DATA STRUCTURES WITH C++

MASTER OF COMPUTER APPLICATIONS (MCA)

SEMESTER-I, PAPER-I

LESSON WRITERS

Dr. Neelima Guntupalli	**Dr. Vasantha Rudramalla**
Assistant Professor Department of CS&E University College of Sciences Acharya Nagarjuna University	Faculty Department of CS&E University College of Sciences Acharya Nagarjuna University
Mrs. Appikatla Puspha Latha	**Dr. U. Surya Kameswari**
Faculty Department of CS&E University College of Sciences Acharya Nagarjuna University	Assistant Professor Department of CS&E University College of Sciences Acharya Nagarjuna University

EDITOR

Dr. Neelima Guntupalli

Assistant Professor
Department of CS&E
University College of Sciences
Acharya Nagarjuna University

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

MCA: DATA STRUCTURES WITH C++

First Edition : 2025

No. of Copies :

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

Published by:

Prof. V. VENKATESWARLU
Director, I/c
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.

MASTER OF COMPUTER APPLICATIONS (MCA)

Semester-I, Paper-I

101MC24: DATA STRUCTURES WITH C++

SYLLABUS

Unit-I

Software Engineering Principles and C++ Classes: Classes: Variable Accessing Class members Operators - Functions and Classes - Reference parameters and Class Objects - Implementation of member function - Constructors - Destructors; Data Abstraction, Classes and ADT-Information Hiding.

Pointers and Array based Lists: Pointer Data types and Pointer variables: Declaring Pointer Variables Address of Operator - Dereferencing Operator Classes, Structures and Pointer Variables - Initializing Pointer Variables - Dynamic Variables - Operators on Pointer Variables.

Unit-II

Linked Lists: Linked List - Properties - Item Insertion and Deletion - Building a Linked List - Linked List as an ADT - Ordered Linked Lists - Doubly Linked Lists - Linked Lists with header and trailer nodes - Circular Linked Lists.

Recursion: Recursive Definitions - Problem solving using recursion - Recursion or iteration - Recursion and Backtracking: n- Queens Puzzle.

Search Algorithms: Search Algorithms: Sequential - Binary search - Performance of binary search insertion into ordered list; Hashing: Hash functions - Collision Resolution - Hashing: Implementation using Quadratic Probing - Collision Resolution: Chaining.

Unit-III

Stacks: Stack operations - Implementation of stacks as arrays - Linked implementation of stacks-Application of stacks.

Queues: Queues: Queue operations - Implementation of Queues as arrays; Linked implementation of Queues; Priority Queue; Application of Queues.

Sorting Algorithms: Selection Sort - Insertion Sort - Quick Sort - Merge Sort - Heap Sort.

Unit-IV

Trees: Binary Trees - Binary Tree Traversal - Binary Search Tree - Non recursive Binary Tree Traversal Algorithms - AVL Trees.

Graphs: Graph Definitions and Notations - Graph Representation - Operations on graphs - Graph as ADT - Graph Traversals - shortest path.

Algorithm - Minimal Spanning Tree.

Prescribed Book:

D.S.Malik, “Data Structures using C++”, Cengage Learning India Edition (2008).

Reference Books:

1. Mark Allen Weiss, “Data structures and Algorithem Analysis in C++”, Third Edition, Pearson Education (2008).
2. Adam Drozdek, “Data Structures and Algorithms in C++”, Cengage Learning, India Edition.

(101MC24)

M.C.A. DEGREE EXAMINATION, MODEL QUESTION PAPER
MCA-First Semester

DATA STRUCTURES with C++

Time: Three hours **Maximum: 70 marks**

SECTION-A

Answer Question No.1 Compulsory: **7 x 2 = 14 M**

1 a) Define Data Abstraction.
b) Define Pointer.
c) What are the differences between Single and Double Linked List?
d) Write the complexity of binary search algorithm.
e) What are the applications of Stack?
f) Application of AVL Tree.
g) What is the purpose of Garbage Collection?

SECTION-B

Answer ONE Question from each unit: **4 x 14 = 56 M**

UNIT-I

2 a) What is Class? How can you define classes in C++.
b) Explain how we can access class members by using Pointer Variable.

OR

c) Write a C++ Program to implement the operations on Complex numbers using classes.
d) Explain about constructors and destructors in C++.

UNIT-II

3 a) Write a procedure to insert an element in an ordered list.
b) Explain Backtracking with an example.

OR

c) Write procedures to delete an element & count number of nodes in Double Linked List.
d) Explain different collision resolution techniques.

UNIT-III

4 a) Define Stack. Implement operations on Stack using arrays.
b) Write the procedure for selection sort.

OR

c) What is Priority Queue? Write the procedure for implementing the operations on Priority Queue.
d) Write a C++ program for sorting ‘n’ elements using Merge Sort technique.

UNIT-IV

5 a) Write a procedure to find minimum & maximum element in a binary search tree.
b) Write the non-recursive algorithm for post order.

OR

c) Explain Different Graph traversal techniques.
d) With an example graph. Explain how to generate minimum cost spanning tree

CONTENTS

LESSON-1

SOFTWARE ENGINEERING PRINCIPLES AND C++ CLASSES

OBJECTIVES:

The objectives of this lesson are to:

1. Understand the fundamental concepts of software, including its importance and role in daily tasks.
2. Learn about the software life cycle and its stages, from development to maintenance.
3. Explore the phases of software development, including analysis, design, implementation, and testing.
4. Gain knowledge of classes, objects, and object-oriented programming principles like encapsulation, inheritance, and polymorphism.

STRUCTURE:

1.1 INTRODUCTION:

Software refers to computer programs designed to perform specific tasks. For instance, word processing software enables users to create documents, resumes, or even books. It has revolutionized daily tasks, replacing typewriters and manual processes with efficient digital tools. The widespread adoption of software has transformed how we communicate and work, with terms like "Internet" now commonplace. Software allows real-time stock trading, instant messaging, and many other activities. Without it, computers are mere hardware incapable of executing meaningful tasks. Software development is a complex process, guided by principles from the field of software engineering.

1.1.1 What is Software?

Software refers to computer programs designed to perform specific tasks. Examples include:

- **Word processing software**: Enables users to create documents, resumes, or books.
- **Communication tools**: Real-time stock trading, instant messaging, etc.

Software has revolutionized daily tasks, replacing manual processes with efficient digital tools. Without software, computers are merely hardware incapable of executing meaningful tasks.

1.1.2 Importance of Software Development

- Software development is guided by principles from the field of software engineering.
- It ensures efficiency, usability, and reliability in solving real-world problems.

1.2 SOFTWARE LIFE CYCLE:

The software life cycle represents the stages a program undergoes from its inception to retirement.

Stages of the Software Life Cycle

1. **Development**:
   - Creating software to address specific problems.
   - A well-developed program minimizes future maintenance costs.
2. **Use**:
   - Deploying and utilizing the software by end users.
   - Feedback highlights potential improvements.
3. **Maintenance**:
   - Resolving issues and enhancing functionality.
   - Major updates may result in new software versions.
   - Programs are retired when maintenance becomes uneconomical.

1.3 SOFTWARE DEVELOPMENT PHASES:

Software development is typically divided into four key phases:

1.3.1 Analysis

- Understanding the problem and requirements.
- Defining user interactions, data manipulations, and expected outputs.
- Breaking complex problems into manageable subproblems.

1.3.2 Design

- Creating algorithms to solve the problem.
- Approaches:
  - **Structured Design**: Divides problems into subproblems.
  - **Object-Oriented Design (OOD)**: Identifies objects and their interactions.

1.3.3 Implementation

- Translating designs into code.
- Ensures modularity and reusability by organizing functions to perform specific tasks.

The implementation phase involves translating designs into code. Functions are written to perform specific operations, ensuring modularity and reusability. Users need only understand the function's purpose, not its internal workings.

Example: Conversion Function

The following function converts a measurement in inches to centimeters:

```
double inchesToCentimeters(double inches) {
   if (inches < 0) {
      cout << "The given measurement must be nonnegative." << endl;
      return -1.0;
   }
   return 2.54 * inches;
}
```

- **Pre condition**: Input must be nonnegative.
- **Post condition**: Returns equivalent centimeters or -1.0 for invalid input.

Pre conditions and Post conditions

- **Pre condition**: A condition that must be true before a function is called.
- **Post condition**: A condition that is guaranteed after the function executes.

Including preconditions and post conditions ensures that functions are used correctly and reliably.

1.3.4 Testing and Debugging

- **Testing**: Ensures the program performs as intended.
- **Debugging**: Identifies and resolves errors to improve reliability.

Test Cases

A test case includes inputs, initial conditions, and expected outputs. For example, if a function checks whether a number is within a range, test cases might include boundary values and general cases.

Types of Testing

- **Black-Box Testing**
  - Focuses on inputs and outputs without knowledge of internal implementation.
  - Example: Testing boundary values like -1, 0, 100 for a range-check function.
- **White-Box Testing**
  - Examines internal structures to ensure all parts of the code are executed.
  - Ensures correctness of control structures like loops and conditionals.

1.4 STRUCTURED AND OBJECT-ORIENTED DESIGN:

Structured Design

- Also known as **top-down design** or **modular programming**.
- Divides a problem into smaller, manageable subproblems.

Object-Oriented Design (OOD)

Object-oriented design identifies components (objects) in the problem domain and defines their interactions. For example, in a video rental system, objects might include "Video" and "Customer." Each object encapsulates data and operations related to it, such as managing stock levels or customer transactions**.** The principles of OOD are

1. **Encapsulation**: Combining data and operations within a single unit (class).
2. **Inheritance**: Creating new data types from existing ones.
3. **Polymorphism**: Using the same expression to denote different operations.

1.5 INTRODUCTION TO ALGORITHM ANALYSIS:

Algorithm analysis is a method used to evaluate the efficiency of an algorithm in terms of the resources it consumes, such as time and space. By understanding how an algorithm's performance scales with input size, developers can choose optimal solutions for various problems.

1.5.1 Importance of Algorithm Analysis

Analyzing algorithms helps in:

- **Comparing Solutions**: Identify the most efficient algorithm for a task.
- **Optimizing Resource Usage**: Choose algorithms that consume less time and space.
- **Scalability**: Ensure the algorithm performs well as input size grows.

Example Scenarios: Delivery Routes

Scenario 1: Sequential Delivery

A driver delivers 50 packages sequentially, driving one mile between houses.

- **Total Distance**:

  Distance = 1 + 1 + 1 + ... + 1 = 50 miles

  Round trip = 50 + 50 = 100 miles

- **Time Complexity**: O(n), where n is the number of packages.

Scenario 2: One Package Per Trip

Another driver delivers one package per trip, returning to the shop after each delivery.

- **Total Distance**:

  Total Distance = 2 × (1 + 2 + 3 + ... + 50) = 2550 miles

- **Time Complexity**: O(n^2).

Generalization:

- **Sequential Delivery**: O(n)
- **One Package Per Trip**: O(n^2)

1.5.2 Measuring Algorithm Efficiency

Number of Operations

Algorithm efficiency is determined by the number of operations, irrespective of hardware speed. For example in searching a list, the efficiency depends on the number of comparisons made.

Example 1: Maximum of Two Numbers

```
void main()
{
  int num1, num2, max;
  cout << "Enter two numbers: ";
  cin >> num1 >> num2;
  if (num1 >= num2)
    max = num1;
  else
    max = num2;
  cout << "Maximum is: " << max << endl;
  getch();
}
```

Operations:

- Constant number of steps, regardless of input size.
- **Time Complexity**: O(1) (constant time).

Example 2: Sum and Average of Numbers

```
void main() {
  int num, sum = 0, count = 0;
  double average;
```

```
    cout << "Enter numbers (end with -1): ";
    cin >> num;
    while (num != -1) {
        sum += num;  // Accumulate sum
        count++;     // Increment count
        cin >> num;  // Next input
    }
    if (count != 0)
        average = sum / (double)count;
    else
        average = 0;
    cout << "Sum: " << sum << ", Average: " << average << endl;
    getch();
}
```

Operations:

- Pre-loop: 3 operations.
- Loop body: 3 operations per iteration.
- Post-loop: 2-3 operations.

Time Complexity: O(n), where n is the number of inputs.

1.5.3 Dominant Terms and Growth Rates

Dominant Terms

The **dominant term** in an algorithm determines its growth rate as input size increases. **Example**: For $n^2 + n$, n^2 dominates for large n.

Common Growth Rates

1. O(1) : Constant time (e.g., finding the maximum of two numbers).
2. O(n) : Linear time (e.g., summing numbers in an array).
3. O(n^2) : Quadratic time (e.g., nested loops).

1.5.4 Common Big-O Notations

O(1) : Constant Time

- **Description**: The algorithm's runtime is fixed, independent of input size.
- **Example**:

```
int getFirstElement(int arr[ ]) {
    return arr[0];
}
```

O(log n): Logarithmic Time

- **Description**: Runtime increases slowly with input size.
- **Example**: Binary search in a sorted array.

O(n) : Linear Time

- **Description**: Runtime grows directly with input size.
- **Example**:

```
int sumArray(int arr[], int n) {
    int sum = 0;
    for (int i = 0; i < n; i++) {
        sum += arr[i];
    }
    return sum;
}
```

O(nlog n): Quasilinear Time

- **Description**: Faster than quadratic but slower than linear.
- **Example**: Sorting algorithms like Merge Sort.

O(n^2): Quadratic Time

- **Description**: Runtime grows proportionally to n2n^2n2.
- **Example**:

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        cout << i * j;
    }  }
```

O(2^n): Exponential Time

- **Description**: Runtime doubles with each additional input.
- **Example**: Solving the Traveling Salesperson problem using brute force.

1.5.5 Practical Considerations in Algorithm Analysis

Choosing Algorithms

When selecting an algorithm, consider:

1. **Time Complexity**: How fast does the algorithm run?
2. **Space Complexity**: How much memory does the algorithm require?
3. **Scalability**: Does it handle large inputs efficiently?

Examples:

- **Searching**: Analyze the number of comparisons required.
- **Matrix Multiplication**: Focus on the number of multiplications, as they are resource-intensive.

1.6 CLASSES IN C++

A **class** in C++ is a user-defined data type that serves as a blueprint for creating objects. It groups related data members (variables) and member functions (methods) under one entity to model real-world entities. Classes are a fundamental part of **Object-Oriented Programming (OOP)** and help encapsulate data and functionality.

1.6.1 What is a Class?

A **class** in C++ is a user-defined data type that serves as a blueprint for creating objects. It groups related:

- **Data Members** (variables).
- **Member Functions** (methods).

1.6.2 Syntax of a Class

```
class ClassName {
private:
   DataType variable1;
   DataType variable2;


public:
   ClassName(); // Constructor
   ~ClassName(); // Destructor
   void memberFunction(); // Member function
};
```

1.6.3 Components of a Class

1. Access Specifiers

- Define the level of accessibility for members of the class. They determine where the members can be accessed.
- **Common Specifiers**:
    - **private**: Members are only accessible within the class.
    - **public**: Members are accessible outside the class.
    - **protected**: Members are accessible within the class and its derived (child) classes.

2. Data Members

- Variables that store the attributes or state of the class.
- Example:

  ```
  int age; // A data member
  double salary;
  ```

3. Member Functions

- Functions that define the behaviors or actions of the class.
- They can manipulate or access the class's data members.
- Example:

```
void setAge(int a) {
   age = a;
}
int getAge() {
   return age;
}
```

4. Constructor

- A special member function used to initialize objects.
- Characteristics:
  - Has the same name as the class.
  - Does not return any value.
- Example:

```
ClassName() {
   age = 0; // Initialization
}
```

5. Destructor

- A special member function used to clean up resources when an object is destroyed.
- Characteristics:
  - Has the same name as the class prefixed by ~.
  - No return type or parameters.
- Example:

```
~ClassName() {
   // Cleanup code
}
```

6. Encapsulation

- The concept of bundling data and methods together while restricting direct access to some members using private or protected.

Example Code

```
class Person {
private:
   string name;
```

```
    int age;

public:
    // Constructor
    Person(string n, int a) {
        name = n;
        age = a;
    }

    // Member function to set age
    void setAge(int a) {
        age = a;
    }

    // Member function to get name
    string getName() {
        return name;
    }

    // Destructor
    ~Person() {
        cout << "Destructor called for " << name << endl;
    }
};

int main() {
    // Create an object of the class
    Person person1("John", 30);

    // Use member functions
    cout << "Name: " << person1.getName() << endl;

    person1.setAge(35); // Update age

    return 0;
}
```

Explanation of the Example

1. **Access Specifiers**:
   - name and age are private, so they cannot be accessed directly from outside the class.
   - The member functions setAge and getName are public, allowing controlled access to private data.
2. **Constructor**:
   - Initializes the name and age variables when the object is created.
3. **Destructor**:
   - Displays a message when the object is destroyed.
4. **Encapsulation**:
   - Private members (name and age) are accessed and modified via public member functions, protecting the data from direct access.

This modular structure makes classes efficient and powerful in handling real-world problems in programming.

1.6.4 Public and Private Access Specifiers in C++

Access specifiers in C++ determine the **visibility** and **accessibility** of class members (data members and member functions). They are crucial for **encapsulation** in object-oriented programming.

1. Public Access Specifier

Definition

- Members declared as public are accessible:
  - From **outside the class** using an object.
  - From **anywhere in the program** where the object is visible.

Characteristics

- Ideal for defining the **interface** of the class (e.g., methods users interact with).
- Provides direct access to data or functions without restrictions.

Syntax

```
class ClassName
{
public:
  DataType memberVariable;
  void memberFunction();
};
```

Example

```
class PublicExample {
public:
    int publicVariable; // Public data member

    // Public member function
    void display() {
        cout << "Public Variable: " << publicVariable << endl;
    }
};

int main() {
    PublicExample obj; // Create an object of the class

    obj.publicVariable = 42; // Directly access public member
    obj.display();           // Call public function

    return 0;
}
```

Output

```
Public Variable: 42
```

Key Points

- publicVariable is directly accessible and modifiable.
- The function display() can be called from outside the class.

2. Private Access Specifier

Definition

- Members declared as private are:
  - Accessible **only within the class** where they are defined.
  - Not accessible directly from outside the class.

Characteristics

- Used to **hide implementation details** and protect data.
- Access is provided through **public member functions** (getters and setters).

Syntax

```
class ClassName {
private:
    DataType memberVariable;
    void memberFunction();
};
```

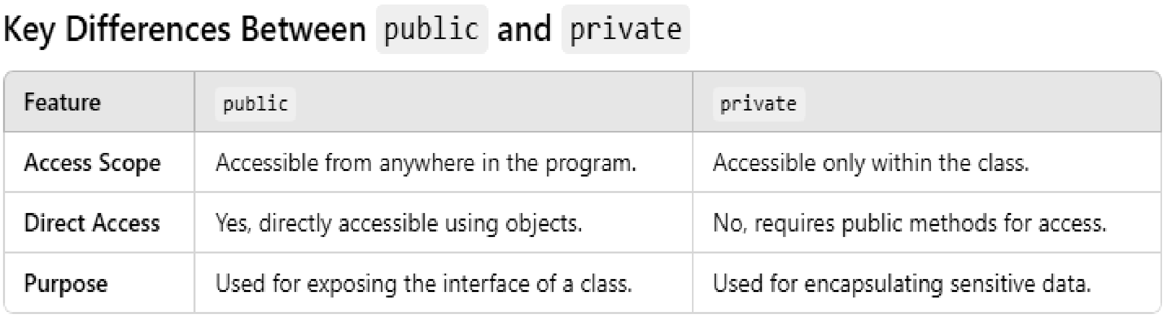
Example

```
class PrivateExample {
private:
    int privateVariable; // Private data member

public:
    // Public member function to set privateVariable
    void setPrivateVariable(int value) {
        privateVariable = value;
    }

    // Public member function to get privateVariable
    int getPrivateVariable() {
        return privateVariable;
    }
};

int main() {
    PrivateExample obj; // Create an object of the class

    // Cannot access privateVariable directly
    // obj.privateVariable = 42; // Error: 'privateVariable' is private

    // Access privateVariable using public member functions
    obj.setPrivateVariable(42);
    cout << "Private Variable: " << obj.getPrivateVariable() << endl;

    return 0;
}
```

Output

```
Private Variable: 42
```

Key Differences Between public and private



Key Differences Between `public` and `private`

Feature	`public`	`private`
Access Scope	Accessible from anywhere in the program.	Accessible only within the class.
Direct Access	Yes, directly accessible using objects.	No, requires public methods for access.
Purpose	Used for exposing the interface of a class.	Used for encapsulating sensitive data.

- **public**:
  - Openly accessible; used to define the class interface.
  - Ideal for methods or attributes meant to interact with the outside world.
- **private**:
  - Restricted access; ensures encapsulation by hiding sensitive data.
  - Provides controlled access through public methods.

By combining public and private access specifiers, robust, secure, and modular programs can be created in C++.

1.7 OBJECTS IN CLASSES:

An **object** in C++ is an instance of a class. When a class is defined, it acts as a blueprint, and the object is the actual entity created based on that blueprint. Objects store data (attributes) and have behaviors (member functions) as defined by the class..

1.7.1 Defining and Creating an Object

1. Define the class.
2. Declare an object:

   ClassName objectName;

1.7.2 Accessing Members Using Objects

- Use the **dot operator (.)**:

  objectName.memberFunction (parameters);

  objectName.dataMember = value;

Example: Object Definition and Accessing Member Functions

```
// Define a class
class Rectangle {
public:
   // Data members
   int length;
   int width;
```

```
    // Member function to calculate area
    int calculateArea() {
        return length * width;
    }

    // Member function to display dimensions
    void displayDimensions() {
        cout << "Length: " << length << ", Width: " << width << endl;
    }
};

int main() {
    // Create an object of the class
    Rectangle rect;

    // Access public data members using the object
    rect.length = 10;
    rect.width = 5;

    // Access member functions using the object
    rect.displayDimensions(); // Output: Length: 10, Width: 5
    cout << "Area: " << rect.calculateArea() << endl; // Output: Area: 50

    return 0;
}
```

Explanation:

1. **Object Creation**:
    - Rectangle rect; creates an object rect of the Rectangle class.
2. **Accessing Data Members**:
    - rect.length and rect.width are accessed directly to assign values because they are public.
3. **Accessing Member Functions**:

- rect.displayDimensions() calls the member function to display the dimensions of the rectangle.
- rect.calculateArea() calls the member function to compute the area of the rectangle.

Key Points:

1. Objects provide access to the public members of a class.
2. The dot operator (.) is used to access members.
3. The behavior of an object depends on the member functions and data defined in its class.

1.8 KEY TERMS:

Software, Software Life Cycle, Object-Oriented Design (OOD), Encapsulation, Algorithm Analysis, Big-O Notation.

1.9 REVIEW QUESTIONS:

1) What is software, and why is it important for modern computing?
2) Define the software life cycle and explain its key stages?
3) Compare and contrast structured design and object-oriented design?
4) What are pre conditions and post conditions in the context of function implementation? Provide an example?
5) Explain the concept of Big-O notation and its significance in algorithm analysis?
6) What is encapsulation, and how does it contribute to object-oriented programming?

1.10 SUGGESTED READINGS:

1) “Object-Oriented Software Engineering: A Use Case Driven Approach” by Ivar Jacobson.
2) **“**Introduction to Algorithms” by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein.
3) **“**The C++ Programming Language” by Bjarne Stroustrup.
4) **“**Clean Code: A Handbook of Agile Software Craftsmanship” by Robert C. Martin.

Dr. Neelima Guntupalli

LESSON-2
CONSTRUCTORS AND DESTRUCTORS

OBJECTIVES:

1. Understand the concept and purpose of constructors in C++, including their properties and types.
2. Learn the differences and use cases of default, parameterized, copy, and overloaded constructors.
3. Explore the role and importance of destructors in resource management and cleanup.
4. Gain familiarity with UML class diagrams and their application in representing class structure and relationships.
5. Develop a practical understanding of constructor and destructor invocation through hands-on examples.

STRUCTURE:

2.1 INTRODUCTION:

In C++, variables are not automatically initialized when declared. Constructors provide a way to guarantee that the data members of a class are initialized when an object is created. A **constructor** is a special member function with unique properties designed for object initialization.

2.1.1 Key Properties of Constructors

1. **Same Name as the Class**: The name of the constructor must match the name of the class.
2. **No Return Type**: Constructors do not have a return type (not even void).
3. **Automatic Execution**: Constructors are executed automatically when an object is created.
4. **Overloading**: A class can have multiple constructors with different parameter lists (constructor overloading).
5. **Cannot Be Called Explicitly**: Constructors cannot be called like normal functions.
6. **Default Constructor**: A constructor without parameters is called a default constructor.
7. **Parameterized Constructor**: A constructor that accepts parameters to initialize an object with specific values.

2.2 TYPES OF CONSTRUCTORS:

There are different types of constructors depending on their functionality and how they initialize objects. These types are:

2.2.1 Default Constructor (Constructor without Parameters)

Definition:

A **default constructor** is a constructor that takes no arguments or has all default arguments. It initializes an object with default values.

Characteristics:

1. Automatically invoked when an object is created without arguments.
2. Can be explicitly defined or provided by the compiler if no other constructors exist.
3. Ensures all data members are initialized, avoiding uninitialized variables.

Syntax:

```
class ClassName {
public:
    ClassName() {
        // Initialization code
    }
};
```

Example

```
class Clock {
public:
    Clock() {  // Default constructor
        hr = 0;
        min = 0;
        sec = 0;
    }

    void displayTime() {
        cout << hr << ":" << min << ":" << sec << endl;
    }

private:
    int hr, min, sec;
};

int main() {
    Clock clock1;          // Default constructor is called
    clock1.displayTime();   // Output: 0:0:0
    return 0;
}
```

2.2.2 Parameterized Constructor

A **parameterized constructor** takes arguments to initialize an object with specific values provided during its creation.

Characteristics:

1. Allows custom initialization of data members.
2. Requires arguments to be passed when the object is created.
3. Supports overloading when combined with other constructors.

Syntax:

```
class ClassName {
public:
    ClassName(int param1, int param2) {
        // Initialization using parameters
    }
};
```

Example

```
class Clock {
public:
    Clock(int h, int m, int s) {  // Parameterized constructor
        hr = (h >= 0 && h < 24) ? h : 0;
        min = (m >= 0 && m < 4) ? m : 0;
        sec = (s >= 0 && s < 4) ? s : 0;
    }

    void displayTime() {
        cout << hr << ":" << min << ":" << sec << endl;
    }

private:
    int hr, min, sec;
};

int main() {
    Clock clock1(12, 30, 45);  // Parameterized constructor is called
    clock1.displayTime();      // Output: 12:30:45
    return 0;
}
```

2.2.3 Copy Constructor

Definition:

A **copy constructor** creates a new object as a copy of an existing object. It is primarily used for copying the values of one object to another.

Characteristics:

1. Takes a reference to an object of the same class as its parameter.
2. If no copy constructor is defined, the compiler provides a default one that performs a shallow copy.
3. Deep copying may be implemented for dynamic memory allocation.

Syntax:

```
class ClassName {
public:
   ClassName(const ClassName &obj) {
      // Copy initialization
   }
};
```

Example:

```
class Clock {
public:
   Clock(int h, int m, int s) {  // Parameterized constructor
      hr = h;
      min = m;
      sec = s;
   }

   Clock(const Clock &obj) {  // Copy constructor
      hr = obj.hr;
      min = obj.min;
      sec = obj.sec;
   }
```

```
    void displayTime() {
        cout << hr << ":" << min << ":" << sec << endl;
    }

private:
    int hr, min, sec;
};

int main() {
    Clock clock1(12, 30, 45);  // Parameterized constructor is called
    Clock clock2 = clock1;     // Copy constructor is called
    clock1.2.2.displayTime();      // Output: 12:30:45
    return 0;
}
```

2.2.4 Overloaded Constructors

Definition:

Constructor overloading allows a class to have multiple constructors with different parameter lists. This provides flexibility in object creation.

Characteristics:

1. Each constructor must have a unique signature (different number or types of parameters).
2. The correct constructor is selected based on the arguments provided during object creation.

Syntax:

```
class ClassName {
public:
    ClassName();  // Default constructor
    ClassName(int param);  // Parameterized constructor
    ClassName(int param1, int param2);  // Another parameterized constructor
};
```

Example

```
class Clock {
public:
   Clock() {  // Default constructor
      hr = 0;
      min = 0;
      sec = 0;
   }

   Clock(int h, int m, int s) {  // Parameterized constructor
      hr = (h >= 0 && h < 24) ? h : 0;
      min = (m >= 0 && m < 4) ? m : 0;
      sec = (s >= 0 && s < 4) ? s : 0;
   }

   void displayTime() {
      cout << hr << ":" << min << ":" << sec << endl;
   }

private:
   int hr, min, sec;
};

int main() {
   Clock defaultClock;       // Calls default constructor
   defaultClock.displayTime();  // Output: 0:0:0

   Clock customClock(4, 20, 30);  // Calls parameterized constructor
   customClock.displayTime();      // Output: 4:20:30

   return 0;
}
```

2.3 UNIFIED MODELING LANGUAGE (UML) DIAGRAMS:

A **UML class diagram** graphically represents a class, its data members, and member functions.

Structure of a UML Diagram

1. **Top Box**: The name of the class.
2. **Middle Box**: Data members and their data types.
3. **Bottom Box**: Member functions, their parameter lists, and return types.

UML for Clock Class

```
Clock
----------------------
- hr: int
- min: int
- sec: int
----------------------
+ setTime(int, int, int): void
+ displayTime(): void
+ Clock(): void
+ Clock(int, int, int): void
    •   +: Public members.
    •   -: Private members.
```

2.4 VARIABLE DECLARATION AND CONSTRUCTOR INVOCATION:

When you declare an object of a class in C++, the constructor is invoked automatically to initialize the object's data members. Depending on how the object is declared, either the **default constructor** or the **parameterized constructor** is called.

2.4.1 Default Constructor Invocation

Definition

The **default constructor** is invoked when an object is declared without passing any arguments. If no constructor is explicitly defined in the class, the compiler generates a default constructor automatically.

Syntax

```
className objectName;
```

How It Works

1. The default constructor initializes the data members of the object with default or specified values.
2. If the user defines a custom default constructor, it executes the user-defined initialization logic.

Example

```
class Clock {
public:
   Clock() {  // Default constructor
      hr = 0;
      min = 0;
      sec = 0;
      cout << "Default constructor called!" << endl;
   }

   void displayTime() {
      cout << hr << ":" << min << ":" << sec << endl;
   }

private:
   int hr, min, sec;
};

int main() {
   Clock myClock;  // Default constructor is called
   myClock.displayTime();  // Output: 0:0:0
   return 0;
}
```

Output

```
Default constructor called!
0:0:0
```

2.4.2 Parameterized Constructor Invocation

Definition

The **parameterized constructor** is invoked when an object is declared and arguments are passed to initialize it. This allows objects to be created with specific values.

Syntax

```
className objectName(arg1, arg2, ...);
```

How It Works:

1. The arguments passed during the object's declaration are used to initialize its data members.
2. The parameterized constructor ensures the object starts in a state defined by the user-provided values.

Example

```
class Clock {
public:
  Clock(int h, int m, int s) {  // Parameterized constructor
    hr = (h >= 0 && h < 24) ? h : 0;
    min = (m >= 0 && m < 60) ? m : 0;
    sec = (s >= 0 && s < 60) ? s : 0;
    cout << "Parameterized constructor called!" << endl;
  }

  void displayTime() {
    cout << hr << ":" << min << ":" << sec << endl;
  }

private:
    int hr, min, sec;
};


int main() {
    Clock myClock(4, 20, 30);  // Parameterized constructor is called
    myClock.displayTime();     // Output: 4:20:30
    return 0;
}
```

Output

```
Parameterized constructor called!
4:20:30
```

2.4.3 Incorrect Syntax: Default Constructor with Empty Parentheses

Explanation

When you include empty parentheses () after the object name during declaration, the compiler treats the statement as a **function declaration** instead of an object declaration. This causes a syntax error because the compiler assumes you're declaring a function returning an object of the class.

Incorrect Example

```
Clock myClock();  // Error: Compiler treats this as a function declaration
```

Reason:

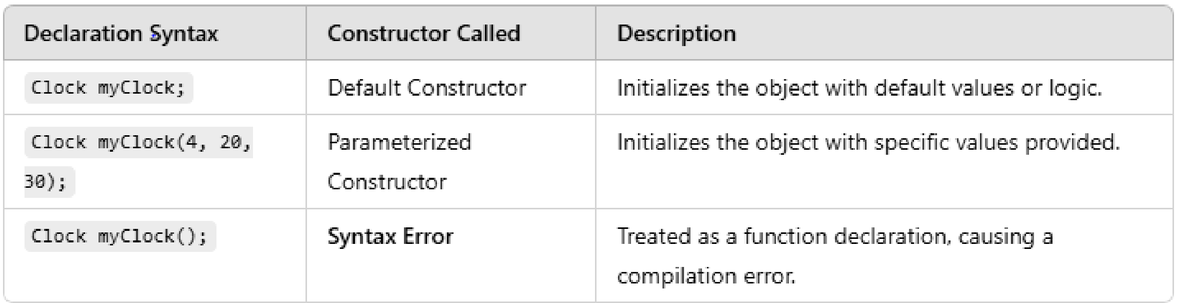
- The compiler interprets Clock myClock(); as a function named myClock that returns an object of type Clock.

Correct Usage

Simply omit the parentheses when invoking the default constructor:

```
Clock myClock;  // Default constructor is correctly invoked
```

Table 2.1. Constructor Invocation Based on Object Declaration



Declaration Syntax	Constructor Called	Description
`Clock myClock;`	Default Constructor	Initializes the object with default values or logic.
`Clock myClock(4, 20, 30);`	Parameterized Constructor	Initializes the object with specific values provided.
`Clock myClock();`	**Syntax Error**	Treated as a function declaration, causing a compilation error.

2.5 DESTRUCTORS IN C++

A **destructor** in C++ is a special member function that is automatically invoked when an object goes out of scope or is explicitly destroyed. Destructors are used to release resources allocated during the object's lifetime, ensuring proper cleanup.

2.5.1 Definition and Key Properties of Destructors

Definition

A **destructor** is a member function of a class that:

1. Has the same name as the class but is preceded by a tilde (~).
2. Does not accept any arguments.
3. Does not return a value, not even void.
4. Is called automatically when the object goes out of scope or is deleted.

Key Properties

1. **Automatic Invocation**: Destructors are automatically called when:
   - o The object goes out of scope (e.g., at the end of a function).
   - o The object is explicitly deleted using the delete keyword.
2. **No Overloading**: A class can have only one destructor. It cannot be overloaded.
3. **Purpose**: Primarily used for releasing dynamically allocated memory, closing files, or releasing other resources.
4. **Non-Returnable**: Since destructors do not have a return type, they cannot return values.

2.5.2 Syntax of a Destructor

The syntax for defining a destructor is as follows:

```
class ClassName {
```

```
public:
    ~ClassName() {
        // Cleanup code
    }
};
```

2.5.3 Example: Basic Destructor

Code Example

```
class Example {
public:
    Example() {  // Constructor
        cout << "Constructor called!" << endl;
    }

    ~Example() {  // Destructor
        cout << "Destructor called!" << endl;
    }
};

int main() {
    cout << "Creating an object..." << endl;
    Example obj;  // Constructor is called

    cout << "End of the program." << endl;
    return 0;  // Destructor is automatically called here
}
```

Output

```
Creating an object...
Constructor called!
```

```
End of the program.
Destructor called!
```

Explanation

1. When obj is created, the constructor initializes the object.
2. At the end of the main() function, obj goes out of scope, and the destructor is automatically invoked to clean up.

2.5.4 Example: Destructor for Releasing Resources

Problem

When dynamically allocating memory or opening files, it's important to free memory or close files when the object is no longer needed. Destructors handle this cleanup.

Code Example

```
class DynamicArray {
private:
  int* arr;
  int size;

public:
  DynamicArray(int s) {  // Constructor
    size = s;
    arr = new int[size];  // Dynamically allocate memory
    cout << "Array of size " << size << " created." << endl;
  }

  ~DynamicArray() {  // Destructor
    delete[] arr;  // Free allocated memory
    cout << "Array of size " << size << " destroyed." << endl;
  }
};

int main() {
  DynamicArray array(5);  // Constructor is called
  cout << "Using the array..." << endl;

  // Destructor is automatically called when array goes out of scope
  return 0;
}
```

Output

```
Array of size 5 created.
Using the array...
Array of size 5 destroyed.
```

Explanation

1. The DynamicArray class dynamically allocates memory in the constructor.
2. The destructor ensures that the allocated memory is freed when the object goes out of scope.

2.5.5 Use Cases of Destructors

1. **Releasing Memory**: Free dynamically allocated memory to prevent memory leaks.
   - Example: delete[] for arrays or delete for pointers.
2. **Closing Files**: Ensure that file handles are closed to avoid file corruption.
3. **Releasing Locks or Network Connections**: Ensure proper release of system resources.

2.5.6 Destructor Behavior with Multiple Objects

When multiple objects of the same class are created, their destructors are called in **reverse order** of creation.

Code Example

```
class Test {
public:
    Test() {
        cout << "Constructor called!" << endl;
    }
    ~Test() {
        cout << "Destructor called!" << endl;
    }
};

int main() {
    cout << "Creating two objects..." << endl;
    Test obj1;
    Test obj2;


    cout << "End of the program." << endl;
    return 0;
}
```

Output

```
Creating two objects...
Constructor called!
Constructor called!
End of the program.
Destructor called!
Destructor called!
```

Explanation

- obj1 is created first, then obj1.2.2.
- Destructors are invoked in reverse order: first obj2 is destroyed, then obj1.

Key Points:

1. **Automatic Invocation**: You don’t need to explicitly call a destructor; it is invoked automatically.
2. **One Destructor per Class**: A class can have only one destructor, and it cannot be overloaded.
3. **Reverse Order**: Destructors for objects in the same scope are called in reverse order of their creation.

4. **Resource Management**: Destructors play a critical role in preventing resource leaks and ensuring program stability.

2.6 KEY TERMS:

Constructor**,** Default Constructor, Parameterized Constructor, Copy Constructor, Overloaded Constructor, Destructor, Unified Modeling Language (UML) Diagram.

2.7 REVIEW QUESTIONS:

1. What are the key properties of constructors in C++?
2. Differentiate between default constructors and parameterized constructors with examples?
3. Explain the concept of a copy constructor and its use in C++?
4. What is the purpose of a destructor, and how does it help in resource management?
5. Describe the structure of a UML class diagram and its components?
6. Why are destructors called in reverse order of object creation? Provide an example?

2.8 SUGGESTED READINGS:

1. “The C++ Programming Language” by Bjarne Stroustrup.
2. “Object-Oriented Programming in C++” by Robert Lafore.
3. “Programming: Principles and Practice Using C++” by Bjarne Stroustrup.
4. “Design Patterns: Elements of Reusable Object-Oriented Software” by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides.

Dr. Neelima Guntupalli

LESSON-3
POINTERS AND ARRAY BASED LISTS

OBJECTIVES:

The objectives of the lesson are to

1. Understand the concept and operations of pointers in C++ and their role in memory management and dynamic programming.
2. Learn the principles of dynamic memory allocation and proper usage of new and delete operators.
3. Explore shallow and deep copy mechanisms and their implications in resource management.
4. Master the fundamentals of array-based lists, including initialization, insertion, deletion, search, and traversal operations.

STRUCTURE:

3.1 INTRODUCTION:

Pointers and array-based lists are fundamental components of C++ programming, offering powerful tools for efficient memory management and data manipulation. A pointer is a variable that stores the memory address of another variable, allowing direct access and modification of memory locations. This capability is crucial for dynamic memory allocation, efficient function parameter passing, and implementing advanced data structures. Array-based lists, on the other hand, provide a sequential collection of elements stored in contiguous memory locations, enabling efficient random access and operations like insertion, deletion, and traversal. Combined, pointers and array-based lists form the backbone of numerous algorithms and data structures in C++, allowing developers to write optimized, flexible, and robust code for a wide range of applications.

3.1.1 Pointers:

A **pointer** is a special type of variable that stores the memory address of another variable instead of a direct value. Pointers allow you to directly access and manipulate the memory locations of variables, making them a crucial feature in C++ for tasks such as dynamic memory allocation, efficient data handling, and implementing advanced data structures.

Key Features:

- **Stores Addresses**: A pointer contains the address of a variable, not the variable's value.
- **Memory Access**: Pointers allow low-level memory access, giving programmers fine-grained control over resource usage.

- **Dynamic Operations**: Used to allocate and free memory at runtime, which is essential for managing resources efficiently.

Real-World Analogy:

Think of a pointer as a "reference" to a specific house (memory location) on a street. Instead of directly describing the house (data), you are providing its address (pointer).

3.1.2 Declaration of Pointer Variables

The syntax to declare a pointer in C++ is:

```
DataType* pointerName;
```

Here:

- DataType specifies the type of variable the pointer will point to (e.g., int, float, double).
- pointerName is the name of the pointer variable.
- The asterisk (*) indicates that the variable is a pointer.

Example:

```
int* ptr;       // A pointer to an integer
double* dptr;   // A pointer to a double
char* cptr;     // A pointer to a character
```

Memory Representation:

When a pointer is declared, it occupies memory to store the address of another variable. For example:

```
int num = 10;
int* ptr = &num;  // ptr now stores the address of num
```

If num is stored at memory address 0x7ffee4f8, ptr will store the value 0x7ffee4f3.8.

3.1.3 Address-of Operator (&)

The **address-of operator** (&) is used to retrieve the memory address of a variable. It is essential for initializing pointers.

Syntax:

```
pointerName = &variableName;
```

Example:

```
int num = 5;        // Declare an integer variable
int* ptr = &num;    // Assign the address of num to the pointer ptr

cout << "Value of num: " << num << endl;        // Outputs 5
cout << "Address of num: " << &num << endl;    // Outputs the memory address of num
cout << "Pointer value: " << ptr << endl;      // Outputs the same address as &num
```

Output:

```
Value of num: 5
Address of num: 0x7ffee4f8
Pointer value: 0x7ffee4f8
```

Notes:

- The & operator retrieves the address, not the value, of the variable.
- This address is a unique identifier of where the variable resides in memory.

3.1.4 Dereferencing Operator (*)

The **dereferencing operator** (*) is used to access or modify the value stored at the memory location a pointer points to.

Syntax:

```
*pointerName
```

Example:

```
int num = 5;        // Declare an integer variable
int* ptr = &num;    // Pointer stores the address of num

cout << "Value of num: " << *ptr << endl;  // Outputs the value of num (5)
*ptr = 20;                                 // Modify num's value through the pointer
cout << "Updated value of num: " << num << endl;  // Outputs 20
```

Output:

```
Value of num: 5
Updated value of num: 20
```

Explanation:

- *ptr accesses the value stored at the memory address held by ptr.
- Modifying *ptr changes the actual value stored in the variable num.

Additional Example: Address and Dereferencing Combined

```
int main() {
   int num = 42;          // Declare an integer variable
   int* ptr = &num;       // Pointer points to num

   cout << "Address of num: " << &num << endl;   // Address of num
   cout << "Value of num using pointer: " << *ptr << endl;  // Value at address

   *ptr = 100;  // Modify num's value via pointer
   cout << "Updated value of num: " << num << endl;
```

```
    return 0;
}
```

Output:

```
Address of num: 0x7ffee4f8
Value of num using pointer: 42
Updated value of num: 100
```

Practical Use Cases of Pointers:

1. **Dynamic Memory Allocation**: Allocate memory during runtime using new and delete.
2. **Efficient Function Parameters**: Pass large objects or arrays by reference to avoid memory overhead.
3. **Data Structures**: Implement complex structures like linked lists, trees, and graphs.
4. **System Programming**: Low-level memory and resource manipulation.
5.

3.2 DYNAMIC MEMORY ALLOCATION:

Dynamic memory allocation is a feature in C++ that allows memory to be allocated during runtime. Unlike static memory allocation, which is determined at compile time, dynamic memory allocation is highly flexible and efficient, enabling the creation of objects and arrays as needed.

3.2.1 Using the new Operator

The new operator is used to allocate memory on the heap during runtime. It returns the address of the allocated memory, which can then be stored in a pointer variable.

Key Points:

- The memory allocated using new persists until it is explicitly deallocated using the delete operator.
- The new operator initializes the memory to a default value if used with primitive types like int or float.

Syntax:

```
pointerVariable = new DataType;
```

Example:

```
int* ptr = new int;  // Allocates memory for one integer
*ptr = 10;           // Assigns value 10 to the allocated memory
```

```
cout << "Value: " << *ptr << endl;   // Outputs 10
cout << "Address: " << ptr << endl; // Outputs the memory address
```

Explanation:

1. int* ptr = new int: Allocates memory for an integer and assigns the address to ptr.
2. *ptr = 10: Dereferences the pointer to store the value 10 in the allocated memory.

Output:

```
Value: 10
Address: 0x7ffee4f8
```

- If memory allocation fails (e.g., insufficient memory), the new operator throws a std::bad_alloc exception.
- You can use std::nothrow to avoid exceptions:

```
int* ptr = new (std::nothrow) int;
if (!ptr) {
    cout << "Memory allocation failed!" << endl;
}
```

3.2.2 Using the delete Operator

The delete operator is used to deallocate memory that was previously allocated using the new operator. This ensures efficient memory usage and prevents memory leaks.

Key Points:

- It frees the allocated memory, making it available for reuse.
- After using delete, set the pointer to NULL to avoid dangling pointers.

Syntax:

```
delete pointerVariable;
```

Example:

```
int* ptr = new int;  // Allocates memory
*ptr = 20;           // Assigns a value

delete ptr;          // Frees the memory
ptr = NULL;          // Resets the pointer to avoid dangling pointer issues
```

Output:

```
 (No visible output; memory is freed)
```

Notes:

- Deleting memory that was not dynamically allocated results in undefined behavior.
- Attempting to dereference a dangling pointer after memory deallocation causes runtime errors.

3.2.3 Dynamic Arrays

Dynamic arrays allow you to allocate memory for an array at runtime. The size of the array can be specified during allocation.

Allocating Dynamic Arrays:

```
int* arr = new int[5];  // Allocates memory for an array of 5 integers
```

Initializing the Array:

```
for (int i = 0; i < 5; i++) {
   arr[i] = i * 2;  // Assigns values
}
```

Accessing and Displaying Elements:

```
for (int i = 0; i < 5; i++) {
   cout << arr[i] << " ";  // Outputs: 0 2 4 6 8
}
```

Deallocating the Array:

```
delete[] arr;  // Frees the array memory
```

Example:

```
int main() {
   int* arr = new int[5];  // Allocate memory for 5 integers

   for (int i = 0; i < 5; i++) {
      arr[i] = i * 2;  // Initialize array
   }

   cout << "Array elements: ";
   for (int i = 0; i < 5; i++) {
      cout << arr[i] << " ";
   }

   delete[] arr;  // Free the memory
   return 0;
}
```

Output:

```
Array elements: 0 2 4 6 8
```

- Always use delete[] for arrays, not delete.
- Failure to deallocate memory may cause memory leaks.

3.3 SHALLOW vs. DEEP COPY:

When working with pointers, understanding shallow and deep copies is essential to ensure correct behavior, especially when copying objects or assigning pointers.

3.3.1 Shallow Copy

A **shallow copy** duplicates only the pointer, not the data it points to. As a result:

- Both the original and the copied pointers point to the same memory location.
- Modifications via one pointer affect the other.
- Deleting the memory via one pointer leaves the other pointer dangling.

Example:

```
int* ptr1 = new int(10);  // Allocates memory
int* ptr2 = ptr1;         // Shallow copy (both pointers point to the same location)

cout << *ptr1 << " " << *ptr2 << endl;  // Outputs: 10 10

delete ptr1;  // Frees the memory
// ptr2 is now a dangling pointer
```

Output:

```
 10 10
```

Issue:

Accessing *ptr2 after delete ptr1 results in undefined behavior.

3.3.2 Deep Copy

A **deep copy** duplicates both the pointer and the data it points to. As a result:

- Each pointer has its own copy of the data.
- Modifications via one pointer do not affect the other.

Implementation:

```
class Sample {
public:
   int* data;

   // Constructor
   Sample(int value) {
      data = new int(value);  // Allocates memory
   }

   // Deep Copy Constructor
   Sample(const Sample& obj) {
      data = new int(*obj.data);  // Copies the value
   }

   // Destructor
   ~Sample() {
      delete data;  // Frees memory
   }
};
```

Usage:

```
int main() {
   Sample obj1(42);       // Creates an object
   Sample obj2 = obj1;    // Deep copy is made

   cout << *obj3.1.data << " " << *obj3.2.data << endl;  // Outputs: 42 42

   *obj3.1.data = 100;      // Modify obj1's data
```

```
    cout << *obj3.1.data << " " << *obj3.2.data << endl;  // Outputs: 100 42


    return 0;
}
```

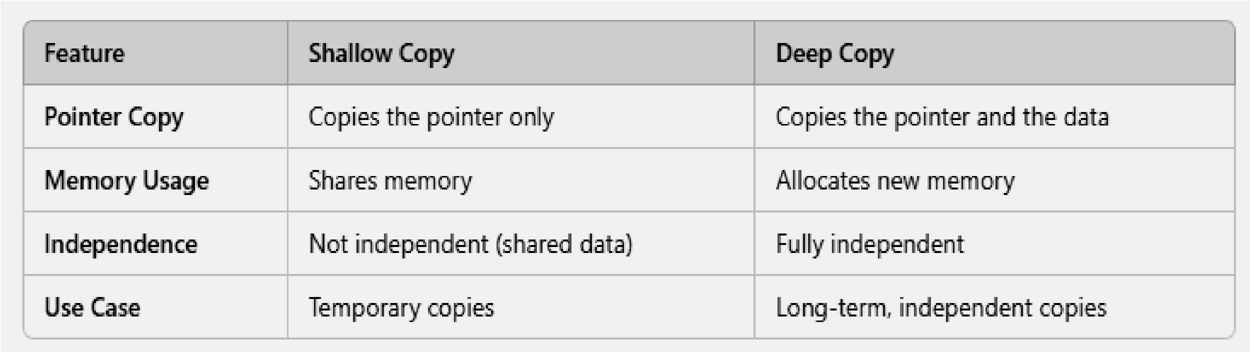
Output:

```
 42   42
100   42
```

Explanation:

- obj1 and obj2 have separate memory allocations.
- Changing obj3.1.data does not affect obj3.2.data.

Table 3.1 Comparison between Shallow Copy and Deep Copy



Feature	Shallow Copy	Deep Copy
Pointer Copy	Copies the pointer only	Copies the pointer and the data
Memory Usage	Shares memory	Allocates new memory
Independence	Not independent (shared data)	Fully independent
Use Case	Temporary copies	Long-term, independent copies

Dynamic memory allocation and proper copy mechanisms are fundamental for managing resources and ensuring robust, efficient programs in C++.

3.4 CLASSES, STRUCTURES, AND POINTER VARIABLES:

Classes, structures, and pointer variables together form the backbone of programming paradigms. While classes support abstraction and modularity, structures provide a simple way to group data, and pointers offer dynamic access and manipulation of memory.

3.4.1 Classes and Pointer Variables

In C++, **classes** are user-defined data types that encapsulate data and functions. Pointer variables can be used with classes to dynamically allocate memory for objects and access their members.

Example:

```
class Sample {
public:
    int value;
    void display() {
        cout << "Value: " << value << endl;
    }
};

int main() {
    Sample* obj = new Sample();  // Dynamically allocate memory for the object
    obj->value = 10;             // Access member using the arrow operator
    obj->display();              // Call member function
    delete obj;                  // Free allocated memory
    return 0;
}
```

Key Points:

- **Arrow Operator (->)**: Used to access class members when working with pointers.
- **Dynamic Object Creation**: new is used to allocate memory for class objects.
- **Memory Deallocation**: delete is used to free dynamically allocated memory.

3.4.2 Structures and Pointer Variables

In C++, **structures** are similar to classes but default to public access for members. Pointer variables can also be used with structures to dynamically allocate memory and access members.

Example:

```
struct Point {
    int x, y;
};

int main() {
    Point* p = new Point;  // Dynamically allocate memory for a structure
    p->x = 5;
    p->y = 10;

    cout << "Point: (" << p->x << ", " << p->y << ")" << endl;

    delete p;  // Free allocated memory
    return 0;
}
```

- **Dynamic Allocation**: Structures can be dynamically allocated like classes.
- **Pointer Access**: The arrow operator (->) is used to access members.

3.5 INITIALIZING POINTER VARIABLES:

Pointer variables need to be initialized before use. Uninitialized pointers can lead to undefined behavior, including runtime crashes.

3.5.1 Initialization Methods

1. **NULL Pointer Initialization**:
   - A pointer is initialized to NULL (C++11 and later) or NULL to indicate it does not point to any memory location.

```
int* ptr = NULL;
if (ptr == NULL) {
   cout << "Pointer is null." << endl;
}
```

2. **Pointing to a Variable**:
   - A pointer is assigned the address of a variable using the address-of operator (&).

```
int num = 10;
int* ptr = &num;
cout << "Address: " << ptr << ", Value: " << *ptr << endl;
```

3. **Dynamic Memory Allocation**:
   - A pointer is initialized to point to memory dynamically allocated using the new operator.

```
int* ptr = new int(20);
cout << "Value: " << *ptr << endl;
delete ptr;  // Free memory
```

3.6 DYNAMIC VARIABLES:

Dynamic variables are variables created at runtime using dynamic memory allocation. These variables exist on the **heap**, not the stack.

3.6.1 Creating Dynamic Variables

The new operator is used to allocate memory dynamically, and the delete operator is used to free it.

Example:

```
int* num = new int(15);  // Dynamically allocate memory for an integer
cout << "Value: " << *num << endl;

delete num;  // Free allocated memory
```

3.6.2 Dynamic Arrays

Dynamic memory can be allocated for arrays as well.

```
int* arr = new int[5];  // Dynamically allocate memory for an array
for (int i = 0; i < 5; i++) {
   arr[i] = i * 2;
}
for (int i = 0; i < 5; i++) {
   cout << arr[i] << " ";
}
delete[] arr;  // Free array memory
```

Advantages of Dynamic Variables

- **Flexible Memory Usage**: Allocate memory as needed during runtime.
- **Efficient Resource Utilization**: Memory is used only when required, avoiding wastage.

Disadvantages of Dynamic Variables

- **Memory Leaks**: Forgetting to deallocate memory with delete leads to memory leaks.
- **Dangling Pointers**: Using a pointer after its memory is deallocated results in undefined behavior.

3.7 OPERATORS ON POINTER VARIABLES:

C++ provides several operators to work with pointer variables.

i. Address-of Operator (&)

- Used to retrieve the address of a variable.

```
int num = 5;
int* ptr = &num;
cout << "Address of num: " << &num << endl;
```

ii. Dereferencing Operator (*)

- Accesses the value stored at the memory location the pointer points to.

```
int num = 10;
int* ptr = &num;
cout << "Value: " << *ptr << endl;
```

3.7.1 Pointer Arithmetic

Incrementing a Pointer (++)

- Moves the pointer to the next memory location of the same data type.
- For example:

  cpp

  Copy code

  ```
  int arr[3] = {10, 20, 30};
  int* ptr = arr;  // Points to arr[0]

  cout << *ptr << endl;  // Outputs 10
  ptr++;  // Moves to the next element (arr[1])
  cout << *ptr << endl;  // Outputs 20
  ```

Decrementing a Pointer (--)

- Moves the pointer to the previous memory location of the same data type.
- For example:

  ```
  int arr[3] = {10, 20, 30};
  int* ptr = &arr[1];  // Points to arr[1]

  cout << *ptr << endl;  // Outputs 20
  ptr--;  // Moves to the previous element (arr[0])
  cout << *ptr << endl;  // Outputs 10
  ```

Addition with Pointers (+)

- Adds an integer value to a pointer, moving it forward by the specified number of elements.
- For example:

  ```
  int arr[3] = {10, 20, 30};
  int* ptr = arr;  // Points to arr[0]

  ptr = ptr + 2;  // Moves forward by 2 elements (points to arr[2])
  ```

```
cout << *ptr << endl;  // Outputs 30
```

Subtraction with Pointers (-)

- Subtracts an integer value from a pointer, moving it backward by the specified number of elements.
- For example:

```
int arr[3] = {10, 20, 30};
int* ptr = &arr[2];  // Points to arr[2]

ptr = ptr - 1;  // Moves backward by 1 element (points to arr[1])
cout << *ptr << endl;  // Outputs 20
```

Pointer Difference

You can calculate the number of elements between two pointers pointing to the same array:

```
int arr[5] = {10, 20, 30, 40, 50};
int* start = &arr[0];
int* end = &arr[4];

cout << "Distance between pointers: " << end - start << endl;  // Outputs 4
```

3.7.2 Pointer Arithmetic with Arrays

Since arrays are stored in contiguous memory, pointer arithmetic allows efficient traversal.

Example: Iterating Over an Array with a Pointer

```
int main() {
   int arr[5] = {10, 20, 30, 40, 50};
   int* ptr = arr;  // Points to the first element

   for (int i = 0; i < 5; i++) {
      cout << *ptr << " ";  // Outputs the value at the pointer
      ptr++;  // Moves to the next element
   }

   return 0;
}
```

Output:

```
10 20 30 40 50
```

3.7.3 Pointer Arithmetic and Data Types

Pointer arithmetic is influenced by the size of the data type the pointer points to:

- Incrementing a pointer increases its value by the size of the data type it points to.
- For instance:
  - If an int is 4 bytes, incrementing an int* pointer increases its value by 3.4.
  - If a char is 1 byte, incrementing a char* pointer increases its value by 3.1.

Example:

```
int main() {
   int arr[3] = {1, 2, 3};
   char str[3] = {'a', 'b', 'c'};

   int* intPtr = arr;
   char* charPtr = str;

   cout << "Initial int pointer address: " << intPtr << endl;
   cout << "Initial char pointer address: " << (void*)charPtr << endl;

   intPtr++;
   charPtr++;

   cout << "After increment, int pointer address: " << intPtr << endl;
   cout << "After increment, char pointer address: " << (void*)charPtr << endl;

   return 0;
}
```

Output:

```
Initial int pointer address: 0x7ffee4f8
Initial char pointer address: 0x7ffee4f4
After increment, int pointer address: 0x7ffee4fc
After increment, char pointer address: 0x7ffee4f5
```

3.7.4 Invalid Operations in Pointer Arithmetic

1. **Pointer Out-of-Bounds**:
   - Moving a pointer beyond the allocated memory results in undefined behavior.
   - Example:

```
int arr[3] = {1, 2, 3};
int* ptr = arr + 3;  // Undefined behavior
```

2. **Arithmetic Between Pointers Pointing to Different Arrays**:
   - Pointer arithmetic is valid only within the same array or memory block.
   - Example:

```
int arr1[3] = {1, 2, 3};
int arr2[3] = {4, 5, 6};
int* ptr1 = arr1;
int* ptr2 = arr2;

// cout << ptr1 - ptr2;  // Undefined behavior
```

Key Points

1. **Pointer Arithmetic Basics**:
   - Increment/Decrement adjusts the pointer by the size of the data type.
   - Addition/Subtraction allows moving the pointer by multiple elements.
2. **Contiguous Memory**:
   - Pointer arithmetic is most effective with arrays due to their contiguous memory storage.
3. **Type Safety**:
   - Pointer arithmetic respects the size of the data type it points to.
4. **Efficiency**:
   - Enables efficient traversal of arrays and memory blocks.

Mastering pointer arithmetic is essential for understanding memory management, dynamic arrays, and low-level programming in C++. It provides a powerful way to navigate memory efficiently, but it must be used carefully to avoid errors and undefined behavior.

3.7.5 Comparison Operators

- Pointers can be compared using relational operators (==, !=, <, >).

Example:

```
int x = 10, y = 20;
int* ptr1 = &x;
int* ptr2 = &y;

if (ptr1 != ptr2) {
   cout << "Pointers are different." << endl;
}
```

3.7.6 NULL Pointer (NULL)

- Introduced in C++, NULL represents a pointer that does not point to any memory.

```
int* ptr = NULL;
if (ptr == NULL) {
  cout << "Pointer is null." << endl;
}
```

Key Points:

1. **Classes and Structures**:
   - Pointers can dynamically allocate memory for class objects and structures.
   - Use the -> operator to access members through pointers.
2. **Pointer Initialization**:
   - Always initialize pointers to NULL or valid memory.
   - Avoid using uninitialized pointers.
3. **Dynamic Variables**:
   - Use new for allocation and delete for deallocation.
   - Be cautious of memory leaks and dangling pointers.
4. **Pointer Operators**:
   - Address-of (&) retrieves the address.
   - Dereferencing (*) accesses the value.
   - Pointer arithmetic navigates memory.

3.8 ARRAY-BASED LISTS:

An **array-based list** is a linear data structure that uses an array to store a collection of elements. It supports various operations such as insertion, deletion, traversal, and search. The list size can be fixed (static arrays) or dynamic (using dynamic memory allocation).

Key Characteristics:

- **Sequential Storage**: All elements are stored in contiguous memory locations.
- **Index-Based Access**: Elements can be accessed directly using their index, enabling efficient random access.
- **Fixed Size**: The maximum size of the list is determined at the time of array declaration for static arrays.

Advantages:

- **Fast Access**: Elements can be accessed in O(1)O(1)O(1) time using their index.
- **Simple Implementation**: Easy to implement and understand.

Limitations:

- **Fixed Capacity**: Cannot resize dynamically in static arrays.
- **Inefficient Insertions/Deletions**: Operations involving shifting elements take O(n)O(n)O(n) time in the worst case.

3.8.1 Initialization

To create and initialize an array-based list, we define an array and specify its size. Optionally, the array elements can be initialized to specific values.

Example:

```
const int SIZE = 10;  // Define the maximum size of the array
int list[SIZE] = {0};  // Initializes an array of size 10 with all elements set to 0
```

Notes:

- If specific values are not provided during initialization, elements may contain garbage values (for non-static arrays).
- Initialization ensures that the array starts in a predictable state.

3.8.2 Operations on Array-Based Lists

Array-based lists support several fundamental operations for managing and manipulating data. Each operation has specific time and space complexities depending on its implementation.

i. Insertion

Insertion is the process of adding an element at a specific position in the array. If the position is not at the end, elements need to be shifted to make room for the new value.

Steps:

1. Ensure there is space in the array for the new element.
2. Shift elements to the right starting from the last element up to the specified position.
3. Insert the new value at the desired position.
4. Increment the size of the list.

Characteristics:

- **Time Complexity**:
  - Best Case: O(1) (if inserting at the end).
  - Worst Case: O(n) (if inserting at the beginning or in the middle).
- **Space Complexity**: O(1), as no additional memory is required.

ii. Deletion

Deletion involves removing an element from a specific position in the array. After removing the element, all subsequent elements need to be shifted left to fill the gap.

Steps:

1. *Loc*ate the element at the specified position.
2. Shift all elements to the left, starting from the next element.
3. Decrement the size of the list.

Characteristics:

- **Time Complexity**:
  - Best Case: O(1) (if deleting the last element).
  - Worst Case: O(n) (if deleting from the beginning or middle).
- **Space Complexity**: O(1), as no additional memory is required.

iii. Search

Search involves finding the position of an element in the array-based list. The search can be **linear** or **binary**, depending on whether the array is sorted.

Linear Search:

- Checks each element sequentially until the desired value is found or the end of the array is reached.

Steps:

1. Start from the first element.
2. Compare each element with the target value.
3. If found, return the position. Otherwise, return -3.1.

Characteristics*:*

- **Time Complexity**: O(n), where n is the number of elements.
- **Space Complexity**: O(1), as no additional memory is required.

Limitations:

- Linear search is inefficient for large datasets compared to binary search (in sorted arrays).

iv. Traversal

Traversal involves visiting and processing each element in the array sequentially. It is commonly used to display the elements or perform operations on each element.

Steps:

1. Start from the first element.
2. Visit each element sequentially up to the last element.
3. Perform the desired operation (e.g., print the element).

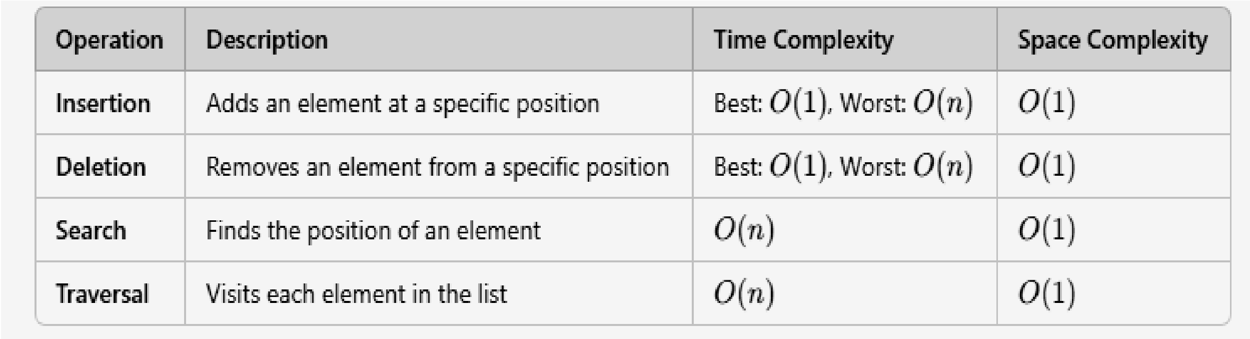
Characteristics:

- **Time Complexity**: O(n), where n is the number of elements.
- **Space Complexity**: O(1), as no additional memory is required.

Use Cases:

- Displaying the list contents.
- Applying a transformation to each element.

Table 3.2 Summary of Array-Based List Operations



Operation	Description	Time Complexity	Space Complexity
Insertion	Adds an element at a specific position	Best: $O(1)$, Worst: $O(n)$	$O(1)$
Deletion	Removes an element from a specific position	Best: $O(1)$, Worst: $O(n)$	$O(1)$
Search	Finds the position of an element	$O(n)$	$O(1)$
Traversal	Visits each element in the list	$O(n)$	$O(1)$

3.9 KEY TERMS:

Pointer, Dynamic Memory Allocation, Shallow Copy, Deep Copy, Array-Based List, Pointer Arithmetic

3.10 REVIEW QUESTIONS:

1) What is a pointer, and how is it declared in C++?
2) Explain the difference between shallow copy and deep copy with examples?
3) How does the new operator work for dynamic memory allocation, and why is it important?
4) What are the main steps involved in the insertion and deletion of elements in an array-based list?
5) How does pointer arithmetic affect traversal of arrays?

3.11 SUGGESTED READINGS:

1) “The C++ Programming Language” by Bjarne Stroustrup.
2) “Effective C++: 55 Specific Ways to Improve Your Programs and Designs” by Scott Meyers.
3) “Programming: Principles and Practice Using C++” by Bjarne Stroustrup.
4) “Data Structures and Algorithm Analysis in C++” by Mark Allen Weis.

Dr. Neelima Guntupalli

LESSON-4

LINKED LISTS AND THEIR TYPES

OBJECTIVES:

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:

4.1 INTRODUCTION:

Linked lists are dynamic data structures used to efficiently manage collections of data. Each node in a linked list contains two components:

- **Data Field**: Stores the actual data.
- **Pointer Field**: Stores the memory address of the next node (and sometimes the previous node, in the case of doubly linked lists).

Key Properties of Linked Lists:

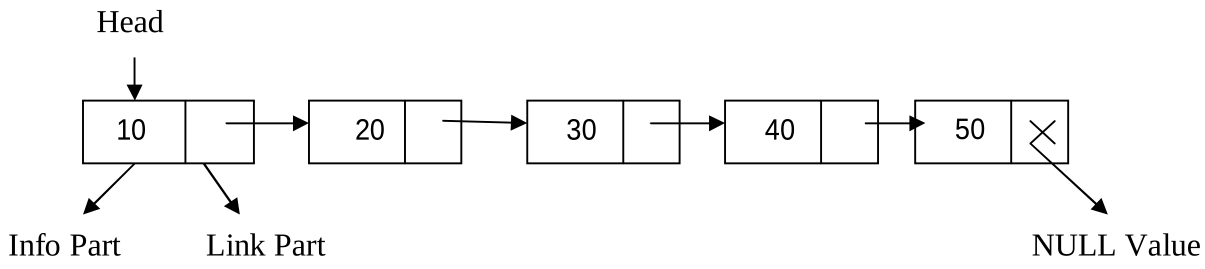
1. Nodes are not stored in contiguous memory locations.
2. Memory is dynamically allocated, allowing the list to grow or shrink as needed.
3. Linked lists provide efficient insertion and deletion operations compared to arrays.

4.2 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 simple node structure in C++ can be defined as follows:

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

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.

4.3 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
- Doubly Circular Linked List

4.3.1. 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++:

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

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:

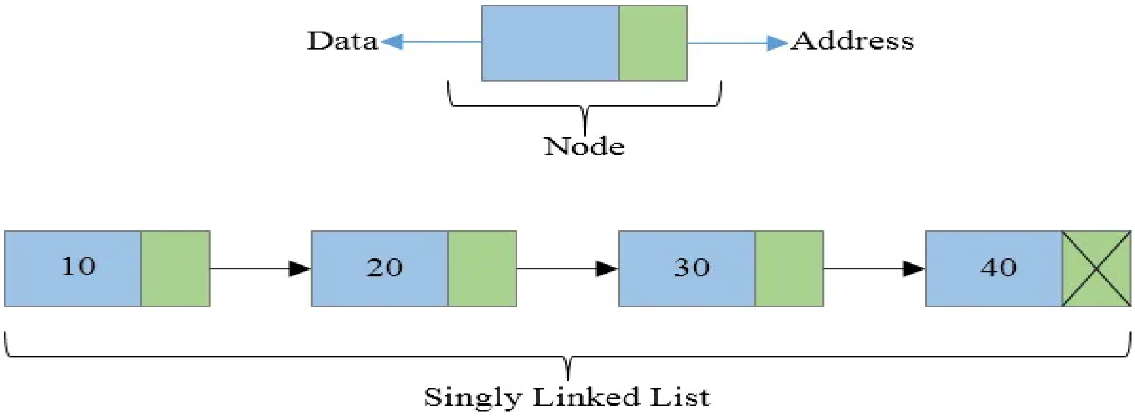


Figure 4.1 Single Linked List representation

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.

4.3.2. 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++:

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

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

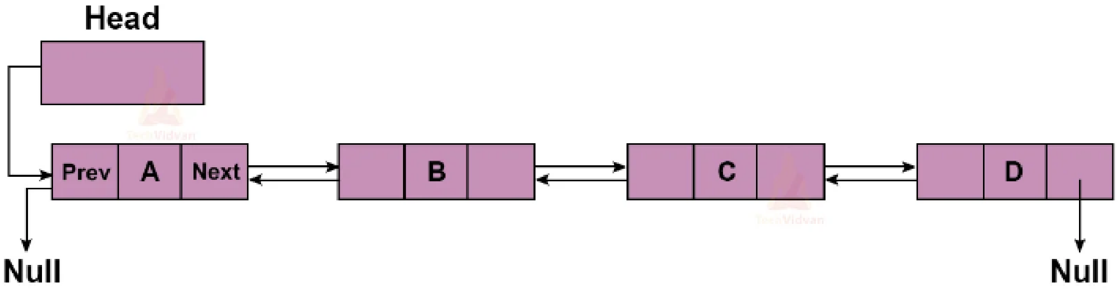


Figure 4.2 Double Linked List representation

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.

4.3.3 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:

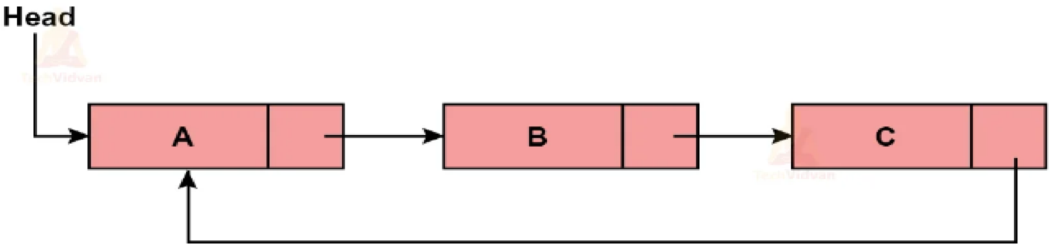


Figure 4.3 Circular Linked List representation

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.

4.3.4 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:

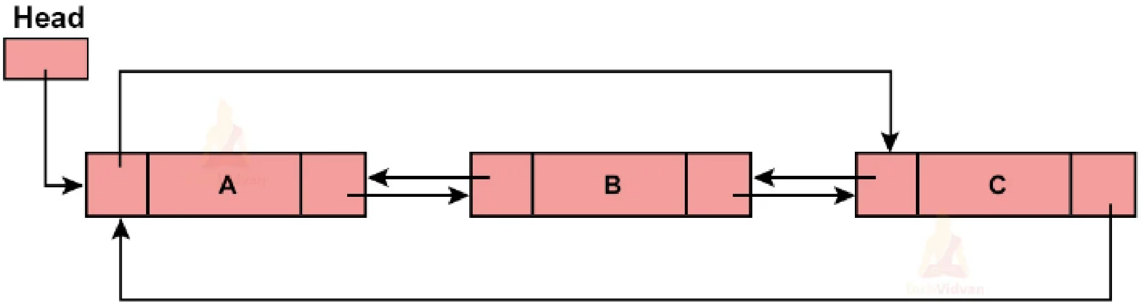


Figure 4.4 Doubly Circular Linked List representation

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:

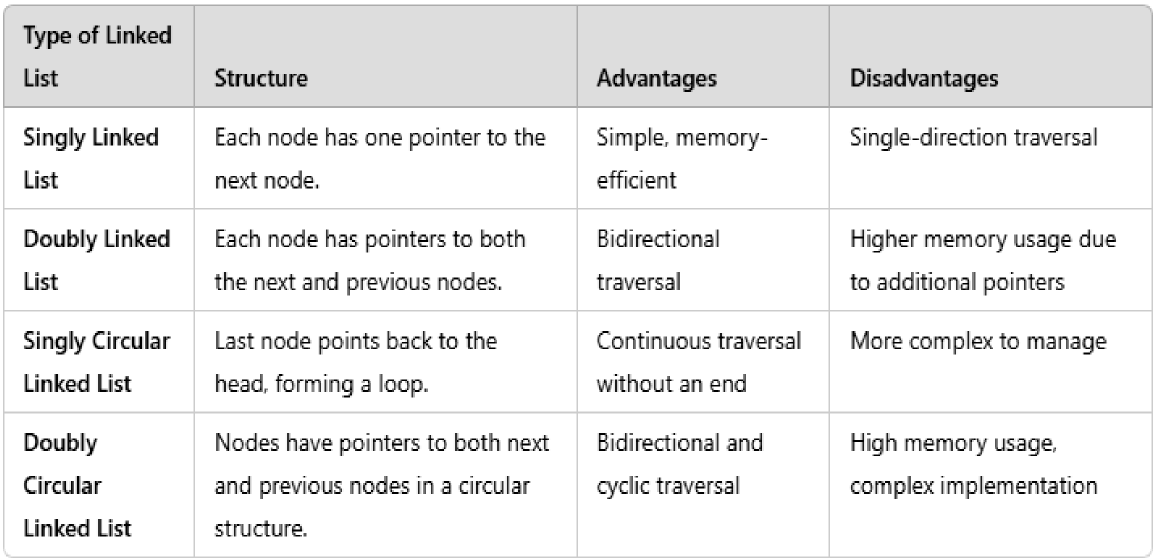
- Allows bidirectional and cyclic traversal.
- Efficient for applications that need continuous, two-way traversal.

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 4.1 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

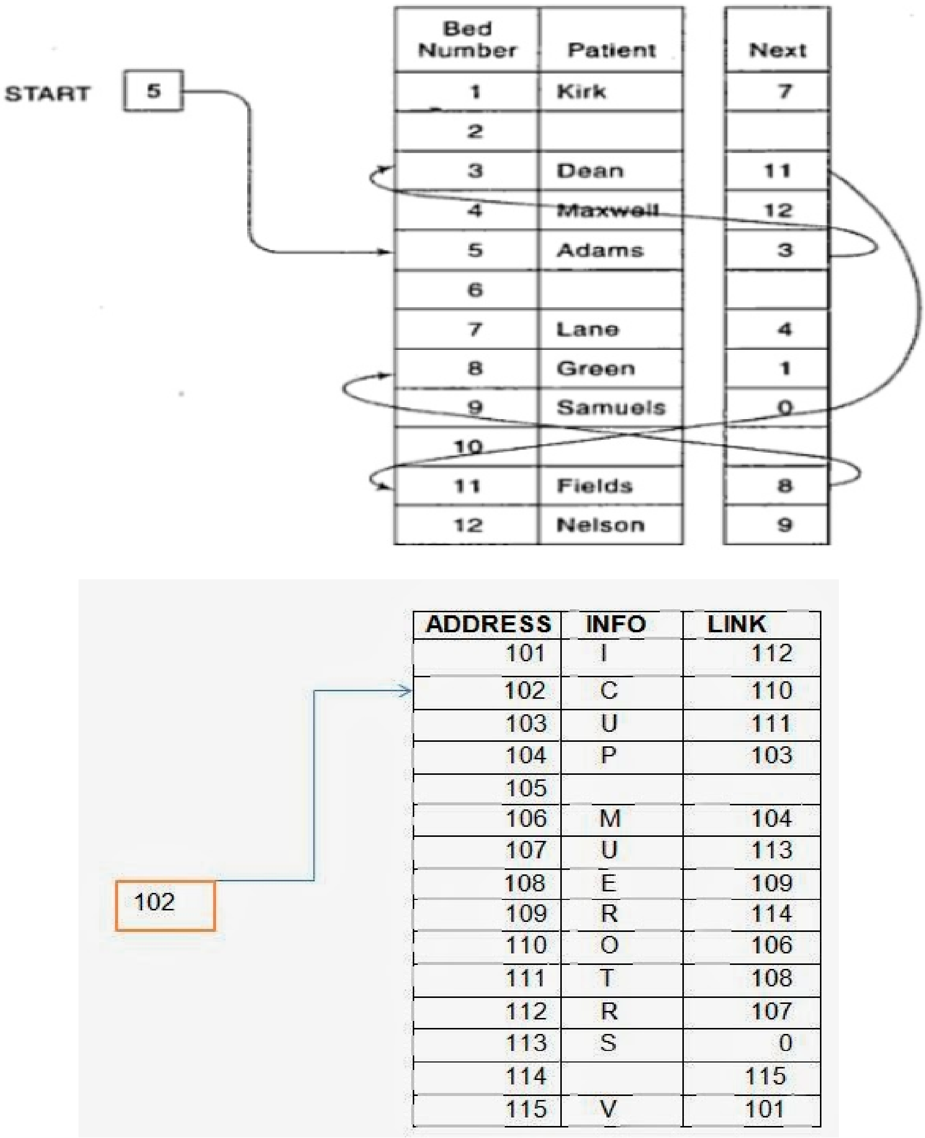
4.4 SIGNIFICANCE OF LINKED LISTS:

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.

4.5 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.



Bed Number	Patient	Next
1	Kirk	7
2		
3	Dean	11
4	Maxwell	12
5	Adams	3
6		
7	Lane	4
8	Green	1
9	Samuels	0
10		
11	Fields	8
12	Nelson	9

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 4.5 Memory representation of Linked Lists

4.5.1 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.

4.5.2 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;

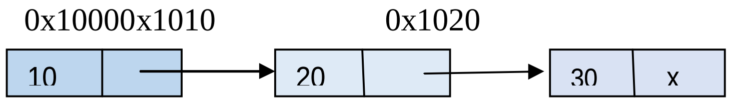
The pointer-based structure is what makes linked lists flexible and dynamic. However, it also requires careful pointer manipulation to ensure the structure remains intact, especially during insertions and deletions.

4.5.3 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.

4.6 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.

4.6.1 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
   delete 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.

4.6.2 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

4.7 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:

4.7.1 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).

4.7.2 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.

4.7.3. 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.

4.7.4 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.

4.7.5 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.

4.7.6 Resizing and Memory Allocation:

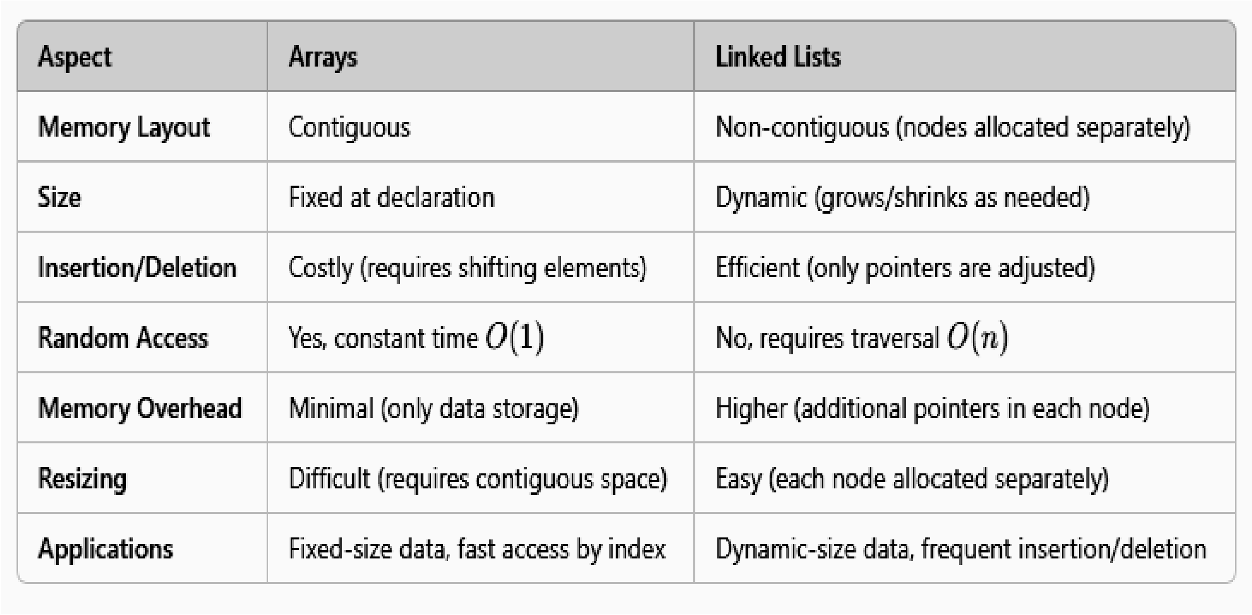
- **Arrays**: Arrays require contiguous memory for all elements, which means resizing can be a challenge. If an array needs to be expanded, a new, larger block of memory must be allocated, and all elements must be copied to this new block. This process can be time-consuming, especially if it needs to 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.

4.7.7 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 4.2 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

4.8 KEY TERMS:

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

4.9 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?

4.10 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. Neelima Guntupalli

LESSON-5
OPERATIONS ON LINKED LISTS

OBJECTIVES:

The objectives of the lesson are to

1. Understand the concept and structure of linked lists, including singly and doubly linked lists.
2. Learn and implement basic operations such as traversal, insertion, deletion, and list building for linked lists.
3. Explore the key differences and advantages of singly and doubly linked lists.
4. Gain practical knowledge of linked list implementation using examples and step-by-step instructions.

STRUCTURE:

5.1 INTRODUCTION:

Linked lists are an essential dynamic data structure where elements (nodes) are linked using pointers. Each node stores data and a reference to the next node, forming a chain. This structure allows efficient insertion, deletion, and traversal. The lesson covers operations for singly and doubly linked lists, highlighting key concepts, practical examples, and their applications. Through clear explanations and code snippets, you will understand how linked lists operate and how to use them effectively.

5.2 LINKED LIST OPERATIONS:

Linked lists are a fundamental data structure widely used in programming for dynamic data management. They consist of nodes where each node contains data and a pointer to the next node. This structure allows efficient insertion and deletion of elements. The following notes cover key operations performed on linked lists, including traversing**,** insertion**,** deletion**,** and building linked lists (both forward and backward). Each operation is explained with step-by-step instructions, code examples, and scenarios to ensure a clear understanding of how linked lists work.

5.2.1 Traversing a Linked List

In a linked list, basic operations like searching for an item, inserting a new item, or deleting an existing item often require going through the entire list. To do this, we need to traverse the list, starting from the first node and moving through each subsequent node.

Imagine we have a pointer, head, which points to the first node in the list. The last node in the list has its link set to NULL, indicating the end of the list. We cannot use the head pointer directly for traversal because doing so would result in losing the reference to the beginning of the list.

This issue arises because a singly linked list only links nodes in one direction:

- The head pointer stores the address of the first node.
- The first node contains the address of the second node.
- The second node links to the third node, and so on.

If we move the head pointer to the second node, the reference to the first node is lost unless we explicitly save it. Continuing this process would lead to losing all previous nodes unless we store their addresses, which is inefficient and uses additional time and memory.

To avoid this problem, we always keep head pointing to the first node in the list. Instead of using head for traversal, we use a different pointer, such as current. This way, current moves through the nodes while head remains unchanged, preserving the reference to the start of the list.

Below is a sample code snippet for traversing a linked list using the current pointer:

```
Node* current = head; // Start from the first node
while (current != NULL) {
   // Process the current node (e.g., print the data)
   cout << current->data << " ";
   // Move to the next node
   current = current->next;
}
```

Here’s how it works:

1. Assign head to current, so current starts at the first node.
2. While current is not NULL (indicating the end of the list), process the current node (e.g., display its data).
3. Move current to the next node by following the link in the current node.

This approach ensures that the head pointer always retains its original position, while the current pointer handles traversal.

5.2.2 Insertion

In the below figure, there is a linked list where p points to a node containing the value 65. A new node with the value 50 needs to be inserted after this node.

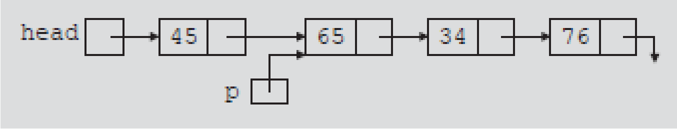


Fig. 5.1 Linked list before item insertion

Steps to Insert the Node:

1. **Create the new node**

   ```
   newNode = new nodeType; // Create a new node
   newNode->info = 50;     // Store the value 50 in the new node
   ```

2. **Adjust the links**

   Execute the following statements in the given order:

   ```
   newNode->link = p->link; // Link the new node to the node after p
   p->link = newNode;       // Link p to the new node
   ```

   This ensures that the new node is properly inserted after p. The sequence of these statements is critical because reversing their order would result in an incorrect configuration where part of the list is lost.

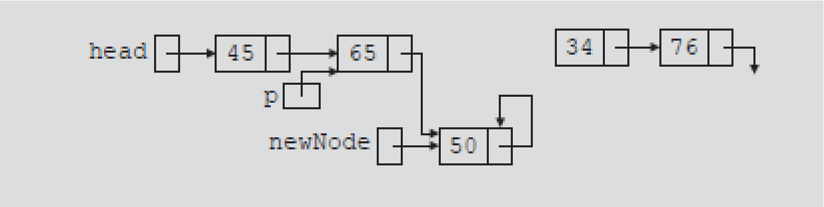


Fig. 5.2. List after the execution of the statement

p->link = newNode; followed by the execution of the statement newNode->link = p->link;

Incorrect Sequence Example:

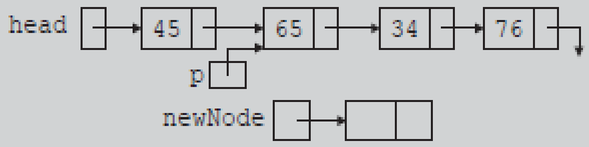
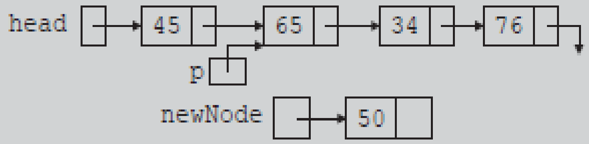
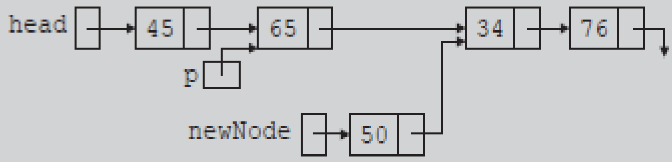
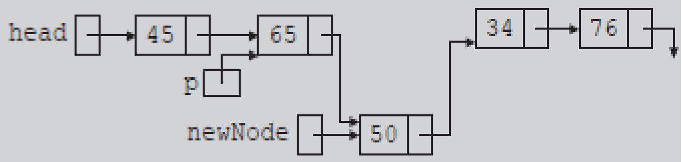
If the statements are reversed:

p->link = newNode;

newNode->link = p->link;

The list would end up in an invalid state, as the new node would link back to itself, and the rest of the list would be disconnected.

TABLE 5.1 INSERTING A NODE IN A LINKED LIST

Statement	Effect
`newNode = new nodeType;`	
`newNode->info = 50;`	
`newNode->link = p->link;`	
`p->link = newNode;`	

Simplifying with Two Pointers:

Using two pointers (p and q), the insertion process can be simplified. Suppose q points to the node after p (e.g., the node with value 34). To insert newNode between p and q, execute:

newNode->link = q; // Link the new node to q

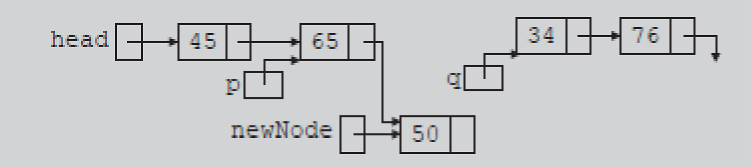
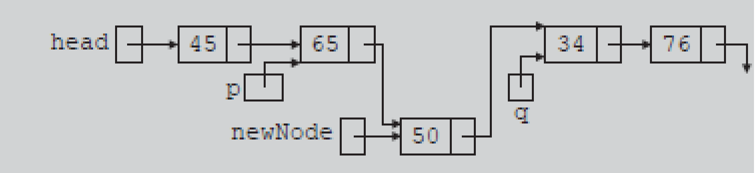
p->link = newNode; // Link p to the new node

In this case, the order of the statements does not matter, as both result in the correct configuration.

Final Code

```
newNode = new nodeType;     // Create the new node
newNode->info = 50;         // Store 50 in the new node
newNode->link = p->link;    // Link the new node to the node after p
p->link = newNode;          // Link p to the new node
```

TABLE 5.2 INSERTING A NODE IN A LINKED LIST USING TWO POINTERS

Statement	Effect
`p->link = newNode;`	
`newNode->link = q;`	

This process ensures that the new node is inserted correctly without breaking the linked list structure. The sequence of operations is crucial to maintain the integrity of the list.

5.2.3 Deletion

Consider the linked list shown in Figure 5-9.

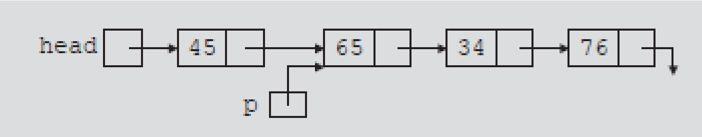


Fig. 5.3 Node to be deleted is with info 34

Suppose that the node with info 34 is to be deleted from the list. The following statement removes the node from the list:

p->link = p->link->link;

Figure 5.4 shows the resulting list after the preceding statement executes.

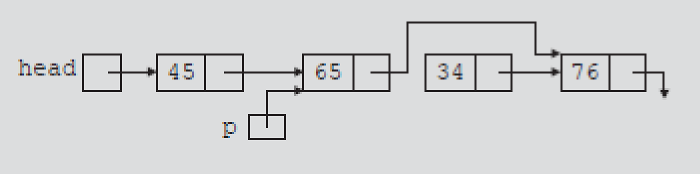


Fig. 5.4 List after the statement p->link = p->link->link; executes

From Figure 5.4, it is clear that the node with info 34 is removed from the list. However, the memory is still occupied by this node and this memory is inaccessible; that is, this node is dangling. To deallocate the memory, we need a pointer to this node. The following statements delete the node from the list and deallocate the memory occupied by this node:

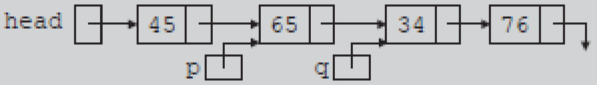
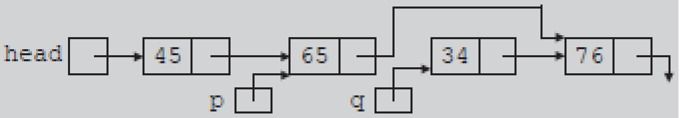
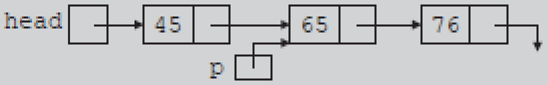
q = p->link;

p->link = q->link;

delete q;

Table 5.3 shows the effect of these statements.

TABLE 5.3 DELETING A NODE FROM A LINKED LIST

Statement	Effect
`q = p->link;`	
`p->link = q->link;`	
`delete q;`	

5.2.4 Building a Linked List

Now that we understand how to insert nodes into a linked list, let's learn how to construct a linked list. When the data is unsorted, the resulting linked list will also be unsorted. A linked list can be built in two ways:

1. **Forward:** New nodes are added to the end of the list.
2. **Backward:** New nodes are added to the beginning of the list.

Let's focus on building a linked list in the **forward** manner.

Building a Linked List Forward

Example:

We want to create a linked list to store the following data:

2, 15, 8, 24, 34

Pointers Needed:

We need three pointers:

1. first - Points to the first node in the list (remains unchanged).
2. last - Points to the last node in the list.
3. newNode - Used to create a new node.

Variable Declaration:

```
nodeType *first, *last, *newNode;
int num;
```

Initializing the List:

At the start, the list is empty, so both first and last are set to NULL:

```
first = NULL;
last = NULL;
```

Steps to Build the Linked List:

1. **Read Data:**

   ```
   cin >> num; // Input a number and store it in num
   ```

2. **Create a New Node:**

   ```
   newNode = new nodeType;  // Allocate memory for a new node
   newNode->info = num;     // Store the value of num in the node
   newNode->link = NULL;    // Set the link of the new node to NULL
   ```

3. **Insert the Node into the List:**

   - If the list is empty (first == NULL):

     ```
     first = newNode;  // Make first point to the new node
     last = newNode;   // Make last point to the new node
     ```

   - If the list is not empty:

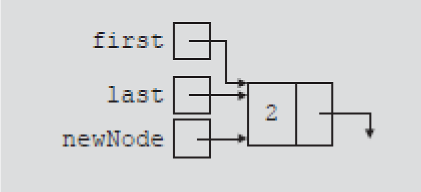
     ```
     last->link = newNode; // Add the new node to the end of the list
     last = newNode;       // Update last to point to the new last node
     ```

Execution Example:

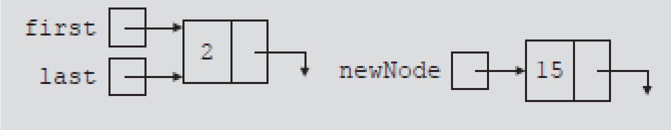
1. Initially, both first and last are NULL.

   When num = 2, a new node is created, and since the list is empty, both first and last point to this new node.

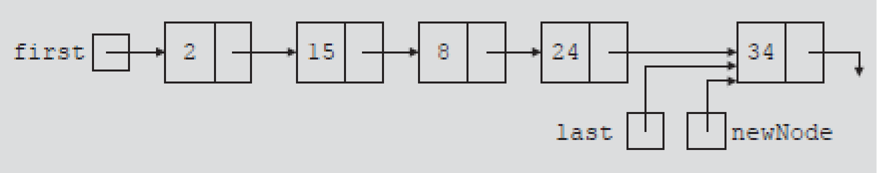
   The list now looks like this:



2. Next, when num = 15, a new node is created with the value 15. Since the list is not empty, this node is added to the end of the list, and last is updated. The list becomes:



3. Similarly, the steps are repeated for the remaining data (8, 24, 34), with each new node added to the end of the list. The final list looks like this:



Code for Building the List Forward

```
first = NULL;
last = NULL;

while (cin >> num) {            // Read numbers until input ends
   newNode = new nodeType;      // Create a new node
   newNode->info = num;         // Store the value in the node
   newNode->link = NULL;        // Set the link to NULL

   if (first == NULL) {         // If the list is empty
      first = newNode;          // Set first to the new node
      last = newNode;           // Set last to the new node
} else {                    // If the list is not empty
    last->link = newNode;   // Add the new node to the end
    last = newNode;         // Update last to point to the new node
}  }
```

This process ensures the linked list is constructed step by step, with each new node appended at the end.

Building a Linked List Backward

In the backward method, new nodes are always added at the beginning of the linked list. Each new node is linked to the current first node, and then the pointer first is updated to point to the new node. This method makes the most recently added node the head of the list. The process continues until all data is added, resulting in the list being built in reverse order.

5.3 LINKED LIST AS AN ABSTRACT DATA TYPE (ADT):

A linked list can be conceptualized as an Abstract Data Type (ADT) with the following operations:

1. **Initialize the List**: Create an empty list.
2. **Check if the List is Empty**: Determine whether the list contains any nodes.
3. **Print the List**: Traverse the list and display its contents.
4. **Find the Length**: Count the number of nodes in the list.
5. **Destroy the List**: Deallocate memory used by all nodes.
6. **Retrieve the First Node**: Access the data in the first node.
7. **Retrieve the Last Node**: Access the data in the last node.
8. **Search for an Item**: Find a node with specific data.
9. **Insert an Item**: Add a new node at a specific position.
10. **Delete an Item**: Remove a node from a specific position.
11. **Make a Copy**: Create a duplicate of the linked list.

These operations allow linked lists to serve as a flexible data structure for various applications.

5.3.1 Structure of Linked List Nodes

A node is the building block of a linked list. It typically includes:

Singly Linked List Node:

```
struct Node {
    int data;     // Data field
    Node* next;   // Pointer to the next node
};
```

Doubly Linked List Node:

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

In a doubly linked list, the addition of the prev pointer enables traversal in both forward and backward directions.

5.3.2 Class Member Variables for Linked Lists

A class to manage linked lists typically includes:

1. **Head Pointer**: Points to the first node of the list.
2. **Length**: Tracks the number of nodes in the list.
3. **Member Functions**: Perform operations like insertion, deletion, and traversal.

Example Class Structure:

```
class LinkedList {
    Node* head;
    int length;

public:
    LinkedList() {
        head = NULL;
        length = 0;
    }

    void insert(int data);
    void display();
```

```
   int getLength() {
      return length;
   }

   ~LinkedList();};
```

5.3.3 Linked List Iterators

An iterator simplifies traversal of a linked list by abstracting node navigation.

Example Iterator Class:

```
class LinkedListIterator {
   Node * current;
public:
   LinkedListIterator(Node* start) { current = start; }
   bool hasNext() { return current != NULL; }
   int next() { int d = current->data; current = current->next; return d; }
   };
```

This iterator provides sequential access to each node's data.

5.3.4 Default Constructor

The default constructor initializes a linked list to an empty state:

Example Code:

```
class LinkedList {
private:
   Node* head;

public:
   LinkedList() : head(NULL) {}
};
```

In this example, the head pointer is set to NULL, indicating that the list is initially empty.

5.3.5 Destroy the List

To avoid memory leaks, all nodes should be deleted:

```
~LinkedList() {
   Node* temp;
   while (head != NULL) {
      temp = head;
      head = head->next;
      delete temp;
   }
}
```

5.3.6 Initialize the List

To reset a list, delete all nodes and set the head pointer to NULL.

5.3.7 Print the List

Display all nodes:

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

5.3.8 Length of a List

Calculate the number of nodes:

```
int getLength() {
   int count = 0;
   Node* temp = head;
   while (temp != NULL) {
      count++;
      temp = temp->next;
   }
   return count;
}
```

5.3.9 Retrieve the Data of the First and Last Nodes

First Node

```
int getFirstNodeData(Node* head) {
   if (head == NULL) {
      return -1; // Indicates the list is empty
   }
   return head->data;
}
```

Last Node

```
int getLastNodeData(Node* head) {
   if (head == NULL) {
      return -1; // Indicates the list is empty }
   Node* temp = head;
   while (temp->next != NULL) {
      temp = temp->next; }
   return temp->data; }
```

5.3.10 Begin and End

```
Node* begin(Node* head) {
   return head; }
Node* end() {
   return NULL; // Indicates the end of the list }
```

5.3.11 Copy the List

The process of copying a linked list involves creating a new list where each node has the same data as the corresponding node in the original list. The structure ensures no memory overlap, making the lists independent.

The function copyList creates a deep copy of a linked list. A deep copy means that the new list has entirely new nodes, with data copied from the original list. Here's how the code works step-by-step:

1. Base Case: Check if the Original List is Empty

```
if (head == NULL) {
    return NULL; // Empty list, return NULL
}
```

- If the input linked list (head) is empty (NULL), the function returns NULL since there’s nothing to copy.

2. Create the First Node of the New List

```
Node* newHead = new Node();   // Create a new node
newHead->data = head->data;  // Copy the data from the first node of the original list
newHead->next = NULL;        // Initialize the next pointer to NULL
```

- A new node (newHead) is created to represent the head of the new list.
- The data from the head of the original list is copied to the new node.
- The next pointer of the new node is initialized to NULL.

3. Initialize Temporary Pointers for Traversal

```
Node* tempOld = head->next;  // Start from the second node in the original list
Node* tempNew = newHead;     // Start from the head of the new list
```

- tempOld is a pointer to traverse the original list, starting from the second node.
- tempNew is a pointer to traverse the new list, starting from newHead.

4. Traverse the Original List and Copy Each Node

```
while (tempOld != NULL) {
    Node* newNode = new Node();  // Create a new node
    newNode->data = tempOld->data;  // Copy the data from the current node of the original
list
    newNode->next = NULL;  // Initialize the next pointer of the new node to NULL

    tempNew->next = newNode;  // Link the new node to the new list
    tempNew = newNode;        // Move tempNew to the new node
    tempOld = tempOld->next;  // Move tempOld to the next node in the original list
}
```

- **Inside the Loop:**
    - A new node is created (newNode) for the new list.
    - The data of the current node in the original list (tempOld) is copied to newNode.

- The next pointer of the last node in the new list (tempNew) is updated to point to the newly created node.
- Both tempNew (new list pointer) and tempOld (original list pointer) are moved to their respective next nodes.

5. Return the Head of the New List

return newHead; // Return the head of the new list

- After copying all nodes, the function returns newHead, which is the head of the new linked list.

Key Points:

1. **Deep Copy:** This function creates a new list where each node is a completely new object.
2. **Independent Lists:** Changes made to the original list will not affect the copied list, and vice versa.
3. **Traversal Logic:** The function uses two pointers (tempOld and tempNew) to move through the original and new lists, respectively.
4. **Memory Allocation:** Each node in the new list is allocated dynamically using new.

Example Execution:

If the original list is:

[10] -> [20] -> [30] -> NULL

After calling copyList, the new list will be:

[10] -> [20] -> [30] -> NULL

- The data is the same, but each node in the new list is a distinct copy, not a reference to the original.

5.4 UNORDERED LINKED LISTS:

The class unordered Linked List is derived from the abstract class linkedListType. This class implements key operations such as search, insertFirst, insertLast, and deleteNode.

5.4.1 Search the List

To search for an item in an unordered linked list:

- Traverse through the list starting from the head.
- Compare the value of each node with the target value.
- If the target is found, return its position or confirm its presence; otherwise, return a message indicating it’s not in the list.

```
bool search(Node* head, int target)
```

```
{
  Node* temp = head; // Start from the head of the list
  while (temp != NULL) { // Traverse the list
    if (temp->data == target) // Check if the current node contains the target
      return true; // Target found
    temp = temp->next; // Move to the next node
  }
  return false; // Target not found
}
```

- **head**: The pointer to the first node of the linked list.
- **target**: The value you are searching for in the list.
- Traverse the list using a temporary pointer (temp).
- Compare the data of each node with the target.
- If a match is found, return true.
- If the end of the list is reached (temp == NULL), return false.

This function can be called in your program with the head of the linked list and the target value as arguments.

5.4.2 Item Insertion

Case 1: Insertion at the Beginning

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

- A new node is created.
- The next pointer of the new node is set to point to the current head.
- The head pointer is updated to point to the new node.

Case 2: Insertion at the End

```
void insertAtEnd(Node*& head, int data) {
  Node* newNode = new Node();
  newNode->data = data;
  newNode->next = NULL;
```

```
    if (head == NULL) {
        head = newNode;
        return;
    }
    Node* temp = head;
    while (temp->next != NULL) {
        temp = temp->next;
    }
    temp->next = newNode;
}
```

- A new node is created, and its next pointer is set to NULL.
- If the list is empty, the head pointer is updated to point to the new node.
- Otherwise, traverse the list to find the last node and update its next pointer to the new node.

Case 3: Insertion at a Specific Position

```
void insertAtPosition(Node*& head, int data, int position) {
    Node* newNode = new Node();
    newNode->data = data;
    if (position == 0) {
        insertAtBeginning(head, data);
        return;
    }
    Node* temp = head;
    for (int i = 0; i < position - 1; ++i) {
        if (temp == NULL) return; // Position out of bounds
        temp = temp->next;
    }
    newNode->next = temp->next;
    temp->next = newNode;
}
```

- If the position is 0, the node is added at the beginning using insertAtBeginning.

- For other positions, traverse the list to the desired position.
- Update the next pointers to insert the new node at the specified position.

5.4.3 Item Deletion

Case 1: Deletion at the Beginning

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

- Check if the list is empty. If not, store the head node in a temporary pointer.
- Update the head pointer to the next node and delete the original first node.

Case 2: Deletion from the End

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

- If the list has only one node, delete it and set the head pointer to NULL.
- For other cases, traverse the list to find the second-last node.
- Delete the last node by updating the next pointer of the second-last node to NULL.

Case 3: Deletion from a Specific Position

```
void deleteFromPosition(Node*& head, int position)

{

   if (position == 0) {

      deleteFromBeginning(head);
```

```
    return;
  }
  Node* temp = head;
  for (int i = 0; i < position - 1; ++i) {
    if (temp == NULL || temp->next == NULL) return; // Position out of bounds
    temp = temp->next;
  }
  Node* toDelete = temp->next;
  temp->next = toDelete->next;
  delete toDelete;
 }
```

- If the position is 0, the node is deleted from the beginning using deleteFromBeginning.
- For other positions, traverse to the previous node of the target position.
- Update the next pointer to skip the target node and delete the node.

5.5 DOUBLE LINKED LIST:

A doubly linked list is a type of linked list where each node contains two pointers:

1. next-Points to the next node in the list.
2. prev-Points to the previous node in the list.

All the operations we discussed for a singly linked list-like search, insertion (at the beginning, end, or specific position), and deletion-can also be performed on a doubly linked list. However, there are minor changes due to the additional prev pointer.

Key Modifications for Doubly Linked List

1. **Insertion:**
   - In addition to updating the next pointer, the prev pointer of the new node and adjacent nodes must also be updated.
   - Example:
     - When inserting at the beginning, set the prev pointer of the original first node to the new node.

2. **Deletion:**
   - o Update both the next pointer of the previous node and the prev pointer of the next node to remove a node from the list.
   - o Example:
     - ▪ When deleting a node, ensure the node before it points to the node after it, and vice versa.
3. **Traversal:**
   - o A doubly linked list can be traversed in **both forward and backward directions** using the next and prev pointers, respectively.

Advantages of Doubly Linked List

- Easier to traverse in both directions.
- Efficient deletion or insertion before a node since the prev pointer is readily available.

Operations on a doubly linked list are similar to those on a singly linked list but require updates to both the next and prev pointers, making it slightly more complex.

5.6 KEY TERMS:

Node, Head Pointer, Traversal, Insertion, Deletion, Doubly Linked List

5.7 REVIEW QUESTIONS:

1) What is the primary difference between a singly linked list and a doubly linked list?
2) Explain how to traverse a linked list without losing the reference to its head?
3) Write the steps to insert a node at the end of a singly linked list?
4) How is the prev pointer used in a doubly linked list for deletion?
5) What are the advantages of using a doubly linked list over a singly linked list?

5.8 SUGGESTED READINGS:

1) “Introduction to Algorithms” by Cormen, Leiserson, Rivest, and Stein.
2) “Data Structures and Algorithm Analysis in C++” by Mark Allen Weiss.
3) “Algorithms in C” by Robert Sedgewick.
4) “Data Structures Through C in Depth” by S.K. Srivastava and Deepali Srivastava.

Dr. Vasantha Rudramalla

LESSON-6
RECURSION

OBJECTIVES:

1. Understand the concept of recursion and its components, including base case, recursive case, and termination conditions.
2. Learn how recursion is implemented in C++ through real-world examples like factorial calculation, sum of natural numbers, and Fibonacci series.
3. Explore the concept of backtracking, its role in solving problems like n-Queens Puzzle, Tower of Hanoi, and Sudoku.
4. Gain insights into memory management during recursion using the call stack and learn how to optimize recursive functions.

STRUCTURE:

6.1 INTRODUCTION:

Recursion is a powerful problem-solving technique that is widely used in programming. In simple terms, recursion is a process where a function calls itself, either directly or indirectly, to solve a problem. Each recursive call breaks the problem down into smaller, more manageable subproblems until a condition (called the base case) is satisfied, and the recursion stops.

6.1.1 Definition of Recursion

Recursion is the process in which a function calls itself in order to divide the original problem into smaller subproblems, solving each sub problem individually until a simple case, known as the base case, is reached. Once the base case is met, the function returns a value, which is then used in previous calls to combine and ultimately solve the problem.

Conceptual Example

Think of recursion as solving a problem by dividing it into several simpler versions of the original problem. A good example is painting balls: if you paint them one by one, it might take a long time. But, if you break the task into smaller tasks-assigning a ball to each friend-the task becomes faster. In the same way, recursion speeds up problem-solving by breaking a larger task into simpler ones.

6.2 RECURSION IN C++

6.2.1 General Structure

A recursive function in C++ follows this structure:

```
return_type recursive_func(parameters) {
   // Base Case: The stopping condition
   // Recursive Case: Logic to solve smaller subproblems
   // Recursive Call: Calls the function with modified arguments
}
```

6.2.2 Components of Recursive Function

1. Return Type

The return type defines the type of value the recursive function will give back. This could be:

- **int** if the function returns an integer.
- **float** if the function returns a floating-point number.
- **void** if the function doesn't return anything (i.e., it just does a task).
- Or any **user-defined type** like string, char, or a custom class.

For example:

- If the function calculates and returns a number, it would have the return type int or float.
- If it doesn't return anything, like printing output, it would have void.

2. Base Case (Termination Condition)

The **base case** is the most important part of recursion. It's a **stopping point**-a condition where the function **does not call itself further**. This is crucial to avoid an **infinite loop** of function calls, which would eventually cause the program to crash (due to a **stack overflow**).

Without a base case, the function would keep calling itself indefinitely.

For example, in calculating the factorial, the base case is when the number becomes 1 because factorial of 1 is 1. So the function stops there.

Example Base Case:

```
if (n == 1) return 1; // Stops recursion when n equals 1
```

3. Recursive Case (Breaking the Problem)

The **recursive case** is where the function calls itself. In the recursive case, the problem is divided into smaller, simpler versions of itself. The function makes a new call with a modified argument, often a smaller or reduced version of the original problem.

This is where the recursion works, solving part of the problem, and then the function calls itself again with a simpler version of the same problem.

For example, to calculate the factorial of a number n, the recursive case would be:

```
return n * factorial(n - 1);  // Calls factorial with n-1
```

This keeps reducing n until it reaches the base case.

6.2.3 Example of Recursion: Factorial Calculation

Let’s take an example of calculating the factorial of a number using recursion:

The **factorial** of a number n is the product of all integers from 1 to n:

- factorial(5) = 5 * 4 * 3 * 2 * 1

```
// Recursive function to find factorial
int factorial(int n) {
    // Base case: If n is 1, stop recursion
    if (n == 1) {
        return 1;
    }

    // Recursive case: Call factorial with n-1
    return n * factorial(n - 1);
}


int main() {
   int number = 5;
   int result = factorial(number);  // Calling the recursive function
   cout << "Factorial of " << number << " is " << result;
   return 0;
}
```

How It Works:

1. **First call:** factorial(5) — This calls factorial(4) (recursive case).
2. **Second call:** factorial(4) — This calls factorial(3) (recursive case).
3. **Third call:** factorial(3) — This calls factorial(2) (recursive case).
4. **Fourth call:** factorial(2) — This calls factorial(1) (recursive case).
5. **Base case reached:** factorial(1) returns 1 (stops recursion).
6. Now, the function begins to **return values** from the stack:
   - factorial(2) returns 2 * 1 = 2
   - factorial(3) returns 3 * 2 = 6
   - factorial(4) returns 4 * 6 = 24
   - factorial(5) returns 5 * 24 = 120

So, factorial(5) returns 120, which is printed in the output.

To summarize:

- The **base case** is the condition that stops the recursion.
- The **recursive case** is where the function calls itself to solve smaller subproblems.
- The **recursive function** continues to call itself, breaking down the problem, until it reaches the base case, at which point it begins returning and solving the problem from bottom to top.

This process of recursion simplifies solving problems that have repetitive or self-similar subproblems, like factorials, tree traversal, and Fibonacci numbers.

6.2.4 Example: Sum of Natural Numbers

Consider the following program that calculates the sum of the first n natural numbers using recursion:

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

int main() {
    int n = 5;
    int sum = Sum(n);
    cout << "Sum = " << sum;
    return 0;
}
```

Explanation:

- **Recursive Function (Sum)**: This function calculates the sum of the first n natural numbers.
- **Base Case**: When n == 0, the function returns 0, stopping further recursion.
- **Recursive Case**: The function calls itself with n - 1 and adds n to the result.

Working of Recursion in C++

To understand recursion better, let's trace the flow of the program step by step. Suppose n = 5:

1. The function Sum(5) is called, which then calls Sum(4), then Sum(3), and so on until Sum(0).
2. The base case n == 0 is reached, which returns 0.
3. Each previous call adds its value to the result, and the recursion "unwinds":
    - o Sum(5) returns 5 + 4 + 3 + 2 + 1 + 0 = 15.

Thus, the final output is the sum of the first 5 natural numbers, which is 15.

6.2.5 Recursion Tree Diagram

The recursion tree for the Sum(5) function looks like this:

```
Sum(5)
  |
Sum(4)
  |
Sum(3)
  |
Sum(2)
  |
Sum(1)
  |
Sum(0)
```

6.3 MEMORY MANAGEMENT IN C++ RECURSION:

In C++, recursion plays a crucial role in breaking down complex problems into smaller subproblems. However, each recursive call requires memory management, as it relies on the **call stack** to store and manage data for the function calls. This makes understanding memory management an essential aspect of recursion.

When a recursive function is called, the following sequence occurs:

1. A **stack frame** (a block of memory) is created in the **call stack**.
2. This stack frame stores:
   - The **local variables** of the function.
   - The **parameters** passed to the function.
   - The **return address** (to know where to continue execution after the function ends).
3. The stack frame is placed **on top** of the existing stack frames.
4. The function executes. If it makes another recursive call, a new stack frame is created, and the process repeats.

Once the base case is reached, the recursion begins to **unwind**:

1. The function at the top of the stack completes its execution and returns its result.
2. Its stack frame is removed (or destroyed).
3. The control goes back to the previous function call (the next stack frame down).
4. This process continues until all recursive calls are resolved and the original call completes.

6.3.1 Call Stack Mechanism

The **call stack** is a special type of memory used for managing function calls. Every time a function is called (recursively or not), a new stack frame is added to the call stack. This stack frame includes:

- **Function arguments**: The values passed to the function.
- **Local variables**: Variables declared inside the function.
- **Return address**: The point in the program to return control after the function finishes.

When the function ends, its stack frame is removed from the stack, and the program control moves back to the calling function.

6.3.2 Example of Call Stack for Recursive Function

Let’s revisit the example of calculating the sum of the first n natural numbers using recursion:

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

When we call Sum(5), here's how the call stack evolves:

1. **First Call**: Sum(5)
   - A stack frame is created for Sum(5) with n = 5.
   - The function calls Sum(4).
2. **Second Call**: Sum(4)
   - A stack frame is created for Sum(4) with n = 4.
   - The function calls Sum(3).
3. **Third Call**: Sum(3)
   - A stack frame is created for Sum(3) with n = 6.
   - The function calls Sum(2).
4. **Fourth Call**: Sum(2)
   - A stack frame is created for Sum(2) with n = 2.
   - The function calls Sum(1).
5. **Fifth Call**: Sum(1)
   - A stack frame is created for Sum(1) with n = 1.
   - The function calls Sum(0).
6. **Sixth Call**: Sum(0)
   - A stack frame is created for Sum(0) with n = 0.
   - Since this is the base case, the function returns 0.

Unwinding the Stack

After reaching the base case, the recursion begins to unwind:

1. The stack frame for Sum(0) is destroyed, and the function returns 0 to Sum(1).
2. In Sum(1), the stack frame calculates 1 + 0 = 1 and returns 1 to Sum(2).
3. The stack frame for Sum(1) is destroyed.

4. In Sum(2), the stack frame calculates 2 + 1 = 3 and returns 3 to Sum(3).
5. The stack frame for Sum(2) is destroyed.
6. This process continues until Sum(5) calculates 5 + 10 = 15 and returns the final result.

6.4 TYPES OF RECURSION IN C++

1. **Direct Recursion:** The function calls itself directly in its body.
   - **Head Recursion:** The recursive call is at the start of the function.
   - **Tail Recursion:** The recursive call is the last statement in the function. Tail recursion is memory-efficient because the compiler can optimize it using **Tail Call Optimization** (TCO).
   - **Tree Recursion:** Multiple recursive calls are made within the function, resulting in a tree-like structure.
2. **Indirect Recursion:** In indirect recursion, a function calls another function, and that function calls the first function, creating a cycle of function calls.

6.4.1 Direct Recursion

Direct recursion occurs when a function calls itself explicitly. This is the most common type of recursion. The function continues to call itself until it reaches a base case, which stops the recursion.

Example:

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

6.4.2 Indirect Recursion

Indirect recursion happens when a function calls another function, which in turn calls the original function. This creates a cycle of function calls.

Example:

```
void functionA(int n) {
   if (n > 0) {
      functionB(n - 1); // Function A calls Function B
```

```
    }
}


void functionB(int n) {
    if (n > 0) {
        functionA(n - 1); // Function B calls Function A
    }
}
```

6.4.3 Infinite Recursion

Infinite recursion occurs when a recursive function does not have a proper base case, or the base case is not reached due to incorrect logic. This can lead to a **stack overflow**, where the program crashes because the call stack runs out of memory.

Example of Infinite Recursion:

```
void infiniteRecursion() {
    infiniteRecursion(); // No base case to stop the recursion
}
```

To avoid infinite recursion, always ensure that the function has a well-defined base case that is reachable.

6.5 PROBLEM SOLVING USING RECURSION:

Recursion is a powerful technique for solving a variety of problems. Below are some examples:

6.5.1 Largest Element in an Array

To find the largest element in an array using recursion:

Algorithm:

1. Compare the current element with the result of the recursive call on the rest of the array.
2. Base case: If the array has only one element, return that element.

Example:

```
int findLargest(int arr[], int n) {
    if (n == 1) {
        return arr[0]; // Base case
    }
    return max(arr[n - 1], findLargest(arr, n - 1)); // Recursive case
}
```

6.5.2 Print a Linked List in Reverse Order

To print a linked list in reverse order using recursion:

Algorithm:

1. Recursively traverse to the end of the list.
2. Print the data while unwinding the recursion.

Example:

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

void printReverse(Node* head) {
    if (head == nullptr) {
        return; // Base case
    }
    printReverse(head->next); // Recursive call
    cout << head->data << " "; // Print while unwinding
}
```

6.5.3 Fibonacci Number

To calculate Fibonacci numbers using recursion:

Algorithm:

1. Base case: Fib(0) = 0 and Fib(1) = 1.
2. Recursive case: Fib(n) = Fib(n-1) + Fib(n-2).

Example:

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

6.5.4 Tower of Hanoi

The Tower of Hanoi is a classic problem that involves moving disks from one rod to another, following specific rules:

1. Only one disk can be moved at a time.
2. A disk can only be placed on top of a larger disk.
3. All disks must be moved to the target rod.

Algorithm:

1. Move n-1 disks from the source rod to the auxiliary rod.
2. Move the largest disk to the target rod.
3. Move the n-1 disks from the auxiliary rod to the target rod.

Example:

```
void towerOfHanoi(int n, char source, char target, char auxiliary) {
    if (n == 1) {
        cout << "Move disk 1 from " << source << " to " << target << endl;
        return;
    }
    towerOfHanoi(n - 1, source, auxiliary, target);
    cout << "Move disk " << n << " from " << source << " to " << target << endl;
    towerOfHanoi(n - 1, auxiliary, target, source);
}
```

6.5.5 Converting a Number from Decimal to Binary

To convert a decimal number to binary using recursion:

Algorithm:

1. Divide the number by 2.
2. Recursively compute the binary representation of the quotient.
3. Print the remainder while unwinding the recursion.

Example:

```
void decimalToBinary(int n) {
    if (n == 0) {
        return; // Base case
    }
    decimalToBinary(n / 2); // Recursive call
    cout << n % 2; // Print remainder
}
```

6.6 BACKTRACKING:

Backtracking is a systematic method for exploring all possible configurations to solve a problem. It involves:

1. Choosing a solution step.
2. Testing if it leads to a valid solution.
3. If not, undoing the step (backtracking) and trying another option.

This technique is commonly used for constraint satisfaction problems like puzzles, combinatorial problems, and optimization tasks.

6.6.1 n-Queens Puzzle

The n-Queens Puzzle involves placing n queens on an n x n chessboard such that no two queens threaten each other. This means:

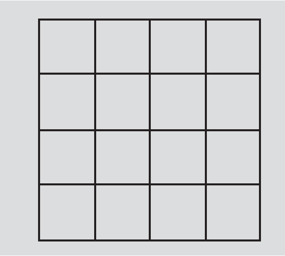
1. No two queens can be in the same row, column, or diagonal.

Algorithm:

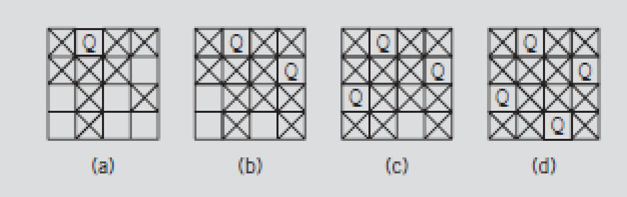
1. Place a queen in a row.
2. Check if the placement is safe.
3. Recursively place queens in the next row.
4. If no valid placement exists, backtrack and try a different column for the previous queen.

6.6.2 Backtracking and the 4-Queens Puzzle

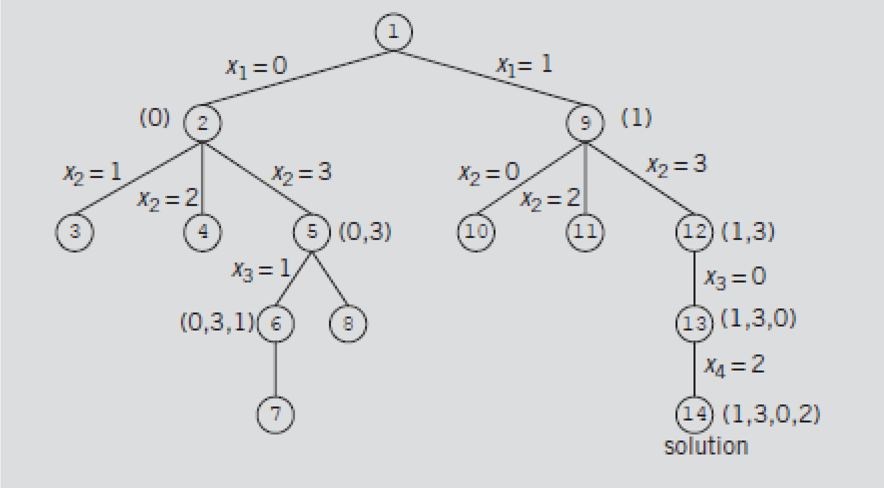
The 4-Queens Puzzle is a specific case of the n-Queens Puzzle for n = 4. The solution follows the same algorithm as the general n-Queens Puzzle, with additional constraints due to the smaller board size. Multiple solutions may exist, and the algorithm can find all solutions by not terminating after finding the first one.



Square board for the 4-queens puzzle



A Solution to the 4-Queens Puzzle



4-Queens Tree

6.6.3 8-Queens Puzzle

The **8-Queens Puzzle** is a classic combinatorial problem where the goal is to place eight queens on an 8x8 chessboard such that no two queens threaten each other. This means:

1. No two queens can be in the same row, column, or diagonal.
2. The challenge lies in finding all valid configurations for placing the queens.

Key Concepts:

- **Row and Column Check:** Ensures no queen is placed in the same row or column.
- **Diagonal Check:** Uses the properties of diagonal positions:
  - For a diagonal from the top-left to bottom-right, the difference (row - column) is constant.
  - For a diagonal from the top-right to bottom-left, the sum (row + column) is constant.

Algorithm: The problem is typically solved using **recursion** and **backtracking**:

1. Start with an empty board.
2. Place a queen in a row, one column at a time.
3. Check if the placement is valid (no conflicts).
4. If valid, proceed to place the next queen in the next row.
5. If no valid placement exists, backtrack and try a different column for the previous queen.

Applications: The 8-Queens Puzzle demonstrates the concepts of backtracking, recursion, and constraint satisfaction, making it a foundational problem in computer science and algorithm design.

To ensure no two queens are on the same diagonal, we need a mathematical approach:

1. **Left-to-Right Diagonal (Upper Left to Lower Right)**:
   - For squares along this diagonal, the difference between the row and column positions remains constant. For example:
     - Squares (0,4), (1,5), (2,6), and (3,7) all satisfy: rowPosition−columnPosition=−4
2. **Right-to-Left Diagonal (Upper Right to Lower Left)**:
   - For squares along this diagonal, the sum of the row and column positions remains constant. For example:
     - Squares (0,6), (1,5), (2,4), and (3,3) all satisfy: rowPosition+columnPosition=6

Using these properties, we can determine whether two queens are on the same diagonal:

- If the **difference** between row and column positions of two queens is the same, they lie on the same left-to-right diagonal.
- If the **sum** of the row and column positions of two queens is the same, they lie on the same right-to-left diagonal.

In general, two queens at positions (i,j)(i, j)(i,j) and (k,l)(k, l)(k,l) are on the same diagonal if:

$$|i-k| = |j-l|$$

This checks the absolute difference between their row and column indices.

Representation of the 8-Queens Puzzle

We represent the solution as an array, queensInRow, where:

- queensInRow[k] specifies the column position of the queen in row kkk.
- For example, queensInRow[0] = 3 means the queen in the first row is placed in the fourth column.

Placing the Queens

To place the queens, the algorithm proceeds as follows:

1. **Start with the First Row**:
    - Begin placing queens one by one in the rows of the chessboard.
2. **Check Valid Placement**:
    - Before placing a queen in column iii of row kkk, ensure:
        - No queen exists in column iii.
        - No queen exists on the same diagonals as square (k,i)(k, i)(k,i).
3. **Use the Function canPlaceQueen(k, i)**:
    - This function checks if a queen can be placed in column iii of row kkk. The implementation:

      ```
      bool canPlaceQueen(int k, int i) {
         for (int j = 0; j < k; j++) {
            if (queensInRow[j] == i // Queen exists in column i
               || abs(queensInRow[j] - i) == abs(j - k)) // Same diagonal
               return false;
         }
         return true;
      }
      ```

    - The loop iterates through all previously placed queens (in rows 0 to k−1k-1k−1):
        - It checks if any queen is already in the same column.
        - It checks if any queen is on the same diagonal using the absolute difference condition.
4. **Backtracking**:
    - If no valid column is found for the kkk-th queen, backtrack to the previous row, adjust the column position, and try again.
5. **Repeat Until All Queens are Placed**:
    - Continue placing queens row by row until all 8 queens are placed.

6.6.4 Recursion, Backtracking, and Sudoku

The Sudoku problem involves filling a 9x9 grid with numbers from 1 to 9 under specific constraints. The grid is divided into nine rows, nine columns, and nine smaller 3x3 grids. Each number must appear exactly once in each row, column, and 3x3 grid.

For example, consider a partially filled grid. The goal is to complete the grid by filling the empty slots while adhering to the constraints. To solve this, we can use a recursive backtracking algorithm.

Understanding the Grid Structure

- The Sudoku grid consists of nine smaller 3x3 grids separated by bold lines.
- The first 3x3 grid spans rows 1-3 and columns 1-3, the second spans rows 1-3 and columns 4-6, and so on.
- The constraints require that:
    - Each row contains the numbers 1 to 9, without duplicates.
    - Each column contains the numbers 1 to 9, without duplicates.
    - Each 3x3 grid contains the numbers 1 to 9, without duplicates.

Algorithm Explanation

The recursive backtracking algorithm operates as follows:

1. **Locate the First Empty Slot**:
    - Starting from the first row, scan for an empty cell. For example, in the partially filled grid, the first empty slot might be in the second row and second column.
2. **Try Numbers from 1 to 9**:
    - For the empty slot, attempt to place numbers between 1 and 9.
    - Before placing a number, check that it is valid:
        - The number must not already exist in the same row, column, or 3x3 grid.
3. **Proceed to the Next Empty Slot**:
    - If a valid number is found, place it in the empty slot and move to the next slot.
4. **Backtrack if Necessary**:
    - If no number can be placed in a slot, backtrack to the previous slot where a number was placed.
    - Change the number in the previous slot to the next possible value and continue the process.
5. **Repeat Until the Grid is Filled or No Solution is Found**:
    - Continue this process until all slots are filled. If no valid numbers can be placed at any point, the algorithm concludes that the Sudoku puzzle has no solution.

Example Walkthrough

- Starting with a partially filled grid, the first empty slot is found in row 2, column 2.
- A valid number, such as 5, is placed in this slot after confirming it does not violate the constraints.

- The algorithm then moves to the next empty slot, filling it similarly.
- If a slot cannot be filled, the algorithm backtracks, revisiting earlier slots to try alternate numbers.

Recursive Nature of the Algorithm

This approach uses recursion to explore all possible combinations:

- Each recursive call tries to place a number in an empty slot.
- The base case is reached when all slots are filled, indicating a solution has been found.
- If a solution is not feasible, the recursion unwinds (backtracks) to explore other possibilities.

This algorithm guarantees that all valid solutions to the Sudoku puzzle are found or confirms that no solution exists. Its systematic nature ensures correctness, but for complex grids, optimizations like constraint propagation can help improve efficiency.

6.7 KEY TERMS:

Recursion, Base Case, Recursive Case, Backtracking, Call Stack, Tail Recursion.

6.8 REVIEW QUESTIONS:

1) What are the components of a recursive function in C++, and why is a base case important?
2) How does the call stack manage memory during recursive function calls, and what happens when recursion reaches the base case?
3) Explain the difference between direct recursion and indirect recursion with examples?
4) Describe how backtracking is used in solving the n-Queens Puzzle and the Tower of Hanoi problem?
5) How can recursion be used to convert a decimal number to binary?

6.9 SUGGESTED READINGS:

1) “The C++ Programming Language” by Bjarne Stroustrup.
2) “Effective C++: 55 Specific Ways to Improve Your Programs and Designs” by Scott Meyers.
3) “Data Structures and Algorithm Analysis in C++” by Mark Allen Weiss.
4) “Introduction to Algorithms” by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein.

Dr. Vasantha Rudramalla

LESSON-7

SEARCH ALGORITHMS AND HASHING TECHNIQUES

OBJECTIVES:

The objectives of this lesson are to

1. Understand the concept and importance of search algorithms and hashing techniques in efficient data management.
2. Learn the types of search algorithms, including linear search and binary search, and their implementation.
3. Explore hashing techniques, hash tables, and the role of hash functions in fast data retrieval.
4. Gain insight into collision resolution techniques such as chaining and open addressing, and their practical applications.

STRUCTURE:

7.1 INTRODUCTION:

Search algorithms and hashing techniques are key tools in computer science for efficiently locating and retrieving data. Search algorithms, such as linear and binary search, help find specific elements in datasets, while advanced methods like depth-first search (DFS) and breadth-first search (BFS) are used in graph traversal. Hashing techniques use hash tables to map data to unique keys, enabling quick storage and retrieval with average constant time O(1)O(1)O(1). These methods are widely applied in database indexing, password verification, and caching, forming the foundation for efficient data management and optimized system performance.

7.2 TYPES OF SEARCH ALGORITHMS:

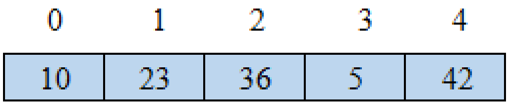
Search algorithms are essential computational techniques designed to locate specific data elements within a dataset efficiently. These algorithms can be broadly categorized based on their approach, data structure, and application domain. Linear search and binary search are fundamental methods, with linear search scanning sequentially and binary search leveraging sorted data for faster results.

7.2.1 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:



0	1	2	3	4
10	23	36	5	42

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.

7.2.2 Binary Search

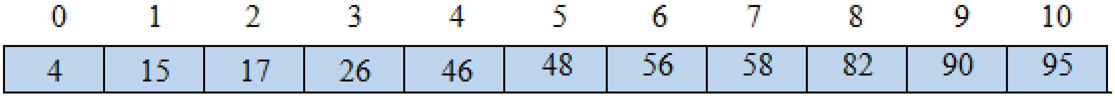
Binary search is an efficient algorithm that leverages ordered data to reduce the time required to find an item. Binary search repeatedly divides the search interval in half, determining which half contains the target item. This "divide and conquer" approach allows binary search to achieve logarithmic 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.

7.3 HASHING:

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.

7.3.1 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.

7.3.2 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.

7.4 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.

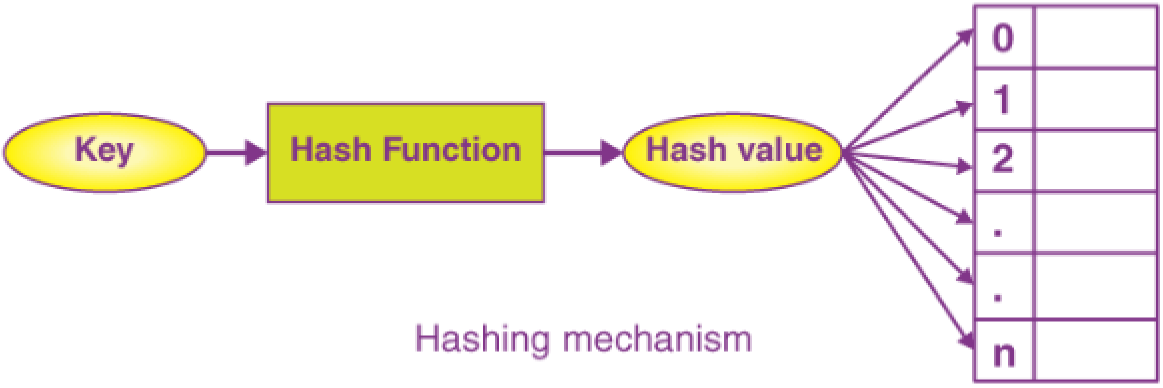


Fig. 7.1. Process of Hashing

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

7.4.1 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

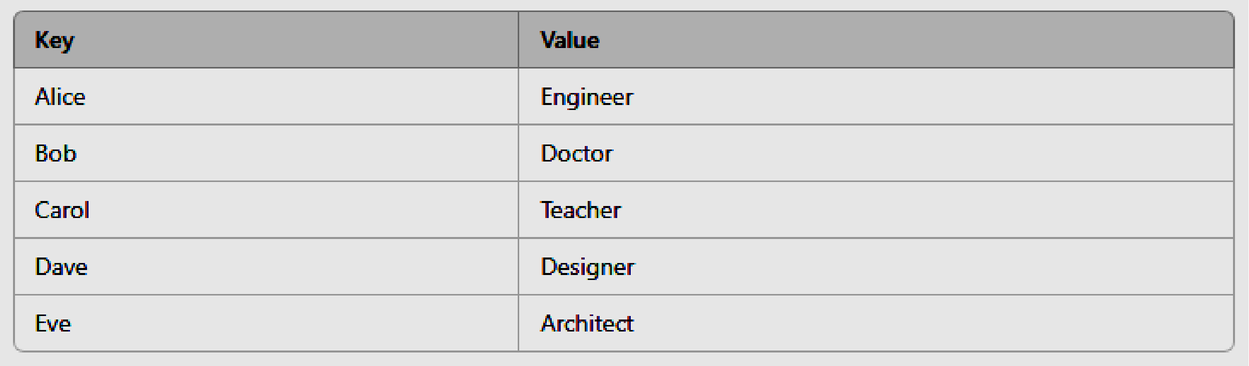
This means "Alice" would be stored at index 8 in the hash table.

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 7.1. 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

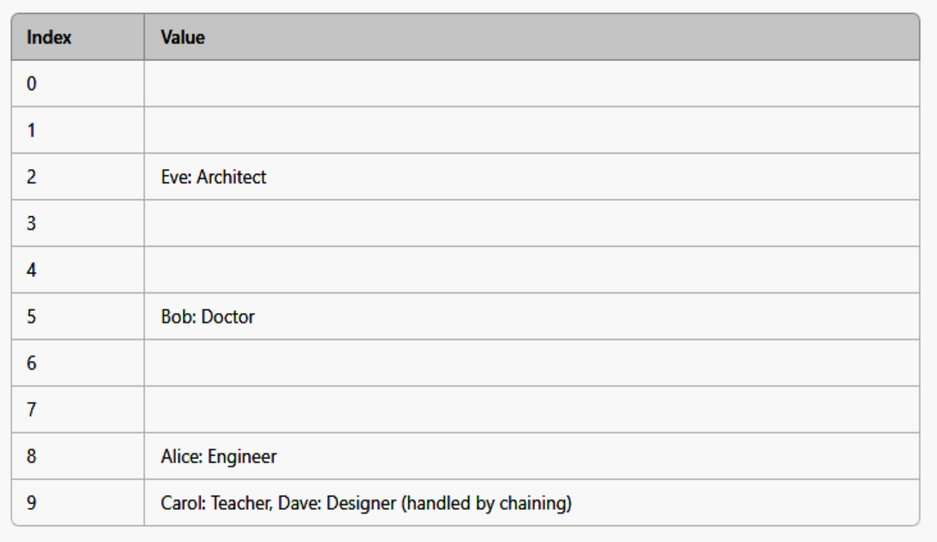
Here’s a step-by-step illustration of how each entry would be added:

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 7.2 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.

7.4.2 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:

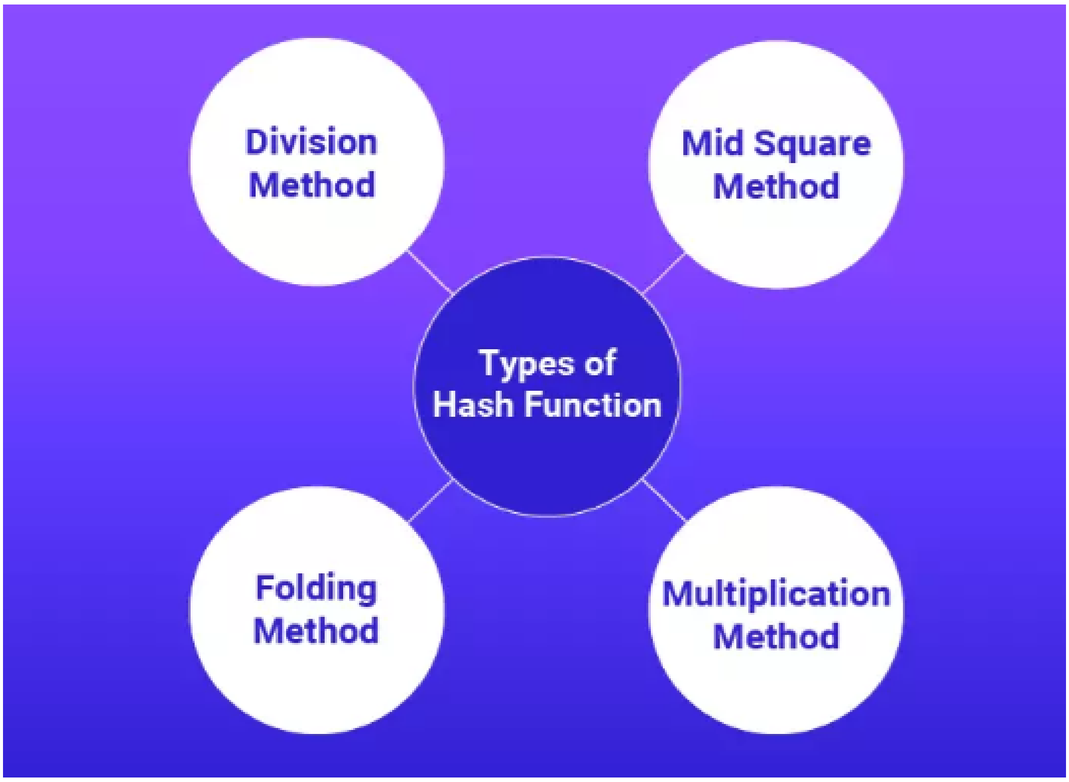


Fig. 7.2 Hash Functions

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 mod 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)=25mod 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.

  Example: 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.

7.4.3 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.

7.4.4 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.

7.5 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.

7.6 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.

7.7 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.

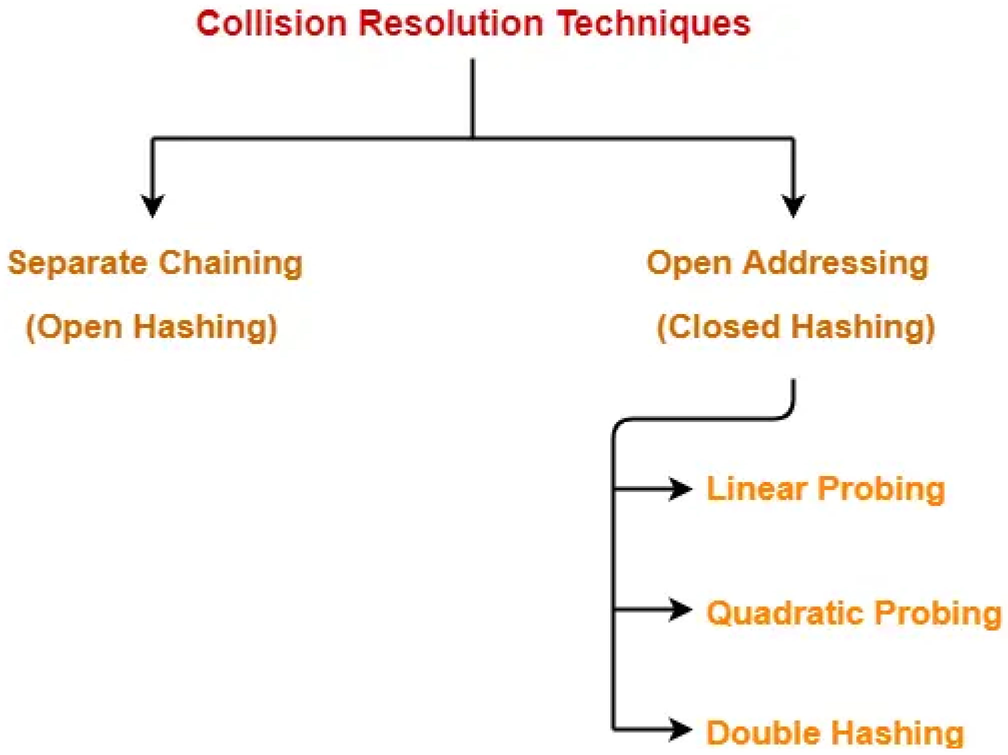


Fig. 7.3 Collision Resolution Techniques

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.

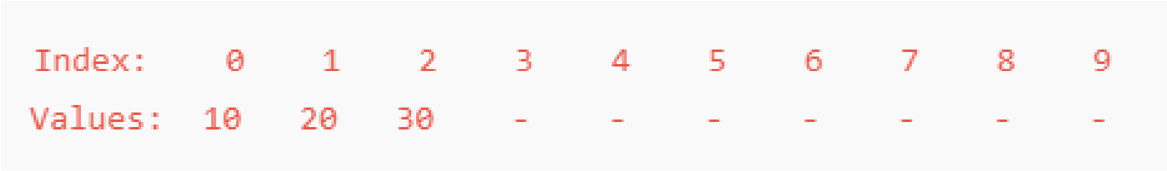
7.7.1 Open Addressing

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…)

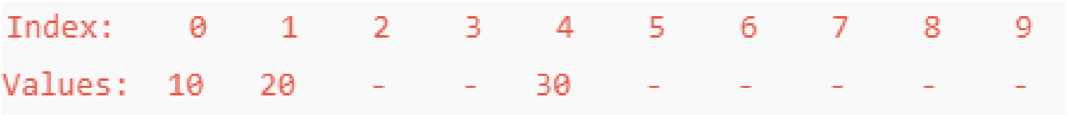
- **How It Works**:

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    -    -    -    -    -
```

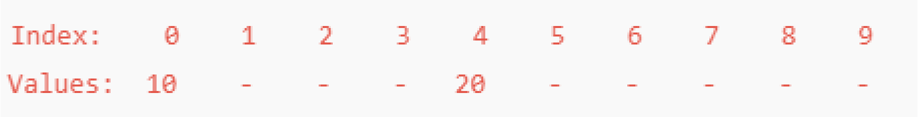
- **Pros**: Reduces primary clustering.
- **Cons**: May still experience secondary clustering, where certain sequences of probes lead to repeatedly used positions.

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.

7.7.2 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:**

- o Each index in the hash table points to a linked list or another dynamic data structure.
- o When a collision occurs, the new entry is simply appended to the linked list at the index.
- o For retrieval, the algorithm searches the linked list at the hashed index to find the correct entry.

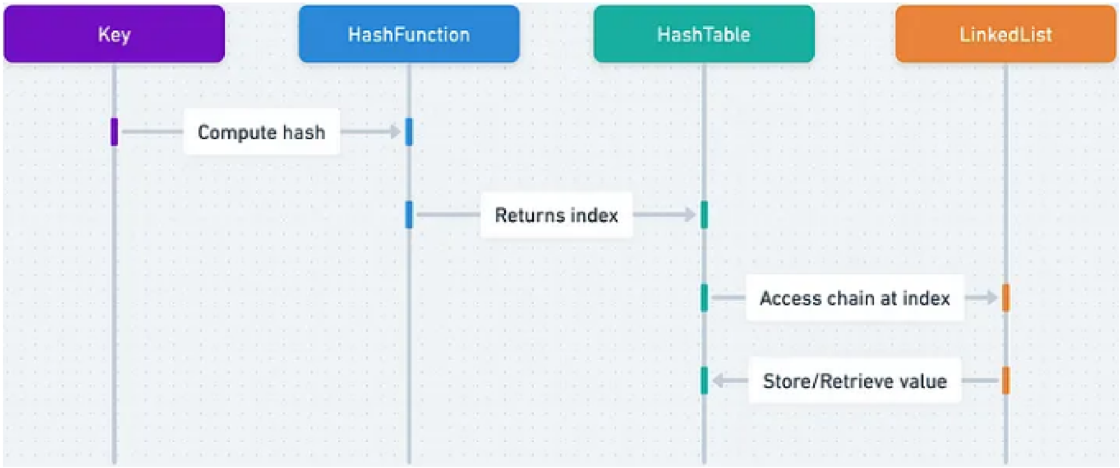


Fig. 7.4 Chaining using Linked List

❖ **Example of Chaining**

- o Suppose we insert keys 15, 25, and 35 into a hash table of size 10 with a hash function that computes key % 10.
- o All three keys hash to index 5, creating a collision.
- o Using chaining, all three entries are stored in a linked list at index 5.

❖ **Types of Chaining Structures**

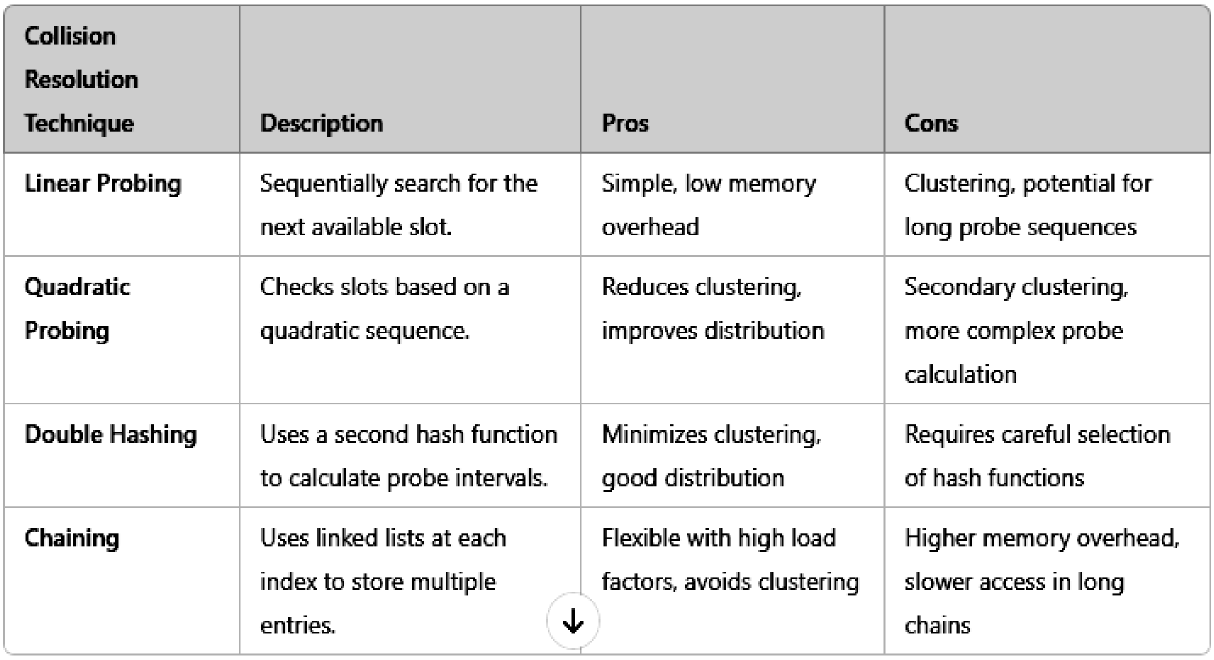
1. **Linked List**: The simplest form, where each entry at a given index points to the next in a singly or doubly linked list.
2. **Binary Search Tree (BST)**: For faster searching within a chain, each index could use a BST rather than a list, allowing O(log n)O(\log n)O(logn) search time within chains.
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 7.3. 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.

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**:
  - o **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.
  - o **Open Addressing**: Double hashing could be used to spread out books with similar ISBNs across different slots in the table.

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.

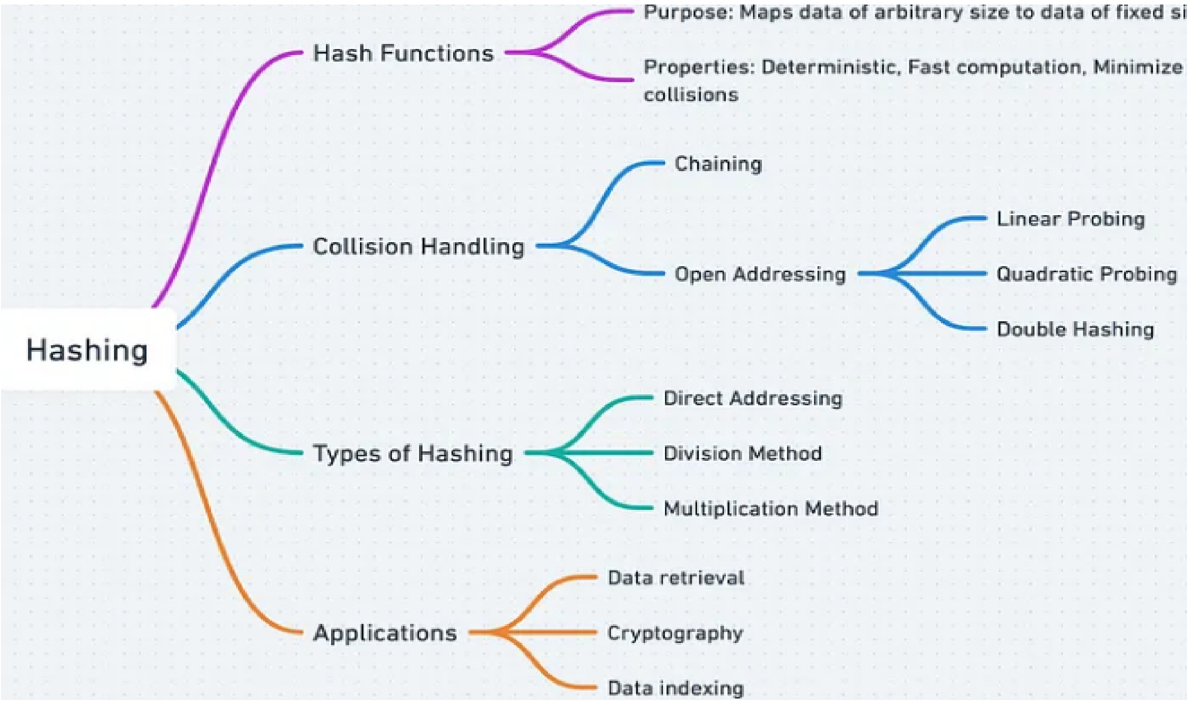


Fig. 7.5. Flow Diagram of Hashing

7.8 KEY TERMS:

Linear Search, Binary Search, Hashing, Hash Function, Hash Table, Collisions, Open Addressing, Chaining, Linear Probing, Quadratic Probing, Double Hashing, Dynamic Hashing.

7.9 REVIEW QUESTIONS:

1) Explain Search algorithms.
2) What is hashing, and how does it help in efficient data management?
3) Explain the difference between static and dynamic hashing?
4) What are collision resolution techniques, and how does chaining differ from open addressing?

5) How does the load factor influence the performance of hash tables?

6) Compare directory-based and directory-less dynamic hashing. What are their advantages and challenges?

7.10 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.

Dr. Vasantha Rudramalla

LESSON-8
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:

8.1 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.

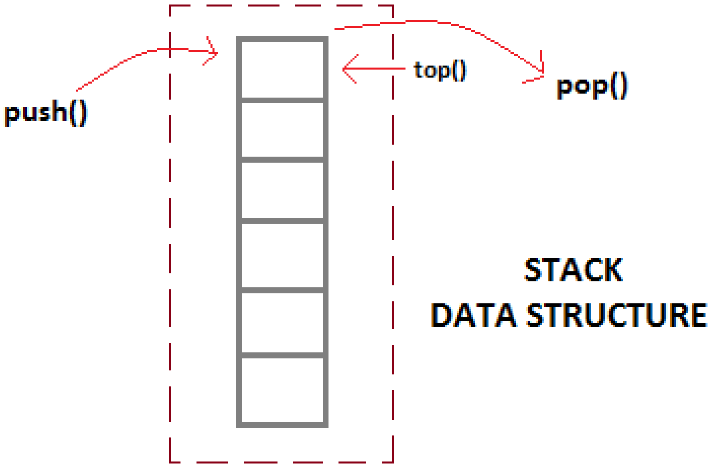


Fig. 8.1 Represenation of Stack

8.2 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.

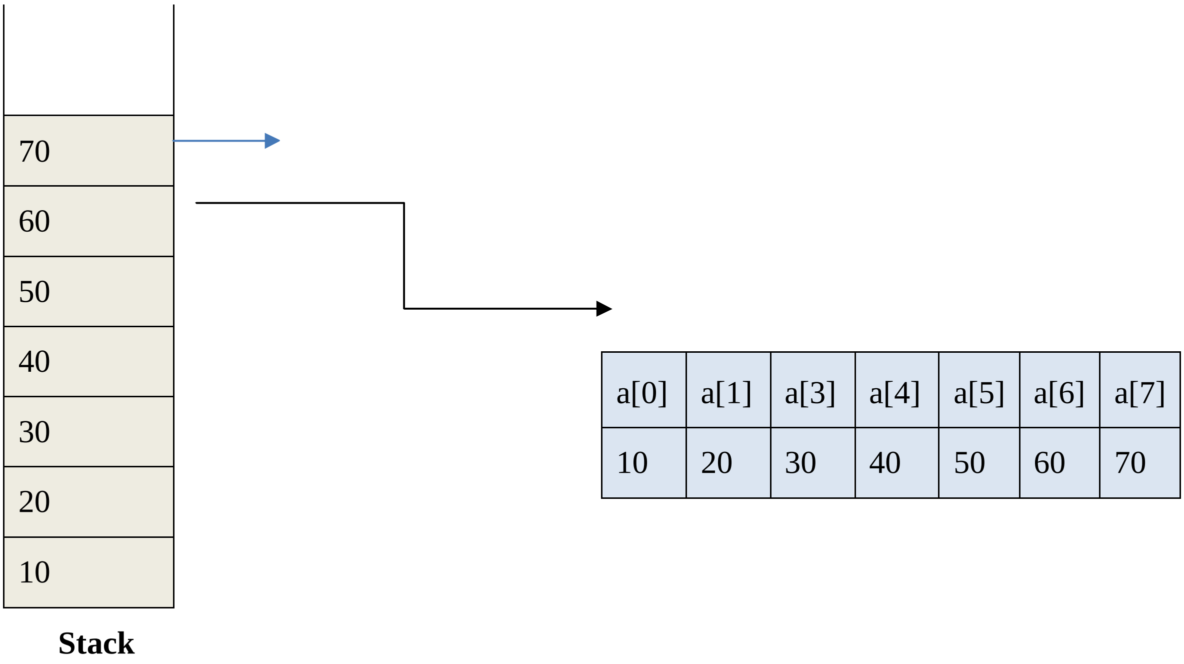
8.3 OPERATIONS ON STACKS:

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.

8.4 IMPLEMENTATION OF STACK OPERATIONS:

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.



a[0]	a[1]	a[3]	a[4]	a[5]	a[6]	a[7]
10	20	30	40	50	60	70

Fig. 8.2 Array Represenation of Stack

A stack is a data structure that can also be represented using a linked list. In this representation, each element of the stack is stored in a node of the linked list, where each node contains two parts: the data (value of the element) and a reference (or pointer) to the next node. The top of the stack corresponds to the head of the linked list. Unlike arrays, the linked list representation of a stack does not have a fixed size, allowing it to dynamically grow or shrink as elements are added or removed. Stack operations such as push (adding an element) and pop (removing an element) are performed by manipulating the head of the linked list. When a new element is pushed onto the stack, a new node is created, and its next pointer is set to the current top node, with the top pointer updated to the new node. Similarly, when an element is popped, the top pointer is updated to the next node, and the removed node is deallocated. This dynamic approach provides flexibility and efficient memory management while ensuring the stack operates on the Last In, First Out (LIFO) principle.

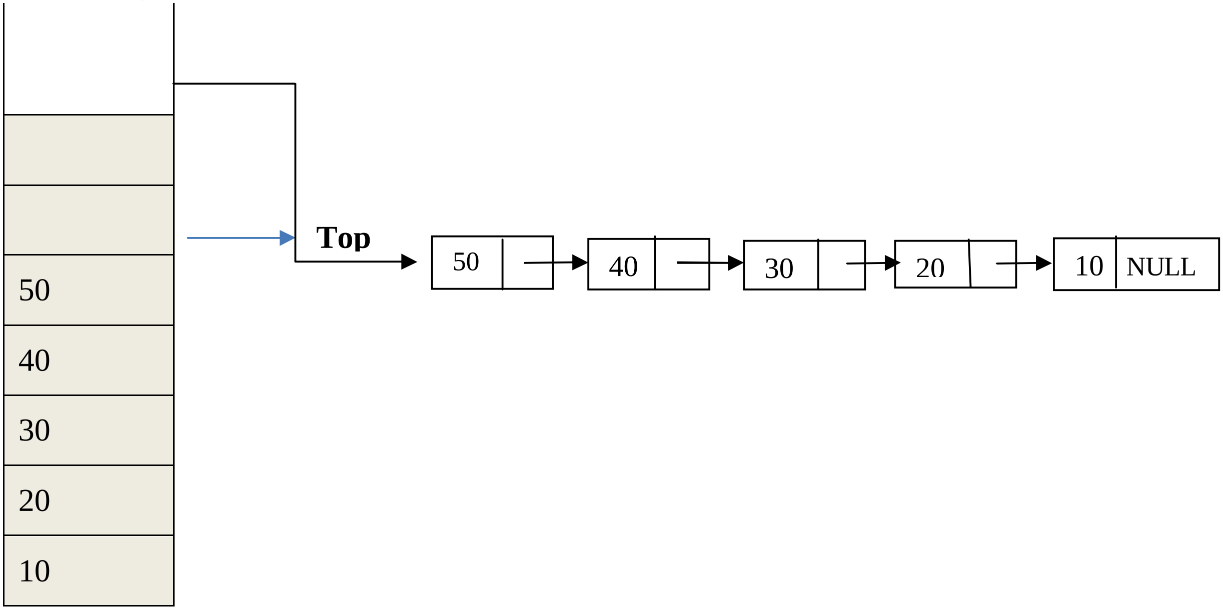


Fig. 8.3 Linked List Representation of Stack

8.4.1 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.

Using array

```
  void push(int a[ ], int &top, int value, int n)
  {
   if (top == n - 1) { // Check if the stack is full
      cout << "Stack Overflow" << endl;
   } else {
      top++;              // Move top to the next position
      a[top] = value;    // Place value at the new top position
      cout << "Pushed " << value << " onto stack" << endl;
   } }
```

The push function adds an element to the stack.

- First, it checks if the stack is full by comparing top with n - 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.

Using Linked List

```
void push(Node* &top, int value)
{
   Node* newNode = new Node(value); // Create a new node
   newNode -> next = top;           // Link the new node to the current top
   top = newNode;                  // Update top to the new node
   cout << "Pushed " << value << " onto stack" << endl;
}
```

The push function adds a new element to the top of a stack implemented using a linked list. It takes a reference to the top pointer (representing the current top of the stack) and the value to be pushed as arguments. A new node is dynamically created using the new keyword, which allocates memory for the node and initializes it with the given value. The next pointer of the new node is set to point to the current top, effectively linking it to the existing stack. Finally, the top pointer is updated to reference the new node, making it the new top of the stack. This ensures that the stack maintains its Last-In, First-Out (LIFO) order, where the most recently added element is always at the top.

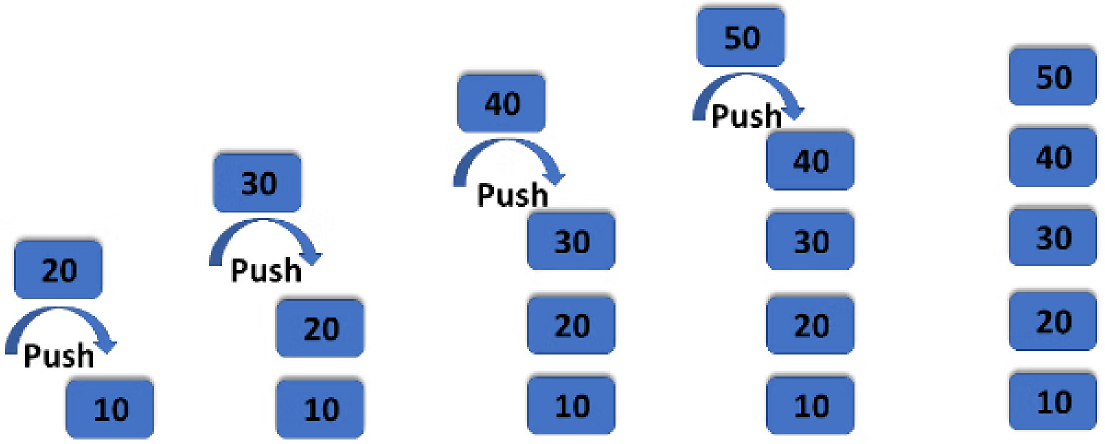


Fig. 8.4 Push Operation

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.

8.4.2 Pop Operation

Using array

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
      cout << "Stack Underflow" << endl;
      return -1; // Return an error value
   } else {
      int value = stack[top]; // Retrieve the top element
      top--;              // Decrement the top pointer
      cout << "Popped " << value << " from stack" << endl;
      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.

Using Linked List

```
int pop(Node* &top) {
   if (top == NULL) { // Check if the stack is empty
      cout << "Stack Underflow" << endl;
      return -1; // Return an error value
   } else {
      Node* temp = top;     // Temporarily store the top node
      int value = temp->data; // Retrieve the value from the top node
```

```
        top = top->next;      // Update the top pointer to the next node
        free(temp);           // Free the memory of the popped node
        cout << "Popped " << value << " from stack" << endl;
        return value;
    } }
```

The pop function removes the top element from a stack implemented using a linked list. It first checks if the stack is empty by verifying if the top pointer is NULL. If the stack is empty, it prints "Stack Underflow" and returns -1 to indicate an error. Otherwise, it temporarily stores the current top node in a pointer temp, retrieves the value from this node, and updates the top pointer to the next node in the stack. The memory allocated for the removed node is then deallocated using free(). Finally, the function prints the value that was popped and returns it. This ensures proper memory management while maintaining the Last-In, First-Out (LIFO) principle.

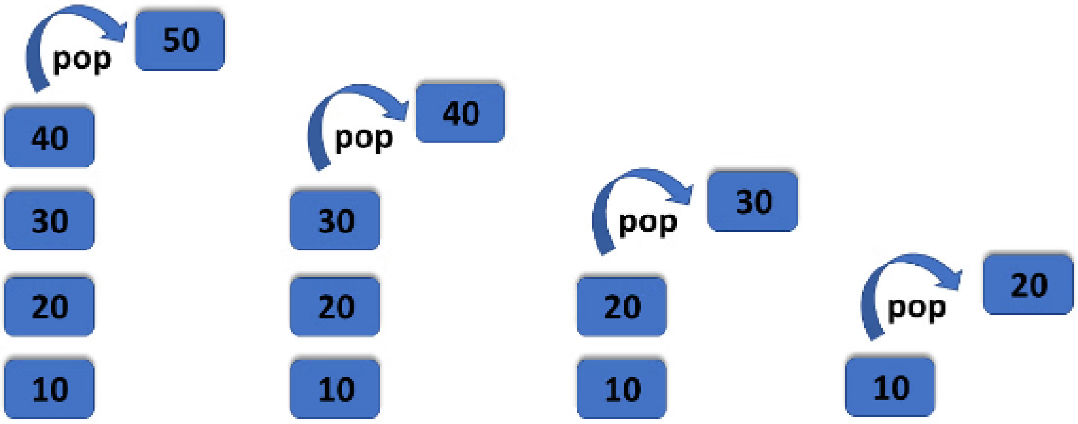


Fig. 8.5 Pop Operation

Fig. 8.5 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.

8.4.3 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.

Using Array:

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

The peek function retrieves the top element of a stack implemented using an array without modifying the stack. It takes the stack array and the top index as parameters. The function first checks if the stack is empty by verifying if top equals -1. If the stack is empty, it prints a message ("Stack is empty") and returns -1 to indicate an error. If the stack is not empty, it accesses the element at the top index of the array, prints its value, and returns it. This operation is non-destructive, meaning the stack remains unchanged after the function is executed, allowing subsequent operations to proceed without affecting the stack's state. This function adheres to the Last-In, First-Out (LIFO) principle of stack operations, providing a convenient way to inspect the top element.

Using Linked List

```
int peek(Node* top) {
   if (top == NULL) { // Check if the stack is empty
      cout << "Stack is empty" << endl;
      return -1; // Return an error value
   } else {
      cout << "Top element is " << top->data << endl;
      return top->data; // Return the value of the top element
   } }
```

The peek function retrieves the top element of a stack implemented using a linked list without removing it. It takes the top pointer, which points to the top node of the stack, as a parameter.

If the top pointer is NULL, it means the stack is empty, so the function prints "Stack is empty" and returns -1 to indicate an error. If the stack is not empty, it accesses the data field of the top node, prints its value, and returns it. This allows the user to view the top element of the stack while keeping the stack unchanged.

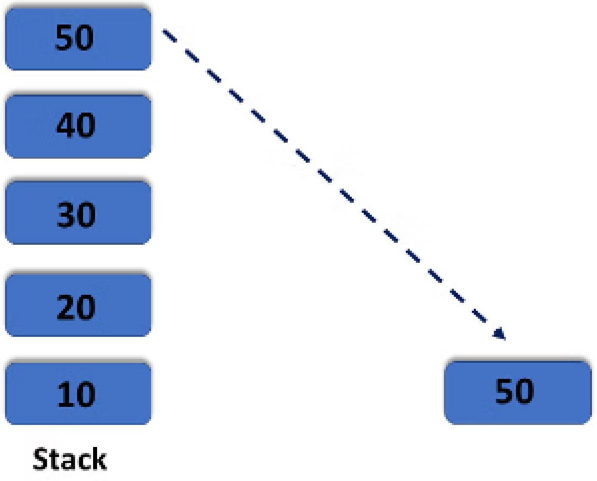


Fig. 8.6 Peek Operation

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.

8.4.4 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.

Using Array:

```
bool isEmpty(int top) {
   return top == -1; // Returns true if stack is empty.
}
```

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.

Using Linked List:

```
bool isEmpty(Node* top) {
   return top == NULL; // Returns true if stack is empty.
}
```

8.4.5 isFull Operation

Using array

```
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 array-based 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.

Using Linked List

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

- The isFull function checks if the linked list-based stack has reached its defined max_size.
- It returns 1 (true) if current_size equals max_size, meaning the stack is full.
- Otherwise, it returns 0 (false), meaning there's still room for more elements.
- If there's no defined max_size, the concept of isFull does not apply to linked list stacks.

8.4.6 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.

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.

Using array

```
void traverse(int stack[], int top)
{     if (top == -1)
    {  cout << "Stack is empty" << endl; // Check if the stack is empty }
   else
   {
      cout << "Stack elements: ";
      for (int i = top; i >= 0; i--)
      {  // Print elements from top to bottom
         cout << stack[i] << " "; }
    }
 }
```

The traverse function is designed to display the elements of a stack. It first checks if the stack is empty by seeing if top is -1, which indicates there are no elements in the stack. If the stack is empty, it prints a message saying "Stack is empty." Otherwise, it starts from the topmost element (stack[top]) and iterates down to the bottom of the stack (stack[0]), printing each element along the way. This ensures the stack's contents are displayed from top to bottom, reflecting the order in which elements would be popped.

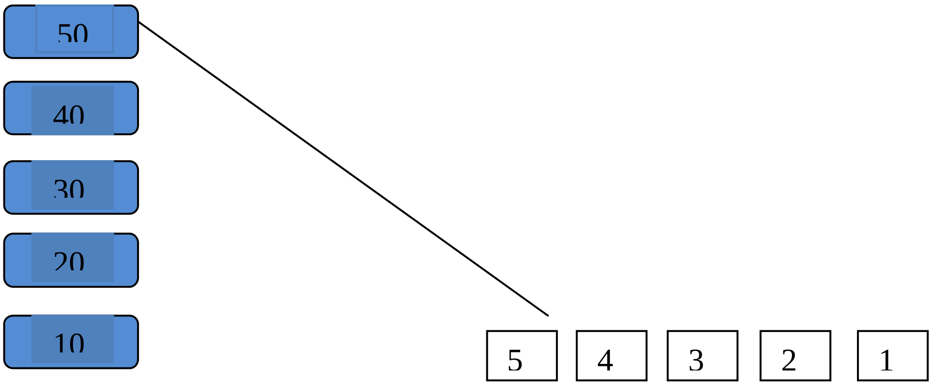


Fig. 8.7 Stack Traversal

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.

Using Linked List

```
void traverse(Node* top)
{
   if (top == NULL)
   {
      // Check if the stack is empty
      cout << "Stack is empty" << endl;
   }
   else
   {
      cout << "Stack elements: ";
      Node* temp = top;
      // Traverse the linked list from top to bottom
      while (temp != NULL)
      {
         cout << temp->data << " ";
         temp = temp->next;
      }
      cout << endl; // Add a newline for better formatting
   }
}
```

The traverse function is designed to display the elements of a stack implemented using a linked list. It takes a pointer top, which represents the top of the stack, as its parameter. The function first checks if top is NULL, indicating that the stack is empty; if so, it prints "Stack is empty." Otherwise, it initializes a temporary pointer temp to the top and iterates through the linked list using a while loop. During each iteration, it prints the data of the current node and moves the pointer to the next node in the list. This process continues until the pointer reaches the end of the list (NULL). After printing all elements, the function adds a newline for better formatting, ensuring that the stack elements are displayed clearly in the order from top to 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.

8.4.7 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.

Using array

```
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
         cout << "Found " << value << " at position " << (top - i) << " from top" << endl;
         return top - i; // Return position from top
      }    }
   cout << "Element " << value << " not found" << endl;
   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.

Using Linked List

```
int search(Node* top, int value)
{
   Node* temp = top;
   int position = 0; // Position from the top
   while (temp != NULL)
   {
      if (temp->data == value)
      {
         // Check if element matches
         cout << "Found " << value << " at position " << position << " from top" << endl;
         return position; // Return position from top
      }
      temp = temp->next;
      position++;
   }
   cout << "Element " << value << " not found" << endl;
   return -1; // Return -1 if not found
}
```

Steps of a Search Function:

- A temporary pointer (temp) is initialized to the top of the stack.
- The function iterates through the linked list, checking if the data of each node matches the given value.
- If a match is found, it prints the position relative to the top and returns it.
- If the loop completes without finding the value, it prints a "not found" message and returns -1.

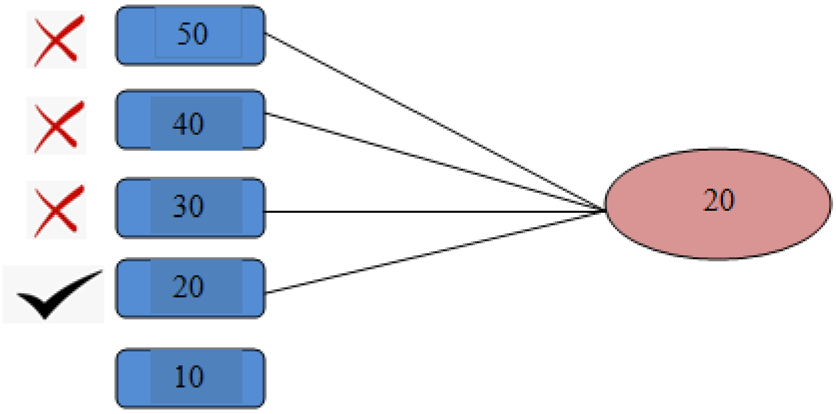


Fig. 8.8 Search Operation

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.

8.5 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 demonstrating their usage:

```
int stack[MAX_SIZE];

int top = -1;


// Push operation

void push(int value) {

    if (top == MAX_SIZE - 1) {

        cout << "Stack Overflow" << endl;

    } else {
        stack[++top] = value;
        cout << "Pushed " << value << " onto stack" << endl;
    }
}

// Pop operation
int pop() {
    if (top == -1) {
        cout << "Stack Underflow" << endl;
        return -1; // Indicates error
    } else {
        return stack[top--];
    }
}

// Peek operation
int peek() {
    if (top == -1) {
        cout << "Stack is empty" << endl;
        return -1; // Indicates error
    } else {
        return stack[top];
    }
}

// Main program
int main() {
    push(10);  // Add 10 to the stack
    push(20);  // Add 20 to the stack
    cout << "Top element is " << peek() << endl; // 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 program demonstrates basic stack operations while handling overflow and underflow conditions.

8.6 APPLICATIONS OF STACKS:

8.6.1 Evaluating Postfix Expressions

A **postfix expression** (Reverse Polish Notation) is a mathematical notation where operators follow their operands, e.g., 3 4 + 5 * instead of (3 + 4) * 5. Computers evaluate it using a stack by pushing operands, applying operators to the top two values, and pushing the result back until a single result remains.**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.

8.6.2 Recursion

Recursion is a technique where a function calls itself to solve smaller subproblems. Internally, recursion uses a **call stack** to keep track of function calls. Each recursive call pushes a new frame onto the stack, and when the base case is reached, the stack unwinds.

Applications of Recursion with Stacks:

1. **Expression Evaluation (e.g., Postfix, Prefix)**:
   - Recursive calls can replace explicit stack usage for parsing and evaluating expressions.
   - Example: Evaluate a postfix expression recursively by processing operators and operands.
2. **Tower of Hanoi**:
   - Solves the problem of moving disks between rods using recursion.
   - Each recursive call represents a move, tracked by the call stack.
3. **Tree Traversals**:
   - Recursion simplifies operations like in-order, pre-order, and post-order traversal of binary trees.

- o Each recursive call represents visiting a node, and the stack handles backtracking.

4. **Backtracking Problems**:
   - o Examples: N-Queens, Sudoku solving.
   - o Recursion and the stack track decisions and backtrack when no solution is found.

8.7 KEY TERMS:

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

8.8 REVIEW 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?

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

Dr. Vasantha Rudramalla

LESSON-9
QUEUES

OBJECTIVES:

The objectives of this lesson are to

1. Understand the concept and functionality of queues as a data structure.
2. Explore different types of queues, including linear, circular, priority and deque.
3. Implement queue operations using arrays and linked lists, such as enqueue, dequeue, and traversal.
4. Analyze real-world applications of queues in simulations, task scheduling, BFS, and data buffering.

STRUCTURE:

9.1 INTRODUCTION:

A queue is a linear data structure that follows the First In, First Out (FIFO) principle, meaning the element inserted first is removed first. It is similar to a real-world queue, such as a line at a ticket counter. Queues are widely used in various applications like task scheduling, managing resources in operating systems, and data buffering in communication systems. Basic operations include enqueue (insertion) and dequeue (removal), ensuring orderly processing of elements. Variants like circular queues, priority queues, and double-ended queues (deque) extend its functionality for specific use cases.

9.2 QUEUES:

Queues can be implemented using either arrays or linked lists. An array-based queue uses a fixed-size array, 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 them suitable for applications where the order of processing is crucial.

9.2.1 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.

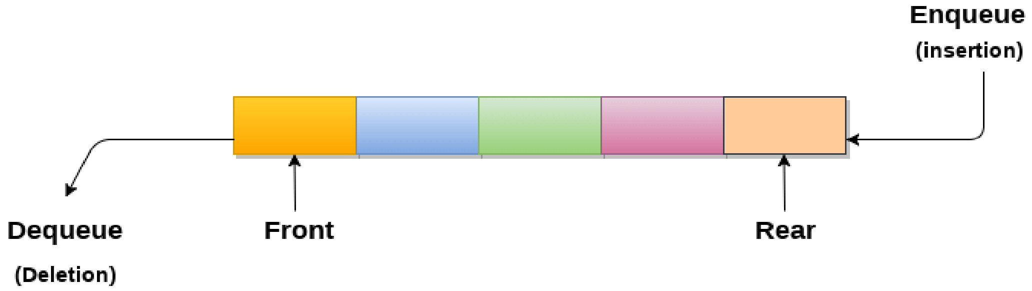


Fig. 9.1 Representation of Queue

9.2.2 Features of Queue

Queues are specialized data structures that follow the **First-In, First-Out (FIFO)** principle. 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 **AddQueue** operation adds new elements to the queue, while the **DeleteQueue** 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).

9.3 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. It is typically implemented using arrays. However, once full, it does not reuse empty spaces created by removed elements.

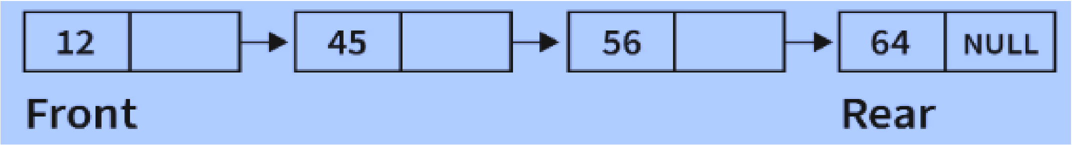


Fig. 9.2 Linear Queue

2. **Circular Queue:** A more efficient queue that reuses empty spaces by wrapping around to the beginning of the array, making it suitable for scenarios that require dynamic resizing.

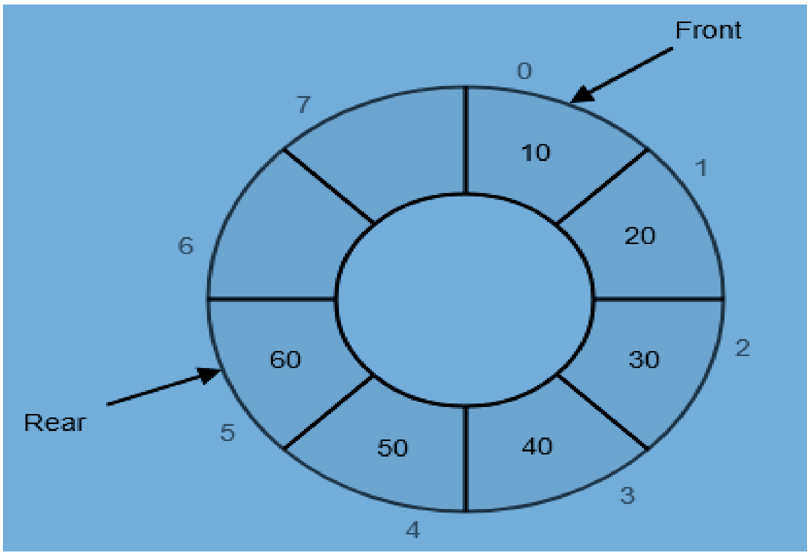


Fig. 9.3 Circular Queue

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, often implemented with linked lists.

9.4 QUEUE OPERATIONS:

Queues are widely used in various applications such as task scheduling, data streaming, and BFS algorithms. The primary operations performed on queues are:

1. **AddQueue:** Adds an element to the rear of the queue. If the queue is full, it results in a queue overflow condition.
2. **DeleteQueue:** Removes the element from the front of the queue. If the queue is empty, it results in a queue underflow condition.
3. **Front:** Retrieves the element at the front of the queue without removing it. This operation allows inspection of the front value without modifying the queue.
4. **isEmpty:** Checks whether the queue is empty. This is useful for preventing underflow errors before a deleteQueue operation.
5. **isFull:** Checks whether the queue is full, mainly in array-based queues, to prevent overflow errors during an addQueue operation.
6. **Traversal:** Displays all elements in the queue from front to rear without altering the queue. This operation helps in examining the queue's contents.
7. **Search:** Searches for a specific element within the queue and returns its position relative to the front, or an indication if it's not present.

9.5 IMPLEMENTATION OF QUEUE OPERATIONS:

Queues can be implemented using arrays or linked lists. In both cases, the **AddQueue** operation adds elements at the rear, and the **DeleteQueue** operation removes elements from the front.

Array-Based Queue

In an **array-based queue**, two pointers, **front** and **rear**, are used to track the position of the first and last elements, respectively.

- **Array:** A fixed-size array is used to store the elements of the queue. The maximum size of this array is pre-defined.
- **Front Pointer:** This pointer represents the front of the queue, where elements are deleteQueued. Initially, **front = -1**.
- **Rear Pointer:** This pointer represents the rear of the queue, where new elements are addQueued. Initially, **rear = -1**.

The queue operations modify these pointers to manage elements in the queue. The addQueue and deleteQueue operations increment or decrement these pointers to add or remove elements, respectively.

Linked List-Based Queue

In a **linked-list queue**, each element is stored as a node containing the data and a pointer to the next node. The **front** and **rear** pointers keep track of the first and last nodes in the queue, respectively.

- **Node:** Each element in the queue is stored as a node with two parts: the data and a pointer to the next node.
- **Front Pointer:** This pointer points to the first node in the queue.
- **Rear Pointer:** This pointer points to the last node in the queue.

In a linked-list-based queue, the **AddQueue** operation involves adding a new node at the end (rear) of the queue, and the **DeleteQueue** operation involves removing the node at the front.

9.5.1. AddQueue (Enqueue) Operation:

Array Implementation:

The AddQueue operation is used to add an element to the rear of the queue. In an array-based implementation, this operation first checks if there is space available in the queue (i.e., if it is full). If the queue is not full, the element is inserted at the position pointed to by queueRear, and then queueRear is incremented.

```
if (rear == MAX - 1) {
   cout << "Queue is full!" << endl;
} else {
   if (front == -1) front = 0;  // Queue is empty, initialize front
   rear++;
   queue[rear] = value;
}
```

Explanation:

- **Full Check**: If rear reaches MAX-1, it indicates that the queue has no more space.
- **Adding Element**: If the queue is not full, we add the element at queue[rear], then increment rear to point to the next available position.

Linked List Implementation:

In a linked list-based implementation, AddQueue involves creating a new node and linking it to the rear of the queue. The new node is assigned as the new rear of the queue.

```
Node* newNode = new Node(value);
if (rear == nullptr) {
   front = rear = newNode;  // First element
} else {
   rear->next = newNode;
   rear = newNode;
}
```

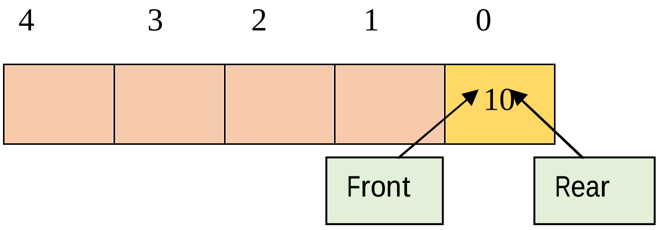
Explanation:

- **Creating a New Node**: A new node is created and linked to the current rear.
- **Empty Queue Check**: If the queue is empty, both front and rear are set to the new node.

Example:

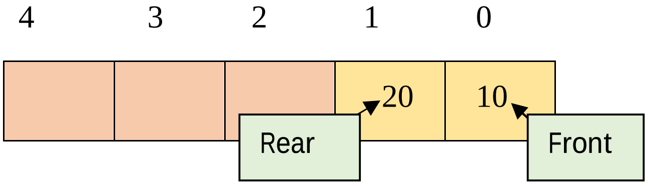
Let's say we have an array A of size 5 and we want to insert elements 10, 20, 30, 40, 50 in sequence. For this implementation array of size 5 is taken.

Visual Representation of the above example is

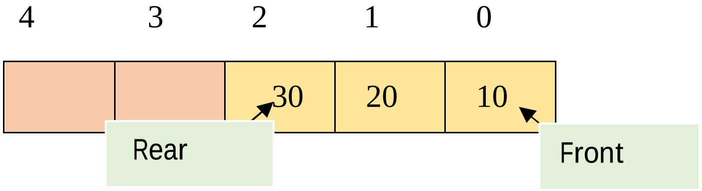


Initially, Front = Rear = -1. Before inserting the first element front and rear values should be initialised to 0.

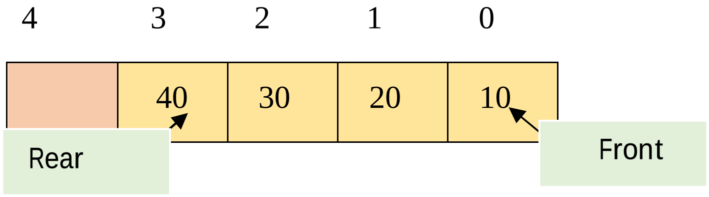
Front = Rear = 0, A[Rear] = 10



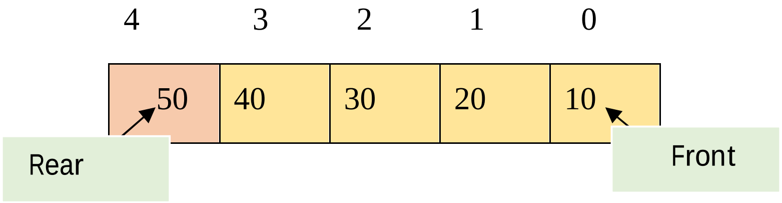
Front = 0 , Rear = 1, A[Rear] = 20



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.

9.5.2 DeleteQueue (Dequeue) Operation:

Array Implementation:

The DeleteQueue operation removes the element at the front of the queue. We retrieve the element at queueFront, then increment the queueFront pointer. If queueFront exceeds queueRear, we reset both pointers to -1, indicating the queue is empty.

```
if (front == -1) {
   cout << "Queue is empty!" << endl;
} else {
   front++;
   if (front > rear) front = rear = -1;  // Reset queue if empty
}
```

Explanation:

- **Empty Queue Check**: If front is -1, the queue is empty, and no deletion can occur.
- **Decreasing Front**: After deletion, we move front to the next position.

Linked List Implementation:

In the linked list-based queue, DeleteQueue involves removing the node pointed to by front. After that, front is updated to the next node. If the queue becomes empty, both front and rear are set to nullptr.

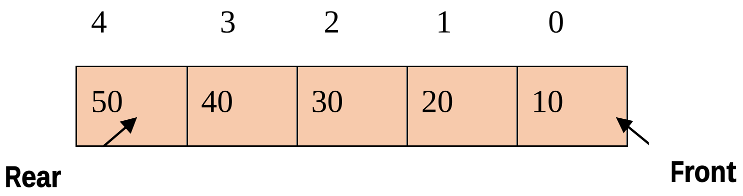
```
if (front == nullptr) {
    cout << "Queue is empty!" << endl;
} else {
    Node* temp = front;
    front = front->next;
    if (front == nullptr) rear = nullptr;  // Queue is empty
    delete temp;
}
```

Explanation:

- **Empty Queue Check**: If front is nullptr, the queue is empty, and no deletion can occur.
- **Updating Pointers**: We move front to the next node and delete the old node. If the queue becomes empty, both front and rear are set to nullptr.

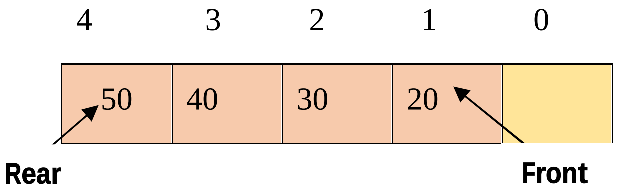
Example:

Visual Representation of the above example

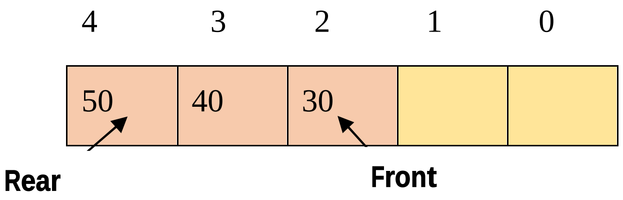


If suppose delete( ) operation has to be performed on the queue, then the first element that was inserted into the queue would be deleted first. So it can be said that the element at the front end of the queue would be deleted.

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.

9.5.3 Front Operation

Array Implementation:

The Front operation simply returns the element at the front of the queue, which is pointed to by queueFront. If the queue is empty, it returns an error.

Code:

```
if (front == -1) {
   cout << "Queue is empty!" << endl;
} else {
   cout << "Front element: " << queue[front] << endl;
}
```

Explanation:

- **Empty Queue Check**: If front is -1, the queue is empty, so no element can be retrieved.
- **Return Front Element**: If the queue is not empty, it retrieves the element at queue[front].

Linked List Implementation:

For a linked list, Front simply returns the data of the node at the front of the queue.

```
if (front == nullptr) {
   cout << "Queue is empty!" << endl;
} else {
   cout << "Front element: " << front->data << endl;
}
```

Explanation:

- **Empty Queue Check**: If front is nullptr, the queue is empty.
- **Return Front Data**: If the queue is not empty, it returns the data of the node at front.

9.5.4 isEmpty Operation

Array Implementation:

The isEmpty operation checks if the queue is empty by checking if queueFront is -1.

```
return (front == -1);
```

Explanation:

- **Empty Queue Check**: If front is -1, the queue is empty, so the function returns true.

Linked List Implementation:

For a linked list, isEmpty checks if the front pointer is nullptr.

```
return (front == nullptr);
```

Explanation:

- **Empty Queue Check**: If front is nullptr, the queue is empty, and the function returns true.

9.5.5 isFull Operation

Array Implementation:

The isFull operation checks if the queue is full by comparing queue rear to the maximum size of the queue.

```
return (rear == MAX - 1);
```

Explanation:

- **Full Queue Check**: If rear equals MAX-1, the queue has reached its maximum size, so the function returns true.

Linked List Implementation:

For linked lists, the concept of being "full" doesn’t apply because linked lists can dynamically grow. So, isFull is usually not implemented for linked lists.

9.5.6 Traversal Operation

Array Implementation:

The Traversal operation prints all elements in the queue from queueFront to queueRear. It checks if the queue is empty before proceeding.

```
for (int i = front; i <= rear; i++) {
   cout << queue[i] << " ";
}
cout << endl;
```

Explanation:

- **Loop Through Queue**: It starts from front and goes up to rear, printing each element in the queue.

Linked List Implementation:

For a linked list, Traversal involves traversing from front to rear and printing each node's data.

```
Node* temp = front;
while (temp != nullptr) {
   cout << temp->data << " ";
   temp = temp->next;
}
cout << endl;
```

Explanation:

- **Traversing the List**: We start at the front node and traverse to the end, printing the data of each node.

9.5.7 Search Operation

Array Implementation:

The Search operation checks if a specific element exists in the queue by iterating from queueFront to queueRear.

```
for (int i = front; i <= rear; i++) {
   if (queue[i] == value) {
      return i;  // Element found
   }
}
return -1;  // Element not found
```

Explanation:

- **Search the Queue**: It checks each element from front to rear. If the element is found, it returns the index. If not, it returns -1.

Linked List Implementation:

For linked lists, Search involves traversing the list from front to rear and checking each node's data.

```
Node* temp = front;
while (temp != NULL) {
   if (temp->data == value) {
        return temp;  // Element found
    }
    temp = temp->next;
}
return nullptr;  // Element not found
```

Explanation:

- **Search the List**: The method traverses the list, comparing each node's data with the value. If the value is found, the node is returned; otherwise, it returns NULL.

9.6 PRIORITY QUEUE:

A **priority queue** is a special type of queue in which each element has a priority level. Unlike a regular queue where elements are processed in the order they arrive (FIFO), a priority queue processes elements based on their priority. The element with the highest priority is dequeued first, regardless of its position in the queue.

1. **Priority Ordering**: Each element is associated with a priority. The queue operates such that the element with the highest priority is always dequeued first.
2. **Heap-Based Implementation**: Priority queues are typically implemented using heaps (either min-heaps or max-heaps). A max-heap is often used for priority queues where the highest priority has the largest value, while a min-heap is used when the highest priority has the smallest value.
3. **Dynamic Behavior**: The priority of elements determines their order of removal from the queue. New elements are inserted based on their priority, and existing elements are re-ordered accordingly.

Operations:

- **Insert (Enqueue)**: Insert an element with a specific priority into the queue.
- **Delete (Dequeue)**: Remove the element with the highest priority from the queue.
- **Peek**: View the element with the highest priority without removing it.

Applications:

- **Scheduling Algorithms**: Used in operating systems to manage tasks based on priority.
- **Dijkstra’s Algorithm**: In graph theory for finding the shortest path where edges have different weights.
- **Real-Time Systems**: Tasks in real-time systems are managed based on their urgency.

Example:

In a **max-priority queue**, if we insert the following elements in order of (element, priority):

- (A, 2)
- (B, 5)
- (C, 1)

The element B will be dequeued first because it has the highest priority.

9.7 KEY APPLICATIONS OF QUEUES:

9.7.1 Simulation

It is a technique in which one system models the behavior of another. For example, physical simulations like wind tunnels are used to test car designs, and flight simulators are used to train pilots. Simulation is particularly useful when it is too expensive or dangerous to experiment with real systems. Using computer models, we can simulate complex systems that are hard to analyze mathematically, gaining insights without the risks or costs associated with real-world testing.

Example: Movie Theater Simulation

Consider a scenario where the manager of a movie theater is concerned about long wait times for customers at the ticket counter. The manager wants to hire enough cashiers to reduce wait times without wasting resources. Instead of experimenting in real life, a simulation can model the theater’s behavior. The simulation can calculate the average wait time for customers based on various numbers of cashiers, arrival rates, and service times.

Queues in Simulation

In simulations, **queues** are crucial because they model the First-In-First-Out (FIFO) behavior seen in real-world systems. For example, when customers arrive at a theater:

- The first customer is served first.
- Remaining customers wait in line until a cashier is free.

Queues are also used in:

- Grocery stores: Customers wait for the next available cashier.
- Banks: Customers queue for tellers.
- Printers: Print jobs are processed in the order they are received.

9.7.2 Design of a Queuing System

A queuing system includes:

1. **Servers:** Objects providing the service, such as cashiers or tellers.

2. **Customers:** Objects waiting for the service.
3. **Queue:** The structure that holds waiting customers.
4. **Transaction Time:** Time required to serve a customer.

When modeling a system like a movie theater:

- Customers arrive at the queue.
- If a server is available, the customer is served immediately. Otherwise, the customer waits.
- The system calculates metrics such as average wait time by tracking arrival and departure times.

Benefits of Simulation with Queues

Simulating systems with queues allows us to:

- **Optimize Resources:** Determine the ideal number of servers needed.
- **Improve Performance:** Identify bottlenecks and minimize customer wait times.
- **Test Scenarios:** Safely experiment with different configurations and customer arrival patterns.

By using computer simulations, businesses can make data-driven decisions to enhance customer satisfaction and operational efficiency.

9.7.3 Other Applications

1. **Task Scheduling in Operating Systems**: Queues are used to manage tasks in operating systems, especially when tasks need to be executed in the order they arrive. For example, when multiple processes are ready to execute, the operating system can use a queue to manage which process gets CPU time first, ensuring that the tasks are processed in the correct order.
2. **Data Streaming and Buffering**: Queues are integral to managing streaming data. For instance, when data is received in a continuous stream (such as network packets or video streams), it is temporarily stored in a queue until it is processed or transmitted. This helps in handling bursts of data efficiently by ensuring that data is processed in the correct order.
3. **Breadth-First Search (BFS)**: In graph theory, BFS is an algorithm that uses a queue to explore nodes level by level. The queue is used to store the nodes that need to be explored, ensuring that nodes are processed in the correct order (first in, first out).
4. **Real-Time Data Processing**: Queues are used in real-time systems, such as online transaction processing (OLTP) systems, where data needs to be processed as it arrives. A queue ensures that the data is processed in the order it arrives, maintaining the integrity of the system.

5. **Print Spooling**: In computer systems where multiple print jobs are queued for a single printer, a queue is used to manage the order in which print jobs are processed. The first print job to arrive is the first one to be printed, ensuring that no job is skipped.

6. **Simulation**: Queues are widely used in simulations where events are processed in the order they occur. For instance, in a simulation of customer service, customers might enter a queue to wait for service, and the first customer in the queue is the first to be served.

9.8 KEY TERMS:

Queue, FIFO, Enqueue, Dequeue, Circular Queue, Priority Queue.

9.9 REVIEW QUESTIONS:

1) What is a queue, and how does it differ from a stack?

2) Explain the FIFO principle with a real-life example.

3) How does a circular queue address the limitations of a linear queue?

4) Describe the key operations performed on a queue and their significance.

5) Discuss two real-world applications of queues in computer science.

9.10 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) **“Data Structures Using C++”** by D. S. Malik.

4) **“The Art of Computer Programming, Volume 1: Fundamental Algorithms”** by Donald E. Knuth.

Mrs. Appikatla Puspha Latha

LESSON-10
SORTING ALGORITHMS

OBJECTIVES:

The objectives of this lesson are to

1. To understand the fundamental concepts and importance of sorting algorithms in computer science.
2. To explore different types of sorting algorithms, their approaches, and time complexities.
3. To analyze the use cases of comparison-based and non-comparison-based sorting techniques.
4. To evaluate and implement sorting algorithms like Selection Sort, Insertion Sort, Quick Sort, Merge Sort, and Heap Sort.

STRUCTURE:

10.1 INTRODUCTION:

Sorting algorithms are fundamental techniques in computer science used to organize data in a specific order, such as ascending or descending. These algorithms play a critical role in enhancing data processing efficiency by arranging elements to facilitate tasks like searching, merging, and data analysis. Sorting can be performed on various data structures, including arrays and linked lists, and is classified into categories such as comparison-based and non-comparison-based algorithms. Popular sorting methods, like Bubble Sort, Merge Sort, Quick Sort, and Heap

Sort, differ in their approach, time complexity, and suitability for specific applications. Understanding sorting algorithms is essential for optimizing performance in both small-scale programs and large-scale data systems.

10.2 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, Selections Sort and 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.

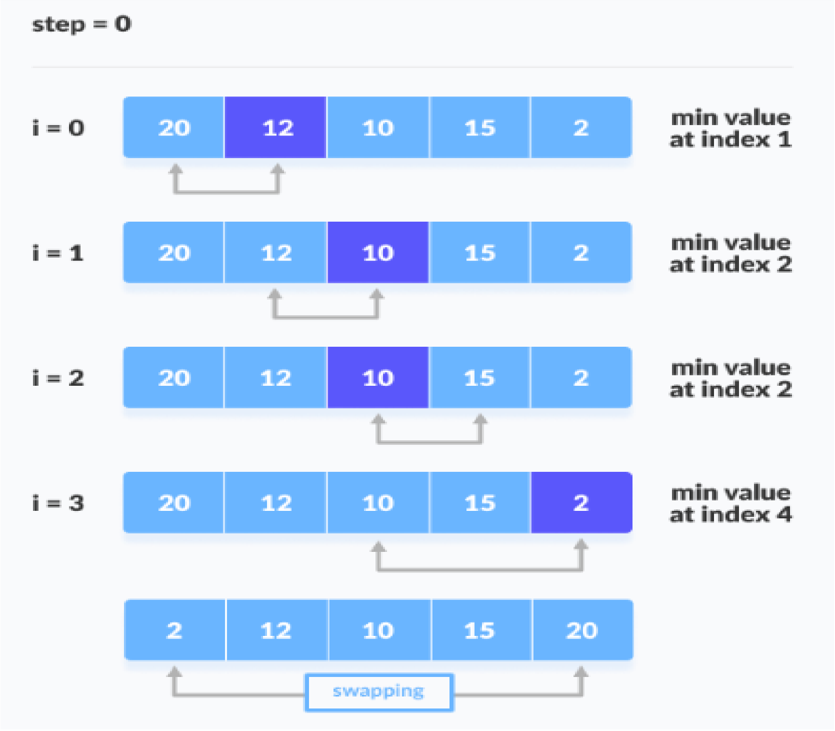
10.3 SELECTION SORT:

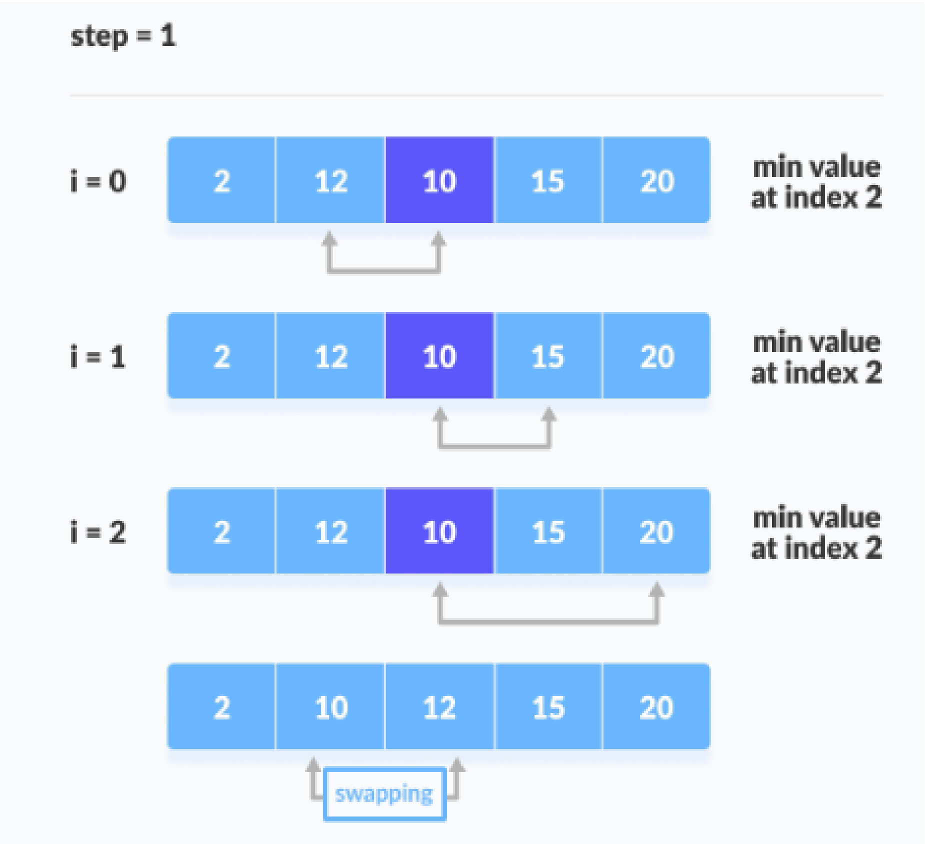
Selection Sort is a simple and intuitive comparison-based sorting algorithm. It works by systematically selecting the smallest (or largest) element from the unsorted portion of an array and placing it in its correct position. This process is repeated until the entire array is sorted. The selection sort algorithm as follows

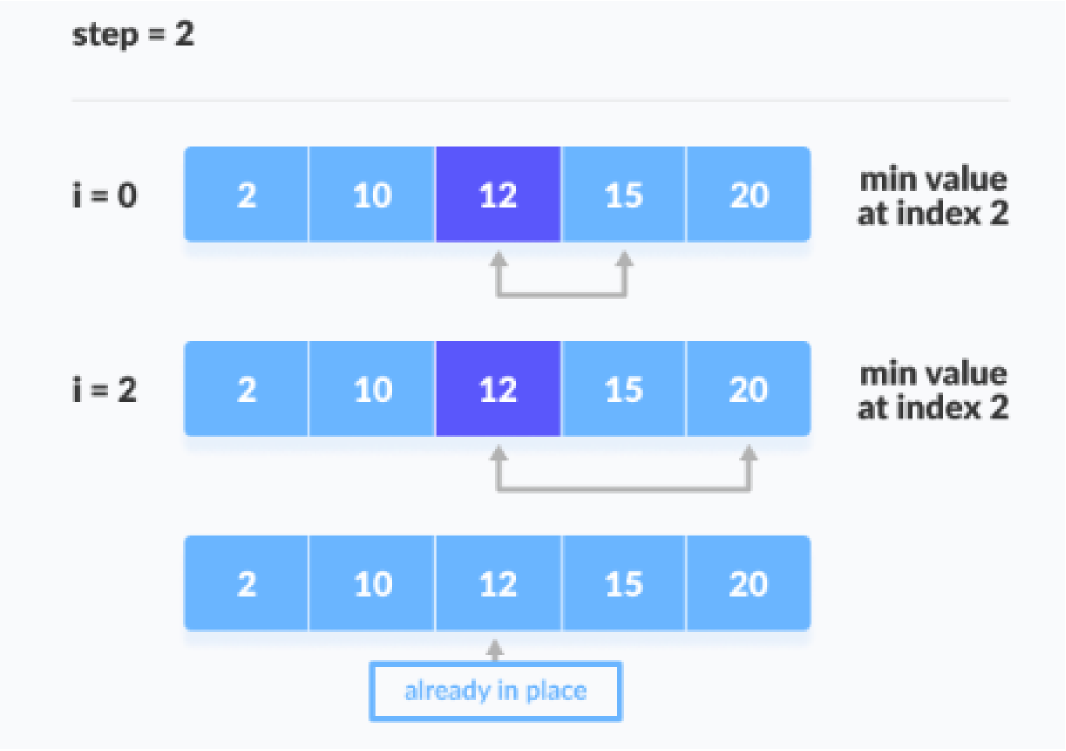
1. Start with the entire array marked as unsorted.
2. Identify the smallest element in the unsorted portion of the array.
3. Swap this smallest element with the first unsorted element, ensuring the smallest value is now in its correct position.
4. Move to the next unsorted portion of the array and repeat the process: find the next smallest element and place it in its correct position.
5. Continue this process until all elements in the array are sorted.

By following these steps, the algorithm gradually builds a sorted portion of the array, one element at a time, until the entire array is in order.

Example:




step = 1
i = 0
2
12
10
15
20
min value at index 2
i = 1
2
12
10
15
20
min value at index 2
i = 2
2
12
10
15
20
min value at index 2
2
10
12
15
20
swapping


step = 2
i = 0
2
10
12
15
20
min value at index 2
i = 2
2
10
12
15
20
min value at index 2
2
10
12
15
20
already in place

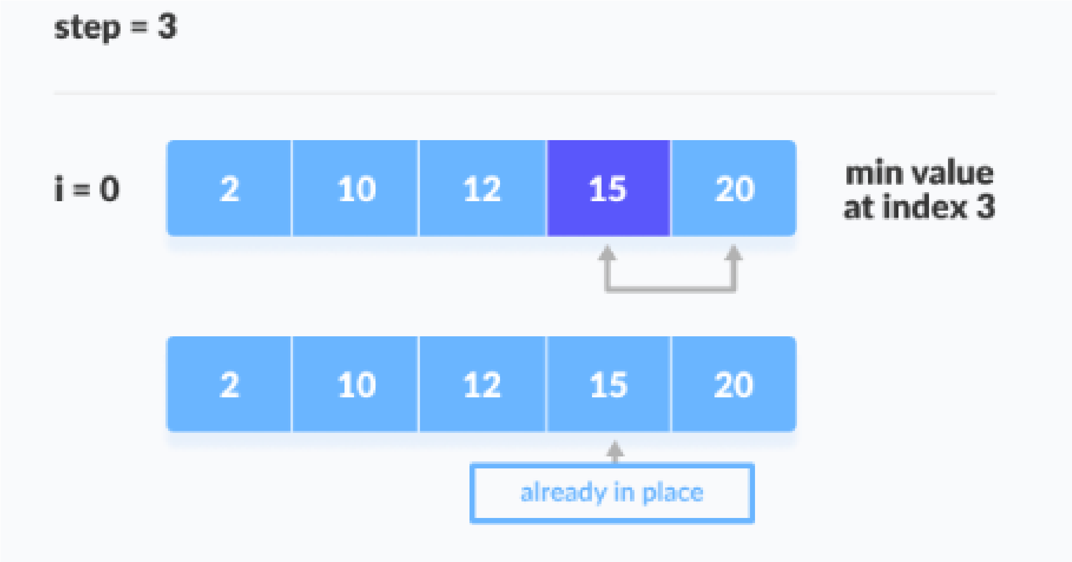


Fig. 10.1 Iterations of Selection Sort

The diagrams provide a step-by-step visualization of the Selection Sort algorithm. Here's a precise explanation for each step:

Step = 0

- **Initial Array:**[20, 12, 10, 15, 2]
- **Process:** The smallest element in the array (2 at index 4) is identified.
- **Swap:**2 is swapped with the first element (20 at index 0).
- **Resulting Array:**[2, 12, 10, 15, 20]

Step = 1

- **Current Array:**[2, 12, 10, 15, 20]
- **Process:** The smallest element in the remaining sub-array (10 at index 2) is identified.
- **Swap:**10 is swapped with the second element (12 at index 1).
- **Resulting Array:**[2, 10, 12, 15, 20]

Step = 2

- **Current Array:**[2, 10, 12, 15, 20]
- **Process:** The smallest element in the remaining sub-array (12 at index 2) is already in its correct position.

- **Swap:** No swap is performed.
- **Resulting Array:**[2, 10, 12, 15, 20]

Step = 3

- **Current Array:**[2, 10, 12, 15, 20]
- **Process:** The smallest element in the remaining sub-array (15 at index 3) is already in its correct position.
- **Swap:** No swap is performed.
- **Resulting Array:**[2, 10, 12, 15, 20]

Final Sorted Array

- The array is fully sorted as [2, 10, 12, 15, 20].

At each step, the smallest element from the unsorted portion is selected and placed in its correct position. The sorted portion of the array grows incrementally from left to right.

10.4 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:

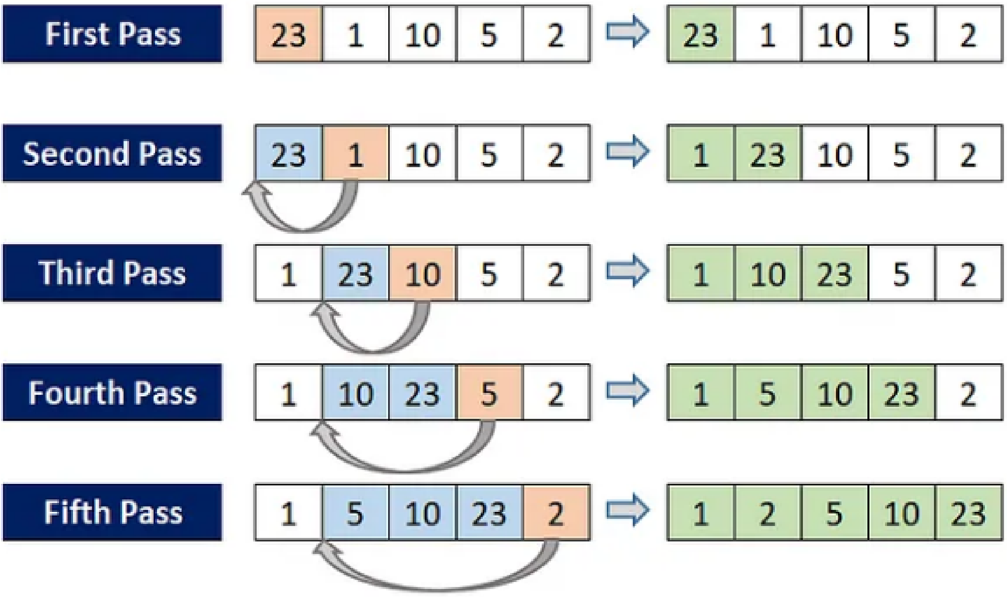


Fig. 10.2 Iterations of Insertion Sort

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.

10.5. 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.

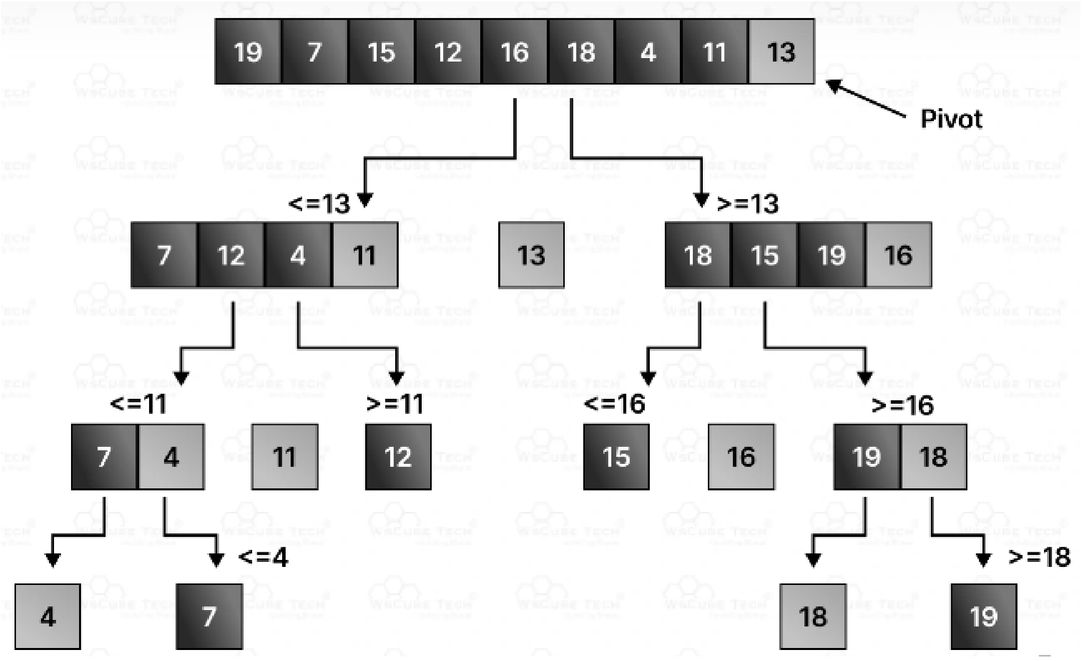


Fig. 10.3 Iterations of Quick Sort

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.

10.6 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.

10.6.1 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.

10.6.2 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.

10.6.3 Recursive Merge Sort

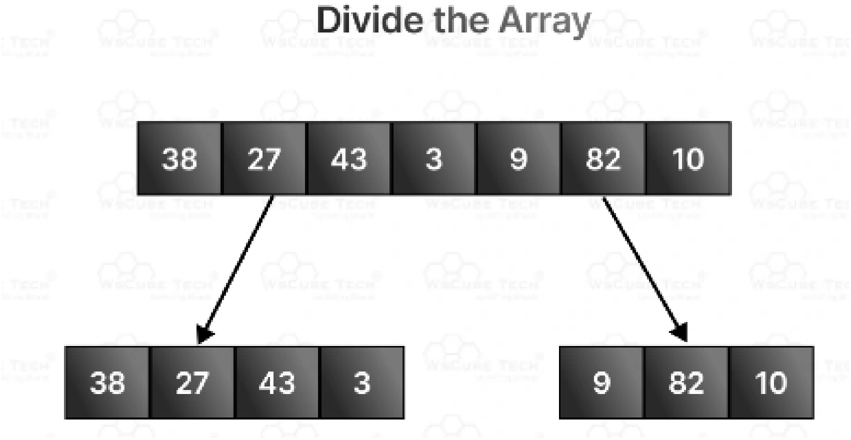
Recursive Merge Sort repeatedly splits the array until single-element subarrays are reached, then merges them recursively to produce the sorted array.

Example of Merge Sort

In the below example, the array [38, 27, 43, 3, 9, 82, 10] is sorted using merge sort.

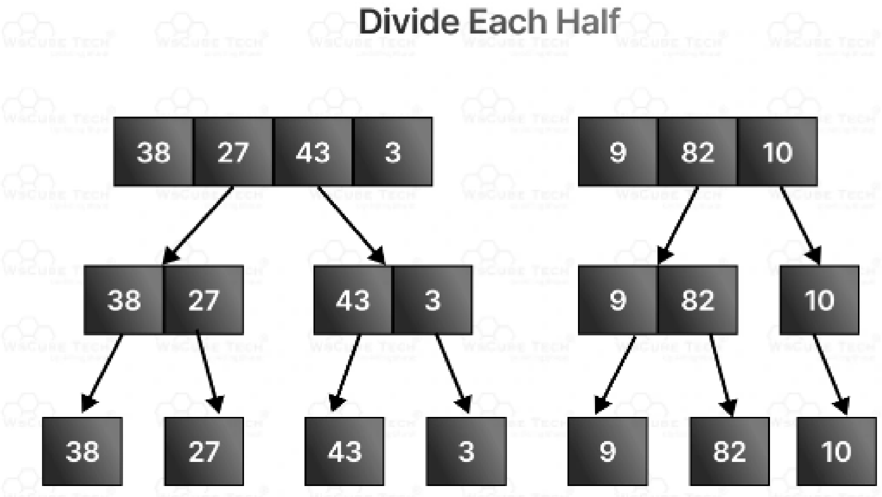
1. Divide the Array

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].

4. 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].

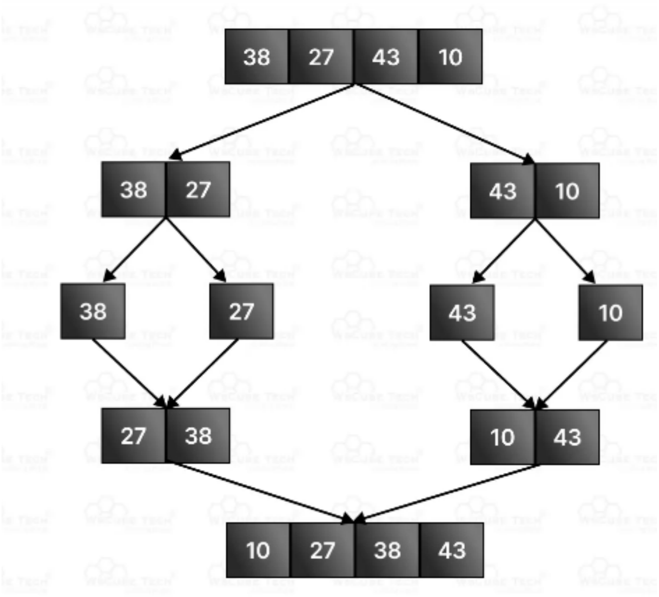


Fig. 10.4 Iterations of Merge Sort

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].

10.7 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.

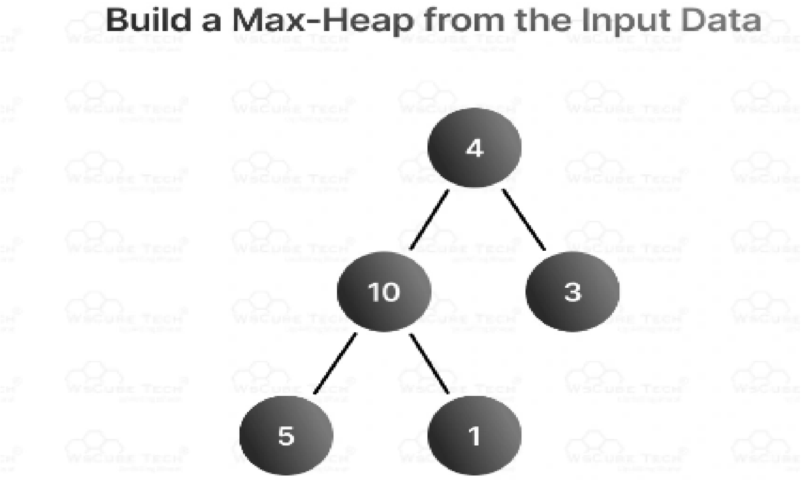
By repeatedly moving the largest number to the end of the list and restructuring the heap, we end up with a sorted list. Heap sort is efficient and works well for large lists.

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]

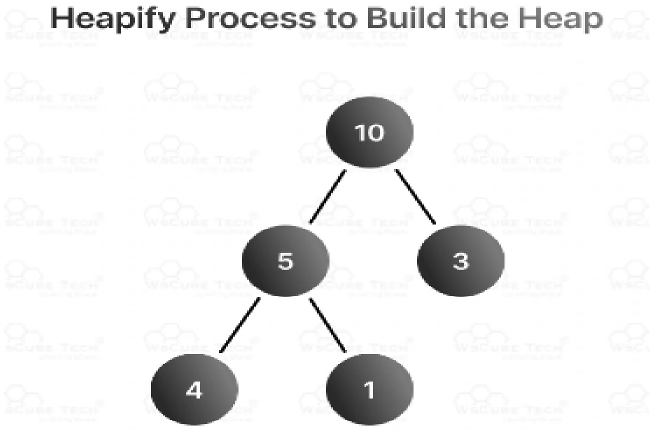


- A **binary tree** structure is created with the elements. Initially, this tree may not satisfy the **max-heap property**.
- The heap property ensures that every parent node is greater than or equal to its child nodes.

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:

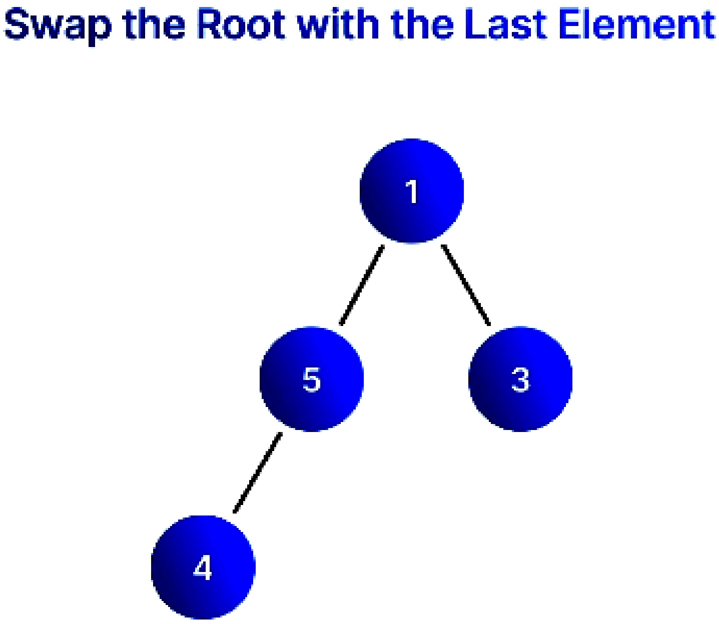


- The heapify operation adjusts the nodes to maintain the **max-heap property**.
- Starting from the bottom-most and right-most parent node, we compare and swap nodes to ensure that each parent node is larger than its child nodes.
- After heapifying, the array becomes [10, 5, 3, 4, 1], and the binary tree satisfies the max-heap property.

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.

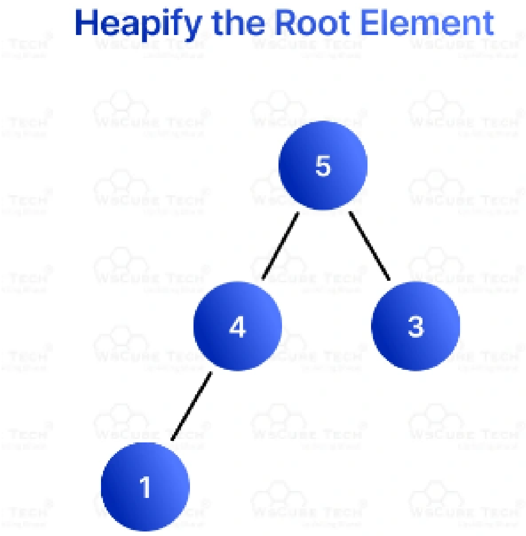
Swap 10 with 1:



- After the swap, the root element (1) may violate the max-heap property.
- The heapify operation is performed again, starting from the root, to restore the heap structure.
- After heapifying, the array becomes [5, 4, 3, 1, 10].

4. Heapify the Root Element:

Restore the heap property by heapifying the root element.

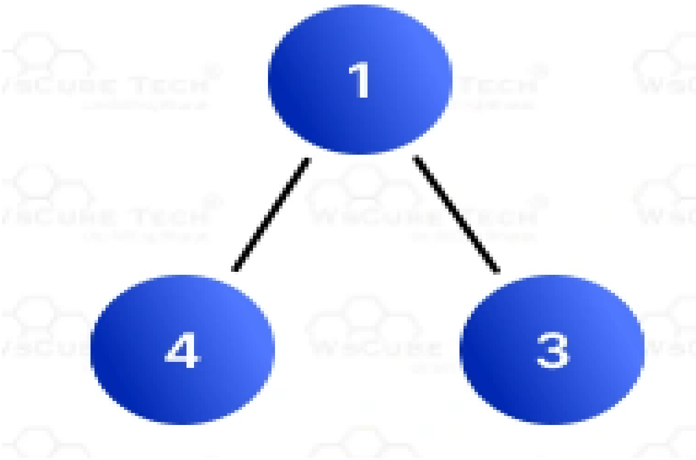


- The root (5) is swapped with the last element in the unsorted portion of the array (1).
- After the swap, the array becomes [1, 4, 3, 5, 10].
- Heapify is performed again, resulting in [4, 1, 3, 5, 10].

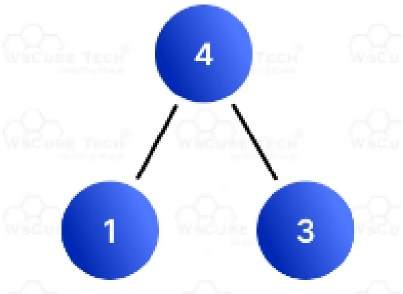
5. Repeat the Process

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:

- The process continues, swapping the root with the last element of the unsorted portion and heapifying.
- Eventually, all elements are placed in their correct positions, resulting in the sorted array [1, 3, 4, 5, 10].

10.8 KEY TERMS:

Sorting, Comparison-Based Sorting, Non-Comparison-Based Sorting, Time Complexity, Stability, Quick Sort, Merge Sort

10.9 REVIEW QUESTIONS:

1) What is the significance of sorting algorithms in computer science?
2) How does Quick Sort differ from Merge Sort in terms of implementation and performance?
3) Explain the stability property of sorting algorithms with examples?
4) Describe the steps involved in the Heap Sort algorithm?
5) Compare the time complexity of Selection Sort and Quick Sort for large datasets?

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) “Algorithms, 4th Edition” by Robert Sedgewick and Kevin Wayne.
4) “The Art of Computer Programming, Volume 3: Sorting and Searching” by Donald E. Knuth.

Mrs. Appikatla Puspha Latha

LESSON-11
BINARY TREE TRAVERSALS AND AVL 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 understand the significance of AVL trees.

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**

```
void preorderTraversal(TreeNode* root)
 {
   if (root != NULL) {
       cout << root->data << " ";        // Visit the root node
       preorderTraversal(root->left);    // Traverse the left subtree
       preorderTraversal(root->right);   // Traverse the right subtree
   }
}
```

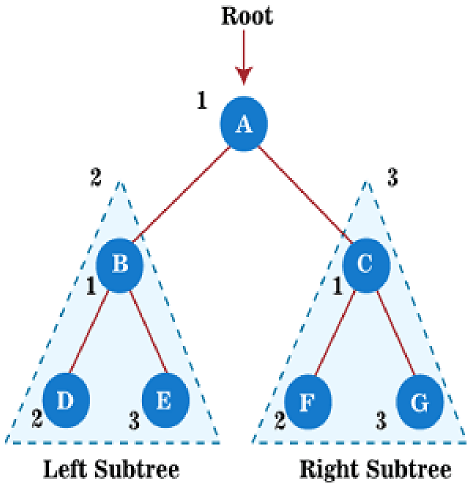


Fig. 11.1. Preorder Traversal

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.

❖ **Function**

```
void inorderTraversal(TreeNode* root)

{
    if (root != NULL) {
        inorderTraversal(root->left);    // Traverse the left subtree
        cout << root->data << " ";       // Visit the root node
        inorderTraversal(root->right);   // Traverse the right subtree
    }
}
```

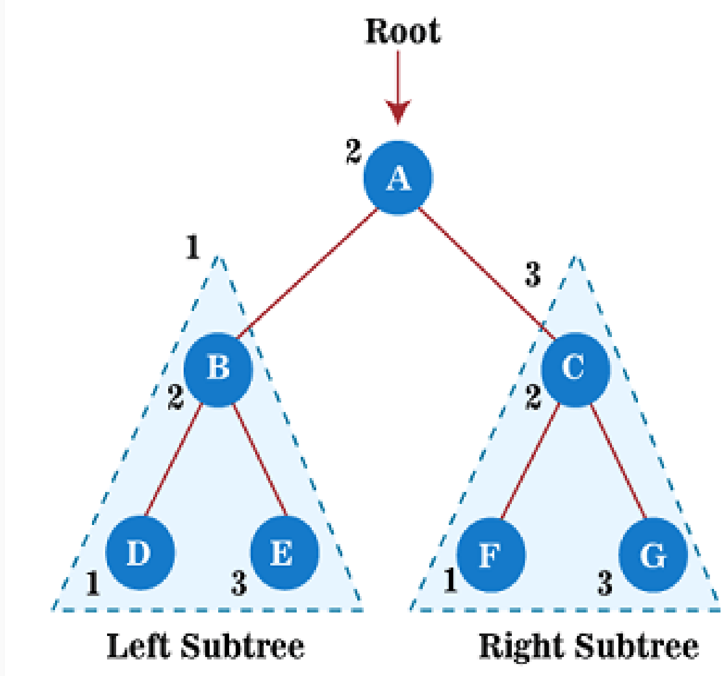


Fig. 11.2. Inorder Traversal

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.

❖ **Function**

```
#include <iostream>
using namespace std;
void postorderTraversal(TreeNode* root) {
   if (root != nullptr) {
      postorderTraversal(root->left);    // Traverse the left subtree
      postorderTraversal(root->right);   // Traverse the right subtree
      cout << root->data << " ";         // Visit the root node
   } }
```

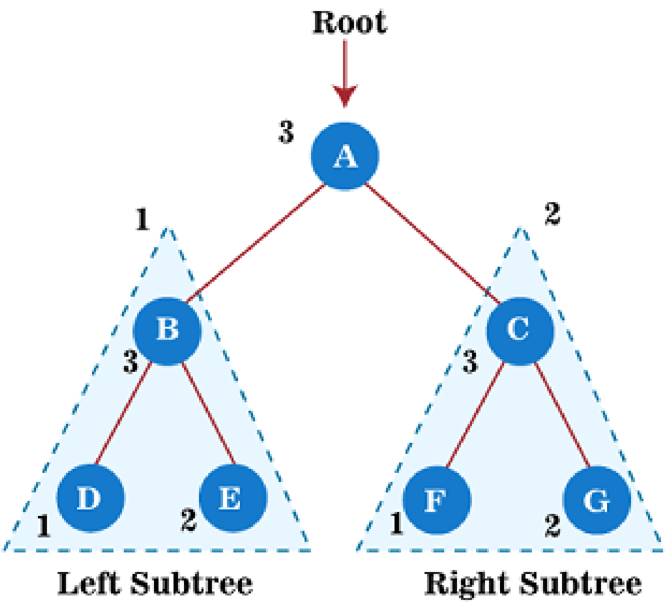


Fig. 11.3. Postorder Traversal

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)

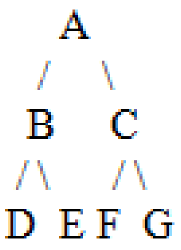
In postorder traversal, we visit each node only after its left and right subtrees have been fully processed. This traversal is particularly useful for applications where we need to process all child nodes before the parent, such as deleting nodes or evaluating postfix expressions in expression trees.

❖ **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 AVL TREES:

AVL trees, named after their inventors Adelson-Velsky and Landis, are a type of self-balancing binary search tree (BST). They ensure that the height difference (balance factor) between the left and right subtrees of any node is no more than 1, thereby maintaining balance in the tree. This balancing property guarantees that the operations on the tree, such as insertion, deletion, and search, are performed in O(log n) time complexity, making AVL trees efficient for dynamic sets of data. Binary Search Trees (BSTs) are efficient when they are balanced. However, without a balancing mechanism, a BST can degenerate into a skewed tree (like a linked list) when elements are inserted in sorted order. This results in operations taking O(n) time, negating the advantages of using a tree. AVL trees address this problem by maintaining balance after every insertion or deletion, ensuring that the tree height remains logarithmic relative to the number of nodes.

11.6.1 Key Concepts of AVL Trees

1. **Self-Balancing Binary Search Tree:**
   An AVL tree is a type of binary search tree (BST) that maintains balance to ensure efficient operations like insertion, deletion, and search.
2. **Balance Factor:**

   The balance factor of a node is the difference in height between its left and right subtrees:

   Balance Factor = Height(Left Subtree) - Height(Right Subtree).

   For an AVL tree, the balance factor of every node must be **-1, 0, or 1**.

3. **Rotations:**
   To restore balance when a node becomes unbalanced (i.e., its balance factor is outside the range [-1, 1]), rotations are performed.
4. **Height Maintenance:**

   The height of a tree is the longest path from a node to a leaf. An AVL tree ensures that the height difference between left and right subtrees at any node is minimal, maintaining logarithmic tree height.
5. **Rebalancing After Operations:**

   Insertion: After adding a node, the tree checks for balance violations and performs rotations if needed.

   Deletion: After removing a node, the tree rechecks balance and applies rotations as required.

6. **Efficiency:**
   By maintaining balance, the height of the AVL tree is kept at **O(log n)**, ensuring efficient operations even with dynamic insertions and deletions.

11.6.2 Types of Rotations

Rotations in AVL trees are used to maintain balance after an insertion or deletion operation causes a node's balance factor to fall outside the range of [-1, 1]. Rotations restructure the tree to restore balance while preserving the binary search tree property.

1. Left Rotation (LL Rotation)

- **When to Use:**

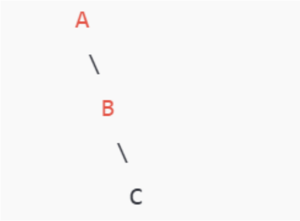
  A left rotation is performed when a node is unbalanced due to a heavier right subtree of its right child.

  Scenario: The balance factor of the node becomes -2, and its right child’s balance factor is ≤ 0.

- **How It Works:**

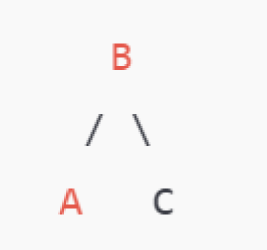
  - The unbalanced node becomes the left child of its right child.
  - The left subtree of the right child (if any) is assigned to the original node’s right pointer.

- **Example:**



```
  A
   \
    B
     \
      C
```

After left rotation:



```
  B
 / \
A   C
```

2. Right Rotation (RR Rotation)

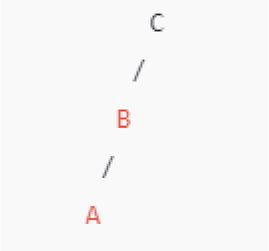
- **When to Use:**

  A right rotation is performed when a node is unbalanced due to a heavier left subtree of its left child.

  Scenario: The balance factor of the node becomes +2, and its left child's balance factor is ≥ 0.

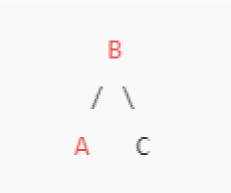
- **How It Works:**
  - The unbalanced node becomes the right child of its left child.
  - The right subtree of the left child (if any) is assigned to the original node's left pointer.

- **Example:**



```
    C
   /
  B
 /
A
```

After right rotation:



```
  B
 / \
A   C
```

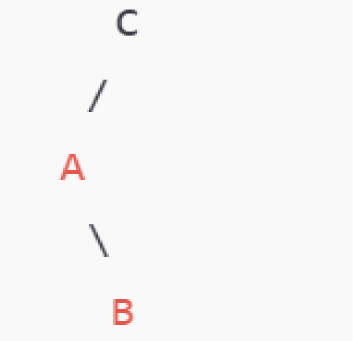
3. Left-Right Rotation (LR Rotation)

- **When to Use:**

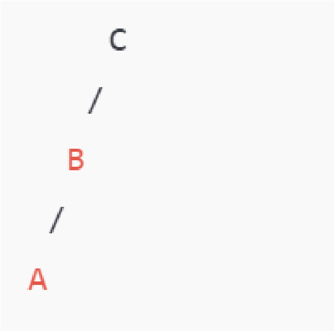
  A left-right rotation is required when a node is unbalanced due to a heavier right subtree of its left child.

  Scenario: The balance factor of the node becomes +2, and its left child's balance factor is -1.

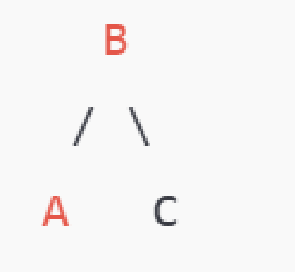
- **How It Works:**
  - A left rotation is first performed on the left child of the unbalanced node.
  - Then, a right rotation is performed on the unbalanced node.
- **Example:**



First, a left rotation is performed on A:



Then, a rotation to the right is done on C:



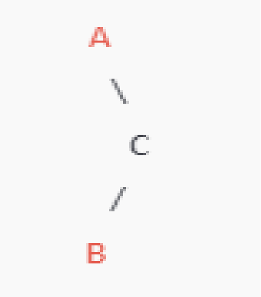
4. Right-Left Rotation (RL Rotation)

- A right-left rotation is required when a node is unbalanced due to a heavier left subtree of its right child.

  Scenario: The balance factor of the node becomes -2, and its right child's balance factor is +1.
- **How It Works:**
  - A right rotation is first performed on the right child of the unbalanced node.
  - Then, a left rotation is performed on the unbalanced node.

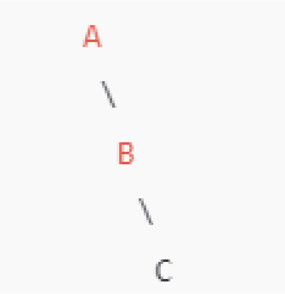
- **Example:**

Before Rotation:



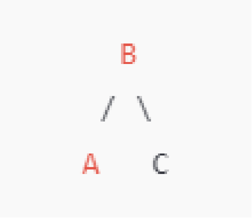
```
  A
   \
    C
   /
  B
```

First, a right rotation is performed on C



```
  A
   \
    B
     \
      C
```

Then, a left rotation is performed on A



```
    B
   / \
  A   C
```

Summary of Rotations:

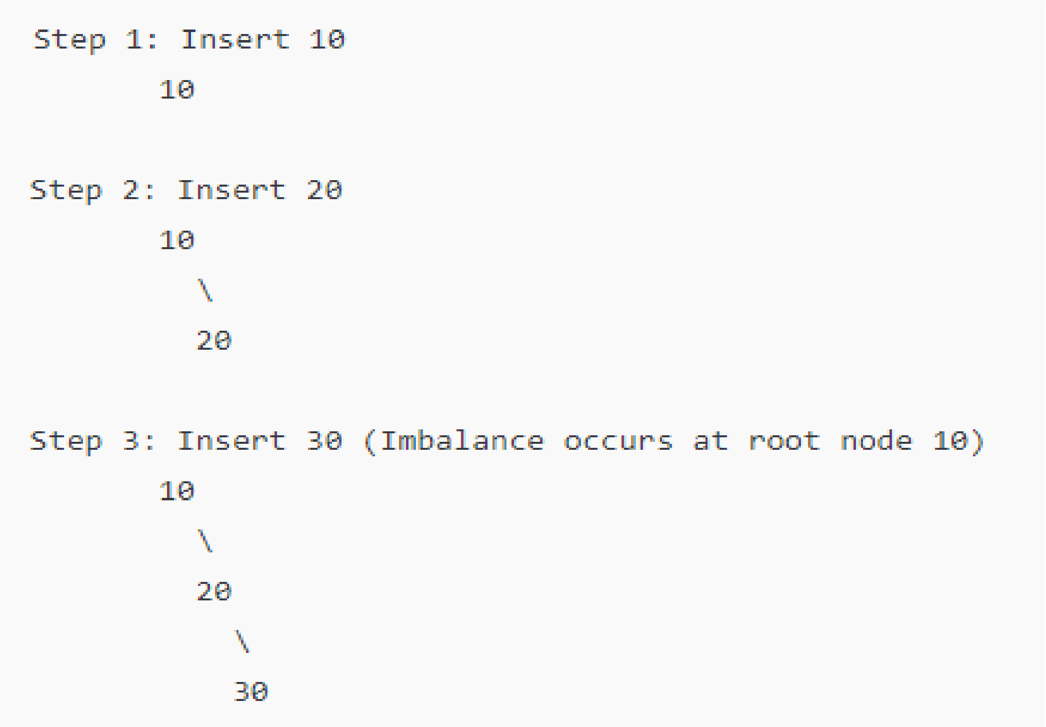
1. **LL Rotation:** For right-heavy imbalance in the right subtree of the right child.
2. **RR Rotation:** For left-heavy imbalance in the left subtree of the left child.
3. **LR Rotation:** For right-heavy imbalance in the left child.
4. **RL Rotation:** For left-heavy imbalance in the right child.

Rotations are crucial in maintaining AVL tree balance, ensuring efficient operations while adhering to the binary search tree property.

11.6.3 Example:

Let's build a tree by inserting the following sequence of numbers:

10, 20, 30, 40, 50, 25

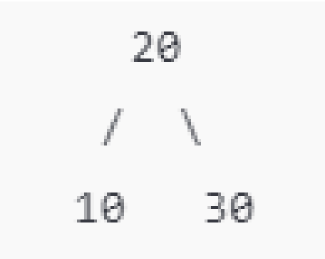


```
Step 1: Insert 10
       10

Step 2: Insert 20
       10
         \
         20

Step 3: Insert 30 (Imbalance occurs at root node 10)
       10
         \
         20
           \
           30
```

Imbalance: Balance factor of node 10 = -2.

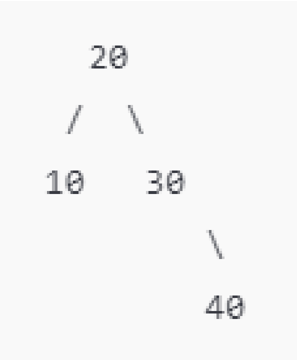
Rotation Needed: Left Rotation at 10.

After Left Rotation:



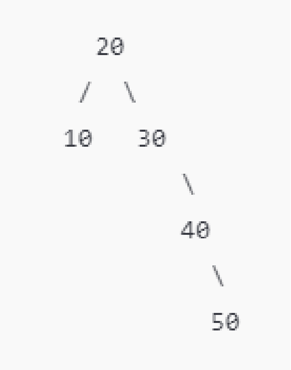
```
  20
 /  \
10   30
```

Step 4: Insert 40



```
  20
 /  \
10   30
        \
        40
```

Step 5: Insert 50 (Imbalance occurs at node 30)

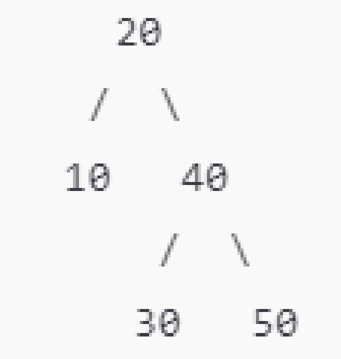


```
    20
   /  \
  10   30
         \
          40
            \
             50
```

Imbalance: Balance factor of node 30 = -2.

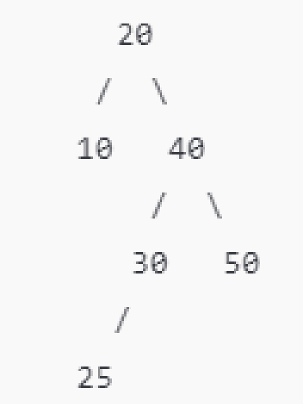
Rotation Needed: Left Rotation at 30.

After Left Rotation:



```
    20
   /  \
  10   40
      /  \
     30   50
```

Step 6: Insert 25 (Imbalance occurs at node 40)



```
     20
    /  \
   10   40
       /  \
      30   50
     /
   25
```

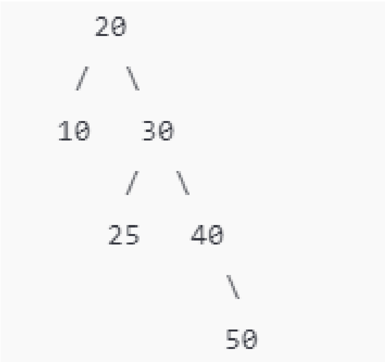
Imbalance: Balance factor of node 40 = +2.

Rotation Needed: Left-Right Rotation at 40.

After Left-Right Rotation:

1. First, perform Left Rotation at 30.
2. Then, perform Right Rotation at 40.

Resulting Tree:



```
    20
   /  \
  10   30
      /  \
     25   40
            \
             50
```

Summary of Rotations in the Example:

1. **Left Rotation** at 10 after inserting 30.
2. **Left Rotation** at 30 after inserting 50.
3. **Left-Right Rotation** at 40 after inserting 25.

This example illustrates how rotations maintain balance in an AVL tree as nodes are added.

11.6.4 Maintaining Balance in AVL Trees

After any insertion or deletion, AVL trees check the balance factor of all ancestor nodes of the affected node. If a violation in the balance factor occurs:

- Determine the type of rotation required.
- Perform the appropriate single or double rotation to restore balance.

The checking of the balance factor and performing rotations is efficient since it only requires traversing the height of the tree O(log n).

11.6.5 Applications of AVL Trees

The major applications of AVL Trees are

1. Databases

- AVL trees are used in databases for indexing and organizing data.
- They ensure efficient searching, insertion, and deletion operations with O(log n) time complexity, which is critical for large datasets.

2. File Systems

- Used in file systems for directory management, where efficient lookup, insertion, and deletion are required.
- Examples: NTFS in Windows utilizes AVL trees for certain indexing tasks.

3. Memory Management

- Used in dynamic memory allocation to manage free blocks of memory.

- AVL trees help efficiently find the nearest available memory block of required size.

4. Routing Algorithms

- Applied in network routers to maintain routing tables.
- Efficiently handles frequent updates in routing tables due to dynamic network topology.

5. Compiler Design

- Used in symbol table management for variables, functions, and classes.
- Ensures quick lookup of identifiers during compilation.

6. Gaming Applications

- AVL trees are used in gaming for managing high-score tables.
- Maintains sorted scores and supports efficient updates when scores change.

7. Geographical Information Systems (GIS)

- AVL trees help store and manage spatial data for location-based services.
- Enables quick lookup for mapping and navigation systems.

8. Telecommunication

- Used for call routing and managing hierarchical data in telecommunications systems.
- Helps in managing prefixes efficiently for quick call routing decisions.

9. Search Engines

- Helps organize and index web pages or documents for efficient search operations.
- Supports dynamic updates in the index as new pages are added.

10. Multimedia Applications

- Used in applications like video games or image rendering for efficient data lookup.
- Balances performance requirements when dealing with large data sets.

11. Operating Systems

- Utilized in process scheduling, where processes need to be sorted by priority or time.
- Ensures efficient priority-based scheduling algorithms.

12. Peer-to-Peer Networks

- Helps in distributed hash tables (DHTs) for managing nodes and ensuring balanced load distribution.

These applications leverage the self-balancing property of AVL trees to ensure consistent and efficient performance in systems where quick data access and dynamic updates are required.

11.7 KEY TERMS:

Binary tree, Depth-first traversal (DFT), Breadth-first traversal (BFT), Threaded binary tree, In-order traversal.

11.8 REVIEW 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) Explain how to choose a specific rotation of AVL trees to balance a tree?

4) What are the advantages of AVL trees?

11.9 SUGGESTED READINGS:

1) **"Introduction to Algorithms"** by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein.

2) **"The Art of Computer Programming, Volume 3: Sorting and Searching"** by Donald E. Knuth.

3) **"Algorithms, 4th Edition"** by Robert Sedgewick and Kevin Wayne.

4) **"Data Structures and Algorithm Analysis in C++"** by Mark Allen Weiss.

Mrs. Appikatla Puspha Latha

LESSON-12
BINARY SEARCH TREES

OBJECTIVES:

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 SIGNIFICANCE:

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.

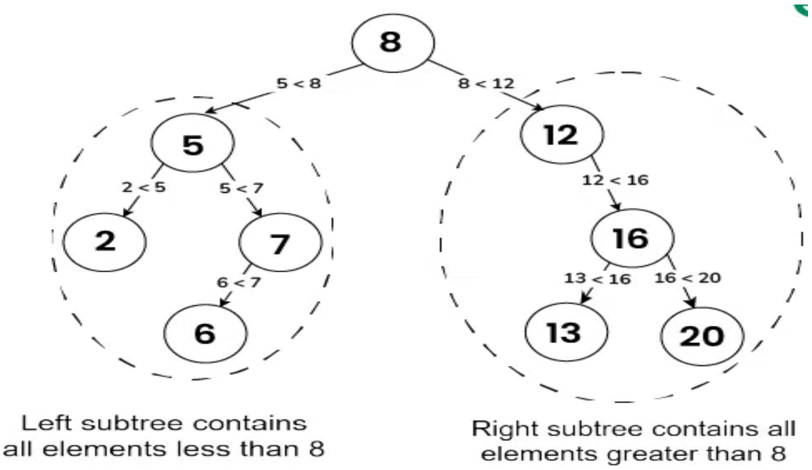

8
5 < 8
8 < 12
5
2 < 5
5 < 7
2
7
6 < 7
6
12
12 < 16
16
13 < 16
16 < 20
13
20
Left subtree contains all elements less than 8
Right subtree contains all elements greater than 8

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:**

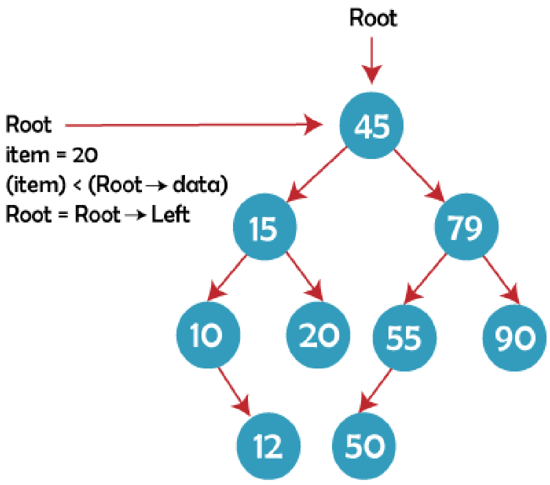
   BSTs support multiple types of tree traversals, such as preorder, inorder, and postorder, which are useful in different contexts like expression evaluation or data processing.

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:



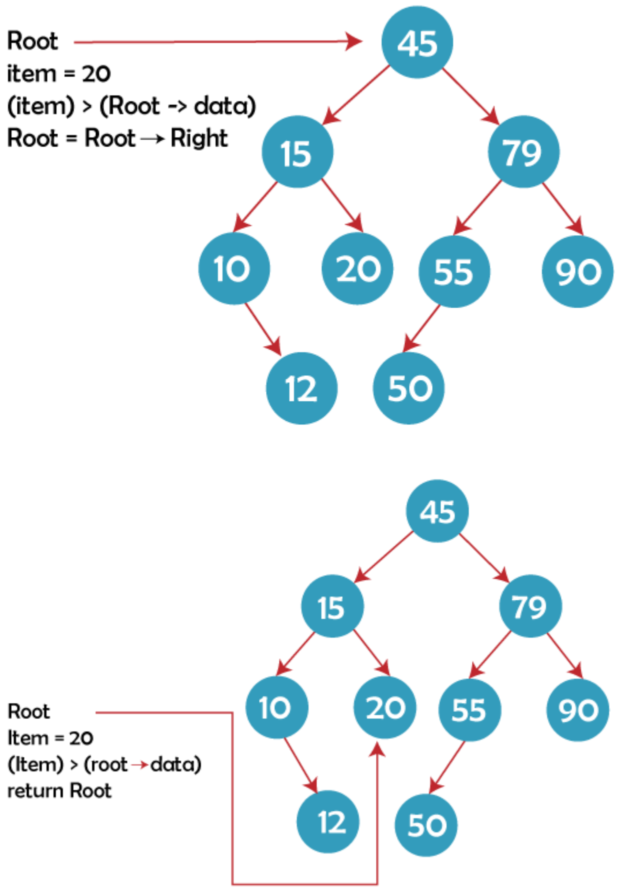


Fig. 12.2 Searching an element in BST

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.

- **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.

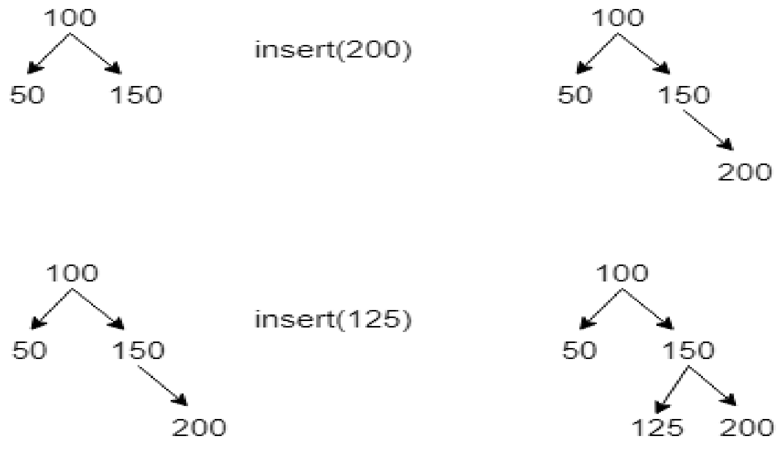


Fig. 12.3 Insertion of node-Example 1

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**

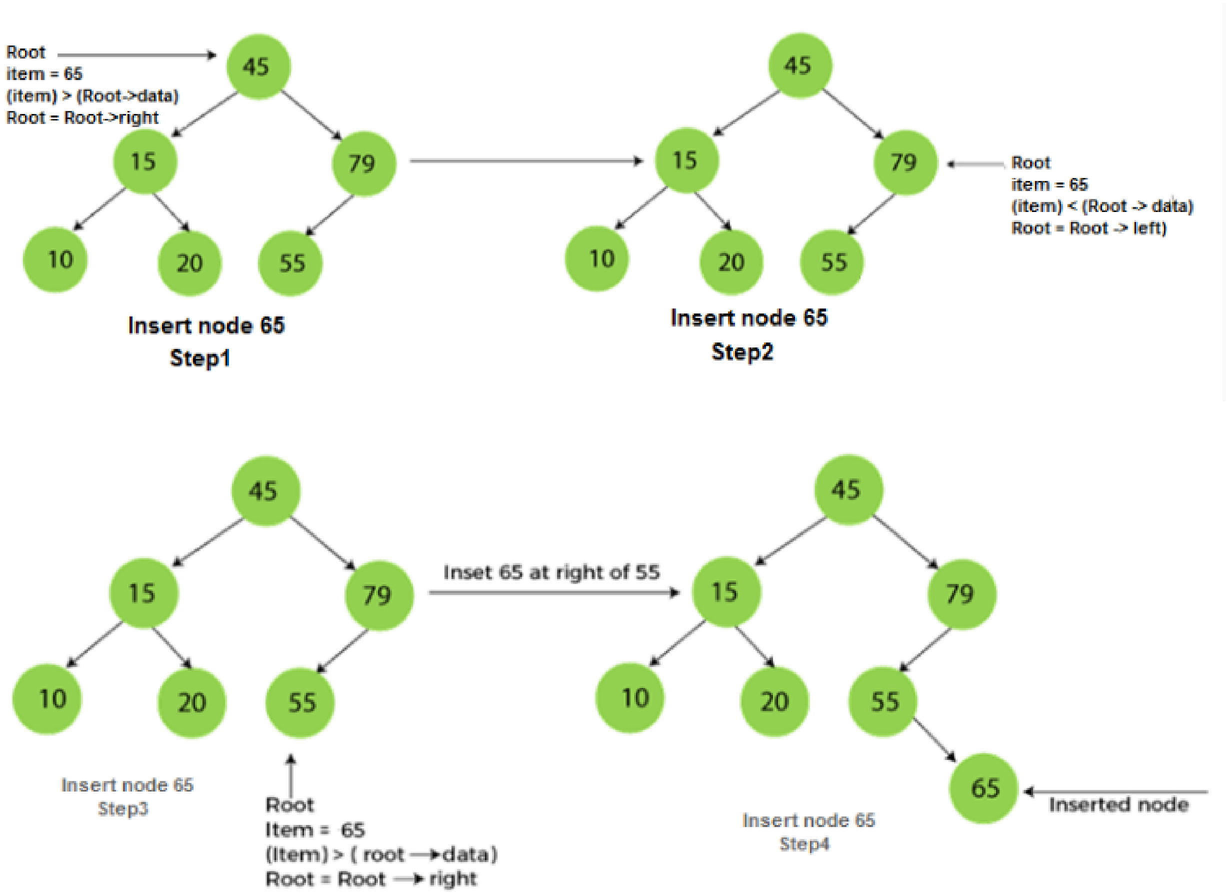


Fig. 12.4 Insertion of node-Example 2

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 Case 1: Deleting a Leaf Node

If the node to be deleted has no children (it is a leaf node), simply remove it from the tree.

Example:

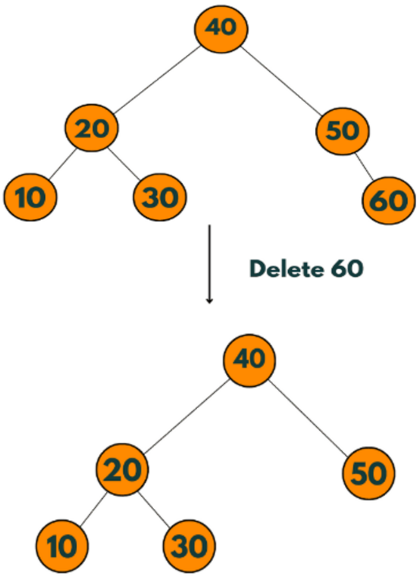


Fig. 12.5 Deletion of a leaf node

- 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.

12.5.2 Case 2: Deleting a Node with One Child

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

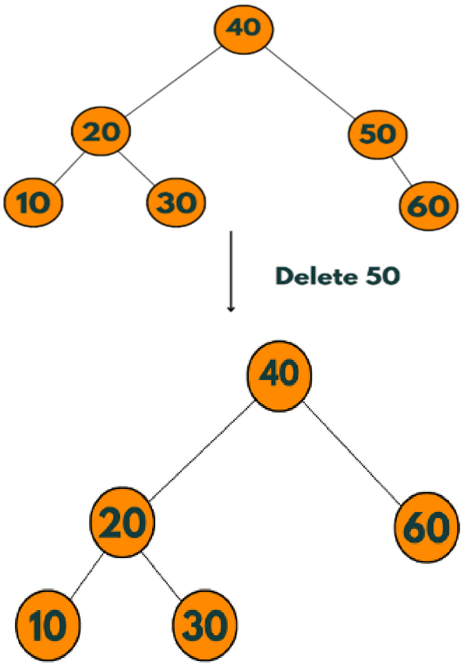


Fig. 12.6 Deletion of a leaf node with one child

- Suppose we want to delete a node with value 50 that has only a right child, 60:
  - Locate the node with value 50.
- 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 Case 3: Deleting a Node with Two Children

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

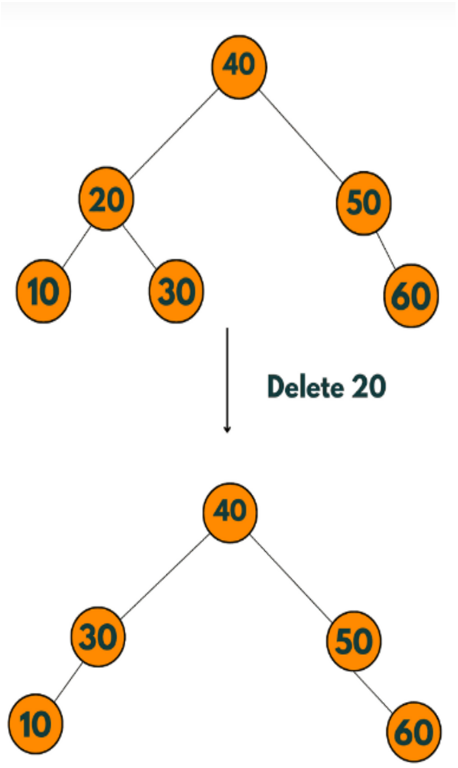


Fig. 12.7 Deletion of a leaf node with two children

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.

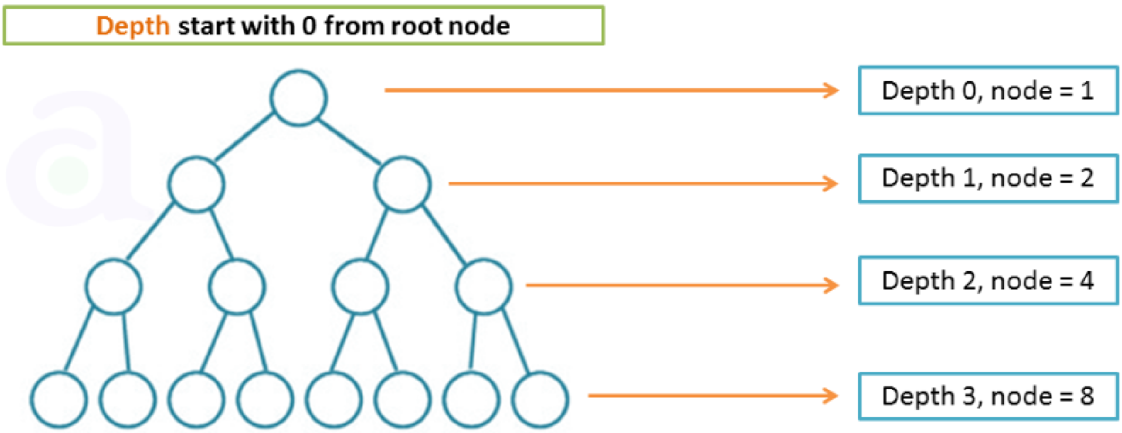


Fig. 12.8 Depth Levels from 0

- **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.

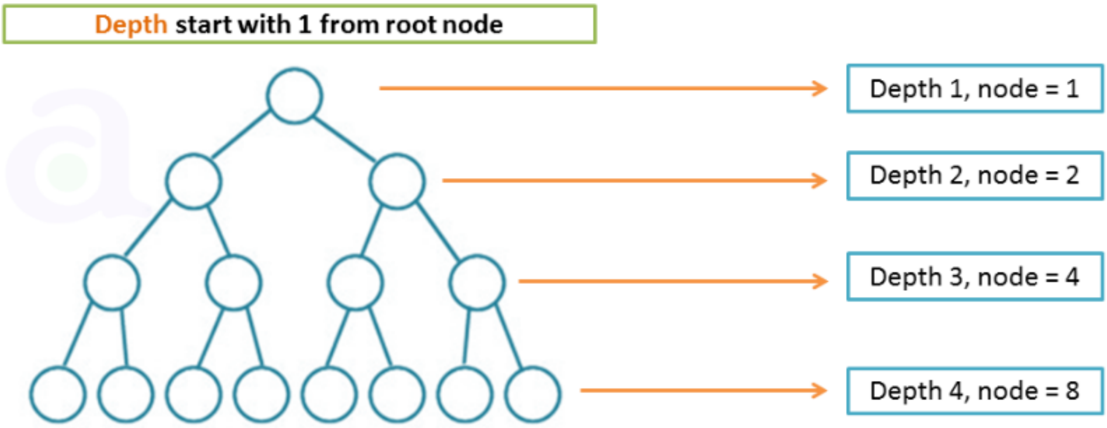


Fig. 12.9 Depth Levels from 1

➢ 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

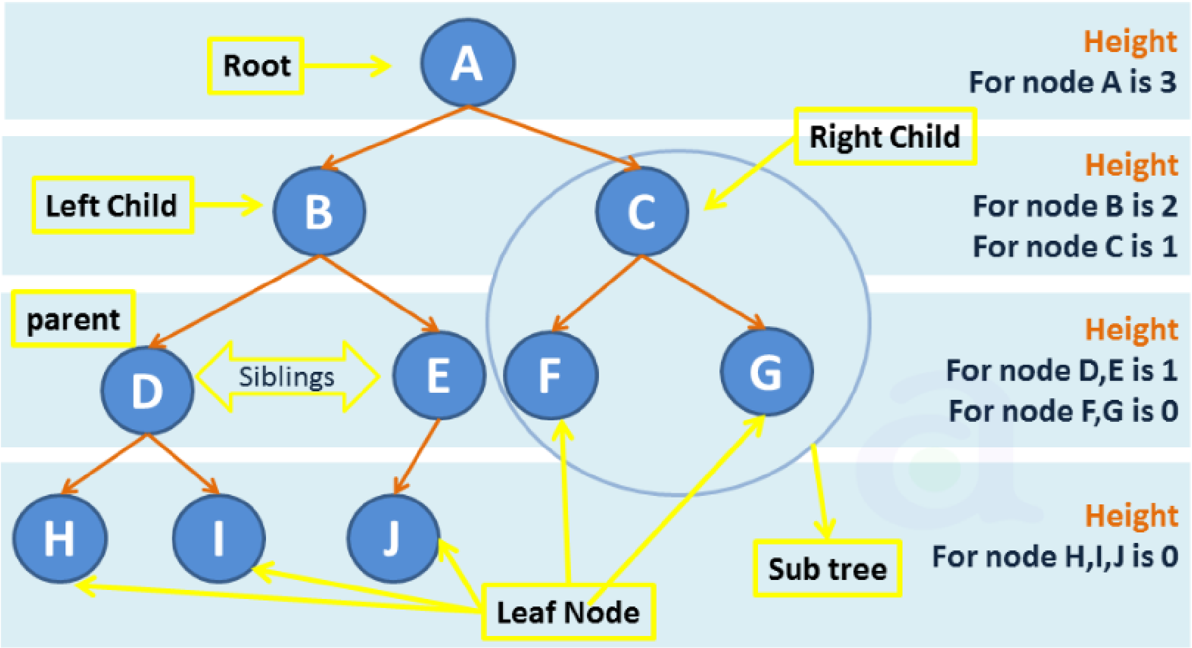


Fig. 12.10 Height of the tree

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 REVIEW 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 Puspha Latha

LESSON-13

GRAPHS AND THEIR OPERATIONS

OBJECTIVES:

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.

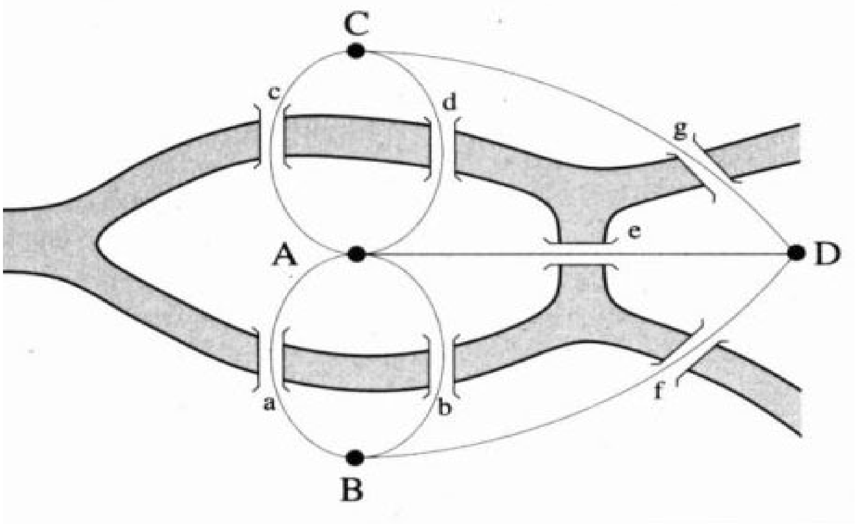


Fig. 13.1.The Koenigsberg Bridge Problem

13.2 IMPORTANT TERMS IN GRAPHS:

Terms which are key for graphs are mentioned below:

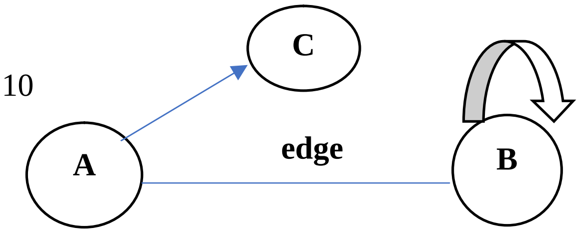


Fig. 13.2. Graph

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 13.1. 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

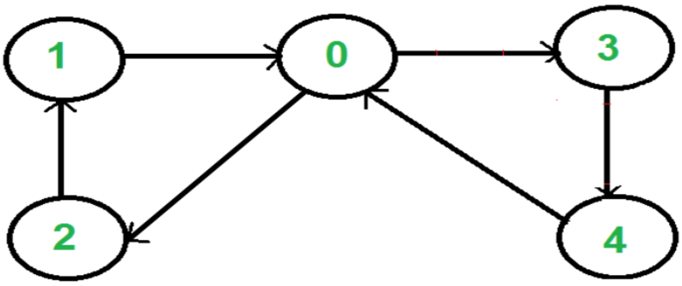


Fig. 13.3. Directed Graph

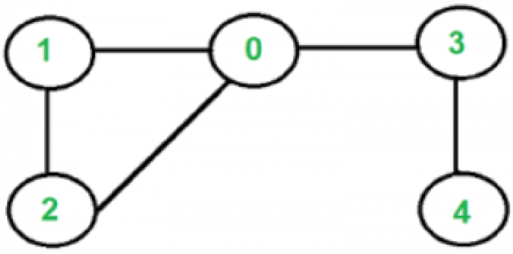


Fig. 13.4. Undirected Graph

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 13.2. 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

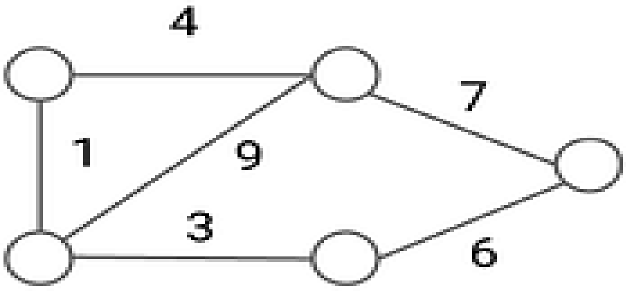


Fig. 13.5 Weighted Graph

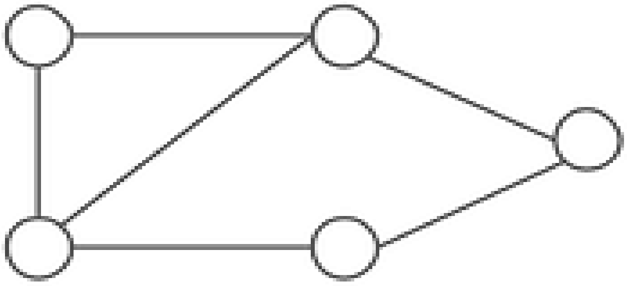


Fig. 13.6. Unweighted Graph

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 13.3. 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

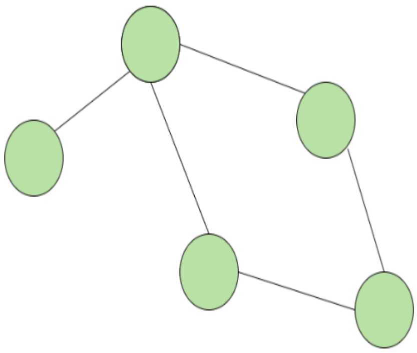


Fig. 13.7 Cyclic graph

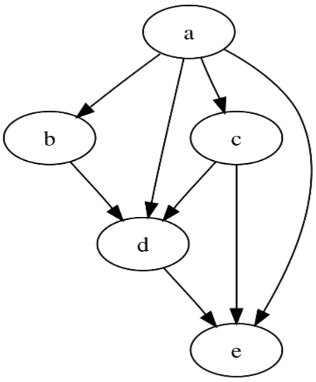


Fig. 13.8. Directed Acyclic Graph

13.3.4 Complete Graphs: Every vertex or node are interconnected with each other; this scenario is named as Complete graphs.

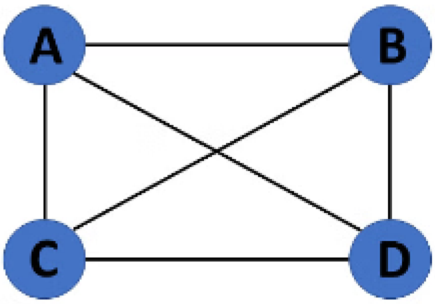


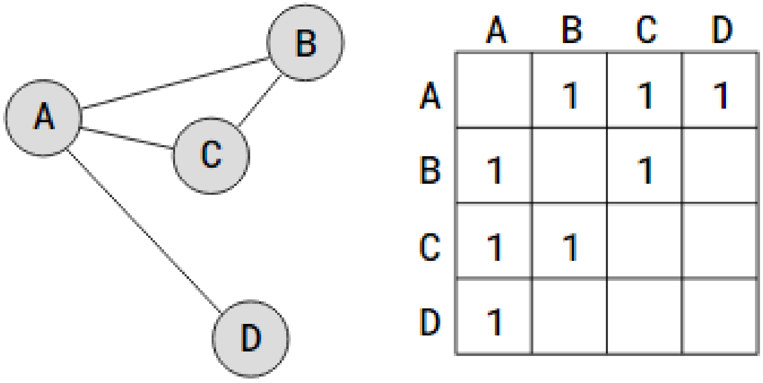
Fig. 13.9. Complete Graph

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:

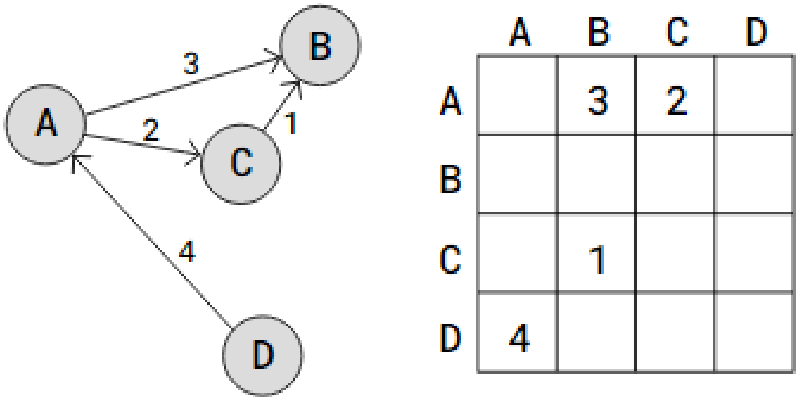
An **adjacency matrix** is a way to represent a graph using a two-dimensional table. Each row and column in the table represents a node in the graph. If there is a connection (or edge) between two nodes, the corresponding cell in the table is marked with a 1 (or the weight of the edge in weighted graphs). 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. 13.10. Adjacency Matrix for Undirected Graph

The Figure 13.10 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. 13.11. Adjacency Matrix for Directed Graph

The Figure 13.11 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.

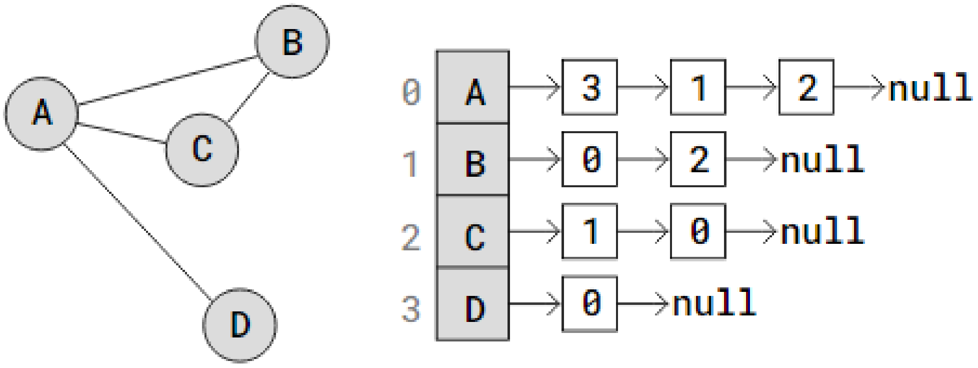


Fig. 13.12 Weighted Directed Graph-Adjacency List

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**.**

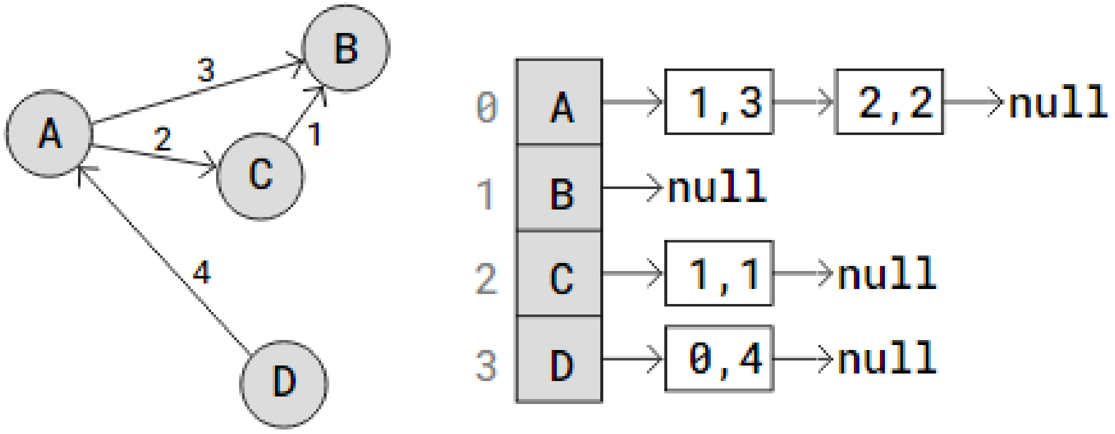


Fig. 13.13 Weighted Directed Graph-Adjacency List

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.

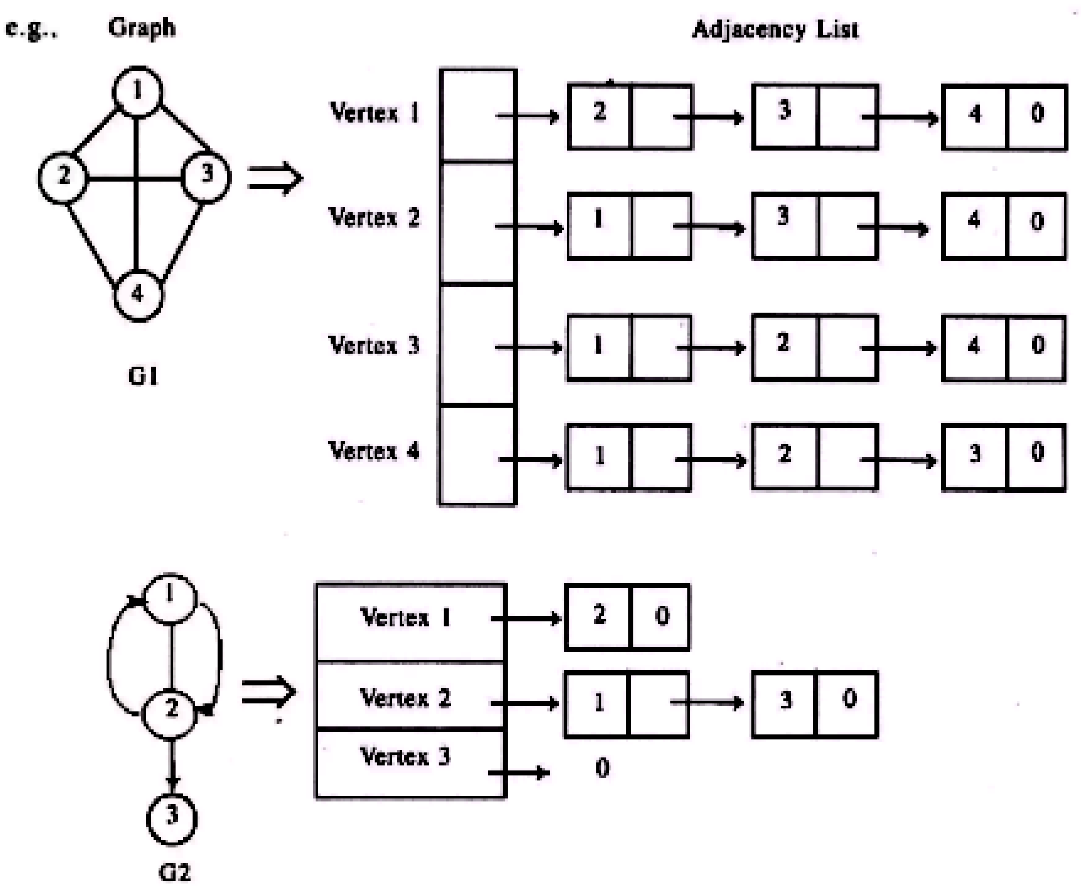


Fig. 13.14 Graph-Adjacency List

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

Adjacency Multi Lists (AML) is a data structure used for representing graphs, particularly **undirected graphs**, in an efficient way. It is an advanced extension of the **adjacency list** representation, designed to manage edges more effectively by ensuring that each edge is stored only once, even in bidirectional graphs. This eliminates redundancy, which is common in basic adjacency list implementations.

Unlike traditional adjacency lists where each vertex maintains its own copy of shared edges, adjacency multi lists allow edges to be shared between the adjacency lists of the two vertices they connect. This makes the structure both memory-efficient and easier to update, especially for operations like edge insertion or deletion.

Why Use Adjacency Multi Lists?

1. **Efficient Edge Representation**:
   - In undirected graphs, the same edge is shared between two vertices. AML ensures that only one copy of the edge is maintained, simplifying the structure.
2. **Simplified Edge Updates**:
   - Since edges are not duplicated, modifications to an edge (e.g., weight updates) only need to be applied once, reducing the risk of inconsistencies.
3. **Applications in Sparse Graphs**:
   - AML is particularly beneficial for sparse graphs, where the number of edges is significantly smaller than the maximum possible.

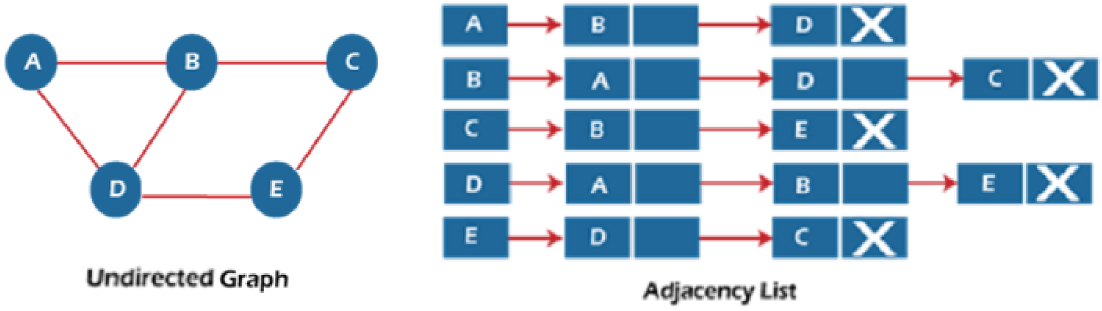


Fig. 13.15 Undirected Graph-AdjacencyList

The above graph represents an undirected graph on the left and its corresponding adjacency list representation on the right. In the adjacency list, each vertex points to a list of its neighbors, efficiently representing the graph's connections.

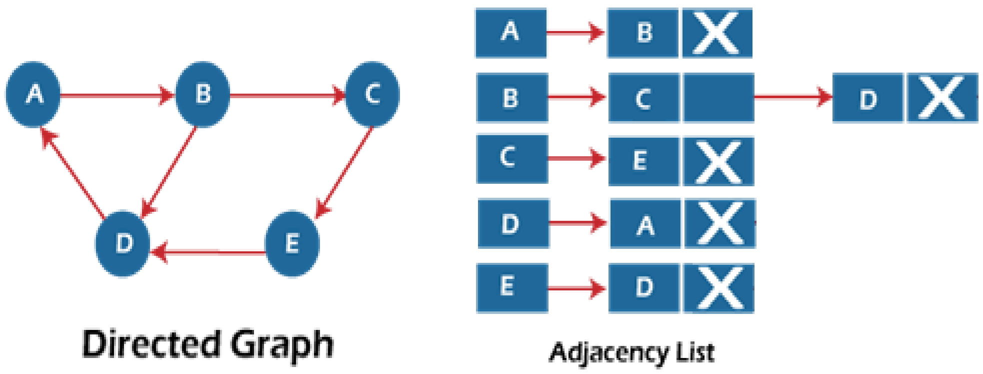


Fig. 13.16 Directed Graph-Adjacency List

The above graph shows a directed graph on the left and its corresponding adjacency list on the right. In the adjacency list, each vertex points to a list of vertices it directs to, effectively representing the graph's directed edges.

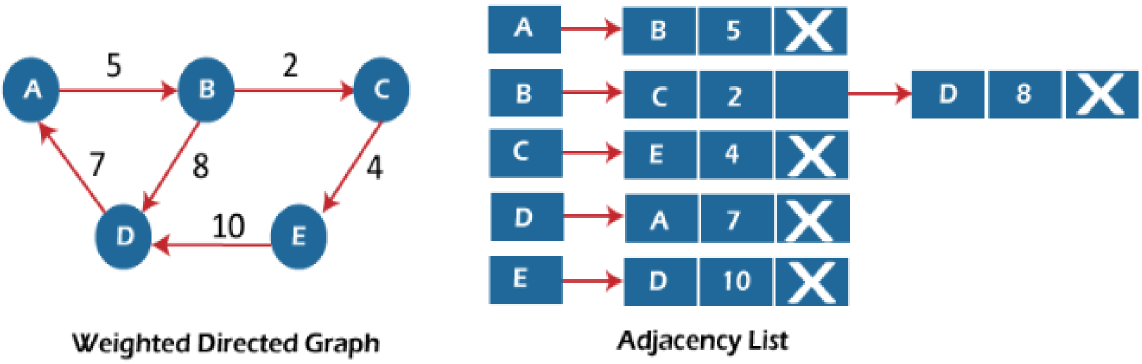


Fig. 13.17 Weighted Directed Graph - Adjacency List

The above graph shows a weighted directed graph on the left and its adjacency list representation on the right. In the adjacency list, each vertex points to its neighbors along with the corresponding edge weights, representing the directed connections and their costs.

13.4.5 Basic Operations on Graphs

1. Create the Graph

To initialize and store the graph in computer memory using a specific representation. The graph can be represented in different ways, depending on the application and memory considerations. Common representations include:

1. **Adjacency Matrix**:
   - A 2D matrix where rows and columns represent vertices, and a cell value indicates whether an edge exists between two vertices.
   - Efficient for dense graphs (graphs with many edges).

2. **Adjacency List**:
    - A list where each vertex points to its neighboring vertices or connected edges.
    - Memory-efficient for sparse graphs (graphs with fewer edges).
3. **Edge List**:
    - A simple list of all edges, where each edge is represented as a pair (or tuple) of vertices and their weight (if applicable).
    - During this operation, all vertices and edges are defined, and the graph structure is built in memory.

2. Clear the Graph

To remove all data from the graph, leaving it empty.

- This operation clears all vertices and edges stored in the graph representation.
- For example:
    - In an adjacency matrix, all values in the matrix would be reset to 0 (or another default value).
    - In an adjacency list, all lists associated with vertices would be deleted.
- This operation is useful when the graph data needs to be reset or when starting a new computation with the same graph object.

3. Check if the Graph is Empty

To verify whether the graph contains any vertices or edges.

- This operation checks:
    - If the graph has zero vertices.
    - If no edges are present (depending on the application, a graph might still be considered empty if it has isolated vertices).
- For an adjacency matrix:
    - Check if all cells in the matrix are 0.
- For an adjacency list:
    - Check if all lists associated with vertices are empty.
- This is often used as a validation step before performing other operations, such as traversal or searching.

4. Traverse the Graph

To systematically visit all vertices and edges of the graph in a specific order.

Traversal is essential for various graph-based algorithms, such as searching, pathfinding, and analyzing connectivity.

Common Graph Traversal Methods:

1. Depth-First Search (DFS):

- Starts at a source vertex and explores as far as possible along one branch before backtracking.
- Uses a stack (or recursion) to keep track of visited nodes.
- Suitable for applications like finding connected components, checking for cycles, etc.

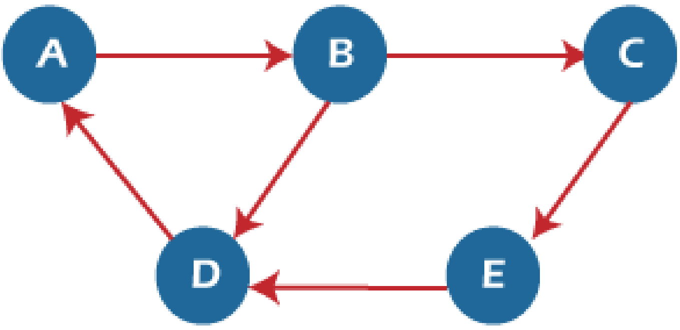
2. Breadth-First Search (BFS):

- Starts at a source vertex and explores all its neighbors before moving to the next level.
- Uses a queue to keep track of vertices to visit next.
- Often used for finding the shortest path in an unweighted graph or determining levels in a graph.

Example:

- For a graph with vertices A -> B -> C, DFS might visit A -> B -> C, while BFS would explore level by level.

5. Print the Graph



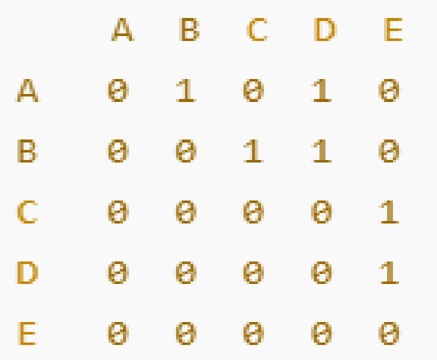
The above diagram represents a directed graph with vertices A, B, C, D, and E connected by directed edges. Below are the different ways to display the structure of the graph in a readable format.

To display the graph's structure, including its vertices and edges, in a way that is easy to understand and debug.

The format of the output depends on the graph's chosen representation:

1. Adjacency Matrix:

Display the graph as a table where rows and columns represent vertices, and each cell indicates the presence (1) or absence (0) of a directed edge.



	A	B	C	D	E
A	0	1	0	1	0
B	0	0	1	1	0
C	0	0	0	0	1
D	0	0	0	0	1
E	0	0	0	0	0

Adjacency Matrix

2. Adjacency List:

- Print each vertex followed by the list of vertices it is directly connected to.

The adjacency list for the above given graph is

A:B

B:C,D

C:E

E : D

D:A

3. Edge List:

- Print all edges as pairs of vertices, optionally including weights (if applicable).

Edge list for the given graph is

(A,B)(D,A)(B,C)(B,D)(C,E)(E,D)

These basic operations provide the foundation for working with graphs. They allow for initialization, management, traversal, and visualization of graph data, enabling the development of more complex algorithms and applications.

13.6 KEY TERMS:

Vertex, Edge, Degree, Path, Cycle, Directed Graph, Undirected Graph, Weighted Graph, Adjacency Matrix, Adjacency List.

13.7 REVIEW 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 Dinesh Mehta (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 in C++).

Dr. U. Surya Kameswari

LESSON-14
GRAPH TRAVERSALS

OBJECTIVES:

The objectives of this lesson are to

1. Understand the concepts of graph traversal methods, including Depth-First Search (DFS) and Breadth-First Search (BFS).
2. Learn how to implement graph traversal algorithms using adjacency lists and adjacency matrices in C++.
3. Explore the applications of graph traversals in solving problems like pathfinding, connectivity analysis, and cycle detection.
4. Analyze the time and space complexity of graph traversal techniques and understand their performance in various scenarios.

STRUCTURE:

14.1 INTRODUCTION:

Graph traversal is a fundamental concept in computer science, where we systematically explore all the vertices and edges of a graph. It is an essential operation in graph theory, widely used in applications like network routing, web crawling, and social network analysis.

Traversal methods like Depth-First Search (DFS) and Breadth-First Search (BFS) help in discovering paths, detecting cycles, and solving real-world problems efficiently. By understanding graph traversals, learners gain the ability to manipulate and analyze graph-based data structures effectively, making it a crucial topic in data structures and algorithms.

14.2 SIGNIFICANCE OF GRAPH TRAVERSALS:

Graph traversals are fundamental operations in computer science and graph theory, offering systematic ways to explore and process graph structures. Their importance stems from their wide applicability and foundational role in solving complex problems. The key significance of graph traversals includes:

1. **Foundation for Graph Algorithms**
   - Graph traversals like Depth-First Search (DFS) and Breadth-First Search (BFS) form the backbone of many advanced graph algorithms, including:
     - **Shortest Path Algorithms** (e.g., Dijkstra’s Algorithm).
     - **Spanning Tree Algorithms** (e.g., Prim's and Kruskal's Algorithms).
     - **Topological Sorting** in directed acyclic graphs (DAGs).
2. **Real-World Applications**
   - Graph traversal techniques are integral to solving real-world problems such as:
     - **Network Routing**: Finding optimal routes in computer networks.
     - **Social Network Analysis**: Discovering communities or connections.
     - **Web Crawling**: Systematically navigating interconnected web pages.
     - **AI and Games**: Finding paths or making decisions in game maps and AI systems.
3. **Understanding Graph Properties**
   - Traversals help identify critical graph characteristics, such as:
     - **Connectivity**: Whether all vertices are reachable.
     - **Cycles**: Detecting circular dependencies in graphs.
     - **Components**: Discovering connected or strongly connected components.
4. **Efficient Data Processing**
   - Traversals allow efficient data search and organization, making them crucial in domains like:
     - Database query optimization.
     - Resource allocation and scheduling.
     - Computational biology for gene interaction analysis.

5. **Algorithmic Problem Solving**
   - Graph traversals enhance algorithmic thinking, enabling students and professionals to design efficient solutions to complex problems. They also teach critical concepts like recursion (DFS) and iterative processing (BFS).

14.3 TYPES OF TRAVERSALS:

Graph traversal refers to the process of visiting all the vertices and edges of a graph in a systematic way. It is a fundamental concept in graph theory and is used to explore the structure and properties of a graph. There are two primary types of graph traversals:

1. **Depth-First Search (DFS)**: In DFS, the traversal starts from a chosen vertex and explores as far as possible along each branch before backtracking. It uses a stack (either explicitly or via recursion) to keep track of the vertices.
2. **Breadth-First Search (BFS)**: In BFS, the traversal starts from a chosen vertex and explores all its neighbors before moving to the next level. It uses a queue to manage the order of exploration.

14.4 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.

14.4.1 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**.**

14.4.2 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.

14.4.3 Algorithm for DFS

Here is the recursive implementation of DFS:

```
// Function to perform Depth First Search
void dfs (int vertex, int graph[MAX][MAX], bool visited[], int n) {
      // Mark the current vertex as visited
   visited[vertex] = true;
      // Print the current vertex
   cout << vertex << " ";
      // Explore all adjacent vertices
   for (int i = 0; i < n; i++) {
      // If there is an edge from 'vertex' to 'i' and 'i' is not visited
      if (graph[vertex][i] == 1 && !visited[i]) {
         // Recursively call DFS on the unvisited adjacent vertex 'i'
         dfs(i, graph, visited, n);
      }     }
```

The above code is explained as below

1. **Purpose**:
   - Implements the Depth-First Search (DFS) algorithm using recursion to traverse a graph represented by an adjacency matrix.
2. **DFS Function**:
   - Takes the following parameters:

    - vertex: The current vertex being explored.
    - graph: The adjacency matrix of the graph.
    - visited: An array to track visited vertices.
    - n: The total number of vertices.
  - Marks the current vertex as visited and prints it.
  - Recursively explores all unvisited adjacent vertices of the current vertex.

3. **Graph Representation**:
   - Uses an adjacency matrix where:
     - graph[i][j] = 1 indicates an edge between vertices i and j.
     - graph[i][j] = 0 indicates no edge.
4. **Visited Array**:
   - A boolean array (visited[]) ensures that each vertex is visited only once to prevent infinite loops.
5. **Main Function**:
   - Initializes the adjacency matrix representing the graph.
   - Initializes the visited[] array to track visited vertices (set to false initially).
   - Calls the dfs function starting from vertex 0.
6. **Traversal Process**:
   - The DFS function recursively explores the graph, visiting each vertex and printing them in the order of traversal.
   - The recursion automatically backtracks when no unvisited neighbors are left.
7. **Output**:
   - Prints the sequence of visited vertices in the order they are traversed.
8. **Key Features**:
   - Suitable for both directed and undirected graphs if represented correctly in the adjacency matrix.
   - Demonstrates the simplicity and efficiency of using recursion for graph traversal.

14.4.4 DFS Traversal Step-by-Step: Example

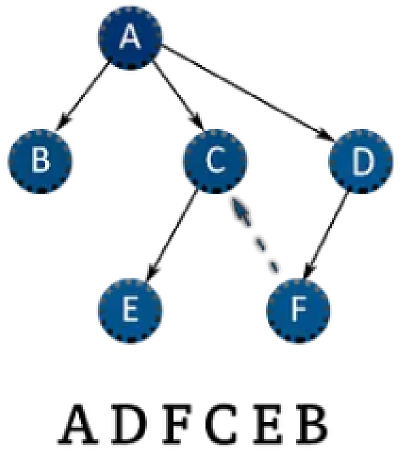


Fig. 14.1. Given Graph for DFS traversal

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.

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:

$$A \rightarrow B \rightarrow C \rightarrow E \rightarrow F \rightarrow D$$

14.4.6 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.

14.4.7 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.

14.5 BREADTH FIRST SEARCH (BFS):

Breadth First Search (BFS) is one of the simplest and most widely used graph traversal algorithms. It explores a graph level by level, visiting all the vertices at the current level (or depth) before moving to the vertices at the next level. This makes BFS ideal for tasks like finding the shortest path in unweighted graphs or exploring all connected components of a graph.

Working of BFS

1. **Starting Point**: BFS begins at a starting vertex, also called the source vertex.
2. **Queue-Based Traversal**: BFS uses a queue data structure to maintain the order in which vertices are visited.
    - A queue follows the First In, First Out (FIFO) principle, meaning the first element added is the first to be processed.
3. **Marking Vertices**: A visited array is used to keep track of which vertices have already been visited, ensuring that no vertex is processed more than once.
4. **Exploring Neighbors**: From the source vertex, BFS visits all its directly connected vertices (its neighbors) and adds them to the queue.
5. **Repeat Until Complete**: The process is repeated for each vertex in the queue, visiting their neighbors and adding unvisited ones to the queue until all reachable vertices have been explored.

14.5.1 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.

14.5.2 Implementation of BFS

```
void bfs(int start, int graph[MAX][MAX], bool visited[], int n) {
   int queue[MAX], front = 0, rear = 0;  // Initialize the queue
      // Mark the starting vertex as visited and enqueue it
   visited[start] = true;
   queue[rear++] = start;
   // Loop while the queue is not empty
   while (front < rear) {
      int current = queue[front++];    // Dequeue the front vertex
      cout << current << " ";          // Process the current vertex
      // Explore all adjacent vertices
      for (int i = 0; i < n; i++) {
         if (graph[current][i] == 1 && !visited[i]) {  // Check if adjacent and unvisited
            visited[i] = true;      // Mark as visited
            queue[rear++] = i;      // Enqueue the adjacent vertex
           }     }     }   }
```

The explanation of the code is represented as

1. **Function Parameters**:
   - start: The starting vertex for the BFS traversal.
   - graph[MAX][MAX]: The adjacency matrix representation of the graph.
   - visited[]: A boolean array to track whether a vertex has been visited.
   - n: The total number of vertices in the graph.
2. **Queue Initialization**:
   - queue[MAX]: The array is used to implement the queue, where vertices are stored for exploration.
   - front and rear: Indices for the queue to track the front and rear of the queue.
   - Initially, front = 0 and rear = 0.

3. **Mark Starting Vertex as Visited**:
   - The starting vertex (start) is marked as visited with visited[start]=true.
   - It is then enqueued into the queue using queue[rear++]=start.
4. **Processing Vertices in the Queue**:
   - The while loop runs as long as there are elements in the queue (front < rear).
   - At each step:
     - The vertex at the front of the queue is dequeued using current= queue[front++].
     - The current vertex is processed (printed to the console).
5. **Exploring Adjacent Vertices**:
   - For each vertex i adjacent to the current vertex (checked using

     graph[current][i] = = 1):
   - If the vertex i has not been visited (!visited[i]), it is:
     1. Marked as visited with visited[i] = true.
     2. Enqueued into the queue with queue[rear++] = i.
6. **Main Function**:
   - Defines the adjacency matrix representation of the graph.
   - Initializes the visited[] array to false.
   - Calls the bfs() function starting from vertex 0.
   - Prints the traversal order to the console.

14.5.3 Example

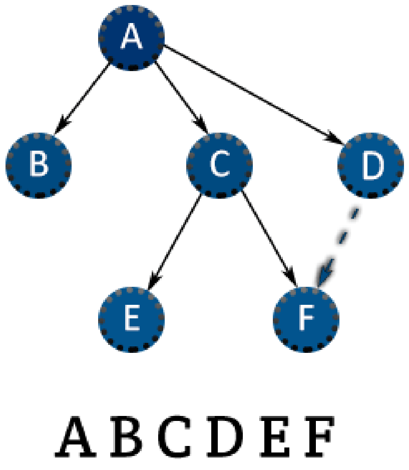


Fig. 14.2.Given Graph for BFS traversal

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 vertex A. 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.

14.5.4 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).

14.6 KEY TERMS:

Depth-First Search (DFS), Breadth-First Search (BFS), Adjacency Matrix, Adjacency List, Graph Traversal, Recursion, Queue.

14.7 REVIEW QUESTIONS:

1. What is the primary difference between Depth-First Search (DFS) and Breadth-First Search (BFS)?
2. How is recursion used in Depth-First Search?
3. Explain the purpose of a visited array in graph traversal algorithms?
4. What data structure is used in Breadth-First Search for maintaining the order of exploration?
5. Provide an example of a real-world application of graph traversals?

14.8 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. “Algorithm Design” by Jon Kleinberg and Éva Tardos.
4. “Data Structures Using C++” by D.S. Malik.

Dr. U. Surya Kameswari

LESSON-15

MINIMUM COST SPANNING TREES

OBJECTIVES:

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.

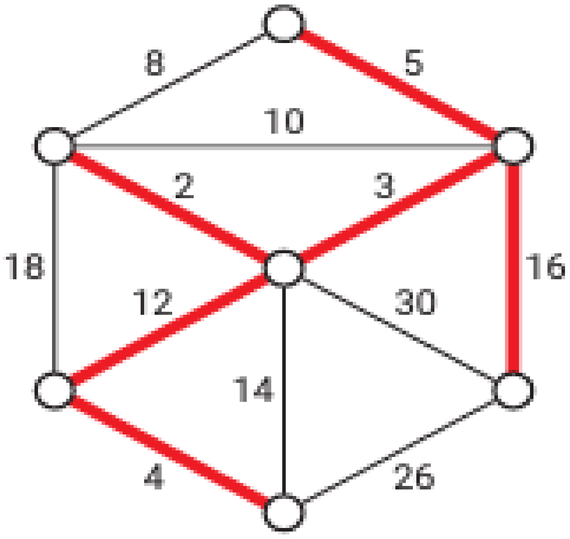


Fig. 15.1.A weighted graph and its minimum cost spanning tree

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

- **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

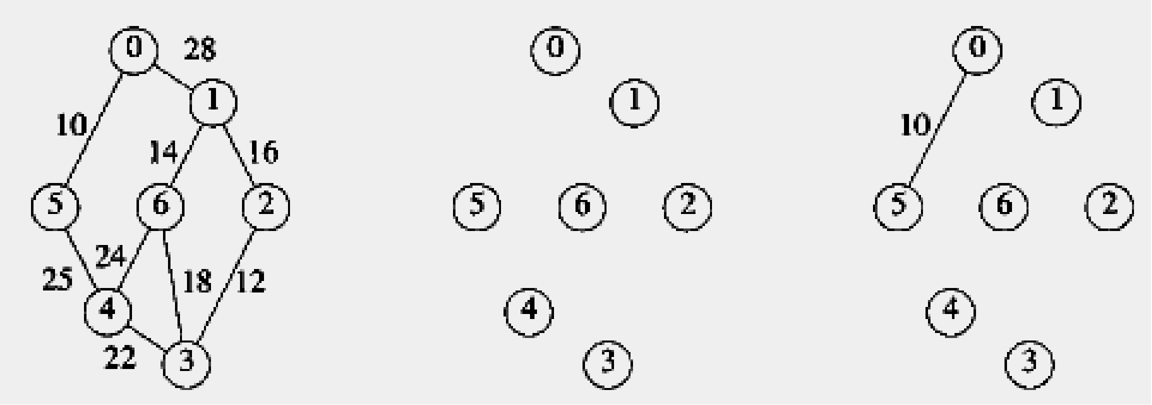
1. **Sort Edges**: All edges in the graph are sorted in non-decreasing order based on their weights.
2. **Initialise Forest**: Each vertex starts as its own independent tree (or set). The algorithm works by gradually merging these trees.

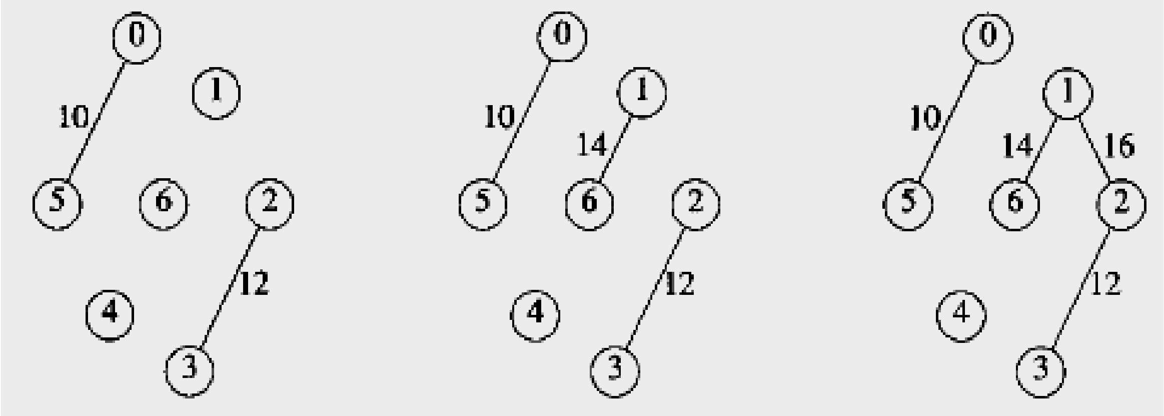
3. **Edge Selection**:
    - Traverse the sorted list of edges.
    - Add an edge to the MST if it connects two separate trees (i.e., if it does not form a cycle).
    - Use a **union-find data structure** to efficiently check for cycles and manage the merging of trees.
4. **Terminate**: The process 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.

15.3.4 Example of Kruskal’s Algorithm





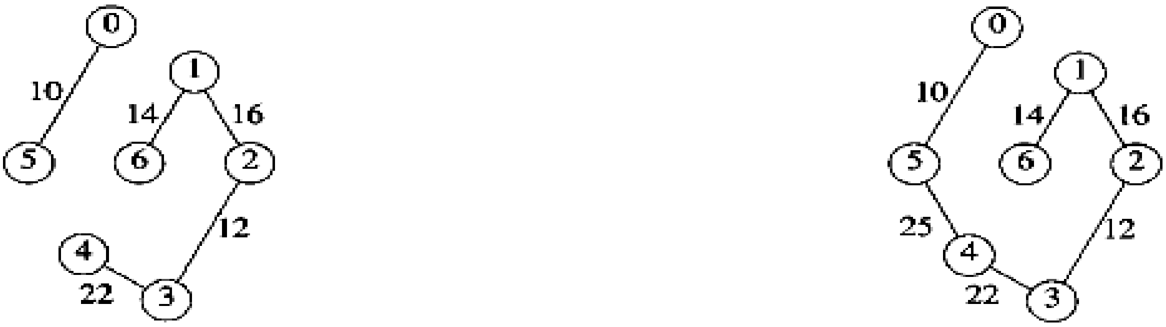


Fig. 15.2 Steps of Kruskal's Algorithm

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.

15.4.3 Example of Prim's Algorithm

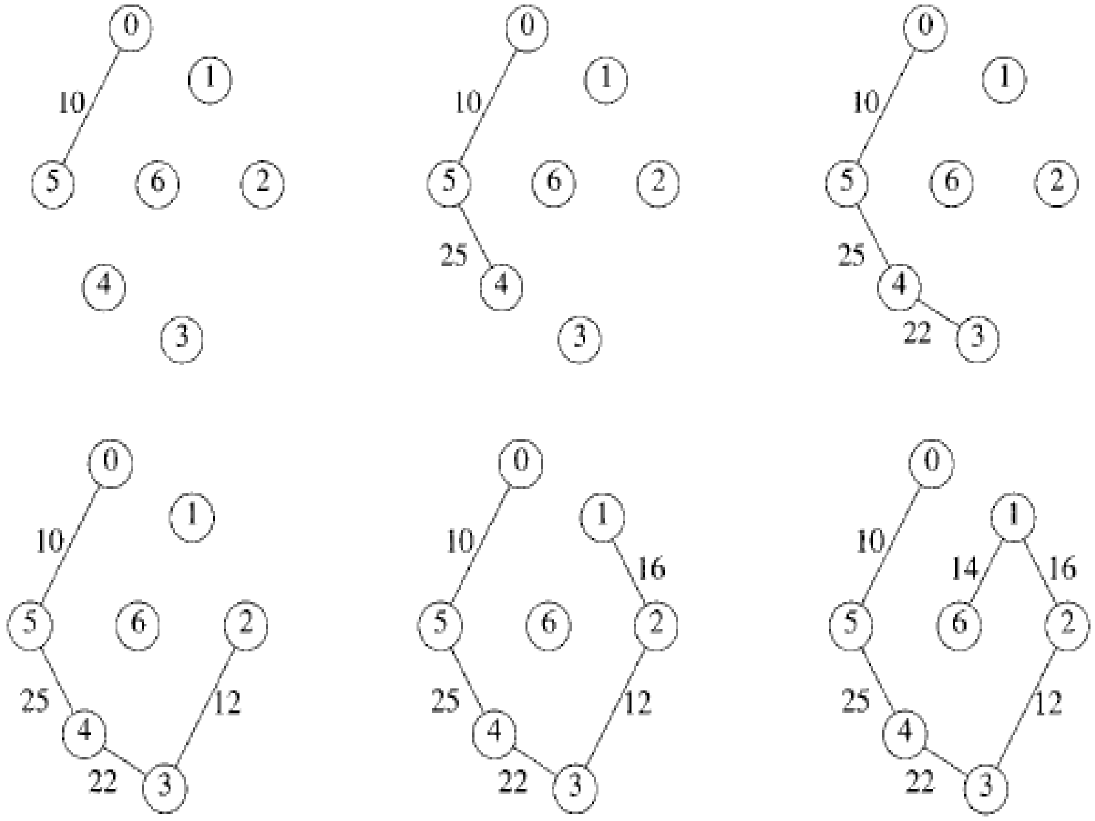


Fig. 15.3 Stages in Prim's Algorithm

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−1edges (where n 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 SHORTEST PATH:

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.

15.5.1 Key Algorithm for Shortest Path

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.

15.6 DIJKSTRA'S ALGORITHM:

Dijkstra's algorithm is one of the most widely used algorithms in computer science and mathematics for solving the shortest path problem in a graph. It was developed by the Dutch computer scientist Edsger W. Dijkstra in 1956 and is designed to compute the shortest paths from a single source node to all other nodes in a graph with non-negative edge weights**.**

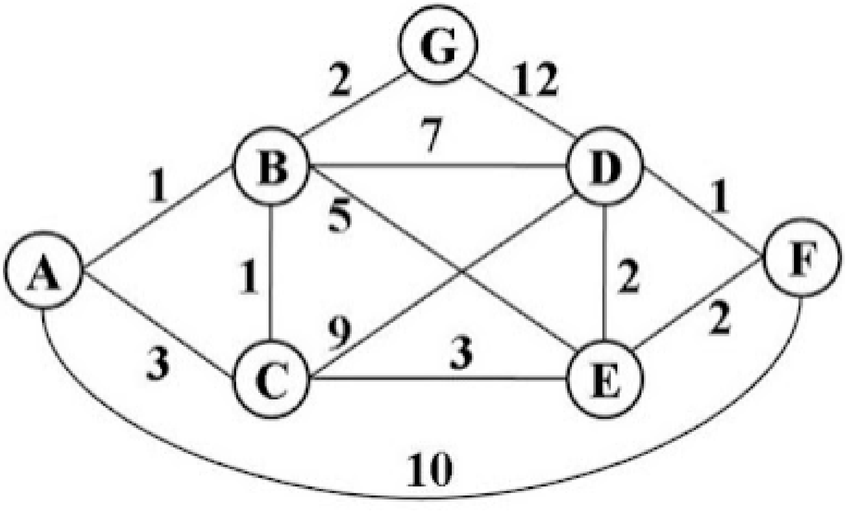


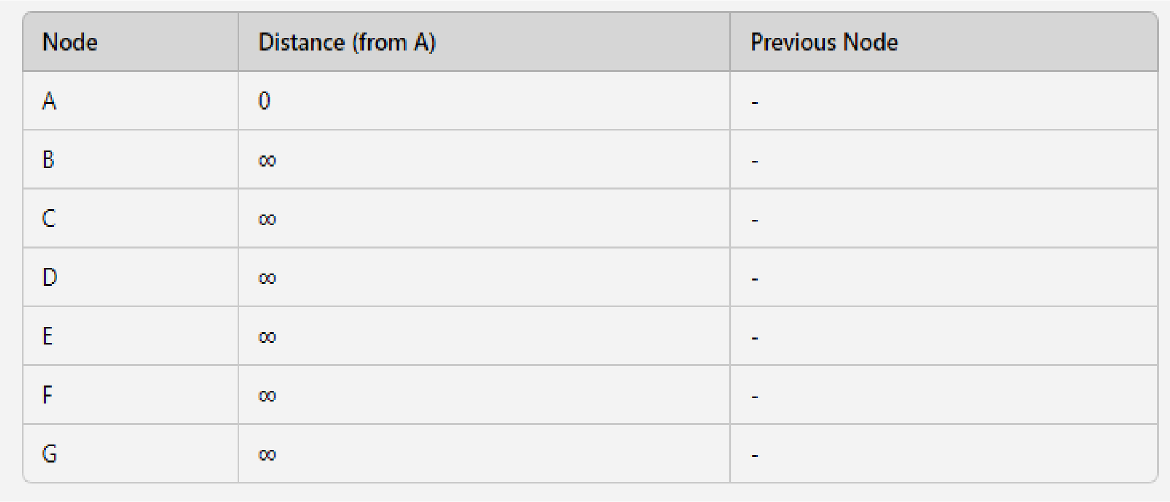
Fig. 15.4 Weighted Directed Graph

Step 1: Initialization

1. Set the distance to the source node (A) as 0.
2. Set the distance to all other nodes as infinity (∞) initially, as we have not yet visited them.

3. Keep a priority queue (or any data structure) to track the node with the smallest distance that hasn't been processed yet.
4. Mark all nodes as unvisited**.**

Initial Table:

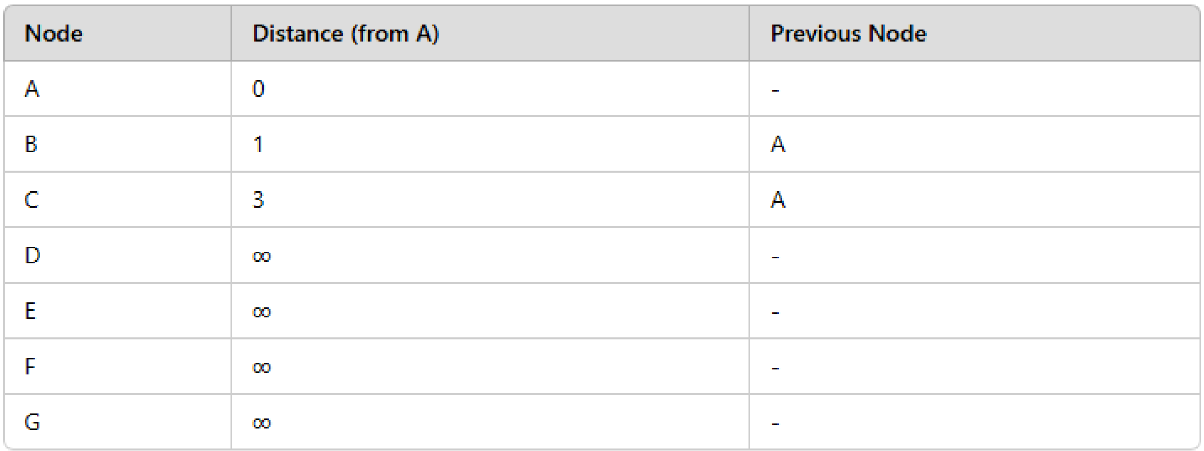


Node	Distance (from A)	Previous Node
A	0	-
B	∞	-
C	∞	-
D	∞	-
E	∞	-
F	∞	-
G	∞	-

Step 2: Process Node A

1. Start with node **A** (distance = 0).
2. Check all its neighbors: **B**, **C**.
   - For **B**: Distance = 0 + 1 = 1. Update the distance and previous node.
   - For **C**: Distance = 0 + 3 = 3. Update the distance and previous node.
3. Mark **A** as processed.

Updated Table:



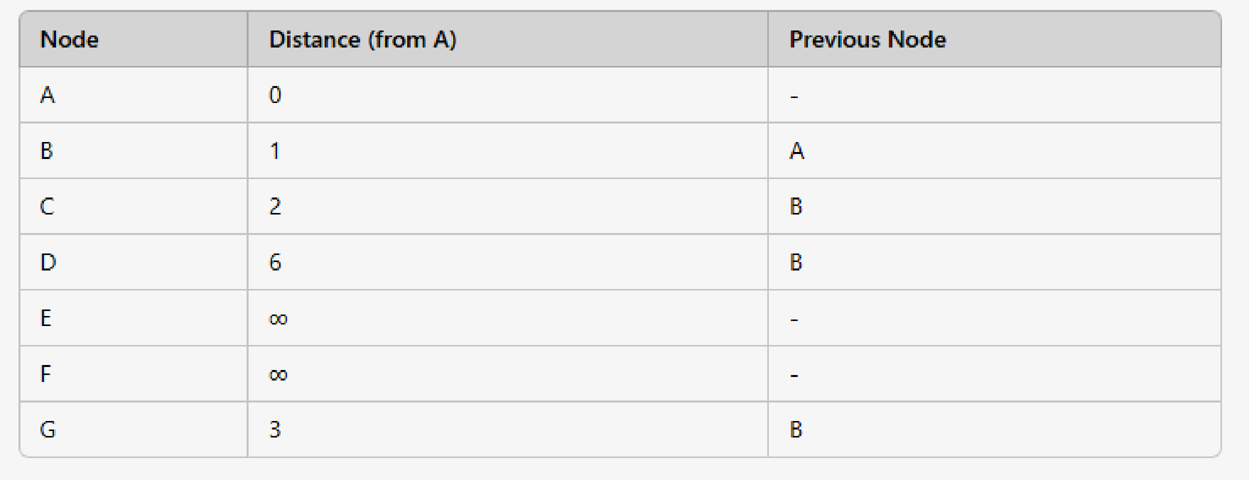
Node	Distance (from A)	Previous Node
A	0	-
B	1	A
C	3	A
D	∞	-
E	∞	-
F	∞	-
G	∞	-

Step 3: Process Node B

1. Pick the unvisited node with the smallest distance: **B** (distance = 1).
2. Check all its neighbors: **C**, **D**, **G**.

   - For **C**: Distance = 1 + 1 = 2. Update (smaller than the current distance of 3).
   - For **D**: Distance = 1 + 5 = 6. Update the distance and previous node.
   - For **G**: Distance = 1 + 2 = 3. Update the distance and previous node.
3. Mark **B** as **processed.**

Updated Table:

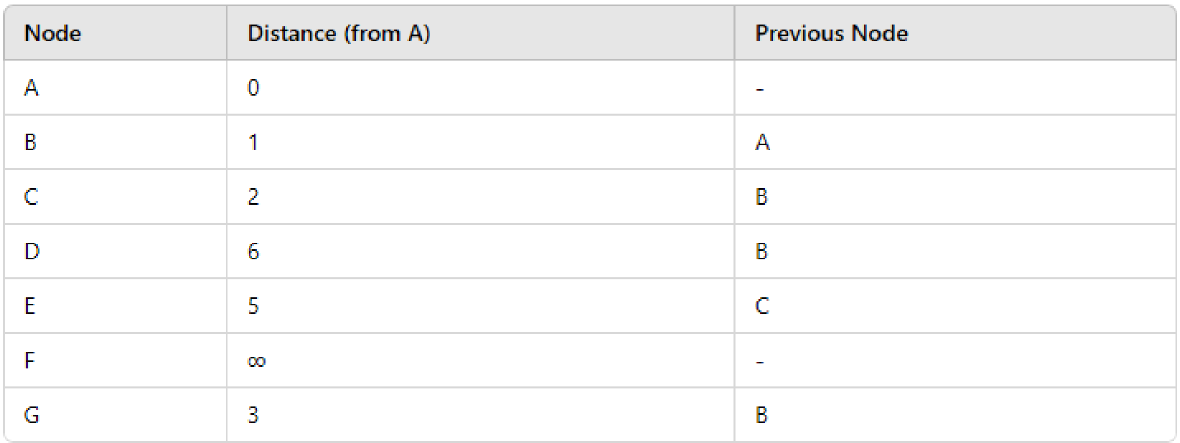


Node	Distance (from A)	Previous Node
A	0	-
B	1	A
C	2	B
D	6	B
E	∞	-
F	∞	-
G	3	B

Step 4: Process Node C

1. Pick the unvisited node with the smallest distance: **C** (distance = 2).
2. Check all its neighbors: **D**, **E**.
   - For **D**: Distance = 2 + 9 = 11. Do not update (current distance to D is 6, which is smaller).
   - For **E**: Distance = 2 + 3 = 5. Update the distance and previous node.
3. Mark **C** as processed.

Updated Table:



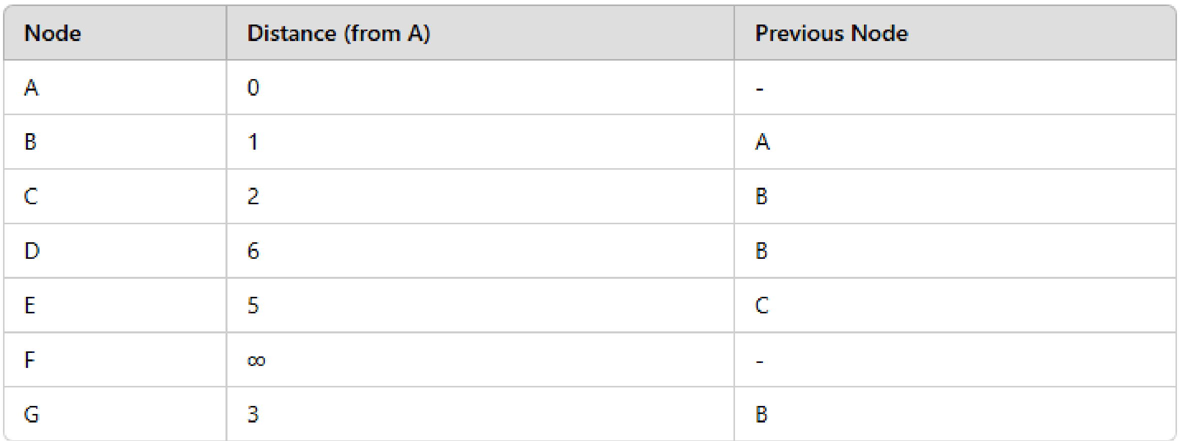
Node	Distance (from A)	Previous Node
A	0	-
B	1	A
C	2	B
D	6	B
E	5	C
F	∞	-
G	3	B

Step 5: Process Node G

1. Pick the unvisited node with the smallest distance: **G** (distance = 3).

2. Check all its neighbors: **D**.
   - For **D**: Distance = 3 + 7 = 10. Do not update (current distance to D is 6, which is smaller).
3. Mark **G** as processed.

Updated Table:

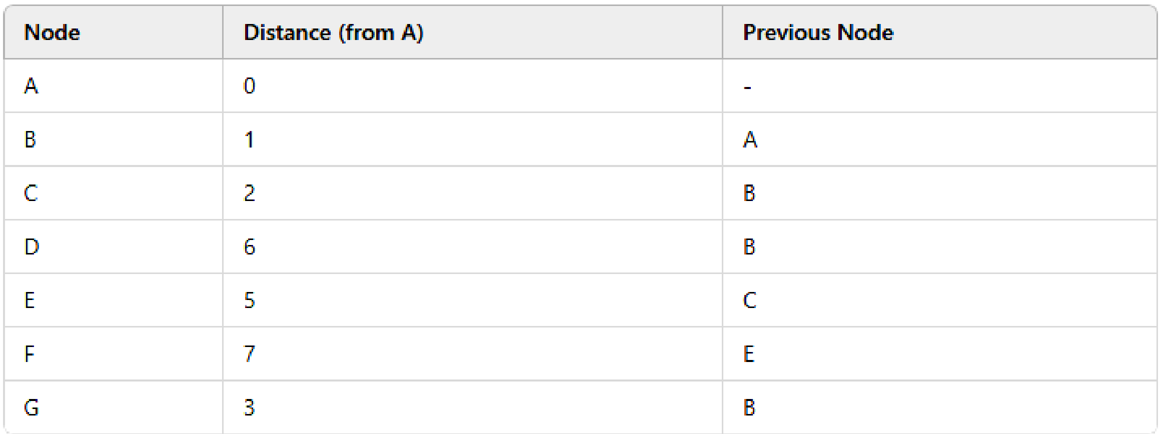


Node	Distance (from A)	Previous Node
A	0	-
B	1	A
C	2	B
D	6	B
E	5	C
F	∞	-
G	3	B

Step 6: Process Node E

1. Pick the unvisited node with the smallest distance: **E** (distance = 5).
2. Check all its neighbors: **D**, **F**.
   - For **D**: Distance = 5 + 2 = 7. Do not update (current distance to D is 6, which is smaller).
   - For **F**: Distance = 5 + 2 = 7. Update the distance and previous node.
3. Mark **E** as processed.

Updated Table:



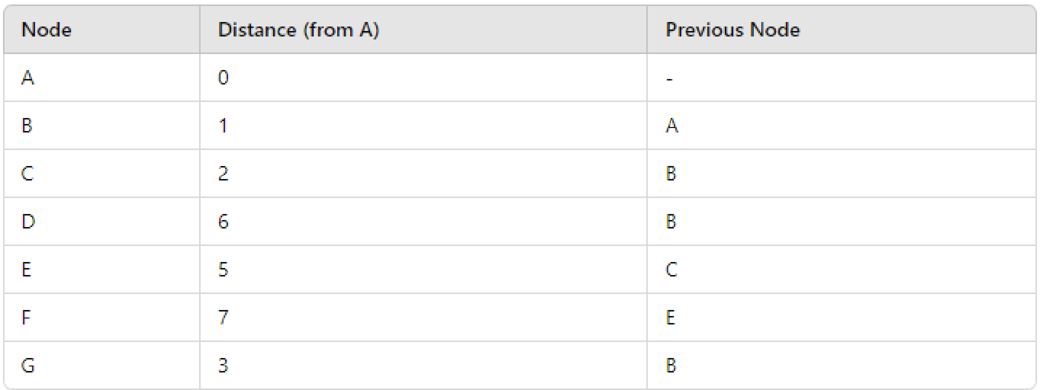
Node	Distance (from A)	Previous Node
A	0	-
B	1	A
C	2	B
D	6	B
E	5	C
F	7	E
G	3	B

Step 7: Process Node D

1. Pick the unvisited node with the smallest distance: **D** (distance = 6).

2. Check all its neighbors: **F**.
   - For **F**: Distance = 6 + 1 = 7. Do not update (current distance to F is 7, which is the same).
3. Mark **D** as processed.

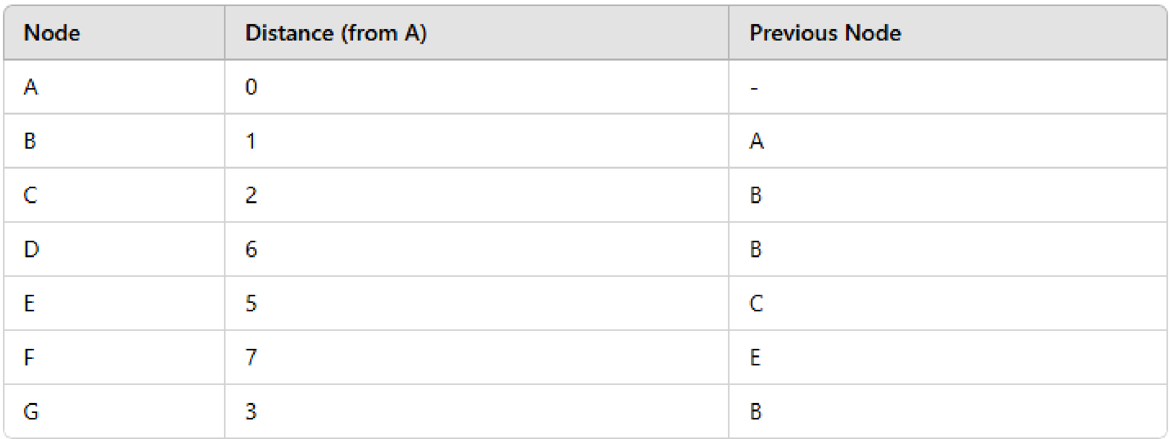
Updated Table:



Node	Distance (from A)	Previous Node
A	0	-
B	1	A
C	2	B
D	6	B
E	5	C
F	7	E
G	3	B

Step 8: Process Node F

1. Pick the unvisited node with the smallest distance: **F** (distance = 7).
2. **F** has no unvisited neighbors to check.
3. Mark **F** as processed.



Node	Distance (from A)	Previous Node
A	0	-
B	1	A
C	2	B
D	6	B
E	5	C
F	7	E
G	3	B

Shortest Paths:

1. **A** → **B**: Distance = 1
2. **A** → **C**: Distance = 2
3. **A** → **G**: Distance = 3
4. **A** → **E**: Distance = 5
5. **A** → **D**: Distance = 6
6. **A** → **F**: Distance = 7

This is how **Dijkstra's Algorithm** works step-by-step for the given graph

15.6.1 Steps of Dijkstra's Algorithm

Step 1: Initialize 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) log V)

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.

15.7 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

15.8 REVIEW 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?

15.9 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.
4) Research articles addressing application of Shortest Route algorithms & MST in the Operations Field & Computer Networks.
5) Dieter Jungnickel's “Graphs, Networks and Algorithms”: extensive discussions of the algorithms of graphs.

Dr. U. Surya Kameswari