itself and add it to the output.
   - With this, we have completed the traversal of the left subtree of **A**.
6. **Right Subtree of A (Rooted at C):**
   - Now, we move to the **right subtree of A**, which is rooted at **C**.
   - For this subtree, we again follow the left-right-root order. We start with the **left child of C**, which is **F**.
7. **Node F:**
   - **F** has no children, so we visit **F** directly and add it to the output.
   - After finishing with **F**, we return to **C**.
8. **Right Child of C (Node G):**
   - Next, we move to the **right child of C**, which is **G**.
   - **G** has no children, so we visit **G** directly and add it to our output.
9. **Visit Node C:**
   - After visiting both the left child (F) and the right child (G) of **C**, we visit **C** itself and add it to the output.
   - Now we have completed the traversal of the right subtree of **A**.
10. **Visit Root Node A:**
    - Finally, after finishing both the left and right subtrees, we visit the root node **A** and add it to the output.

❖ **Postorder Traversal Output:**

The sequence of nodes visited in postorder traversal for this tree is:

D → E → B → F → G → C → A

❖ **Summary of the Steps in Order:**

1. Visit **D** (left child of B)
2. Visit **E** (right child of B)
3. Visit **B** (root of left subtree)
4. Visit **F** (left child of C)
5. Visit **G** (right child of C)
6. Visit **C** (root of right subtree)
7. Visit **A** (root node)

In postorder traversal, we visit each node only after its left and right subtrees have been fully processed. This traversal is particularly useful for applications where we need to process all child nodes before the parent, such as deleting nodes or evaluating postfix expressions in expression trees.

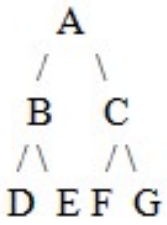
❖ **Applications of Postorder Traversal:**

- Tree Deletion: Postorder traversal is ideal for deleting trees because it ensures all child nodes are processed before their parent, making deletion operations safer and more efficient.

- Postfix Expressions: In expression trees, postorder traversal yields postfix notation (Reverse Polish Notation), where operators are applied after their operands.

11.4 Summary of Tree Traversal Techniques

To summarise, here are the key characteristics and outcomes for each traversal type applied to the sample tree:



Traversal Type	Sequence	Output
Preorder	Root, Left, Right	A → B → D → E → C → F → G
Inorder	Left, Root, Right	D → B → E → A → F → C → G
Postorder	Left, Right, Root	D → E → B → F → G → C → A

11.5 CHOOSING THE RIGHT TRAVERSAL

Each traversal method serves unique applications.

- Preorder Traversal is useful when the root must be processed before its subtrees, such as in tree duplication or prefix notation.
- Inorder Traversal provides sorted output for BSTs and helps in processing expressions in their infix form.
- Postorder Traversal is optimal when subtrees must be processed before the root, as in deletion tasks and postfix notation.

11.6 INTRODUCTION TO THREADED BINARY TREES

A threaded binary tree is a special type of binary tree where empty (null) pointers are replaced with "threads."

In a regular binary tree, if a node doesn't have a left or right child, its pointers would be set to null. But in a threaded binary tree, instead of being null, these pointers are directed to point to the next or previous node in the in-order sequence. This could be the node's "in-order predecessor" (the node that comes just before it) or "in-order successor" (the node that comes just after it), depending on the type of threading:

- **Single Threading**: Only one pointer (usually right) is threaded to the in-order successor.
- **Double Threading**: Both left and right pointers can be threaded to connect with in-order predecessor and successor.

This threading makes it easier to move through the tree in order, without needing extra memory or complex recursion.

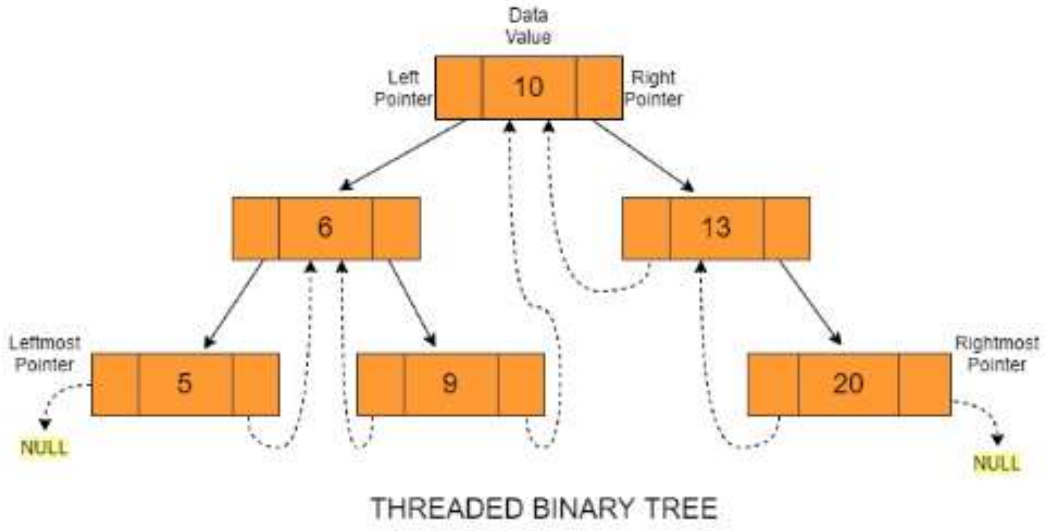


Fig 11.4. Structure of a Threaded Binary Tree Node

To implement threading, each node in a threaded binary tree is extended with two additional fields:

- **Left Thread Indicator**: Indicates whether the left pointer is a thread or a genuine child pointer.
- **Right Thread Indicator**: Indicates whether the right pointer is a thread or a genuine child pointer.

The structure in C would look like:

```
typedef struct threaded_tree
{
   short int left_thread;
   struct threaded_tree* left_child;
   char data;
   struct threaded_tree* right_child;
   short int right_thread;
} threaded_pointer;
```

Here, left_thread and right_thread are used to distinguish between child pointers and threads.

11.6.1 Advantages of Threaded Binary Trees

1. **Efficient In-Order Traversal**: Threaded binary trees allow in-order traversal without using a stack or recursion. Each node has a direct link to its in-order successor, making the traversal process more efficient.
2. **Memory Optimization**: By reusing the null links, threaded binary trees reduce the need for additional stack memory during traversal.

3. **Simpler Traversal Algorithm**: Traversing a threaded binary tree can be done iteratively with minimal code, as each node directly points to its successor or predecessor where needed.

11.6.2 In-Order Traversal of a Threaded Binary Tree

In a threaded binary tree, in-order traversal can be simplified by following these steps:

1. Start from the leftmost node (which is the smallest in the in-order sequence).
2. For each node:
    - Print the data.
    - If the right pointer is a thread, follow it to the in-order successor.
    - If the right pointer is a child pointer, move to the leftmost node in the right subtree.

The following C code shows an in-order traversal in a threaded binary tree:

```
void inorder(threaded_pointer* tree)
 {
   threaded_pointer* temp = tree;
     while (1)
     {
        temp = insucc(temp);
        if (temp == tree) break;
        printf("%c ", temp->data);
     }
 }
```

This code performs an **in-order traversal** on a threaded binary tree, which means it visits each node in the sequence defined by in-order (left-root-right).

Here's a simple breakdown:

1. **Start at the Root**:

   The function starts at the root node, named `tree`, and assigns it to a temporary pointer `temp`.

2. **Loop Through Nodes**:
    - The `while` loop runs continuously until it's told to stop (`break`).
    - Inside the loop, `temp = insucc(temp);` finds the in-order successor of the current node (`temp`). The `insucc` function follows the threads to move to the next node in the in-order sequence.
3. **Check for Completion**:
    - If `temp` reaches back to the starting point (`tree`), the traversal is done, and the loop breaks.
4. **Print the Data**:
    - If the traversal is not complete, the function prints `temp->data`, which is the value stored in the current node.

In summary, this code moves from node to node using the threads in the tree, printing each node's data in in-order sequence without recursion or a stack. The traversal ends when we return to the starting node, completing a full cycle through the tree.

11.6.3 Example

We have nodes in the in-order sequence: **A, B, C, D, E**. The tree is constructed as in the below diagram so that nodes with missing children have pointers (threads) to their in-order predecessor or successor.

Here's a step-by-step construction of the tree:

1. **Root Node (B)**:
   - `B` is the root with two children.
   - Left Child of B: Node `A`
   - Right Child of B: Node `C`
2. **Node A**:
   - `A` has no left child, so it's threaded.
   - Right Child of A: Node `B` (regular link, no threading needed).
3. **Node C**:
   - `C` has no left child, so its left pointer is threaded to `B` (its in-order predecessor).
   - Right Child of C: Node `D` (threaded).
4. **Node D**:
   - `D` has no right child, so its right pointer is threaded to `E` (its in-order successor).
   - Left Child of D: Node `C` (regular link).

Node E:

- `E` has no right child, so its right pointer could be threaded to an end marker (e.g., `NULL`).

In this tree:

- The in-order traversal sequence would be: **A → B → C → D → E**.
- Threads allow traversal from one node to the next in the in-order sequence without recursion or a stack.

11.6.4 Insertion in a Threaded Binary Tree

Insertion in a threaded binary tree is slightly more complex than in a regular binary tree. When inserting a new node:

1. If it has no right child, the right thread is set to point to the in-order successor.
2. If it has a left thread, it points to its in-order predecessor.

The right insertion process in C is managed by `insert-right`, which handles cases where the new node is inserted as the right child of an existing node(Fundamentals-of-Data-St…).

11.6.5 Applications of Threaded Binary Trees

1. **Memory-Constrained Environments**: Threaded binary trees are useful in systems with limited memory since they optimise the use of pointers.
2. **Non-Recursive Traversal Needs**: Systems or applications that need efficient in-order traversal without recursion benefit from threaded binary trees.
3. **In-Order Data Processing**: When data needs to be processed in a sorted order, threaded binary trees provide a straightforward mechanism for in-order traversal.

Threaded binary trees thus provide an effective way to handle tree traversal efficiently by leveraging existing null links and optimising memory usage.

11.7 KEY TERMS

Binary tree, Depth-first traversal (DFT), Breadth-first traversal (BFT), Threaded binary tree, In-order traversal.

11.8 SELF ASSESSMENT QUESTIONS

1. What is the difference between depth-first traversal (DFT) and breadth-first traversal (BFT)?
2. Explain the steps involved in preorder, inorder, and postorder traversals with examples.
3. What are the advantages of threaded binary trees over regular binary trees?
4. How does in-order traversal differ in threaded binary trees compared to regular binary trees?
5. Write a C function to perform preorder traversal on a binary tree.

11.9 SUGGESTED READINGS

1. "Data Structures and Algorithm Analysis in C" by Mark Allen Weiss.
2. "Introduction to Algorithms" by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein.

3. "The Art of Computer Programming, Volume 1: Fundamental Algorithms" by Donald E. Knuth.
4. "Data Structures Using C" by Aaron M. Tenenbaum, Yedidyah Langsam, and Moshe J. Augenstein.
5. "Algorithms, 4th Edition" by Robert Sedgewick and Kevin Wayne.

Mrs. Appikatla Pushpa Latha

Lesson -12
Binary Search Trees

OBJECTIVE

The objective of the Lesson is to

- Understand the structure and properties of Binary Search Trees (BSTs).
- Explain the efficient operations (search, insert, delete) enabled by BSTs.
- Highlight the significance of balanced BSTs in ensuring optimal performance.
- Analyze real-world applications of BSTs in data management and indexing.
- Explore the impact of tree height on the efficiency of operations in BSTs.

STRUCTURE

12.1 INTRODUCTION

A binary search tree (BST) is a type of binary tree that makes it easy to search, insert, and delete elements efficiently. Each node in a BST has a unique key, and it follows specific rules: the left subtree contains keys smaller than the node's key, the right subtree contains keys larger than the node's key, and both subtrees are also binary search trees. This structure keeps the elements in order and allows for quick operations. BSTs are used in many areas, such as databases, dynamic sets, and dictionaries, because they can handle data in a structured and efficient way. The performance of a BST depends on its height—shorter trees are faster. In balanced BSTs, the height is $O(\log_2 n)$, making operations like searching and inserting very quick. However, if the tree becomes unbalanced, the height can grow to $O(n)$, slowing things down. To avoid this, balanced versions of BSTs, like AVL and Red-Black trees, are often used. BSTs are a simple yet powerful way to manage and organize data effectively.

12.2 FEATURES AND SIGNIFIANCE

A **binary search tree** is a specialized form of a binary tree with the following properties:

1. Each element in the tree has a unique key.
2. The keys in the left subtree of a node are smaller than the key of the node.
3. The keys in the right subtree of a node are larger than the key of the node.
4. Both left and right subtrees are themselves binary search trees.

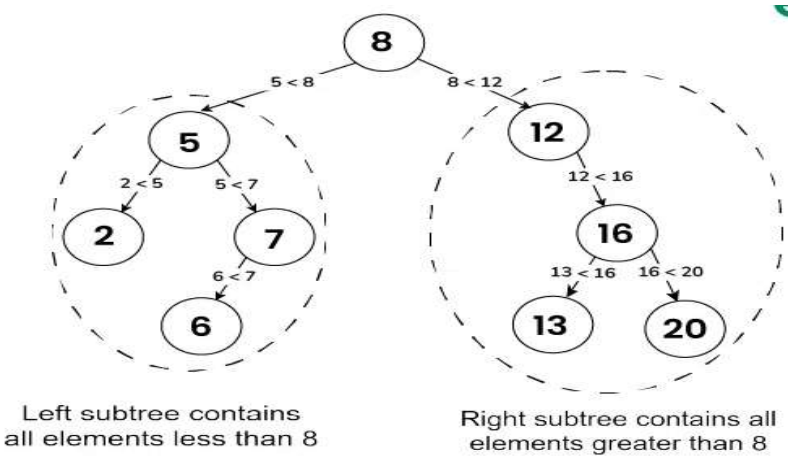


Fig 12.1. Binary Search Tree - Example

12.2.1 Key Features of Binary Search Trees (BSTs)

1. **Unique Key Property**

   Each node in a BST contains a unique key, ensuring no duplicates and enabling precise operations.
2. **Hierarchical Structure**

   The BST is organized hierarchically:
   - The left subtree of a node contains elements smaller than the node's key.
   - The right subtree contains elements larger than the node's key.
   - Both subtrees are themselves binary search trees.
3. **Efficient Searching**

   The tree's structure allows searching for a key in O(h), where hhh is the height of the tree. For balanced BSTs, this is $O(\log_2 n)$.
4. **Dynamic Operations**

   BSTs support efficient insertion, deletion, and traversal operations, adapting dynamically to changes in the dataset.
5. **Sorted Traversals**

   Using **inorder traversal**, BSTs produce elements in sorted order, making them useful for ordered data operations.
6. **Adaptability**

   BSTs can adapt to different balancing techniques, such as AVL trees or Red-Black trees, to optimize performance for specific use cases.

12.2.2 Significance of Binary Search Trees

1. **Efficient Data Management**

   BSTs provide a systematic way to manage and organize data, making it easy to perform dynamic operations like insertions and deletions.
2. **Versatility in Applications**

   BSTs are used in various applications, including database indexing, dynamic set operations, dictionaries, and symbol tables in compilers.

3. **Optimized Performance**
   By maintaining order among elements, BSTs ensure faster access times compared to unstructured data storage methods like arrays or linked lists.
4. **Support for Ordered Data**
   BSTs are ideal for scenarios where ordered data is required, such as range queries or retrieving sorted data subsets.
5. **Foundation for Advanced Data Structures**
   BSTs form the basis for more advanced structures like AVL trees, Red-Black trees, and B-trees, which enhance efficiency in specialized contexts.
6. **Space Efficiency**
   Unlike arrays, BSTs do not require pre-allocation of memory, dynamically adjusting to the size of the data.
7. **Flexibility in Traversals**
   BSTs support multiple types of tree traversals, such as preorder, inorder, and postorder, which are useful in different contexts like expression evaluation or data processing.

Binary search trees are a fundamental data structure in computer science, combining simplicity with powerful functionality to handle a wide range of real-world problems efficiently.

12.3 SEARCHING IN A BINARY SEARCH TREE

1. **Initialize the Search at the Root Node:**
   - Begin the search operation from the root node of the binary search tree.
2. **Compare the Target Key with the Current Node's Key:**
   - If the target key is equal to the current node's key, the search is successful, and the current node is the result.
3. **Determine the Direction of Search:**
   - If the target key is **less than** the current node's key, move to the left child (indicating that the target, if it exists, must be in the left subtree).
   - If the target key is **greater than** the current node's key, proceed to the right child (indicating that the target, if it exists, must be in the right subtree).
4. **Repeat the Comparison and Direction Steps:**
   - Continue the process of comparison and directional choice (steps 2 and 3) as you progress through each node in the tree.

5. **Termination of the Search:**
    - If you reach a NULL node, it indicates that the target key is not present in the tree, and the search is deemed unsuccessful.
    - If the target key is found during the traversal, the search is successful, and the node containing the key is returned as the result**.**

Example

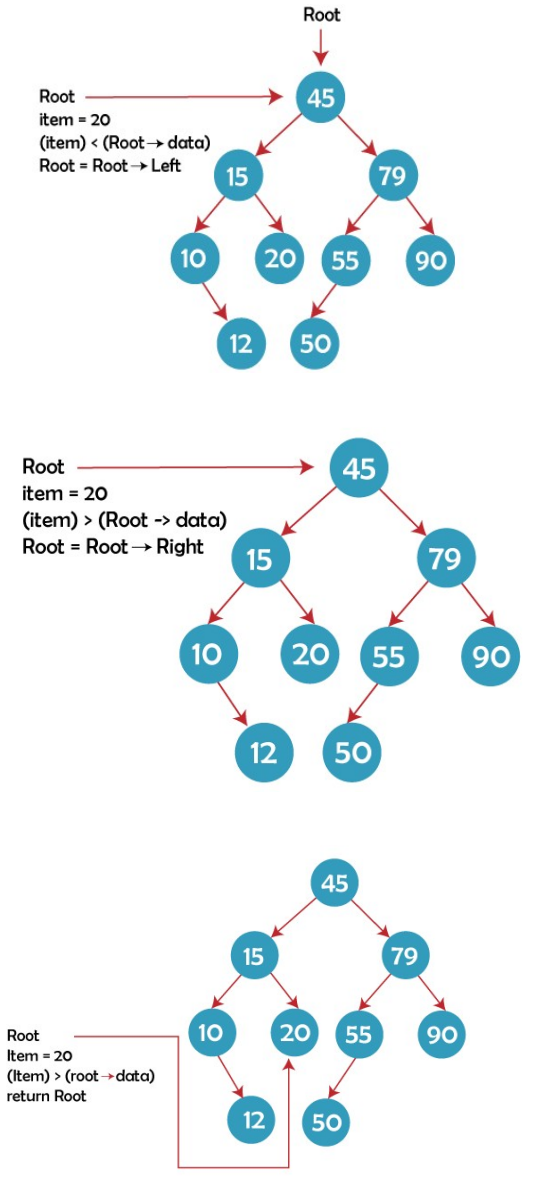


Figure 2: Searching an element in BST

The key value 20 is searched in the given array of elements. The steps of searching process are depicted as below

Step 1

- Begin at the root node with the value 45.
- The item to be searched is 20.
- Since 20 is less than 45 (item < root's data), move to the left child of the root.
- Initial comparison directs the search to the left subtree.

Step 2

- Current node is now 15.
- Compare the item 20 with the current node's data, 15.
- Since 20 is greater than 15 (item > root's data), move to the right child of the current node.
- Comparison shifts the search to the right subtree of the current node.

Step 3

- Current node is now 20.
- Item matches the current node's data (item == root's data).
- Search is successful, and the node containing the value 20 is found.

This method takes advantage of the binary search tree's properties, where each left subtree contains nodes with smaller keys and each right subtree contains nodes with larger keys, enabling an efficient O(h) search time, where h is the height of the tree.

12.4 Inserting into a Binary Search Tree

Inserting a new node into a Binary Search Tree (BST) requires following a few systematic steps to maintain the tree's ordered structure. All the steps are discussed in detail below

12.4.1 Verifying if the Key Already Exists

- Start by searching for the key you wish to insert. In a BST, each node follows this property:

For any given node:

- All nodes in its left subtree contain values less than the node's value.
- All nodes in its right subtree contain values greater than the node's value.

- Begin the search at the root node of the tree.
- Compare the key to be inserted with the current node's value
  - If the key is equal to the current node's value, then it already exists in the tree, and no insertion is needed, as BSTs typically do not allow duplicate values.
  - If the key is less than the current node's value, proceed to the left child.
  - If the key is greater than the current node's value, proceed to the right child.
- Continue this process until you either
  - Find the key, which means it already exists (no insertion needed).
  - Reach a NULL pointer, which means you've found the place where the new node should be inserted.

12.4.2 Key Insertion at Termination Point

If the key is not found (i.e., you reached a NULL pointer), then appropriate position to insert the new node is found.

1. Create a new node with the key value.
2. Attach this new node to the tree at the position where the search terminated.

This means

- If the last comparison suggested going left (the key was smaller than the last examined node), insert the new node as the left child of that node.
- If the comparison suggested going right (the key was larger), insert the new node as the right child of that node.

- **Implementation Code**

```
void insert_node(tree_pointer *node, int num) {
// If the tree is empty, create a new node
```

```
if (!(*node)) {
   tree_pointer ptr = (tree_pointer) malloc (sizeof (tree_pointer));
   if (!ptr) {  // Check if malloc failed
      fprintf(stderr, "Memory allocation failed\n");
      exit(1);
   }
   ptr->data = num;
   ptr->left_child = ptr->right_child = NULL;
   *node = ptr;  // Set the new node as the root
  } else {
   // Search for the correct position to insert
   tree_pointer temp = modified_search(*node, num);

   if (temp) {  // If temp is not NULL, insert the new node
      tree_pointer ptr = (tree_pointer)malloc(sizeof(tree_pointer));
       if (! ptr) {  // Check if malloc failed
          fprintf(stderr, "Memory allocation failed\n");
          exit(1);
      }
      ptr->data = num;
      ptr->left_child = ptr->right_child = NULL;
      // Insert the new node based on the comparison
      if (num < temp->data) {
         temp->left_child = ptr;
      } else {
         temp->right_child = ptr;
      } }    } }
```

The above function insert_node adds a new node with a specific value (num) to a binary search tree. The C function is explained in detail below.

The binary search tree is represented using pointers, where each node has:

1. A data value.
2. Two child pointers: left_child (for smaller values) and right_child (for larger values).

Parameters used in the code:

- tree_pointer *node: A pointer to the root of the tree (or subtree) where the node will be inserted.
- int num: The value to be inserted into the tree.

➢ **Step-by-Step Explanation**

1. Check if the Tree is Empty

   if (!(*node)) {

   - If the tree (or subtree) is empty (*node is NULL), it means there's no root yet.
   - A new node will be created to become the root.

2. Create a New Node

   tree_pointer ptr = (tree_pointer) malloc (sizeof (tree_pointer));

   - Memory is allocated for the new node using malloc.
   - The program checks if malloc was successful. If not, it prints an error and stops the program:

     ```
     if (!ptr) {
        fprintf(stderr, "Memory allocation failed\n");
        exit(1);
     }
     ```

3. Set the New Node's Data

   ```
   ptr->data = num;
   ptr->left_child = ptr->right_child = NULL;
   *node = ptr;
   ```

   - The data field is set to the value num.
   - The left_child and right_child pointers are initialised to NULL (no children yet).
   - The newly created node becomes the root of the tree.

4. If the Tree is Not Empty

   ```
   else {
      tree_pointer temp = modified_search(*node, num);
   ```

- The program looks for the correct position to insert the new node. The function modified_search is used for this.

5. Insert at the Correct Position
   if (temp) {
   - If modified_search returns a valid node (temp), a new node is created similarly as before:
     tree_pointer ptr = (tree_pointer) malloc (sizeof (tree_pointer));
     ptr->data = num;
     ptr->left_child = ptr->right_child = NULL;
6. Attach the New Node to the Parent Node
   if (num < temp->data) {
   temp->left_child = ptr;
   } else {
   temp->right_child = ptr;
   }
   - If the new value (num) is less than the temp node's value, the new node becomes the left_child.
   - Otherwise, it becomes the right_child.

The time Complexity is

- Searching for the insertion point takes O(h).
- The rest of the operation takes constant time, O(1).
- Overall complexity: O(h).

➢ **Example 1**

If the tree initially consists of [100,50,150] and 200 is the key value that is to be inserted, then the insertions are performed as in the below figure.

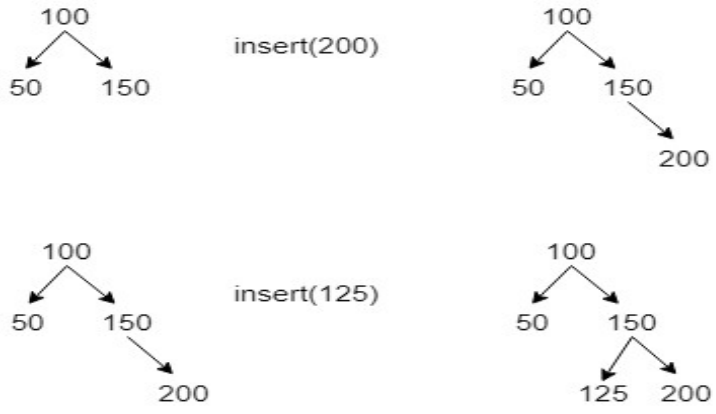


Figure 3. Insertion of node - Example 1

The above figure illustrates the insertion process in a Binary Search Tree (BST) with two steps. Here is a breakdown of each insertion:

Initial Tree Structure

- The root of the tree is 100.
- The left child of 100 is 50.
- The right child of 100 is 150.

First Insertion: insert(200)

1. The value 200 is greater than 100, so we move to the right child of 100, which is 150.
2. 200 is also greater than 150, so we move to the right of 150.
3. Since 150 does not have a right child, 200 is inserted as the right child of 150.

After this insertion, the tree structure is as follows:

- 100 (root)
    - Left child: 50
    - Right child: 150
        - Right child of 150: 200

Second Insertion: insert(125)

1. The value 125 is greater than 100, so we move to the right child of 100, which is 150.
2. 125 is less than 150, so we move to the left of 150.
3. Since 150 does not have a left child, 125 is inserted as the left child of 150.

After this insertion, the final tree structure is as follows:

- 100 (root)
    - Left child: 50
    - Right child: 150
        - Left child of 150: 125
        - Right child of 150: 200

➢ **Example 2**

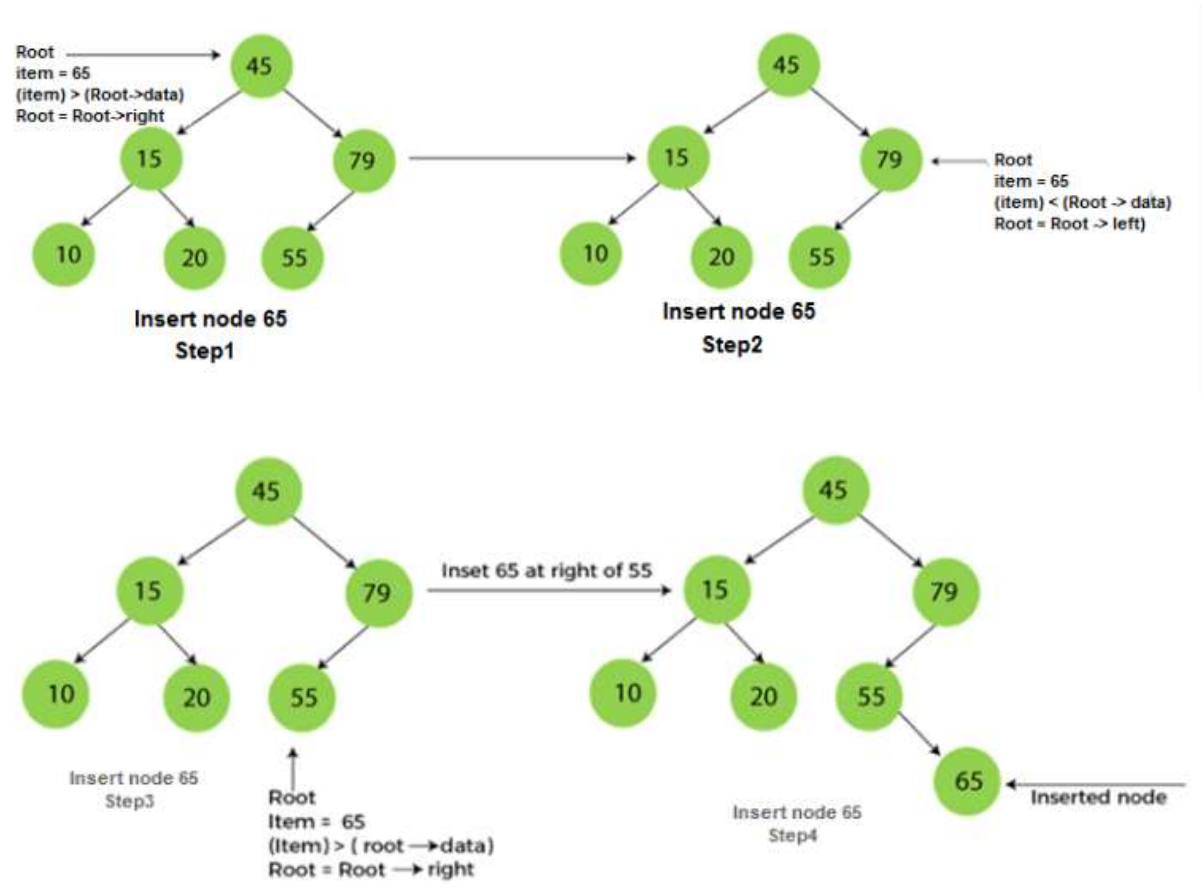


Figure 4. Insertion of node - Example 2

Initial Tree Structure

- The root node is 45.
- The left subtree of 45 contains nodes 15, 10, and 20.
- The right subtree of 45 contains nodes 79, 55.

Step-by-Step Insertion of Node 65

Step 1:

- Start at the root node (45).
- Since 65 is greater than 45, move to the right child of 45, which is 79.

Step 2:

- At node 79:
  - 65 is less than 79, so we move to the left child of 79, which is 55.

Step 3:

- At node 55:
  - 65 is greater than 55, so we move to the right child of 55.
  - The right child of 55 is currently empty, so this is where we'll insert 65.

Step 4:

- Insert 65 as the right child of 55.
- The insertion is now complete, and the tree maintains its binary search property.

12.5 Deletion from a Binary Search Tree

Deleting a node from a Binary Search Tree (BST) involves three main cases, each of which addresses the structure of the tree and ensures that it maintains the BST properties after deletion. Here's an explanation of each case:

12.5.1 Deleting a Leaf Node (No Children) - Case 1

If the node to be deleted has no children (it is a leaf node), simply remove it from the tree.

Example

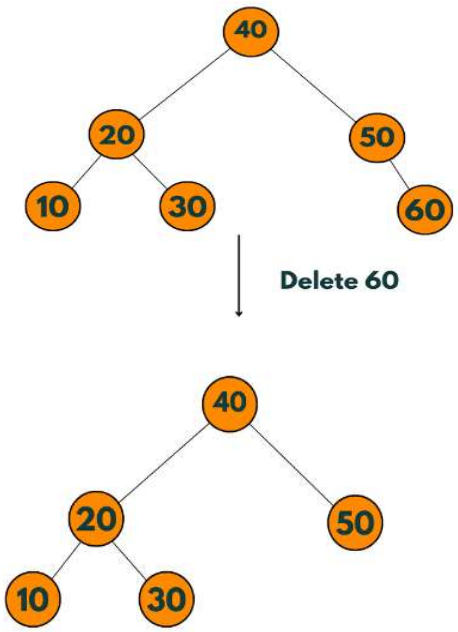


Figure 5. Deletion of a leaf node

- Suppose we want to delete the node with value 60:
  - Locate the node with value 60.
  - Remove it from the tree since it has no children.
- This is the simplest deletion case and doesn't affect the structure of the rest of the tree.

12.5.1 Deleting a Node with One Child - Case 2

If the node to be deleted has only one child, bypass the node to be deleted by linking its parent directly to its child.

Example

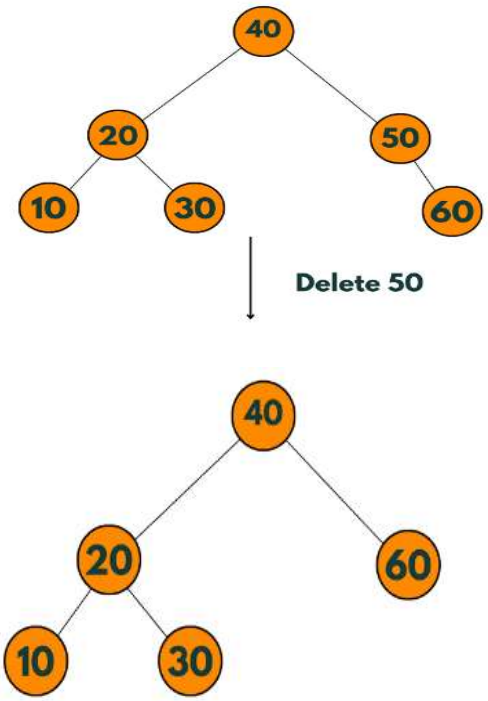


Figure 6. Deletion of a leaf node with one child

- Suppose we want to delete a node with value 50 that has only a right child, 60:
  - Locate the node with value 50.
  - Link the parent of 50 (e.g., 40) directly to 60, bypassing 150.
- After deletion, 40's right child will now be 60, and 50 is removed.

This operation maintains the BST properties because the single child of the deleted node still fits correctly within the tree.

12.5.3 Deleting a Node with Two Children - Case 3

If the node to be deleted has two children, replace it with either:

- The **inorder predecessor** (the largest node in its left subtree), or
- The **inorder successor** (the smallest node in its right subtree).

After replacing the node, delete the inorder predecessor or successor from its original position, as it will have at most one child.

Steps

1. **Find the Inorder Successor or Predecessor**:
   - If replacing with the inorder successor, find the smallest node in the right subtree.
   - If replacing with the inorder predecessor, find the largest node in the left subtree.
2. **Replace the Node**:
   - Replace the value of the node to be deleted with the value of the inorder successor (or predecessor).
3. **Delete the Inorder Successor or Predecessor**:
   - Since the successor or predecessor will have at most one child, delete it following Case 1 or Case 2.

Example

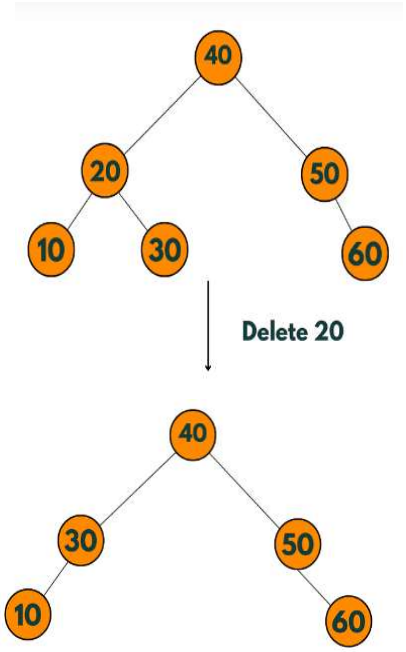


Figure 7. Deletion of a leaf node with two children

Suppose we want to delete a node with value 50 that has only a right child, 60

- **Locate the node with value 50.**
  - Starting from the root node (40), move to the right child, where we find 50.
- **Link the parent of 50 (which is 40) directly to 60, bypassing 50.**

- Since 50 has only a right child (60), we update 40's right pointer to point directly to 60, effectively bypassing and removing 50 from the tree.
- After deletion, 40's right child will now be 60, and 50 is removed.

- This keeps the Binary Search Tree properties intact, as all nodes to the right of 40 are still greater than 40.
-

➢ **Code for delete function**

```
void delete(list_pointer *ptr, list_pointer trail, list_pointer node)
{
   /* delete node from the list, trail is the preceding node
      ptr is the head of the list */
   if (trail)
      trail->link = node->link;
   else
      *ptr = (*ptr)->link;
   free(node); }
```

12.6 Height of a Binary Search Tree

The below figures illustrate the concepts related to the depth and structure of a binary tree: the distribution of nodes across depth levels and the formula for calculating the maximum number of nodes in a binary tree.

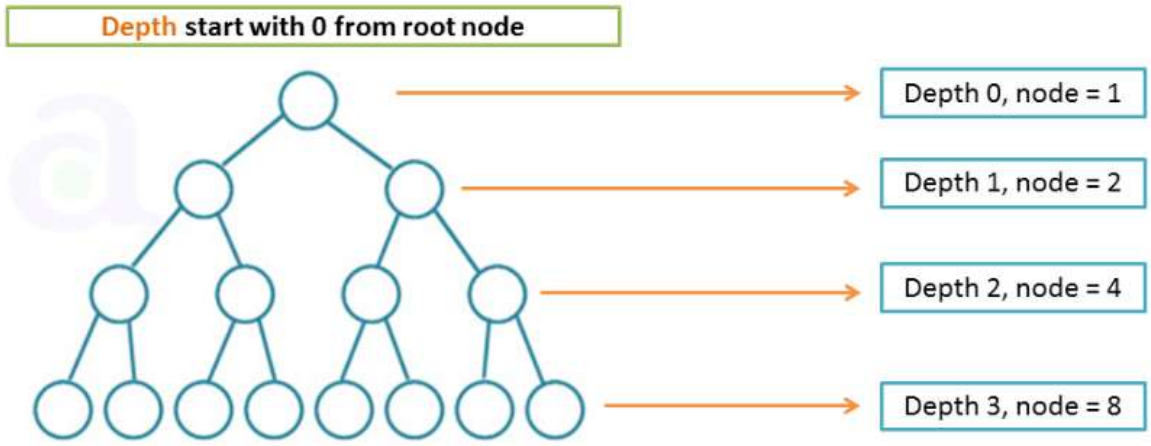


Figure 8. Depth Levels from 0

➢ **Depth Levels and Node Count (Starting from 0)**

- **Depth**: The depth of a node in a binary tree is the distance from the root node. Depth starts from 0 at the root level and increments by 1 as you move down each level.
- **Node Count at Each Depth**:
  - **Depth 0**: Contains 1 node (the root).
  - **Depth 1**: Contains 2 nodes.
  - **Depth 2**: Contains 4 nodes.
  - **Depth 3**: Contains 8 nodes.
- **Observation**: The number of nodes doubles at each depth level as you go deeper into the tree. This exponential growth is a key characteristic of binary trees, where each parent node can have up to two children.

Maximum Number of Nodes in a Binary Tree of Depth k

- **Formula**: The maximum number of nodes in a binary tree of depth k is given by the formula: $2^{k+1}-1$ where $k \geq 0$.
- **Example Calculation**:
  - For a tree of depth k=3:
    $2^{3+1}-1=2^4-1=16-1=15$
  - This result means that a full binary tree of depth 3 can have a maximum of 15 nodes.

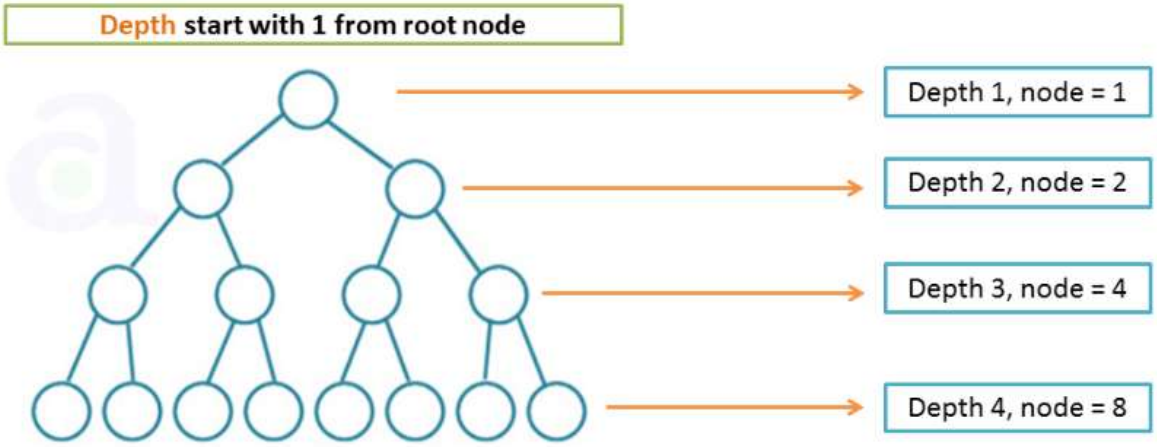


Figure 9. Depth Levels from 1

➢ **Depth Levels in a Binary Tree (Starting from 1)**

- **Depth**: Depth represents the level of a node in the tree, measured from the root node. Here, depth starts at 1 for the root node and increments by 1 as we move down each level.
- **Nodes at Each Depth**:
  - **Depth 1**: Contains 1 node (root node).

  - **Depth 2**: Contains 2 nodes.
  - **Depth 3**: Contains 4 nodes.
  - **Depth 4**: Contains 8 nodes.
- **Observation**: The number of nodes doubles at each subsequent depth level, which is characteristic of a binary tree where each parent can have two children.

Maximum Number of Nodes in a Binary Tree of Depth k

- **Formula**: The maximum number of nodes in a binary tree of depth k is given by: 2^k-1 where $k \geq 1$
- **Example Calculation**:
  - For a binary tree of depth k=4: $2^4-1=16-1=15$
  - This calculation indicates that a full binary tree with depth 4 can contain a maximum of 15 nodes.
- **Breakdown of Nodes by Depth**:
  - Depth 1: 1 node
  - Depth 2: 2 nodes
  - Depth 3: 4 nodes
  - Depth 4: 8 nodes
  - **Total**: 1+2+4+8=15

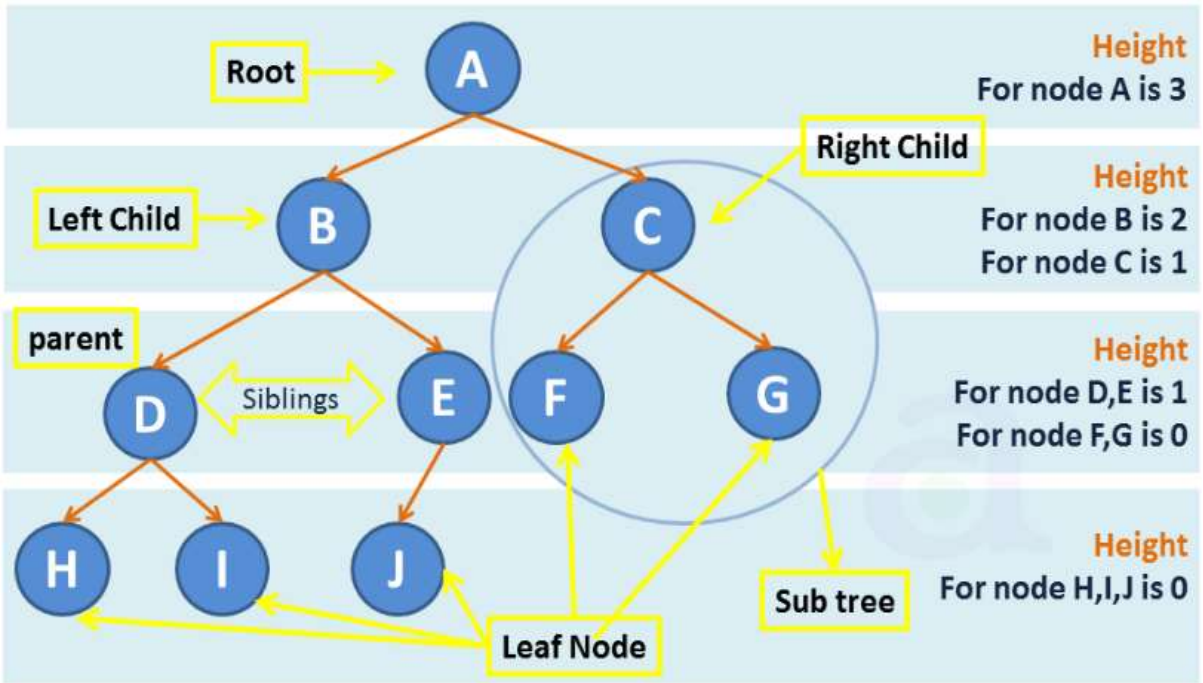


Figure 10. Height of the tree

The above figure shows the concept of tree height in a binary tree. The height of a tree is defined as the longest path from the root node to any leaf node. In this example, the height is 3,

as the root node A has a path of three edges to reach the farthest leaf nodes like H, I, and J. Each node has its own height based on the distance to its farthest child. For instance, node B has a height of 2, while leaf nodes such as H and J have a height of 0 because they have no children. The height of the tree affects the performance of operations like searching and inserting, as a shorter height generally makes these operations faster.

12.7 Balanced Search Trees:

A BST that maintains a balanced structure, with a height in the worst case of O(log 2n)O(\log_2 n)O(log2n), is termed a balanced search tree. These trees ensure optimal performance for BST operations, even in the worst case, because they avoid the issues associated with height degeneration.

Examples of Balanced Search Trees:

- **AVL Trees**: These trees maintain balance by ensuring that the heights of the left and right subtrees of any node differ by at most one. AVL trees perform rotations to maintain this balance, ensuring O(log n)O(\log n)O(logn) operations.
- **2-3 Trees**: A type of balanced search tree where each node can have 2 or 3 children. This structure helps maintain a balanced height, leading to efficient operations.
- **Red-Black Trees**: These trees are a type of balanced binary tree where each node is assigned a color (either red or black) and rotations are applied to maintain a balanced structure. This balance guarantees O(log n)O(\log n)O(logn) time for insertions, deletions, and searches.

Balanced BSTs, through their self-balancing properties, provide a robust solution to the problem of maintaining efficient operation times in dynamic datasets. They ensure that the tree height remains logarithmic, leading to optimal performance.

12.8 KEY TERMS

Binary search tree, Balanced BST, Inorder traversal, AVL trees, Red-Black trees.

12.9 ASSESSMENT QUESTIONS

1. What are the key properties of a binary search tree (BST)?
2. How does the height of a BST affect its performance?
3. Describe the three cases of node deletion in a BST.
4. What is the difference between a balanced and an unbalanced BST?
5. Explain the steps for searching an element in a BST.

12.10 SUGGESTED READINGS

1. "Data Structures and Algorithms in C" by Mark Allen Weiss
2. "Introduction to Algorithms" by Thomas H. Cormen et al.
3. "Data Structures Using C and C++" by Yedidyah Langsam et al.
4. "Fundamentals of Data Structures in C" by Ellis Horowitz et al.
5. Research articles on AVL Trees and Red-Black Trees in IEEE Xplore.

Mrs. Appikatla Pushpa Latha

Lesson – 13
Introduction to Graphs

OBJECTIVE S

The objective of this lesson is to

1. Gain a conceptual understanding of core Graph components, including vertices, edges, paths and cycles.

2. Differentiate between various graph types, such as directed vs. undirected and weighted vs. unweighted.

3. Understand key Graph algorithms and concepts, including graph traversals, shortest path
   algorithms, and minimum cost spanning trees.

Structure

13.1 THE GRAPH ABSTRACT DATA TYPE

The Graph Abstract Data Type (ADT) is a formal representation of a graph structure used to model relationships or connections among a set of entities. It provides a framework for defining the structure and operations applicable to graphs, facilitating efficient implementation and manipulation in computational tasks.

13.1.1 Origins and Significance of Graphs

Graphs, a cornerstone of mathematics and computer science, are essential for representing relationships among objects. Originating in the 18th century, their applications span various fields, making them indispensable tools.

Key points about their origins and significance:

- Graphs represent connections, such as nodes linked by edges.
- Their versatility enables them to model complex systems with ease.
- The study of graphs has evolved into a robust field with wide-ranging applications.

13.1.2 Applications of Graphs in Various Fields

Graphs play an integral role in numerous disciplines, facilitating problem-solving in real-world contexts. Some notable applications include:

1. **Computer Science**:
    - Algorithms like Breadth-First Search (BFS) and Depth-First Search (DFS).
    - Dependency resolution and scheduling tasks.
    - Representing data structures (e.g., trees, networks).
2. **Social Networks**:
    - Representing users as nodes and their relationships as edges.
    - Analysing connectivity, influence, and group dynamics.
3. **Biology**:
    - Gene networks, ecosystems, and protein interactions.
4. **Transport and Logistics**:
    - Traffic optimisation, shortest-path calculations, and delivery route planning.
5. **Electrical Engineering**:
    - Circuit design, where components are vertices, and connections are edges.

13.1.3 The Koenigsberg Bridge Problem

The Koenigsberg Bridge Problem, introduced by Leonhard Euler in 1736, is a historic example of practical graph theory. Euler examined the city of Koenigsberg (modern-day Kaliningrad), where four landmasses were connected by seven bridges. The challenge was to devise a walk crossing each bridge exactly once.This problem is considered the foundation of graph theory, as it introduced the use of mathematical structures to solve real-world problems involving networks and connectivity.

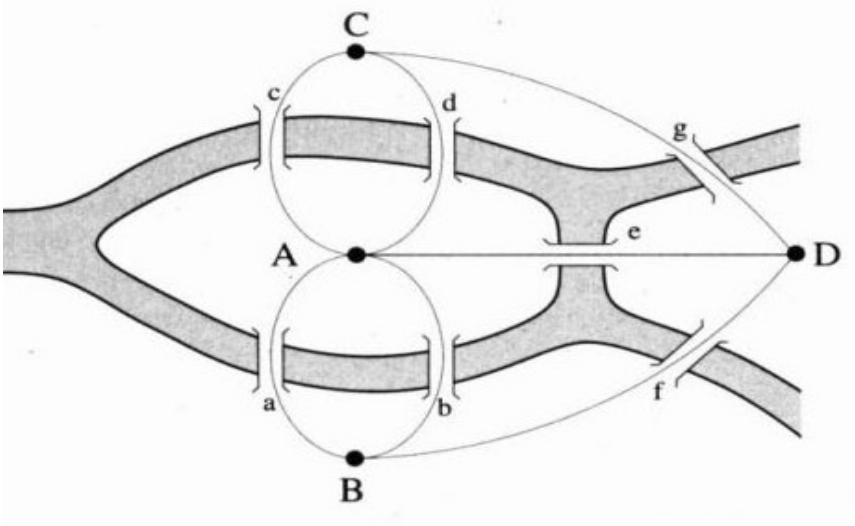


Fig 13.1.The Koenigsberg Bridge Problem

Euler's Formulation of Graph Theory

To solve the problem, Euler transformed the physical layout into a mathematical model, laying the groundwork for graph theory. His abstraction made it possible to reason mathematically about connectivity and the structure of networks.

Euler abstracted the problem into a graph by:

- Representing landmasses as vertices.
- Representing bridges as edges.

This mathematical representation allowed him to focus on the core connectivity issues rather than the physical arrangement, demonstrating the power of abstraction.

Euler's analysis of the Koenigsberg Bridge Problem revealed the conditions required to traverse all edges (bridges) exactly once. He formalized the problem mathematically and provided a solution.

The problem was stated as follows:

- Is it possible to traverse all bridges exactly once and return to the starting point?

Euler's Conclusion:

It was impossible, based on the degree of vertices:

- Each vertex (representing a landmass) must have an even degree (number of edges connected to it) for such a path to exist, except for at most two vertices.
- In the Koenigsberg graph, all four vertices had an odd degree, violating this condition.

This analysis introduced the concepts of:

1. Eulerian Path**:** A path that visits every edge exactly once, which is possible if exactly 0 or 2 vertices have an odd degree.
2. Eulerian Circuit: A path that starts and ends at the same vertex while visiting every edge exactly once, which is possible if all vertices have an even degree.

Euler's findings highlighted the importance of vertex degrees in determining the possibility of such walks, leading to a deeper understanding of graph connectivity.

Contribution to the Field of Graph Theory

Euler's work was a breakthrough in mathematics, as it established graph theory as a field and provided a systematic framework for analyzing networks.

His contributions included:

- **Formalisation of Graph Concepts**: Euler introduced fundamental terms like vertices, edges, paths, and circuits, which form the basis of graph theory today.
- **Definition of Eulerian Paths and Circuits**: These foundational concepts are still used in network design, optimisation, and various algorithmic applications.
- **Applications to Real-World Problems**: Euler's abstraction inspired solutions in diverse areas, including logistics, connectivity analysis, and resource management.
- **Demonstration of Abstraction**: By transforming physical layouts into mathematical models, Euler showcased how abstract reasoning can address real-world challenges.

This pioneering work laid the foundation for modern graph theory, influencing its applications in mathematics, computer science, and beyond. It remains a cornerstone of the study of networks and connectivity.

13.2 IMPORTANT TERMS IN GRAPHS

Terms which are key for graphs are mentioned below:

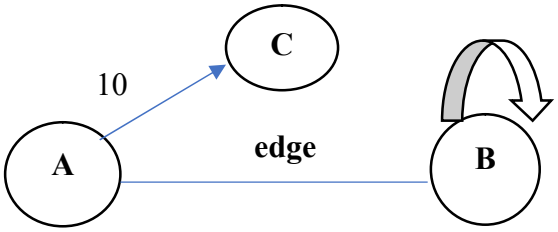


Fig 13.1. Graph

1. **Vertex:** A and B in the above image are called vertices or nodes
2. **Edge:** The connection between A and B is called Edge, an edge can have multiple data linked to it can be directional or non-directional or can be weighted or unweighted.
3. **Degree:** When a vertex or node has multiple connections or edges which states its degree with the same number of connections.
4. **Path :**The way from one vertex to another is called path in the case above path from B to C is from B to A and A to C.

5. **Cycle:** A path which starts and ends at the same point without visiting any other vertex or node. In the above figure there is a cycle from B to B.
6. **In degree**: The indegree of a node in a directed graph is the number of edges that are directed towardthat node. In simpler terms, it counts how many other nodes point to this node. For example, if three edges are pointing to a node, its indegree is 3.
7. **Out degree**: The outdegreeof a node in a directed graph is the number of edges that are directed outward from that node. It shows how many other nodes the given node points to. For example, if a node points to two other nodes, its outdegree is 2**.**
8. **Weight:** The weight of an edge in a graph is a numerical value assigned to the edge, representing a specific attribute or metric of the connection between two vertices. In the above graph the weight of edge A - C = 10.

13.3 TYPES IN GRAPHS

Graphs can be categorised into various types based on their structure, properties, and the relationships they represent.

Below are the figures and is a detailed explanation of the different types of graphs:

13.3.1 Directed vs Undirected : Graphs with an indication of traversal are called directed graphs and without any direction of traversal are called undirected graphs.

Table 1.Comparison: Directed vs Undirected Graphs

Aspect	**Directed Graphs**	**Undirected Graphs**
Edge Representation	u→v (one-way connection)	(u,v) (two-way connection)
Symmetry	Asymmetric	Symmetric
Degree	In-degree and Out-degree	Degree (total edges connected)
Path	Follows the direction of edges	No directional restriction
Applications	Data flow, web links, task scheduling	Social networks, physical networks
Complexity	More complex due to directionality	Simpler, as connections are mutual

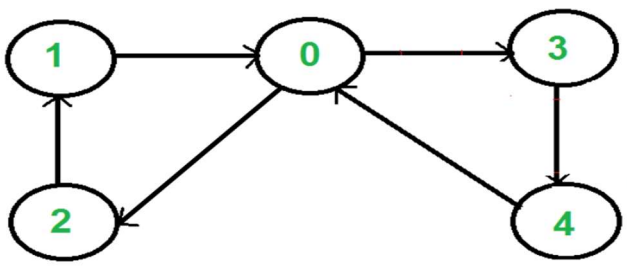


Fig 13.2. Directed graph

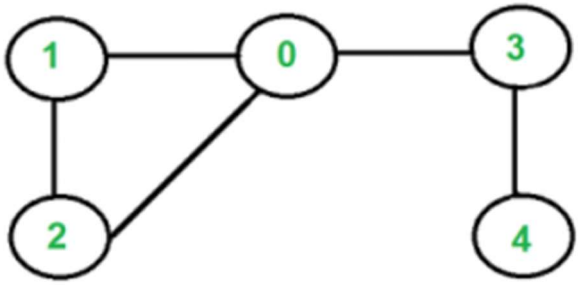


Fig 13.3. Undirected graph

13.3.2 Weighted and Unweighted: When each edge of graph has a distinct value then it is known to be a weighted and a graph without any values to the edges is called unweighted graphs.

Table 2. Comparison: Weighted Graphs and **Unweighted Graphs**

Aspect	**Weighted Graphs**	**Unweighted Graphs**
Edge Representation	Each edge has an associated weight www	Edges are treated equally without weights
Edge Attributes	Quantifies metrics like cost, distance, capacity, or time	Represents binary relationships (exist or not)
Algorithm Suitability	Algorithms like Dijkstra's, Prim's, and Kruskal's	BFS, DFS, and other simple traversal algorithms
Complexity	Requires additional storage and computation for weights	Simpler due to uniform treatment of edges
Applications	Cost-based problems, shortest paths, optimisation	Connectivity, reachability, and basic topology
Examples	Transportation networks (distance), financial costs	Social networks, basic graphs without attributes

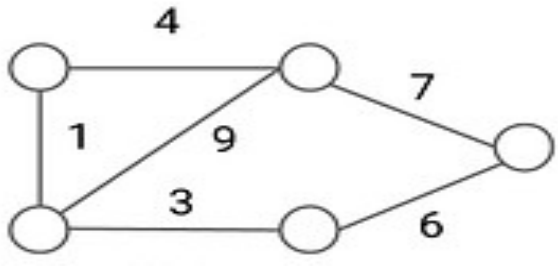


Fig 13.4. Weighted Graph

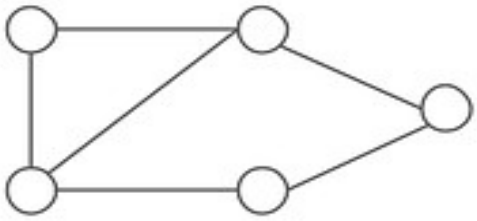


Fig 13.5. UnWeighted Graph

13.3.3 Cyclic Graphs vs Directed Acyclic Graphs: Graphs which consists of one more cycle of subgraphs then they are identified as Cyclic graphs and a graph with no subgraphs which have cycles are called Directed Acyclic graphs.

Table 3. Comparison: Cyclic Graphs and **Directed Acyclic Graphs (DAGs)**

Aspect	Cyclic Graphs	Directed Acyclic Graphs (DAGs)
Definition	Contains at least one cycle where a path starts and ends at the same vertex	A directed graph with no cycles
Edge Direction	Can be directed or undirected	Always directed
Structure	Allows closed paths or loops	No closed paths or loops
Examples	Road networks with circular routes	Task dependency charts, family trees
Applications	Modeling circular processes, networks with feedback	Scheduling, topological sorting, and dependency resolution
Path Characteristics	Cycles can make path traversal infinite	Paths are finite and acyclic
Algorithm Suitability	BFS/DFS can be used to detect cycles	Algorithms like Topological Sorting and Critical Path Analysis
Common Use Case	Electrical circuits with feedback loops	Workflow management, version control systems

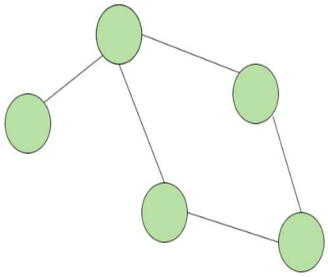


Fig 13.4 Cyclic graph

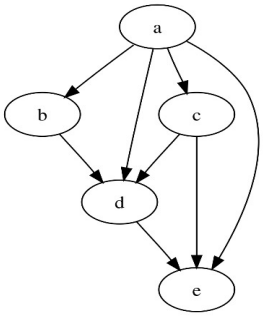


Fig 13.5. Directed Acyclic Graph

13.3.4 Complete Graphs: Every vertex or node are interconnected with each other; this scenario is named as Complete graphs.

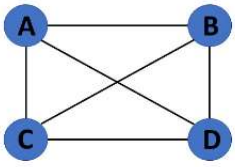


Fig 13.6. Complete Graph

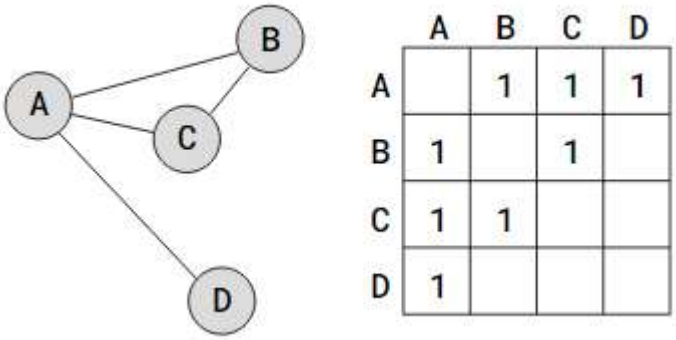
13.4 GRAPH REPRESENTATION

As it has been noted, graph representations are essential when it comes to understanding how to design the way, associations are represented within the data. This paper reveals that how

this representation selection impacts the time complexity of graph algorithms and their space complexity. Graphs can be represented in three main ways:

13.4.1 Adjacency Matrix:

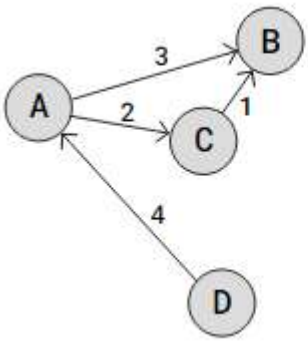
An **adjacency matrix** is a way to represent a graph using a two-dimensional table. Each row and column in the table represents a node in the graph. If there is a connection (or edge) between two nodes, the corresponding cell in the table is marked with a 1 (or the weight of the edge in weighted graphs). If there is no connection, the cell is marked with a 0. For undirected graphs, the table is symmetric because connections go both ways, but for directed graphs, it might not be. While it is simple to understand and allows quick checking of connections, it uses a lot of memory for graphs with very few edges because every possible pair of nodes must be stored, even if they are not connected.



	A	B	C	D
A		1	1	1
B	1		1	
C	1	1		
D	1			

Fig 13.7. Adjacency Matrix for UnDirected Graph

The Figure 13.7 illustrates a graph with four vertices A,B,C, and D, along with its adjacency matrix representation. The graph is undirected, meaning all connections between vertices are bidirectional. For instance, if A is connected to B, B is also connected to A. The adjacency matrix provides a tabular representation of the graph, where each row and column corresponds to a vertex, and a value of 1 indicates the presence of an edge between the respective vertices. For example, row A shows connections to B,C and D while row B reflects connections to A and C. The matrix is symmetric, reflecting the mutual nature of the connections in an undirected graph.



	A	B	C	D
A		3	2	
B				
C		1		
D	4			

Fig 13.8. Adjacency Matrix for Directed Graph

The Figure 13.8 shows a weighted directed graph and its adjacency matrix. The graph consists of four vertices A,B,C,D with directed edges carrying weights that represent costs or distances. For example, A→B has a weight of 3, A→Chas a weight of 2, C→B has a weight of 1, and A→Dhas a weight of 4. The adjacency matrix tabulates these weights, where each row corresponds to the source vertex and each column to the destination vertex. Blank cells indicate the absence of an edge. The graph is both directed and weighted, with the matrix reflecting the directionality and weights of edges.

Advantages of using Adjacency matrix:

- Accessing time or time complexity of using this representation method is *O(1),* among two vertices.
- Implementation is easy and can be used when there are good number of vertices connected with each other

Disadvantages

- **Memory Usage:** Storage is an aspect which makes it inefficient and has the storage in the order of V^2, where V represents number of vertices.

13.4.2 Adjacency List:

An adjacency list is a data structure used to represent a graph by listing all the nodes and their adjacent (connected) nodes. Each node in the graph is associated with a list of other nodes that it shares an edge with. For example, in an undirected graph, if two nodes are connected by an edge, each will appear in the other's adjacency list. This structure is especially efficient for representing sparse graphs (graphs with fewer edges compared to the number of nodes) because it only stores existing edges, saving memory compared to other representations like adjacency matrices. Adjacency lists are commonly implemented using arrays or hash tables, where each node points to a linked list or a dynamic array containing its neighbors. This format is widely used in algorithms like breadth-first and depth-first searches due to its simplicity and efficiency.

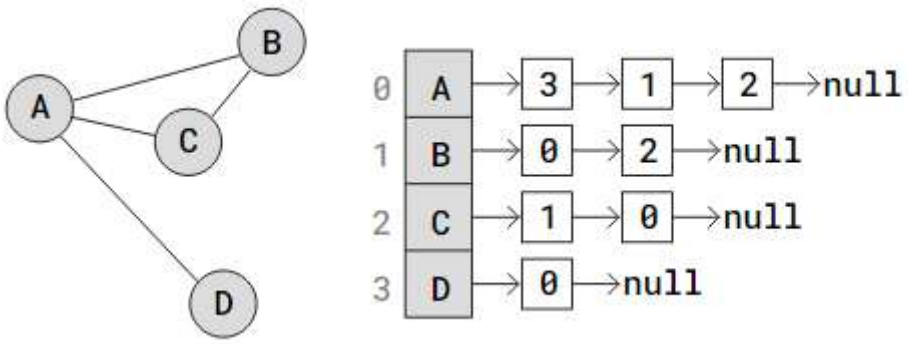
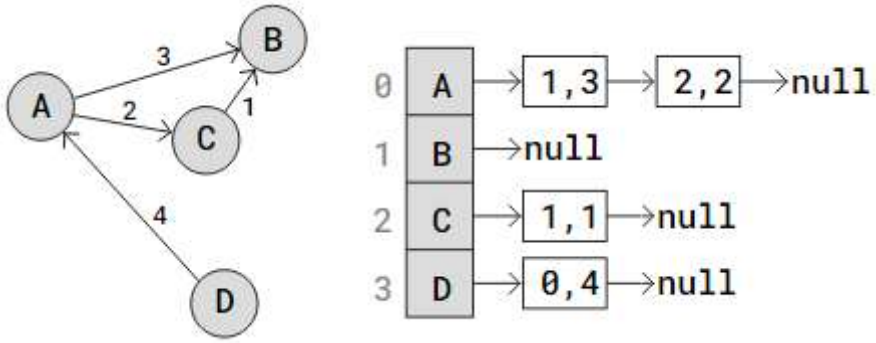


Fig 13.9 Weighted Directed Graph - Adjacency List

The above diagram represents a graph and its adjacency list representation. On the left, the graph consists of four nodes (A, B, C, and D) connected by edges. On the right, the adjacency list shows how each node is linked to others. Each node (e.g., A, B, etc.) is associated with a list of adjacent nodes, represented by their indices. For example, node

A (index 0) is connected to nodes D, B, and C (indices 3, 1, and 2). The adjacency list format efficiently stores the graph's connections, particularly for sparse graphs**.**



.Fig 13.9 Weighted Directed Graph - Adjacency List

The above figure shows a weighted directed graph and its representation using an adjacency list**.** The graph consists of four vertices A,B,C,D with directed edges and associated weights. The adjacency list represents each vertex as a node, followed by a linked list of its outgoing edges and their weights. For example, vertex A is connected to B, C, and D with weights 3, 1, and 2 respectively. Vertex B is connected to C with weight 2, while C is connected to B with weight 1. D has no outgoing edges. This structure is space-efficient and well-suited for sparse graphs

The below diagram represents another example for adjacent list representation of the graph in the memory.

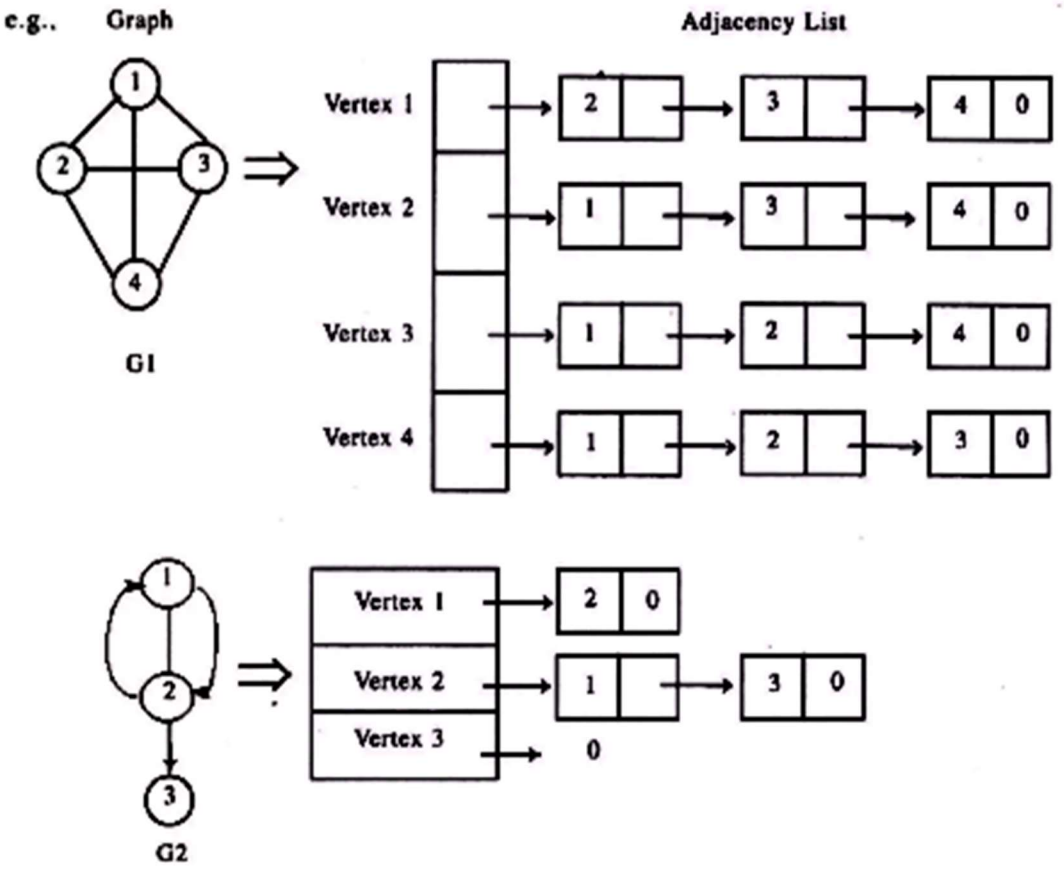


Fig 13.10 Graph - Adjacency List

Advantages:

- **Space:** Saves memory compared to the adjacency matrix and the storage is *O(V+E)* space (E is the number of edges and V is the number of vertices).

- **Efficiency:** Traversing across the graph especially when it comes to distance calculations is very efficient. Thus, this representation suites best for DFS (Depth First Search) and BFS (Breadth First Search).

13.4.3 Edge List

In an edge list, all the resulting edges are combined into a vectors of vertex (or triplets in weighted graphs) pairs.

Advantages:

- **Space:** Requires very less space with a complexity of *O(E)*space, this makes it very efficient in storage in cases where only edge relations are needed.
- **Flexible:** This representation method can be used for any type of graph. Thus, restricting one from using multiple data structures for a graph.

Disadvantages:

- Inefficient in traversals and verifying if vertices are connected.

13.4.4 Adjacency Multi Lists

An adjacency multi-list is a graph representation designed to efficiently store graphs, particularly undirected graphs, where edges are shared between two nodes. Unlike traditional adjacency lists or matrices, adjacency multi-lists use shared nodes to reduce redundancy and save memory, making them an efficient option for certain graph applications. Here is a detailed explanation:

In an adjacency multi-list, each edge is stored only **once** and is shared by the two vertices (nodes) it connects. Instead of duplicating edges in the adjacency lists of both connected nodes (as in a traditional adjacency list), pointers are used to manage the connection between the two nodes.
Each node maintains:

1. A list of edges connected to it.
2. Each edge contains pointers to the two vertices it connects and also pointers to the next edge in the adjacency list for each vertex.

Structure:

An adjacency multi-list has the following components:

1. Vertices (Nodes):
   - Each vertex has a pointer to the first edge in its adjacency list.
2. Edges:
   - Each edge has:
     - Two pointers to the vertices it connects.
     - Two pointers to the next edges in the adjacency list of both vertices.

Key Characteristics:

- Efficient Memory Usage:Edges are not duplicated in the adjacency lists of the two vertices, as they are shared through pointers**.**
- Flexible Navigation:By following pointers, you can easily traverse all edges connected to a specific vertex**.**
- Undirected Graphs:It is particularly useful for undirected graphs, as each edge is shared, avoiding duplication**.**

Example

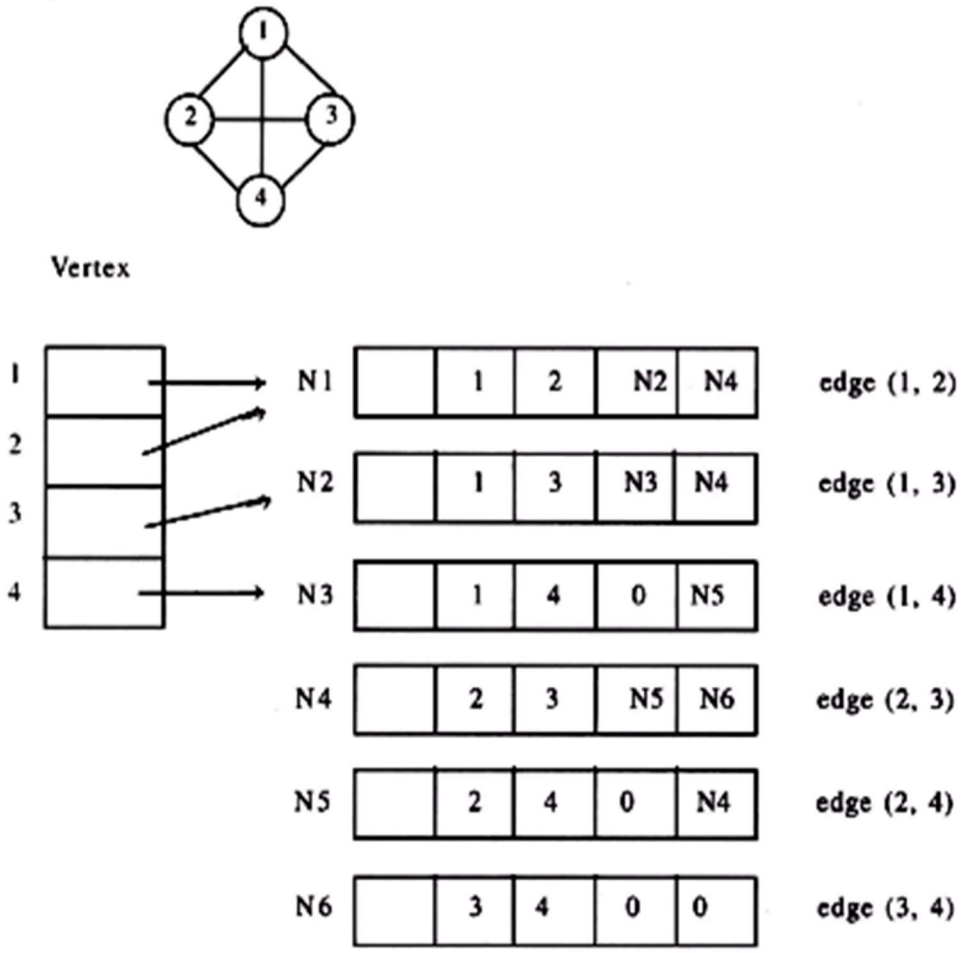


Fig 13.11. Adjacency Multi Lists

The above image represents an **adjacency multi-list** for a graph with four vertices (1, 2, 3, 4) and six edges connecting them. The adjacency multi-list efficiently stores the graph's edges without duplication. Here's an explanation:

Graph Structure:

The graph has:

- Vertices: 1,2,3,4
- Edges: (1,2),(1,3),(1,4),(2,3),(2,4),(3,4)

The adjacency multi-list stores:

1. The vertices as a list.
2. The edges as nodes (labeled N1, N2**,…**) shared between the connected vertices.

Explanation of the Representation:

1. Vertex List (Left Side):
    - Each vertex points to the first edge (node N) connected to it.
    - For example:
        - Vertex 1 points to N1 (representing edge 1−2).
        - Vertex 2 points to N4 (representing edge 2−3).
        - And so on.
2. Edge Nodes (Right Side):
    - Each edge node (e.g., N1,N2) stores:
        - Vertices it connects: Each edge node stores the vertices it connects. For instance:
            - N1 connects vertices 1 and 2 (edge 1−2).
            - N2 connects vertices 1 and 3 (edge 1−3).
        - Next Pointers for Each Vertex: Each edge node has two pointers:
            - One for the next edge in the adjacency list of the first vertex.
            - Another for the next edge in the adjacency list of the second vertex.
            - For example:
                - N1 has pointers to N2(next edge for vertex 1) and N4 (next edge for vertex 2).
3. Edge Labeling:
    - Each edge is labeled with its vertex pair, such as (1,2) for N1(1,3) for N2 and so on.

Traversal:

- To traverse all edges connected to a vertex, follow the pointers in its adjacency list.

  For example, starting from vertex 1:

  - Go to N1 (1−2).
  - Follow N1's pointer to N2 (1−3).
  - Then follow N2's pointer to N3 (1−4).

Advantages of the Representation:

1. **Space Efficiency**: Edges are stored only once, reducing redundancy compared to traditional adjacency lists.
2. **Efficient Traversal**: Pointers allow quick navigation between edges in the adjacency list of each vertex.
3. **Suits Undirected Graphs**: Perfect for representing undirected graphs where edges are shared between nodes.

This adjacency multi-list structure is an efficient way to represent sparse undirected graphs, offering both memory savings and flexibility for traversal.

13.5 Orthogonal Representation of a graph

The orthogonal representation of a graph is a specialized data structure used to efficiently represent a planar graph, where no edges cross each other. This method is particularly useful for graphs that are drawn on a plane and need to preserve the geometric relationships between edges and vertices. It focuses on encoding both the adjacency of vertices and the embedding (spatial layout) of edges around vertices.

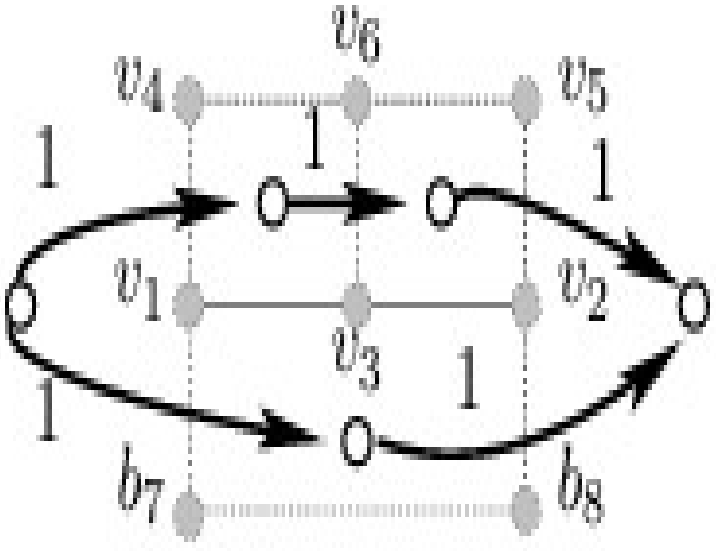


Fig 13.12. Orthogonal Representation of Graph

The above graph represents a directed planar graph with vertices and edges arranged in an orthogonal representation. Here's a detailed explanation:

1. Graph Components:

   Vertices (v1,v2,v3,v4,v5,v6)

   - The graph has six vertices (v1,v2,v3,v4,v5), each positioned at specific points on a grid.
   - The vertices are connected by directed edges (arrows) that indicate the direction of traversal between them.

2. Edges:

   - The edges connect the vertices with horizontal and vertical lines, bending at right angles where necessary.
   - For example:
     - There is an edge from v1 to v2.
     - Another edge goes from v3 to v5 through an intermediary vertex.
   - Each edge has a direction (indicated by the arrowhead).

3. Edge Weights:

   - Some edges are labeled with a weight, such as "1". These weights might represent costs, capacities, or distances in the context of the graph's application.

4. Planarity:

   - The graph is planar, meaning no two edges cross each other.

5. Right-Angle Bends:

   - The edges bend at right angles, maintaining the orthogonal structure. This is evident in the horizontal and vertical arrangement of edges.

Features:

1. Directed Edges:

   - The arrows show the flow of direction between vertices, indicating that this is a directed graph.

2. Weights:

   - Edge weights provide additional information. For example:
     - The edge between **v1** and **v2** is labeled "1".

- This suggests that traversing this edge has a weight or cost of 1.

3. Orthogonal Layout:

- The graph is embedded on a grid with edges restricted to horizontal and vertical segments. This makes the representation clear and systematic.

Possible Applications:

1. **Circuit Design:**

- This graph could represent an electrical circuit, where vertices are components and edges are connections.

2. **Network Flow:**

- It might represent a flow network, where weights denote capacities or distances.

3. **Routing Problems:**

- The graph could model a routing problem, with paths directed and weighted to optimize traversal.

Traversal Example:

- Starting at **v1**:
  - Follow the edge to **v2**.
  - From **v2**, you can traverse to **v5** via intermediate vertices.

This graph effectively models a structured, planar, and directed network, leveraging the orthogonal layout for clarity and practical use.

13.6 KEY TERMS

Vertex, Edge,Degree ,Path ,Cycle ,Directed Graph ,Undirected Graph , Weighted Graph , Adjacency Matrix ,Adjacency List.

13.7 SELF-ASSESSMENT QUESTIONS

1. Differentiate between a directed and an undirected graph with examples.

2. Explain the differences between an adjacency matrix and an adjacency list. Which is more space-efficient for sparse graphs?

3. What is the significance of vertex degrees in graph theory? Provide examples of in-degree and out-degree in directed graphs.

4. Describe how a cyclic graph differs from a directed acyclic graph (DAG). Give real-world applications of DAGs.

5. Discuss the advantages and disadvantages of using adjacency multi-lists for graph representation.

13.8 SUGGESTIVE READINGS

1. Fundamentals of Data Structures in C by Ellis Horowitz, Sartaj Sahni, and Susan Anderson-Freed (Chapter on Graphs and their representations).
2. Introduction to Graph Theory by Douglas B. West (Concepts of vertices, edges, paths, and cycles).
3. Algorithm Design by Jon Kleinberg and Éva Tardos (Graph representations and traversal techniques).

DR U SURYA KAMESWARI

Lesson - 14

Graph Algorithms

OBJECTIVES

The objectives of the lesson are

1. Understand the fundamental operations and traversal techniques in graph theory, including Depth First Search (DFS) and Breadth First Search (BFS).
2. Explore connectivity concepts such as connected components, strongly connected components, and biconnected components, including their applications in network analysis.
3. Learn about graph representation techniques like adjacency matrices and adjacency lists, as well as graph modification operations.
4. Analyse key graph algorithms such as shortest path algorithms, minimum spanning trees (MSTs), and articulation points for solving practical problems.

STRUCTURE

14.1 INTRODUCTION

A graph G=(V,E) consists of a set of vertices V and a set of edges E connecting pairs of vertices. Graphs can be undirected or directed, with the edges represented as ordered or unordered pairs of vertices.

Graph operations encompass the methods and techniques used to manipulate, explore, and analyse graphs, which are mathematical structures consisting of vertices (nodes) and edges (connections). Graphs are a versatile tool to model real-world systems such as social networks, road maps, communication networks, and computational processes. At their core, graph operations enable the processing and querying of relationships within a graph, providing insights into the structure and behaviour of interconnected systems. These operations serve as foundational building blocks for solving complex computational problems in areas like routing, scheduling, and clustering.

14.2 TYPES OF GRAPH OPERATIONS

Graph operations can be broadly categorized into basic operations that define the structural properties of graphs and functional operations that perform analysis or modify the graph. Below is a detailed explanation of the types of graph operations:

14.2.1 Basic Structural Operations

Adding and Removing Vertices and Edges

- **Adding Vertices**: Incorporate new nodes into the graph without altering the existing structure.
- **Removing Vertices**: Delete nodes and their associated edges.
- **Adding Edges**: Add new connections between vertices to modify graph relationships.
- **Removing Edges**: Remove specific edges while keeping vertices intact.

These operations are fundamental in dynamic graphs where the structure evolves, such as in network expansion or editing social graphs.

14.2.2 Traversal Operations

Traversal operations are used to explore all or parts of a graph systematically. They serve as the basis for understanding connectivity and reachability in the graph.

Depth First Search (DFS)

- DFS explores a graph by going as deep as possible along a branch before backtracking.
- It is useful for detecting cycles, topological sorting, and exploring connected components.

Breadth First Search (BFS)

- BFS explores all neighbours of a vertex before moving to vertices at the next level.
- It is commonly used for finding the shortest path in unweighted graphs and checking bipartiteness.

14.2.3 Connectivity Analysis

Connectivity analysis explores the relationships and paths between vertices in a graph, helping to identify cohesive structures and ensure network robustness. It plays a critical role in understanding graph components, clustering, and resilience in networks.

Connected Components

- Identifies subsets of vertices such that there is a path between any two vertices in the subset.
- Used in clustering and network resilience analysis.

Strongly Connected Components (Directed Graphs)

- Identifies subsets of vertices in directed graphs where every vertex is reachable from every other vertex in the subset.

14.2.4 Path and Distance Operations

These operations determine connectivity and optimal routes between vertices.

Shortest Path

Finds the minimum distance between two vertices. Algorithms include:

- Dijkstra's Algorithm for weighted graphs with non-negative edges.
- Bellman-Ford Algorithm for graphs with negative weights.
- Floyd-Warshall Algorithm for finding shortest paths between all pairs of vertices.

Transitive Closure

- Determines whether a path exists between any pair of vertices. Useful in reachability analysis.

14.2.5 Spanning Tree Operations

Minimum Spanning Tree (MST)

- A spanning tree connects all vertices in a graph with the minimum possible total edge weight.
- Algorithms:
  - Kruskal's Algorithm: Uses edge sorting and union-find for MST generation.
  - Prim's Algorithm: Builds MST by expanding the tree one vertex at a time.

Spanning Forest

- For disconnected graphs, a spanning forest is a collection of spanning trees for each connected component.

14.2.6 Graph Modification Operations

Graph Subdivision

- Subdivide edges by inserting additional vertices, often used in graph transformations.

Graph Merging

- Combine multiple graphs into a single graph, merging their vertices and edges.

Graph Complementation

- Transform the graph such that edges present in the original graph are absent in the complement, and vice versa.

14.2.7 Graph Representation Operations

Adjacency Matrix Conversion

- Represent graphs using a square matrix, where entries denote edge presence or weight.

Adjacency List Conversion

- Represent graphs using lists for each vertex, showing connected vertices.

These conversions enable efficient storage and manipulation of graphs based on their density and size.

14.2.8 Graph Search Operations

Cycle Detection

- Identifies cycles within a graph. It is crucial for understanding feedback loops and ensuring acyclic structures, such as in directed acyclic graphs (DAGs).

Topological Sorting

- Produces a linear ordering of vertices in a DAG such that for every directed edge **(u,v)**, vertex **u** precedes **v**. Useful in scheduling tasks with dependencies.

14.2.9 Edge Classification and Articulation Points

Edge Classification

- Classifies edges in DFS traversal as tree, back, forward, or cross edges. This classification is essential for analysing graph cycles and structures.

Articulation Points and Bridges

- Articulation Points: Vertices whose removal disconnects the graph.
- Bridges: Edges whose removal disconnects the graph.

Graph operations are essential tools for analysing and manipulating graphs. From basic structure modification to advanced traversal and pathfinding, they provide the means to solve practical problems across disciplines. Understanding these operations helps develop efficient algorithms for complex systems, enabling better optimization and insight into interconnected data.

14.3 DEPTH FIRST SEARCH (DFS)

Depth First Search (DFS) is a fundamental graph traversal algorithm that explores a graph by traversing as deep as possible along a branch before backtracking. This strategy is analogous to exploring a maze by venturing down one path until you reach a dead end, and then retracing your steps to explore other unvisited paths. DFS is widely used for tasks such as cycle detection, path finding, and connectivity analysis in graphs.

DFS can be applied to both directed and undirected graphs, as well as to weighted and unweighted graphs, though it primarily considers the structure of the graph rather than edge weights.

14.3.1 Working Principle

DFS employs a stack-based approach to track traversal paths. This stack can be implemented explicitly or implicitly through recursion. The algorithm starts from a source vertex, marking it as visited, and then recursively visits its unvisited adjacent vertices. If no unvisited adjacent vertices remain, the algorithm backtracks and continues to explore other vertices.

The main components of DFS are:

1. Visited Array: An array to keep track of whether a vertex has been visited.
2. Adjacency Representation: Either an adjacency list or an adjacency matrix to store the graph structure.
3. Stack (or Recursive Call Stack): To track the current path of exploration.

14.3.2 Steps in DFS

1. Start at a source vertex, mark it as visited, and push it onto the stack (if using an explicit stack).
2. Visit the first unvisited adjacent vertex, mark it as visited, and push it onto the stack.
3. Repeat the process for the current vertex until all adjacent vertices have been visited.
4. If a vertex has no unvisited adjacent vertices, backtrack by popping vertices from the stack until a vertex with unvisited neighbours is found.
5. Continue the process until all vertices reachable from the source vertex are visited.
6. If there are still unvisited vertices in the graph, repeat the process from a new source vertex.

14.3.3 Algorithm for DFS

Here is the recursive implementation of DFS:

```
Void dfs(int vertex, int graph[MAX][MAX], int visited[], int n)
{
// Mark the current vertex as visited
   visited[vertex] = 1;
printf("%d ", vertex);
// Explore all adjacent vertices
for (inti = 0; i< n; i++) {
```

```
if (graph[vertex][i] == 1&& !visited[i]) {
dfs(i, graph, visited, n);
 } }
 }
```

The above code is explained as

- graph[MAX][MAX] is the adjacency matrix of the graph.
- visited[] is an array where each element corresponds to whether a vertex has been visited.
- `n` is the total number of vertices in the graph.
- The function marks the current vertex as visited, prints it, and recursively explores all its unvisited neighbours.

14.3.4 Example

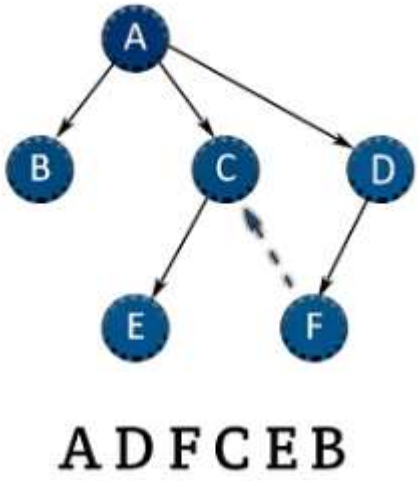


Fig 14.1. Given Graph for DFS traversal

DFS explores a graph as deep as possible along each branch before backtracking.

Let us analyze how DFS would traverse the graph shown in the image step by step.

- Vertices**:** A,B,C,D,E,F
- Edges:
  - A→B,A→C,A→D
  - C→E,C→F
  - D→F
- Starting Point: Let us assume DFS starts at vertex A.

14.3.5 DFS Traversal Step-by-Step

1. Start at Vertex A:
   - Mark A as visited.
   - Move to the first unvisited adjacent vertex of A, which is B.
2. Visit Vertex B:

   - Mark B as visited.
   - Since B has no outgoing edges (no adjacent vertices), backtrack to A.
3. Backtrack to Vertex A:
   - From A, move to the next unvisited adjacent vertex, which is C.
4. Visit Vertex C:
   - Mark C as visited.
   - Move to the first unvisited adjacent vertex of C, which is E.
5. Visit Vertex E:
   - Mark E as visited.
   - Since E has no outgoing edges, backtrack to C.
6. Backtrack to Vertex C:
   - From C, move to the next unvisited adjacent vertex, which is F.
7. Visit Vertex F:
   - Mark F as visited.
   - Since F has no outgoing edges, backtrack to C, then to A.
8. Backtrack to Vertex A:
   - From A, move to the next unvisited adjacent vertex, which is D.
9. Visit Vertex D:
   - Mark D as visited.
   - From D, move to its unvisited adjacent vertex F, but since F is already visited, stop further exploration.

DFS Traversal Order

The order of vertices visited in DFS (starting at A) is:

$$A \rightarrow B \rightarrow C \rightarrow E \rightarrow F \rightarrow D$$

14.3.6 Key DFS Characteristics in the Graph

1. Exploration Depth: The algorithm explores as deep as possible along each branch before backtracking.
2. Recursive Nature: DFS uses recursion (or an explicit stack) to maintain the exploration state. Backtracking occurs when no unvisited adjacent vertices are available.
3. Edge Classification:
   - Tree Edges: These are the edges used during DFS traversal

     (e.g., $A \rightarrow B, A \rightarrow C, C \rightarrow E, C \rightarrow F$)

   - Back Edges: These connect a vertex to an ancestor in the DFS tree (none in this graph since it's acyclic).
4. Visited Status: Vertices B,E,F,D are marked visited as they are explored.

14.3.7 Applications and Complexity Analysis

DFS (Depth First Search) is a fundamental graph traversal algorithm that explores as deep as possible along each branch before backtracking, making it a versatile tool for solving various graph-related problems and computational tasks. Some of the applications of DFS are

1. Pathfinding and Reachability
2. Cycle Detection
3. Topological Sorting
4. Connected Components
5. Graph Coloring
6. Maze and Puzzle Solving

DFS is used in connectivity checking, topological sorting, and cycle detection. The time complexity is O(V+E), where V is the number of vertices and E is the number of edges.

14.4 BREADTH FIRST SEARCH (BFS)

Breadth First Search (BFS) is a graph traversal algorithm that explores all vertices at the current level (or depth) before moving on to vertices at the next level. BFS uses a **queue** data structure to maintain the order of exploration, ensuring that all vertices are visited layer by layer. It is particularly effective for finding the shortest path in unweighted graphs and for exploring all reachable vertices from a given source

14.4.1 Algorithm for BFS

1. Start at the source vertex and mark it as visited.
2. Add the source vertex to a queue.
3. While the queue is not empty:
    - Remove (dequeue) the vertex at the front of the queue.
    - Visit all its unvisited adjacent vertices, mark them as visited, and enqueue them.
4. Repeat until all reachable vertices are visited.

14.4.2 Implementation of BFS

```
void bfs ( int start, int graph[MAX][MAX], int visited[], int n) {

   int queue[MAX], front = 0, rear = 0; // Initialize queue

   visited[start] = 1;                  // Mark the starting vertex as visited

   queue[rear++] = start;               // Enqueue the starting vertex


   while (front < rear) {               // Loop while the queue is not empty

      int current = queue[front++];     // Dequeue the front vertex

      printf("%d ", current);           // Process the current vertex

      for (int i = 0; i< n; i++) {      // Check adjacent vertices

         if (graph[current][i] == 1 && !visited[i]) {
```

```
            visited[i] = 1;          // Mark adjacent vertex as visited

            queue[rear++] = i;       // Enqueue the adjacent vertex

        }   }    } }
```

The explanation of the code is represented as

1. Input Parameters**:**
   - start: The starting vertex for BFS traversal.
   - graph[MAX][MAX**]**: An adjacency matrix representing the graph.
   - visited[]: An array to track whether a vertex has been visited.
   - n: The number of vertices in the graph.
2. Initialization**:**
   - queue[]: A simple array acts as a queue to store vertices to explore.
   - front and rear: Indices to manage the queue (FIFO).
3. Algorithm Steps**:**
   - Start at the **s**tart vertex, mark it as visited, and enqueue it.
   - While the queue is not empty:
     - Dequeue the vertex at the front and process it (print it in this example).
     - For each adjacent vertex of the current vertex:
       - If it is not visited, mark it as visited and enqueue it.
4. Output**:**
   - The traversal order of vertices starting from the source.

14.4.3 Example

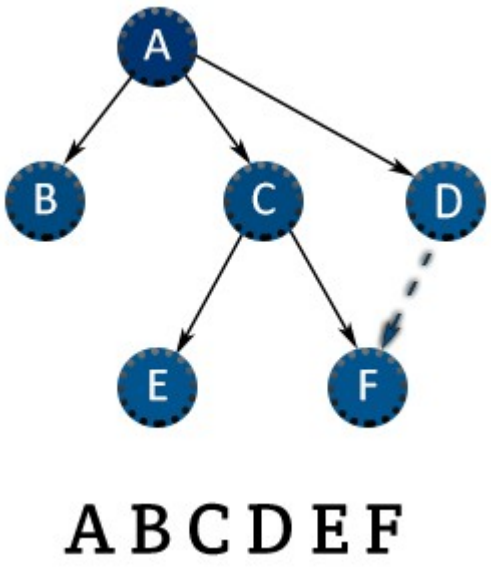


Fig 14.2.Given Graph for BFS traversal

The above graph is a directed graph with six vertices labeled A, B, C, D, E, F. The edges represent one-way connections between the vertices.

Edges

- A→B,A→C,A→D
- C→E,C→F
- D→F

BFS Step-by-Step

BFS explores the graph level by level, starting from a source vertex. Let's assume the traversal starts at vertexA. BFS will visit all vertices reachable from A in the order of their levels in the graph.

1. **Initialization**:
   - Start at vertex A.
   - Mark A as visited and enqueue it.
2. **Level 0**:
   - Dequeue A (current vertex).
   - Visit all its unvisited adjacent vertices B,C,D mark them as visited, and enqueue them.
3. **Level 1**:
   - Dequeue B (next in queue). Since B has no outgoing edges, move to the next vertex in the queue.
   - Dequeue C. Visit its unvisited adjacent vertices E and F, mark them as visited, and enqueue them.
   - Dequeue D. Since F is already visited, no new vertices are added to the queue.
4. **Level 2**:
   - Dequeue E and F sequentially. Since they have no unvisited adjacent vertices, the traversal ends.

14.4.4 BFS Traversal Order

The order in which vertices are visited during BFS (starting from A) is:

A→B→C→D→E→F

Visual Representation of Levels

- Level 0**:** A
- Level 1**:** B,C, D
- Level 2**:** E, F

Applications and Complexity Analysis

BFS is a powerful graph traversal algorithm that explores vertices level by level. It is widely used in various domains, particularly where systematic exploration of nodes or shortest path determination is required. Below are some key applications of BFS:

1. Shortest Path in Unweighted Graphs
2. Connectivity Testing
3. Bipartite Graph Checking
4. Finding Connected Components
5. Web Crawling

BFS is used in finding shortest paths in unweighted graphs and checking bipartiteness. Its complexity is O(V+E).

14.5 CONNECTED COMPONENTS

A connected component in a graph is a maximal subset of vertices such that there exists a path between any two vertices within the subset. In simpler terms:

- In an undirected graph, a connected component is a subgraph where all vertices are reachable from each other.
- In a directed graph, the concept splits into:
  - Weakly Connected Components: When ignoring edge directions, all vertices in the component are connected.
  - Strongly Connected Components (SCCs): All vertices in the component are mutually reachable following edge directions.

14.5.1 Types of Connected Components

1. In Undirected Graphs**:**

   - A connected component includes all vertices that are reachable from any vertex in the component.
   - An undirected graph can have one or more connected components.

2. In Directed Graphs**:**

   - A weakly connected component is formed if the graph is treated as undirected.
   - A strongly connected component is a subset where every vertex can reach every other vertex in the component, considering the direction of edges.

14.5.2 Algorithm to Find Connected Components

1. For Undirected Graphs (Using BFS or DFS)**:**

   - Initialize all vertices as unvisited.
   - Perform BFS or DFS from any unvisited vertex. Mark all reachable vertices as part of the same connected component.
   - Repeat the process for unvisited vertices until all vertices are processed.

2. For Directed Graphs (Strongly Connected Components)**:**

   - Use algorithms like Kosaraju's Algorithm, Tarjan's Algorithm, or Gabow's Algorithm to identify SCCs.
   - These algorithms rely on DFS to explore and process the graph efficiently.

14.6 SPANNING TREES

A spanning tree of a graph is a subgraph that:

1. Includes all the vertices of the original graph.
2. Is connected (there is a path between any two vertices).
3. Contains no cycles (it is a tree).

For a graph with V vertices, a spanning tree will always have V−1 edges. A graph can have multiple spanning trees.

14.6.1 Properties of Spanning Trees

1. **Connection**: A spanning tree ensures that all vertices in the graph are connected.
2. **No Cycles**: By definition, a tree does not have cycles, and neither does a spanning tree.
3. **Minimum Edges**: The number of edges in a spanning tree is always V−1, where V is the number of vertices in the graph.
4. **Unweighted Graphs**: Spanning trees are created without considering edge weights. For weighted graphs, we use algorithms to find a minimum spanning tree (MST).

14.6.2 Types of Spanning Trees

1. Standard Spanning Tree**:**
   - A spanning tree is derived from an unweighted graph and connects all vertices with V−1 edges.
2. Minimum Cost Spanning Tree (MST)**:**
   - A spanning tree for a weighted graph that minimizes the total weight of its edges. Algorithms like Prim's and Kruskal's are used to find MSTs.

14.6.3 Algorithms for Finding Spanning Trees

1. Depth First Search (DFS) or Breadth First Search (BFS)**:**
   - DFS or BFS can be used to construct a spanning tree by exploring all vertices and adding edges to the tree as they are discovered.
2. Kruskal's Algorithm**:**
   - A greedy algorithm that finds the MST by sorting edges by weight and adding them to the spanning tree if they do not form a cycle.
3. Prim's Algorithm**:**
   - A greedy algorithm that starts with a single vertex and grows the MST by adding the smallest edge connecting a vertex in the tree to a vertex outside the tree.

14.6.4 DFS and BFS Spanning Trees

DFS generates a depth-first spanning tree, and BFS generates a breadth-first spanning tree. These structures are useful in circuit analysis and designing efficient communication networks.

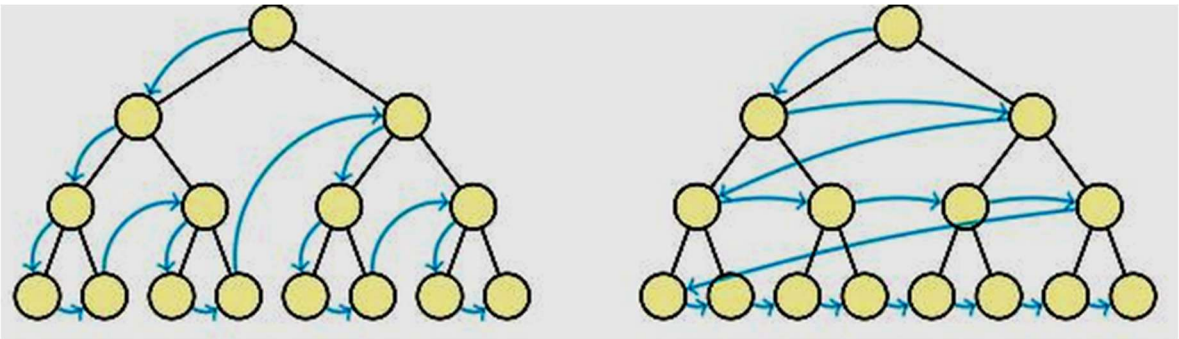


Fig 14.3. DFS and BFS Spanning Trees

14.7 BI-CONNECTED COMPONENTS

A bi-connected component (BCC) is a part of a graph where:

- Removing any single vertex does not disconnect the subgraph.
- All the vertices in this component are strongly connected.

In simpler terms, it's a group of vertices and edges that are highly connected and resilient to the failure of one vertex. The key points can be represented as

1. Bi-connected components may share common vertices called articulation points.
2. BCCs are useful to find strong and reliable parts of a graph.

In a network of cities connected by roads:

- A bi-connected component would represent a group of cities where removing one city does not break the connections between others.

14.8 ARTICULATION POINTS

An articulation point (also known as a cut vertex) is a vertex in a graph whose removal increases the number of connected components. In simpler terms, it is a critical node that holds different parts of the graph together. Removing an articulation point can disconnect parts of the graph or isolate certain vertices.

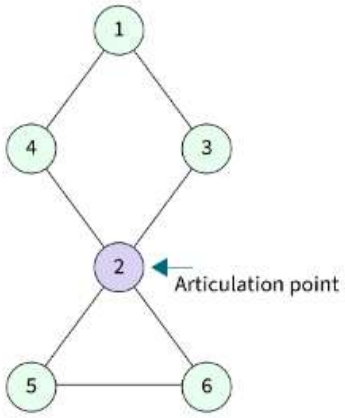


Fig 14.4. Graph with Articulation Point

The figure shows a graph where vertex 2is anarticulation point. Removing vertex 2 disconnects the graph into multiple components, isolating vertices 1,4 from 3,5,6.

14.8.1 Properties of Articulation Points

1. Articulation points are critical for maintaining the connectivity of a graph.
2. A vertex u is an articulation point if:
   - It is the root of the DFS tree and has two or more child subtrees.
   - It is not the root, but at least one child v satisfies low[v]≥discovery[u], where:
     - low[v] is the smallest discovery time reachable from v or its descendants.
     - discovery[u] is the time when u is first visited during DFS traversal.

14.8.2 Identification of Articulation Points Using DFS

Articulation points can be identified using DFS by assigning each vertex a discovery time during the traversal. Alongside this, the algorithm computes a low[ ] value for each vertex, which tracks the earliest reachable vertex (based on discovery time) from any of its subtrees. A vertex is determined to be an articulation point if removing it increases the number of connected components in the graph, which is evaluated based on its discovery and low[ ] values during the DFS traversal.

Algorithm Steps:

- Perform DFS and assign discovery and low values to all vertices.
- Check for articulation point conditions during traversal:
- Root with two or more child subtrees.
- Non-root vertices satisfying low[v]≥discovery[u].

Example 1 :

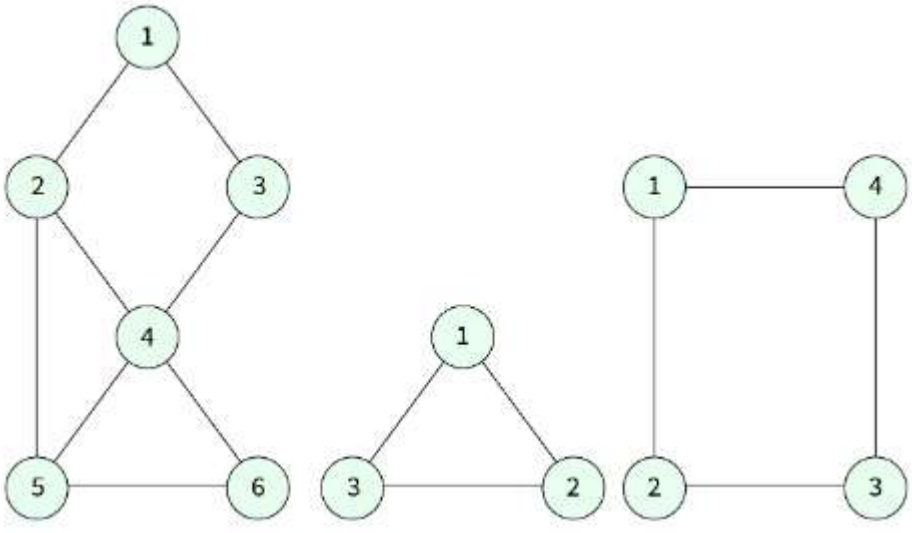


Fig 14.5. Graphs Demonstrating Articulation Points

The above figure consists of three disjoint subgraphs- Graph 1, Graph 2 and Graph 3. Each part has its own connectivity structure, and we will identify the bi-connected components (BCCs**)** within each subgraph. A bi-connected component is a maximal subgraph where removing any single vertex does not disconnect the component.

Graph1

Vertices: {1,2,3,4,5,6}
Edges: {1−2,1−3,2−4,3−4,4−5,4−6}

1. **Bi-Connected Components**:
    - BCC 1: {1,2,4,3}
        - This subgraph forms a cycle where removing any single vertex does not disconnect the rest.
    - BCC 2: {4,5}
        - Vertex 4 connects directly to 5, and removing 4 isolates 5.
    - BCC 3: {4,6}
        - Similarly, vertex 4 connects directly to 6.
2. **Articulation Points**:
    - 4: Removing 4 disconnects 5 and 6 from the rest of the graph.

Graph 2

Vertices: {1,2,3}
Edges: {1−2,1−3,2−3}

1. **Bi-Connected Components**:
    - BCC 1: {1, 2, 3}
        - This subgraph forms a complete triangle (cycle), so it is fully bi-connected.
2. **Articulation Points**:
    - None: All vertices are part of the same bi-connected component, and removing any single vertex does not disconnect the graph.

Graph 3

Vertices: {1,2,3,4}
Edges: {1−2,2−3, 3−4, 4−1}

1. **Bi-Connected Components**:
    - BCC 1: {1, 2, 3, 4}

- This subgraph forms a cycle, so it is bi-connected.

2. **Articulation Points**:
   - None: All vertices are part of the same bi-connected component, and removing any single vertex does not disconnect the graph.

Example 2 :

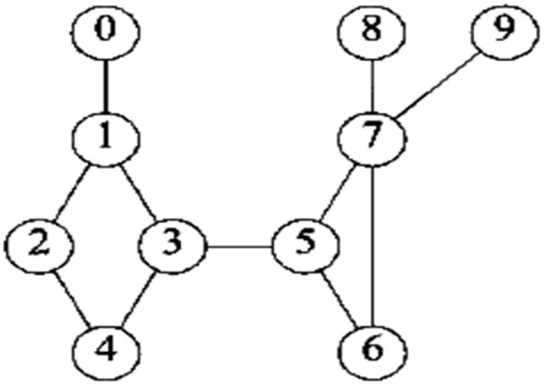


Fig 14.6. Connected Graph

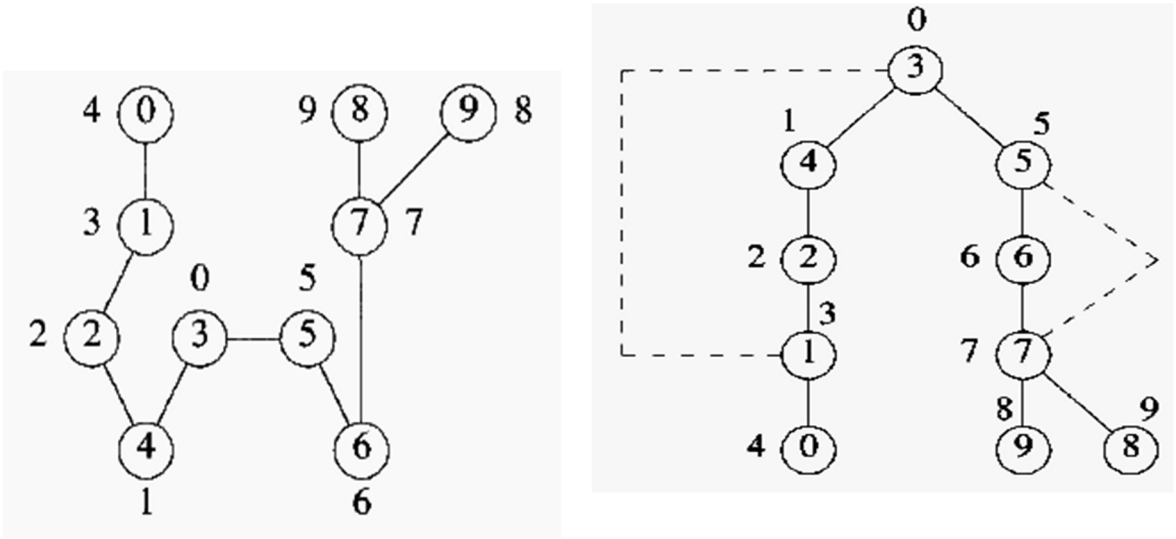


Fig 14.7. Depth First Spanning tree of Fig.14.6

Graph Representation

- The first graph is a connected undirected graph.
- The second graph represents its DFS spanning tree, with the order of visitation (DFS numbers) and back edges marked.
- The third graph redraws the spanning tree, highlighting the articulation points and low values.

14.8.3 Depth-First Search (DFS) in Biconnected Components

DFS is used to traverse the graph and identify biconnected components as follows:

1. Start from any vertex (e.g., vertex 3).
2. Create a Spanning Tree:

   Mark edges as:

   - Tree edges: Edges in the DFS spanning tree.
   - Back edges: Edges that connect a vertex to one of its ancestors in DFS.

Calculating Key Values

DFS Number (dfn):

The DFS number is the order in which a vertex is visited during the DFS traversal. For example:

- In the graph, dfn(3) = 0, dfn(0) = 4, and dfn(9) = 8.

Low Value (low):

The low value of a vertex (low[u]) is the smallest DFS number that can be reached from the vertex u by:

1. Its descendants.
2. At most one back edge.

Formula for low[u]:

low[u] = min(dfn(u),
min(low(w)) for all children w of u,
min(dfn(v)) for all back edges (u, v))

Example:

In the DFS spanning tree:

- Vertex 3 (DFS root):
  - low[3] = min(dfn(3), low(4), low(5), ...).
- Vertex 7:
  - low[7] = min(dfn(7), low(8), dfn(9)).

Identifying Articulation Points

A vertex u is an articulation point if:

1. It is the root of the DFS spanning tree and has two or more children.
2. For any child w of u: low(w) >dfn(u).

From the spanning tree:

- Vertex 1:
  - It is an articulation point because low(0) > dfn(1).
- Vertex 7:
  - It is an articulation point because low(8) > dfn(7).
- Vertex 3:
  - It is the root and has multiple children.

Finding Biconnected Components

Biconnected components are identified by partitioning the edges based on articulation points:

- If low[w] >dfn[u], the edge between u and w starts a new biconnected component.
- Using a stack, edges are grouped as components.

The key observations are

1. Two biconnected components share at most one vertex (an articulation point).
2. No edge belongs to more than one biconnected component.

Algorithm Overview

```
void dfnlow(int u, int v) {

dfn[u] = num++;  // Assign DFS number

   low[u] = dfn[u]; // Initialize low value

   for (each child w of u) {

      if (dfn[w] == 0) { // If w is unvisited

         dfnlow(w, u);  // Recursively call DFS

          low[u] = MIN(low[u], low[w]);

          if (low[w] >dfn[u]) {

             // u-w edge is part of a biconnected component

          }

      } else if (w != v) {

         // Update low value for back edge

         low[u] = MIN(low[u], dfn[w]);

      } } }
```

Steps to Identify Components

1. Start DFS traversal from any vertex.
2. Track dfn and low values.
3. Use a stack to store edges.
4. When low[w] >dfn[u], pop edges from the stack to form a biconnected component.

Example Walkthrough:

Input Graph is Fig.14.6

1. Perform DFS starting at vertex 3.
2. Build the DFS spanning tree and record dfn values.

DFS Spanning Tree representation in Fig 14.7

- Tree edges and back edges are clearly marked.
- DFS numbers (order of traversal) are assigned to vertices.

Articulation Points in Fig 14.8

- Articulation points are highlighted:
  - Vertex 3: Root with multiple children.
  - Vertex 7: low(8) >dfn(7).
  - Vertex 1: low(0) >dfn(1).

Key Takeaways

- Bi-connected components and articulation points are critical in understanding the connectivity of a graph.
- The DFS algorithm provides an efficient way to compute these properties using dfn and low values.
- Articulation points help identify vulnerabilities in a network, while biconnected components partition the graph for analysis.

This process is not only theoretical but also foundational in graph algorithms used in network analysis, circuit design, and other real-world applications.

14.9 KEY TERMS

Graph, Depth First Search Breadth First Search ,Shortest Path ,Minimum Spanning Tree (MST) ,Connected Components ,Adjacency Matrix ,Adjacency List ,Articulation Points .

14.10 SELF-ASSESSMENT QUESTIONS

1. How does Depth First Search (DFS) differ from Breadth First Search (BFS)?

2. Define Minimum Spanning Tree along with some of its applications.

3. Finding Connected Components in an undirected graph.

4. What is an adjacency matrix, and why would one use it to represent a graph?

5. What is the importance of articulation points in terms of graph connectivity?

14.11 SUGGESTIVE READINGS

1. "Introduction to Algorithms" by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein.
2. "Graph Theory with Applications" by J.A. Bondy and U.S.R. Murty.
3. "Algorithms" by Robert Sedgewick and Kevin Wayne.
4. "Data Structures and Algorithm Analysis in C" by Mark Allen Weiss.
5. "Graph Algorithms in the Language of Linear Algebra" by Jeremy Kepner and John Gilbert.

Dr U SURYA KAMESWARI

Lesson - 15
Minimum Cost Spanning Trees

OBJECTIVE

The objectives of the lesson are

1. To understand the concept of Minimum Cost Spanning Trees (MSTs) and their properties, including spanning tree characteristics, weighted graphs, and optimization goals.
2. To explore algorithms for constructing MSTs, such as Kruskal's, Prim's, and Sollin's (Borůvka's) algorithms, with emphasis on their steps, features, and applications.
3. To study shortest path algorithms, including Dijkstra's, Bellman-Ford, and Floyd-Warshall, for solving single-source, all-pairs, and transitive closure problems.
4. To analyse the applications of MSTs and shortest paths in real-world problems like routing, logistics, and network optimization.
5. To reinforce understanding through examples, self-assessment questions, and suggested readings for further exploration of graph algorithms.

STRUCTURE

15.1 INTRODUCTION

A Minimum Cost Spanning Tree (MST**)** is a special subgraph of a connected, weighted, undirected graph. It is a spanning tree that connects all the vertices in the graph with the minimum possible total edge weight, adhering to the following rules:

1. **Spanning Tree Properties**:
    - The MST must include all vertices in the graph.
    - It must have exactly n−1 edges if the graph has n vertices.
    - The MST must not contain any cycles.
2. **Weighted Graph**:
    - Each edge in the graph has a weight (or cost), which represents some attribute such as distance, time, or expense.
3. **Optimization Goal**:
    - The MST minimizes the sum of the weights of the selected edges while ensuring the graph remains connected.

Below is the example of a weighted graph with its minimum cost spanning tree.

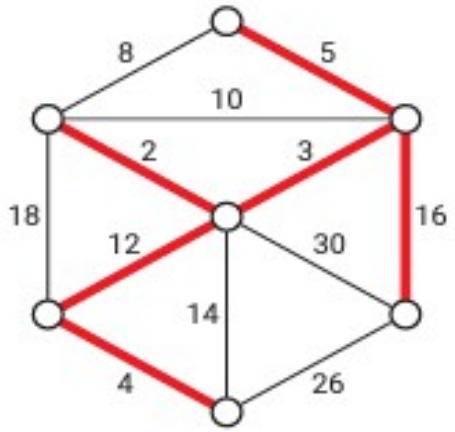


Fig 15.1.A weighted graph and its minimum cost spanning tree

The cost of a spanning tree is the sum of the weights of the edges included in it. A minimum cost spanning tree (MST) is the spanning tree that has the smallest possible total cost among all possible spanning trees for the graph.

15.2 ALGORITHMS TO FIND MST

Three well-known algorithms can be used to find the MST of a connected, weighted, undirected graph:

1. Kruskal's Algorithm
2. Prim's Algorithm
3. Sollin's Algorithm (Borůvka's Algorithm)

Each algorithm relies on a strategy called the greedy method, which constructs an optimal solution step by step.

The Greedy Method

The greedy method involves:

1. **Building a solution incrementally in stages**: At each step, a decision is made based on the current state of the solution.
2. **Making the best decision at each stage**: The decision is based on a specific criterion (e.g., selecting the least costly edge in this case).
3. **Ensuring feasibility**: Every decision must maintain the constraints of the problem and guarantee that the solution remains valid.

In the context of MSTs, the greedy method focuses on selecting the least-cost edge at each step while adhering to the following constraints:

1. **Edges within the graph**: Only the edges of the given graph can be used.
2. **Exactly n−1 edges**: The solution must use exactly n−1edges to ensure it forms a spanning tree.
3. **No cycles**: Edges that create cycles cannot be included, as spanning trees must be acyclic.

15.3 KRUSKAL'S ALGORITHM

Kruskal's Algorithm is a fundamental algorithm in graph theory, used to find the Minimum Cost Spanning Tree (MST) of a connected, weighted, undirected graph. The algorithm is based on the greedy method, which ensures that at each step, the edge with the smallest weight is considered, provided it does not form a cycle. This approach guarantees the construction of an MST with the minimum total edge weight.

15.3.1 Overview of Kruskal's Algorithm

- **Purpose**: To find the spanning tree with the least cost that connects all vertices in the graph without forming cycles.
- **Key Idea**: Sort all edges of the graph in non-decreasing order of their weights and add them to the MST one by one, as long as they do not form a cycle.
- **Greedy Approach**: The algorithm selects edges based solely on their weight, making the best local decision at each step to achieve a globally optimal solution.

15.3.2 Steps of Kruskal's Algorithm

1. **Sort Edges**: All edges in the graph are sorted in non-decreasing order based on their weights.
2. **Initialise Forest**: Each vertex starts as its own independent tree (or set). The algorithm works by gradually merging these trees.
3. **Edge Selection**:
    - Traverse the sorted list of edges.
    - Add an edge to the MST if it connects two separate trees (i.e., if it does not form a cycle).
    - Use a **union-find data structure** to efficiently check for cycles and manage the merging of trees.
4. **Terminate**: The process continues until n−1n - 1n−1 edges are included in the MST (where nnn is the number of vertices).

15.3.3 Key Features

- **Cycle Prevention**: The use of the union-find data structure ensures that edges forming cycles are excluded from the MST.
- **Optimality**: By always selecting the least-cost edge available, Kruskal's algorithm guarantees the MST is optimal.
- **Applicability**: The algorithm is well-suited for sparse graphs (graphs with relatively few edges compared to the number of vertices) because its time complexity depends on the number of edges.

Example

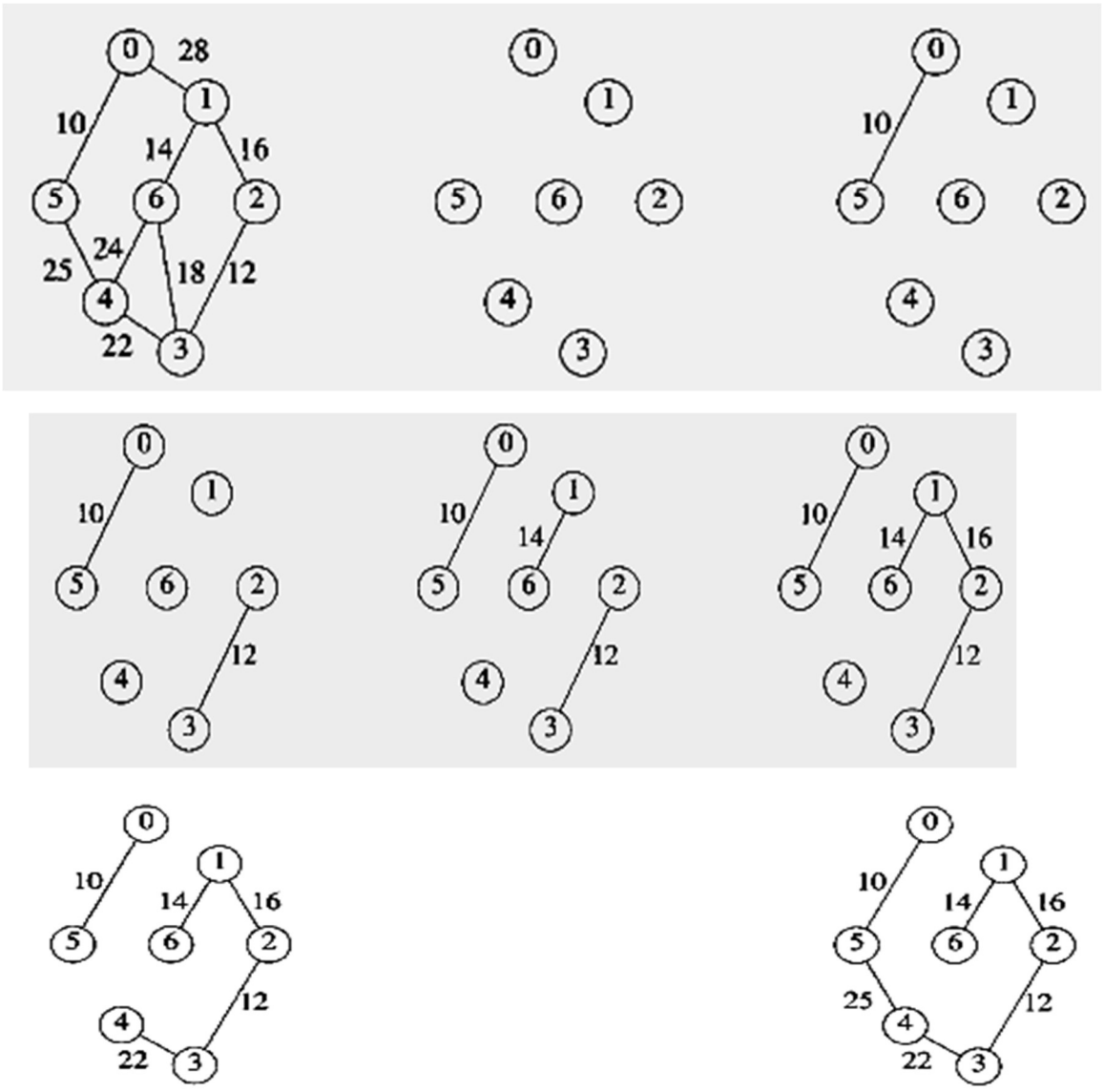


Fig 15.2 Steps of Kruskal's Algorithm

Steps of Kruskal's Algorithm for Fig 15.2 is given below

Step 1: Represent the Graph

- The input graph (as shown in the first image) is an undirected, weighted graph where edges have specific weights.
- Each vertex is represented as a node, and edges connect these nodes with a given weight.

Step 2: Sort the Edges by Weight

- List all the edges of the graph and sort them in non-decreasing order of their weights. For the given graph:
  - Sorted edges: (5,0,10), (2,3,12), (6,1,14), (6,2,16), (4,3,22), (5,4,25), (0,1,28)
- Sorting ensures that the least costly edges are considered first.

Step 3: Initialize MST

- Create an empty MST. Initially, each vertex is treated as an individual set or component.

Step 4: Process Each Edge

Iteratively add edges to the MST from the sorted list, following these rules:

1. **Cycle Prevention**: An edge is added only if it does not form a cycle in the MST.
2. **Stop Condition**: Stop adding edges once the MST contains n−1n-1n−1 edges (where nnn is the number of vertices).

Detailed Steps:

1. **Add Edge (5,0,10)**:
   - No cycle is formed, so include this edge in the MST.
   - Current MST: {(5,0)}
2. **Add Edge (2,3,12)**:
   - Adding this edge does not form a cycle.
   - Current MST: {(5,0),(2,3)}
3. **Add Edge (6,1,14)**:
   - This edge connects vertex 6 and vertex 1, forming no cycle.
   - Current MST: {(5,0),(2,3),(6,1)}
4. **Add Edge (6,2,16)**:
   - This edge connects vertex 6 and vertex 2, forming no cycle.
   - Current MST: {(5,0),(2,3),(6,1),(6,2)}
5. **Add Edge (4,3,22)**:

- Adding this edge connects vertex 4 and vertex 3, forming no cycle.
- Current MST: {(5,0),(2,3),(6,1),(6,2),(4,3)}

6. **Skip Edge (5,4,25)**:
   - Adding this edge would form a cycle between vertices 5 and 4.
   - The MST remains unchanged.
7. **Skip Edge (0,1,28)**:
   - Adding this edge would form a cycle between vertices 0 and 1.
   - The MST remains unchanged.

Step 5: Terminate

- The algorithm stops when n−1n-1n−1 edges (5 edges for 6 vertices) are added to the MST.
- Final MST: {(5,0),(2,3),(6,1),(6,2),(4,3)}
- Total cost of the MST: 10+12+14+16+22=74.

Summary

1. Sort edges by weight.
2. Add the least-cost edge to the MST, ensuring no cycles are formed.
3. Repeat until n−1n-1n−1 edges are included in the MST.

This greedy method ensures that the resulting MST has the minimum possible cost. If you need clarification on any specific graph or part of the algorithm, feel free to ask!

15.4 PRIM'S ALGORITHM

Prim's algorithm is another method for constructing the **Minimum Cost Spanning Tree (MST)** of a weighted, connected, undirected graph. Unlike Kruskal's algorithm, which works edge-by-edge, Prim's algorithm builds the MST vertex by vertex. It starts from an arbitrary vertex and grows the MST by adding the least-cost edge that connects a vertex in the MST to a vertex outside it.

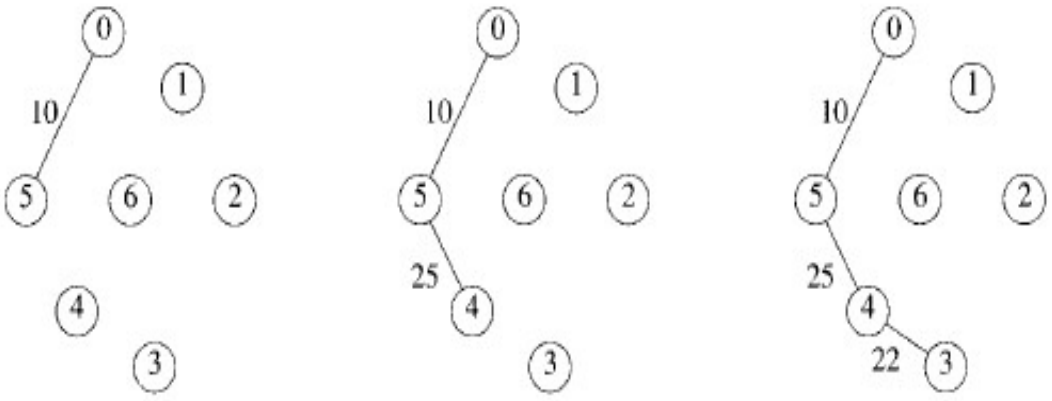
15.4.1 Key Characteristics

1. **Vertex-Based Growth**: It expands the MST by adding edges that connect a vertex in the MST to a new vertex outside the MST.
2. **Greedy Strategy**: At every step, it selects the edge with the smallest weight that is feasible.
3. **Priority Queue**: The algorithm often uses a priority queue to efficiently find the next smallest edge.

15.4.2 Steps of Prim's Algorithm

1. **Initialise**:
    - Select an arbitrary starting vertex (say vertex 0).
    - Initialise an empty MST.
    - Maintain a set of vertices already included in the MST.
    - Use a data structure (e.g., a priority queue) to track the edges with the smallest weight for connecting new vertices.
2. **Add the First Vertex**:
    - Add the starting vertex to the MST.
    - Mark it as visited.
    - Add all its adjacent edges to the priority queue.
3. **Iterative Edge Selection**:
    - Select the edge with the smallest weight from the priority queue.
    - If the edge connects a vertex already in the MST to a vertex outside the MST, add the edge and the new vertex to the MST.
    - Mark the new vertex as visited.
    - Add all edges of the new vertex that connect to unvisited vertices to the priority queue.
    - Repeat until n−1 edges have been added (where nnn is the number of vertices).
4. **Terminate**:
    - Stop when the MST contains n−1 edges. The resulting graph is the MST.

Example



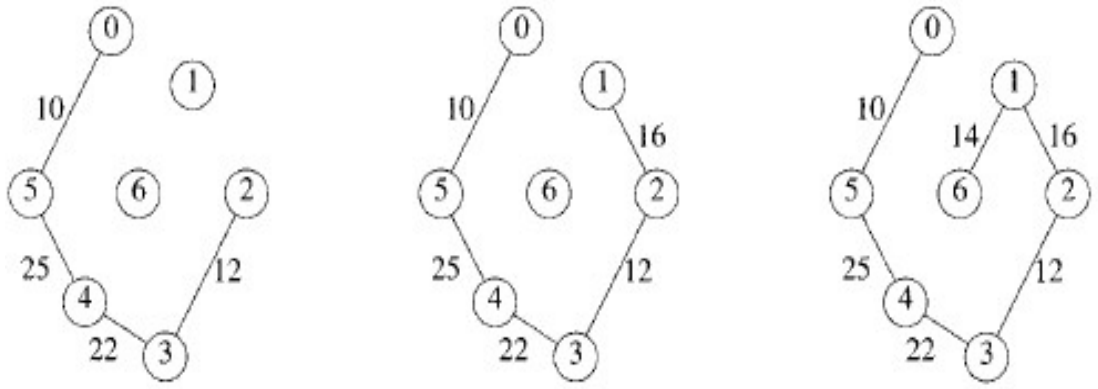


Fig 15.3 Stages in Prim's Algorithm

Prim's algorithm constructs the Minimum Cost Spanning Tree (MST) by starting from an arbitrary vertex and incrementally adding edges that connect the MST to the remaining vertices with the **least weight**. The algorithm ensures that no cycles are formed during the process.

Step-by-Step Execution of Fig 15.3 is represented below

Step 1: Start from Any Vertex

- Choose a starting vertex (in this example, vertex 0).
- Include this vertex in the MST.
- Add all edges from this vertex to a priority queue, which sorts the edges based on their weights.

MST after Step 1:

- Vertex: {0}
- Edge added: (0,5) with weight 10.

Step 2: Add the Edge with Minimum Weight

- From the priority queue, select the edge with the smallest weight that connects a vertex in the MST to a vertex outside the MST.
- Add this edge to the MST.
- Add the new vertex (in this case, vertex 5) to the MST.
- Update the priority queue with edges from the new vertex.

MST after Step 2:

- Vertices: {0, 5}
- Edges: {(0,5)}
- Add edge (5,4) with weight 25 to the queue.

Step 3: Add the Next Minimum Weight Edge

- Select the edge with the smallest weight from the updated priority queue.
- Add edge (5,4) with weight 25 to the MST, and include vertex 4.

MST after Step 3:

- Vertices: {0, 5, 4}
- Edges: {(0,5), (5,4)}
- Add edge (4,3) with weight 22 to the queue.

Step 4: Continue Adding Edges

- Add edge (4,3) with weight 22 to the MST and include vertex 3.

MST after Step 4:

- Vertices: {0, 5, 4, 3}
- Edges: {(0,5), (5,4), (4,3)}
- Add edge (3,2) with weight 12 to the queue.

Step 5: Expand MST

- Add edge (3,2) with weight 12 to the MST and include vertex 2.

MST after Step 5:

- Vertices: {0, 5, 4, 3, 2}
- Edges: {(0,5), (5,4), (4,3), (3,2)}
- Add edge (2,6) with weight 16 to the queue.

Step 6: Complete the MST

- Add edge (2,6) with weight 16 to the MST and include vertex 6.

MST after Step 6:

- Vertices: {0, 5, 4, 3, 2, 6}
- Edges: {(0,5), (5,4), (4,3), (3,2), (2,6)}
- Finally, add edge (6,1) with weight 14.

Final MST

The MST includes all vertices and has exactly n−1n-1n−1 edges (where nnn is the number of vertices):

- Edges in MST: (0,5),(5,4),(4,3),(3,2),(2,6),(6,1)
- Total Weight: 10+25+22+12+16+14=99.

Key Features of Prim's Algorithm

1. Vertex-based Growth: Starts with a single vertex and incrementally adds edges.
2. Greedy Method: Selects the smallest weight edge connecting the MST to a new vertex at every step.
3. Efficient with Priority Queues: Using a priority queue optimizes the selection of the next smallest edge.

15.5 SOLLIN'S ALGORITHM (BORŮVKA'S ALGORITHM)

Sollin's algorithm (also known as Borůvka's algorithm) is another greedy algorithm for finding the Minimum Cost Spanning Tree (MST) of a connected, weighted, undirected graph. It builds the MST in stages by merging connected components of the graph until only one tree (the MST) remains.

The algorithm begins with each vertex as its own individual tree (or component). At each step, it selects the smallest edge connecting each component to another component, adding these edges to the MST. This process reduces the number of components until only one remains.

15.5.1 Steps of Sollin's Algorithm

Step 1: Initialise Components

- Treat each vertex of the graph as an independent tree (component).
- Start with an empty MST.

Step 2: Select Minimum Weight Edges

- For each component, find the **minimum weight edge** that connects it to another component.
- Add these edges to the MST.

Step 3: Merge Components

- Merge all components connected by the edges selected in the previous step.
- Replace the merged components with a single component.

Step 4: Repeat Until One Component Remains

- Repeat Steps 2 and 3 until all vertices belong to a single component (i.e., the MST is complete).

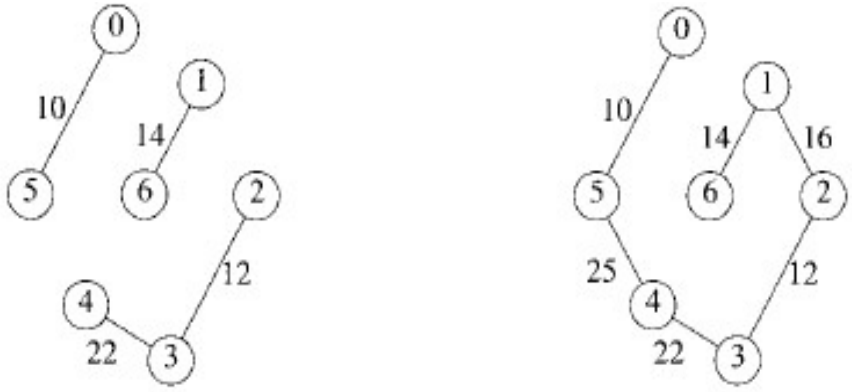


Fig 15.4 Stages in Sollin's Algorithm

15.5.1 Stages in Sollin's Algorithm

Sollin's Algorithm constructs the Minimum Cost Spanning Tree (MST) in stages, merging smaller connected components into larger ones by selecting the smallest edge from each component. Below is an explanation of the stages shown in the provided images.

Initial State

- Each vertex is treated as a separate component:
- Components: {0},{1},{2},{3},{4},{5},{6}
- Goal: Merge these components into a single connected MST

Stage 1: Select the Smallest Edge for Each Component

1. For each vertex, identify the smallest edge connecting it to another component:
    - Vertex 0: Selects edge (0,5) with weight 10.
    - Vertex 1: Selects edge (1,6) with weight 14.
    - Vertex 2: Selects edge (2,3) with weight 12.
    - Vertex 3: Selects edge (3,4) with weight 22.
    - Vertex 4: Selects edge (4,5) with weight 25.
    - Vertex 6: Selects edge (6,2) with weight 16.
2. Add the selected edges to the MST.

Stage 2: Merge Components

- The selected edges merge the following components:
    - {0,5}
    - {1,6}
    - {2,3}
    - {3,4}
- New components: {0,5},{1,6},{2,3,4}.

Stage 3: Select Smallest Edges for New Components

1. For each merged component, select the smallest edge connecting it to another component:

- Component {0,5}: Selects edge (5,4) with weight 25.
- Component {1,6}: Selects edge (6,2) with weight 16.
- Component {2,3,4}: No smaller edges left to add (already connected).

2. Add the selected edges to the MST.

Stage 4: Final Merge

- All components are now connected into a single component:
  - MST Edges: (0,5),(1,6),(2,3),(3,4),(5,4),(6,2).
- Total Weight: 10+14+12+22+25+16=99.

Key Observations

1. Each stage reduces the number of components by merging smaller ones.
2. The algorithm stops when all vertices are part of a single connected component.
3. At every stage, the **greedy choice** (minimum weight edge) ensures the MST has the smallest total weight.

This systematic merging approach is what makes Sollin's algorithm efficient, especially in parallel implementations.

15.6 SHORTEST PATHS AND TRANSITIVE CLOSURE

Shortest paths and transitive closure are important topics in graph theory, often used in routing, connectivity analysis, and optimization problems. Below, we explore these concepts in detail.

15.6.1 Shortest Paths

The **shortest path** between two vertices in a weighted graph is the path that has the minimum total weight of edges. This concept is crucial in fields like transportation, computer networks, and operations research.

Types of Shortest Path Problems

1. **Single-Source Shortest Path**:
   - Finds the shortest paths from a single source vertex to all other vertices in the graph.
   - Common algorithms:
     - **Dijkstra's Algorithm**: For graphs with non-negative edge weights.
     - **Bellman-Ford Algorithm**: Handles graphs with negative weights.
2. **All-Pairs Shortest Path**:
   - Finds the shortest paths between every pair of vertices in the graph.
   - Common algorithms:
     - **Floyd-Warshall Algorithm**: Dynamic programming-based approach.
     - **Johnson's Algorithm**: Efficient for sparse graphs.
3. **Single-Pair Shortest Path**:
   - Finds the shortest path between a specific pair of vertices.
   - Often a subset of the above problems.

15.6.2 Key Algorithms for Shortest Path

a) Dijkstra's Algorithm:

- A greedy algorithm that computes the shortest path from a source vertex to all other vertices in a graph with non-negative weights.
- Steps:
    1. Initialise the distance of all vertices as infinite except the source vertex, which is set to 0.
    2. Use a priority queue to repeatedly select the vertex with the smallest distance.
    3. Update distances of adjacent vertices if a shorter path is found.

b) Bellman-Ford Algorithm:

- Computes the shortest paths from a single source vertex even when the graph contains negative edge weights.
- Steps:
    1. Initialise distances as in Dijkstra's algorithm.
    2. Relax all edges V−1V - 1V−1 times (where VVV is the number of vertices).
    3. Check for negative weight cycles.

c) Floyd-Warshall Algorithm:

- Solves the all-pairs shortest path problem using dynamic programming.
- Updates the shortest paths iteratively by considering intermediate vertices.
- Steps:
    1. Initialise a distance matrix with direct edge weights (or infinity if no direct edge exists).
    2. Update the matrix by iterating over all pairs of vertices, considering each vertex as an intermediate.

15.6.3 Transitive Closure

The **transitive closure** of a directed graph is a graph that indicates whether a path exists between each pair of vertices. If there is a path from vertex uuu to vertex vvv, the transitive closure will have a directed edge from uuu to vvv.

Key Concepts

1. **Adjacency Matrix Representation**:
    - The transitive closure can be represented as a matrix T where:
        - T[i][j]=1 if there exists a path from vertex i to vertex j, otherwise T[i][j]=0.
2. **Warshall's Algorithm**:
    - A modified version of the Floyd-Warshall algorithm to compute transitive closure.
    - Steps:

1. Initialise a matrix with A[i][j]=1if there is a direct edge from i to j, otherwise A[i][j]=0.
2. For each vertex kkk, update the matrix such that T[i][j]=T[i][j]∨(T[i][k]∧T[k][j])

Applications:

Database query optimizations.

Determining reachability in graphs.

Network flow and connectivity analysis.

15.6.4 Applications of Shortest Paths and Transitive Closure

1. Routing:
   - Finding optimal routes in transportation and communication networks.
2. Social Network Analysis:
   - Identifying connectivity between individuals or groups.
3. Supply Chain Optimization:
   - Minimising transportation costs.
4. Web Crawlers:
   - Analyzing reachability between pages.

15.7 SINGLE SOURCE ALL DESTINATIONS

The **Single-Source All-Destinations Shortest Path** problem involves finding the shortest path from a single source vertex to **all other vertices** in a weighted graph. The objective is to compute the minimum cost (or distance) required to travel from the source vertex to every other vertex in the graph.

This problem is common in transportation networks, routing protocols, and logistics planning.

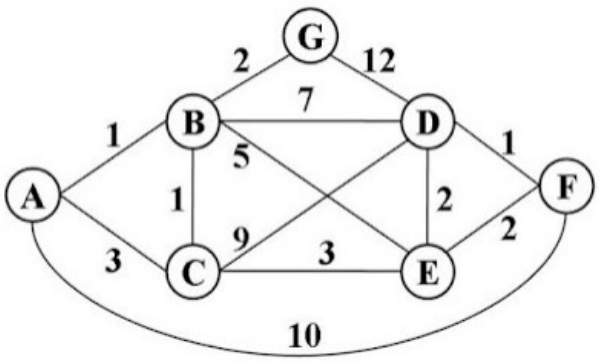


Fig 15.5 Single-Source All-Destinations Shortest Path

The **Single-Source All-Destinations** problem involves determining the shortest path from a single source vertex to all other vertices in a weighted, directed graph. Using the above diagram the process of how Dijkstra's algorithm is applied is explained

- **Graph Details**:
  - The graph is a **directed weighted graph** with vertices A,B,C,D,E,F,GA, B, C, D, E, F, GA,B,C,D,E,F,G.
  - Each edge has a non-negative weight representing the cost to travel between vertices.
- **Goal**:
  - Find the shortest paths from the source vertex AAA to every other vertex in the graph.
- **Method**:
  - Use **Dijkstra's algorithm**, which relies on a greedy approach to incrementally find the shortest path to each vertex.

Explanation Using the Diagram

1. **Graph Representation**:
   - Vertices: {A,B,C,D,E,F,G}.
   - Weights on edges represent costs. For example:
     - (A,B)=1, (A,C)=3, (B,G)=2 (C,D)=9.
2. **Algorithm Setup**:
   - Initialise a set S to keep track of vertices whose shortest paths from A have been found.
   - Maintain a **distance array** where distance[v] represents the current shortest distance from A to vertex v. Initially:
     - distance[A]=0 (source vertex),
     - distance[v]=∞ for all other vertices v≠A
   - Use a **cost adjacency matrix** or adjacency list to represent the graph.

15.7.1 Steps of Dijkstra's Algorithm

Step 1: Initialise the Source

- Start with vertex A as the source.
- Add A to the set S, as its shortest path is already known (distance[A]=0).

Step 2: Update Distances for Neighbors of A

- Explore edges from A:
  - (A,B):distance[B]=1
  - (A,C):distance[C]=3
  - (A,E):distance[E]=10
- For other vertices, the distances remain ∞

Step 3: Select the Vertex with Minimum Distance

- Among the vertices not in S, select the one with the smallest distance:
  - distance[B]=1, the smallest distance.

- Add B to S.

Step 4: Update Distances for Neighbors of B

- Explore edges from B:
  - (B,G):distance[G]=distance[B]+weight(B,G)=1+2=3.
  - (B,D):distance[D]=distance[B]+weight(B,D)=1+7=8
  - 8(B,D):distance[D]=distance[B]+weight(B,D)=1+7=8.

Step 5: Repeat the Process

- Continue selecting the vertex with the smallest distance and updating distances for its neighbors:
  - **Select C**: Update distance[D]=6via C→D(3+3).
  - **Select G**: No updates, as all its neighbors are already in S
  - **Select D**: Update distance[E]=8 via D→E(6+2).
  - **Select E**: Update distance[F]=9 via E→F (8+1).

Final Distance Array

- distance[A]=0 (source vertex).
- distance[B]=1 (shortest path: A→B).
- distance[C]=3 (shortest path: A→C).
- distance[G]=3(shortest path: A→B→G).
- distance[D]=6 (shortest path: A→C→D).
- distance[E]=8 (shortest path: A→C→D→E).
- distance[F]=9 (shortest path: A→C→D→E→F).

Key Insights from the Algorithm

1. **Greedy Approach**:
   - The shortest path to any vertex always passes through vertices in SSS, whose shortest paths are already known.
2. **Relaxation**:
   - After adding a vertex to SSS, the algorithm updates (or "relaxes") distances for its neighbors, ensuring the shortest path is maintained.
3. **Efficiency**:
   - With a priority queue, the algorithm runs in O((V+E)logV)

Applications

- **Routing and Navigation**:
  - Finding shortest paths in road networks.
- **Network Optimization**:
  - Optimizing data packet routing in computer networks.
- **Logistics**:
  - Determining the shortest transportation routes in supply chains.

By applying Dijkstra's algorithm, the shortest paths from the source vertex to all other vertices in the graph are efficiently computed.

15.8 ALL PAIRS SHORTEST PATH

The **All-Pairs Shortest Path (APSP)** problem is about finding the shortest path between every pair of vertices in a directed graph G=(V,E). The goal is to compute a matrix A such that A[i][j] represents the length (or cost) of the shortest path from vertex i to vertex j. Below, we explain the algorithm, its formulation, and key considerations in solving the problem.

- The graph is represented using a **cost adjacency matrix**, cost(i,j), defined as:
  - cost(i,i)=0, for all i.
  - cost(i,j)=∞, if there is no edge from i to j.
  - cost(i,j) = length of the edge (i,j) in E(G).

OBJECTIVE

- Determine a matrix A such that A[i] [j] represents the shortest path distance between vertices i and j.

KEY ASSUMPTIONS

1. The graph G does not contain any negative weight cycles.
   - A negative cycle allows paths to be arbitrarily small, making the shortest path undefined.
2. cost(i,j)≥0 is not strictly required. However, the algorithm requires no negative cycles to ensure correctness.

15.9 DYNAMIC PROGRAMMING FORMULATION

1. Principle of Optimality

- If the shortest path from vertex i to j passes through an intermediate vertex k, then:
  - The sub-path from i to k is the shortest path from i to k.
  - The sub-path from k to j is the shortest path from k to j.
- This principle allows us to break the shortest path problem into subproblems, making it suitable for dynamic programming.

2. Recurrence Relation

The Recurrence relation is listed as below

- Let $A^k(i, j)$ be the length of the shortest path from i to j, such that the path passes through no vertex with an index higher than k.
- The recurrence relation is:

$$A^k(i, j) = \min\left(A^{k-1}(i, j), A^{k-1}(i, k) + A^{k-1}(k, j)\right)$$

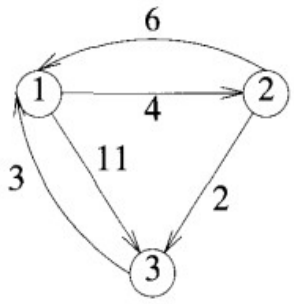
- If the path does not pass through vertex k, $A^k(i, j) = A^{k-1}(i, j)$.
- If the path passes through vertex k, $A^k(i, j) = A^{k-1}(i, k) + A^{k-1}(k, j)$.

3. Initialization

- For $k = 0$, the initial matrix $A^0(i, j)$ is the cost adjacency matrix:

$$A^0(i, j) = \begin{cases} \text{cost}(i, j), & \text{if } i \neq j, \\ 0, & \text{if } i = j. \end{cases}$$

Example



(a) Example digraph

A^0	1	2	3
1	0	4	11
2	6	0	2
3	3	∞	0

(b) A^0

A^1	1	2	3
1	0	4	11
2	6	0	2
3	3	7	0

(c) A^1

A^2	1	2	3
1	0	4	6
2	6	0	2
3	3	7	0

(d) A^2

A^3	1	2	3
1	0	4	6
2	5	0	2
3	3	7	0

(e) A^3

Fig 15.6 Floyd-Warshall Algorithm

The example showcases the Floyd-Warshall algorithm, a dynamic programming method for solving the All-Pairs Shortest Path (APSP) problem in a directed graph. Below, I will explain the steps using the provided graph and matrices.The step by step explanation of the above example is represented in detail below.

Table 15.1 Intialisation and First iteration

Step	From/To	Formula	Calculation	Result
Initialization	$A^0(i,j)$	$A^0(i,j) = \begin{cases} \text{cost}(i,j), & \text{if } i \neq j \\ 0, & \text{if } i = j \end{cases}$	Based on the graph's adjacency matrix	Refer to A^0
Iteration 1 ($k = 1$)	$A^1(3,2)$	$A^1(3,2) = \min\left(A^0(3,2), A^0(3,1) + A^0(1,2)\right)$	$\min(\infty, 3 + 4) = 7$	Updated $A^1(3,2) = 7$
	A^1	Repeat for all vertex pairs using $k = 1$	-	Refer to A^1

Table 15.2 Second and Third iterations

Step	From/To	Formula	Calculation	Result
Iteration 2 ($k = 2$)	$A^2(1,3)$	$A^2(1,3) = \min\left(A^1(1,3), A^1(1,2) + A^1(2,3)\right)$	$\min(11, 4 + 2) = 6$	Updated $A^2(1,3) = 6$
	A^2	Repeat for all vertex pairs using $k = 2$	-	Refer to A^2
Iteration 3 ($k = 3$)	$A^3(2,1)$	$A^3(2,1) = \min\left(A^2(2,1), A^2(2,3) + A^2(3,1)\right)$	$\min(6, 2 + 3) = 5$	Updated $A^3(2,1) = 5$
	A^3	Repeat for all vertex pairs using $k = 3$	-	Refer to A^3
Final Result	A^3	The final matrix A^3 contains the shortest paths between all vertex pairs.	-	Refer to A^3

15.9 KEY TERMS

Minimum Cost Spanning Tree, Weighted Graph, Prime's Algorithm, Sollin's Algorithm, Shortest Path, Transitive Closure, Dijkstra's Algorithm, Bellman-Ford Algorithm, Floyd-Warshall Algorithm

15.10 SELF-ASSESSMENT QUESTIONS

1. What are the main characteristics of the Minimum Cost Spanning Tree?
2. Compare Kruskal's, Prim's and Sollin's algorithm for finding MST.
3. What is a transitive closure of a graph used for and how is it calculated?
4. How does Dijkstra's algorithm work; its limitations.
5. When will you use Bellman-Ford instead of Dijkstra?

15.11 Suggested Readings

1. Some selected chapters from the Alma Mater book of Cormen, Leiserson, Rivest and Stein by the title “Introduction to Algorithms” dealing with Graph Algorithms.
2. Algorithm Design written by Jon Kleinberg and Éva Tardos including sections on greedy algorithms and the general theory of graph problems.
3. Mark Allen Weiss’s Data Structures and Algorithm Analysis in C++ – MST and Shortest Path Sections.
5. Research articles addressing application of Shortest Route algorithms & MST in the Operations Field & Computer Networks.
6. Dieter Jungnickel's "Graphs, Networks and Algorithms": extensive discussions of the algorithms of graphs.

Dr. U. SURYA KAMESWARI