

DEPARTMENT OF INFORMATION TECHNOLOGY & COMPUTER ENGINEERING



LECTURE NOTES

SUBJECT: DATA STRUCTURE

BRANCH: 3rd SEM, IT

FACULTY NAME: DR. ANITARANI BRAHMA

Department of Information Technology

Vision:

To focus on development of skilled and confident personalities of today and tomorrow by using cutting edge technology in the department of Information Technology to accept need based challenges with a sense of social responsibilities.

Mission:

- **M1:** To impart quality education by implementing state-of-the-art teaching-learning methods to enrich the academic competency, credibility and integrity of the students.
- **M2:** To implement the educational program in our department from fundamental engineering to recent technology as per emerging trends in the field of Information Technology.
- **M3:** To facilitate a conducive ambience and infrastructure to develop professional skills and nurture innovation in students.
- **M4:** To inculcate sensitivity towards society, respect for environment and promote high standards of ethics.

LEARNING OUTCOMES:

After completion of the course, the students will be able to:

- Define the basic terminologies and classifications of data structures.
- Explain the representation and operations of linear data structures like stacks, queues, and linked lists.
- Implement algorithms for sorting and searching using appropriate data structures.
- Analyze the time and space complexity of data structure operations and algorithm paradigms like dynamic programming and backtracking.
- Design solutions using advanced data structures for real-world applications, such as shortest path problems or flow-based algorithms.

UNIT-I

(INTRODUCTION TO DATA STRUCTURE)

Basic Terminology

Data: Data are raw, fact or value.

Information: Processed data.

Data Structure:

- Describes the way of organizing and storing the data in primary memory.
- Also it describes the operations associated with the data structure.

Abstract Data Types (ADT):

- The definition of ADT only mentions what operations are to be performed but not how these operations will be implemented.
- It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations.
- It is called “abstract” because it gives an implementation-independent view. The process of providing only the essentials and hiding the details is known as abstraction.

Classification of Data Structure:

Classification by Logical Structure

1. Linear Data Structures

In linear data structures, elements are arranged sequentially, and each element is connected to its previous and next element. Examples include:

- **Array:** A collection of elements stored at contiguous memory locations.
- **Linked List:** A sequence of nodes where each node contains data and a reference to the next node.
- **Stack:** Follows Last-In-First-Out (LIFO) principle; elements are added and removed from the top.
- **Queue:** Follows First-In-First-Out (FIFO) principle; elements are added at the rear and removed from the front.

2. Non-linear Data Structures

Non-linear data structures do not store data in a sequential manner. They are used to represent hierarchical relationships. Examples include:

- **Tree:** A hierarchical structure with a root node and child nodes, used to represent data with a parent-child relationship.
 - **Graph:** Consists of nodes (vertices) and edges, representing complex relationships between data points.
-

Classification by Memory Allocation

1. Static Data Structures

These have a fixed size, and memory is allocated at compile-time. Example:

- **Array:** Once declared, the size of an array cannot be altered during program execution.

2. Dynamic Data Structures

These can grow or shrink in size during program execution, with memory allocated at runtime. Examples:

- **Linked List:** Allows efficient insertion and deletion of elements.
- **Stack and Queue:** Can be implemented using linked lists to allow dynamic resizing.

Data Type:

- Describes the type of value, size and scope of the value.

Types of data type:

1. Primary/Basic/Built-in data type:

Type of data type	Type of value	Size (OS specific)	Format Specifier
int	Integer value:0,1,....n	2 bytes	%d
float	Real Value: 4.5, 7.8,.....	4 bytes	%f
double	Long Real Value: 110000.8, 99888.7	8 bytes	%lf
character	Character, symbol, numbers	1 byte per character	%c

2. Derived data type: Derived from primary data type

e.g. array, function, pointer

3. User Defined Data type: Created by user as per requirement

e.g. struct, union

Operations on Data structure:

Traversing :	Visiting each elements of the data structure only once is called traversing.
Searching :	Finding an element in a data structure that satisfies one or more conditions.
Inserting :	Adding elements of the same type in a data structure is called inserting.
Deleting :	Removing an element from a data structure is called deletion. 
Sorting :	Sorting the elements in a data structure in ascending and descending order is called sorting.
Merging :	Merging is the storage of elements located in two different files by combining them into one data file.

Algorithm:

Step wise computational procedure for solving a problem.

Flow Chart:

Graphical or pictorial representation of algorithm.

Trade-off of algorithm:

A **space–time** or **time–memory trade-off** in computer science is a case where an algorithm or program trades increased space usage with decreased time.

Here, *space* refers to the data storage consumed in performing a given task (RAM, HDD, etc), and *time* refers to the time consumed in performing a given task (computation time or response time).

Algorithm Analysis?

Algorithm analysis is the process of determining the **efficiency** of an algorithm in terms of:

- **Time Complexity:** How much time the algorithm takes to run.
- **Space Complexity:** How much memory the algorithm uses.

Asymptotic Analysis:

Asymptotic analysis is a method of describing the **behavior of an algorithm as the input size (n) becomes very large**.

It analyses how much time and space required by the algorithm.

It helps us understand the performance of an algorithm **independently of machine and compiler specifications**.

Asymptotic Notations:

- These are mathematical tools to represent time complexity. Asymptotic analysis allows for the comparison of algorithms' space and time complexities by examining their performance characteristics as the input size varies.

Notation	Name	Description
$O(n)$	Big-O	Upper bound: Worst-case time complexity. <i>It is the most widely used notation for Asymptotic analysis. It specifies the upper bound of a function. The maximum time required by an algorithm or the worst-case time complexity. It returns the highest possible output value(big-O) for a given input.</i>
$\Omega(n)$	Omega	Lower bound: Best-case time complexity. <i>Omega notation represents the lower bound of the running time of an algorithm. Thus, it provides the best case complexity of an algorithm.</i>
$\Theta(n)$	Theta	Tight bound: Average-case or exact time complexity. <i>It represents the upper and the lower bound of the running time of an algorithm,</i>

Types of Asymptotic Notations:

a) Big-O Notation (O)

- Describes the **worst-case** scenario.
- Shows the **maximum time** an algorithm can take.
- Example: If an algorithm takes at most n^2 steps for input size n , it is **$O(n^2)$** .

b) Omega Notation (Ω)

- Describes the **best-case** scenario.
- Shows the **minimum time** an algorithm can take.
- Example: **$\Omega(n)$** means it takes at least n steps.

c) Theta Notation (Θ)

- Describes the **average-case** or **exact** runtime.
 - It gives both the lower and upper bounds.
 - Example: **$\Theta(n \log n)$** means the algorithm always takes time proportional to $n \log n$.
-

Worst-Case Analysis

- It is the most commonly used method in algorithm analysis.
- Focuses on the **maximum time** an algorithm may take for any input of size n .
- Helps in **guaranteeing performance** even in the most difficult cases.

Example: Linear Search

- Worst-case: If the element is not in the list $\rightarrow$ we check all elements $\Rightarrow$ **$O(n)$** .
- Best-case: If the element is the first $\rightarrow$ **$\Omega(1)$** .

Example: Linear Search Algorithm

Suppose you have an array:

$A = [10, 20, 30, 40, 50]$

You want to find a number in the array.

Algorithm

```
LinearSearch(A, n, key)
1. for i = 0 to n-1
2.     if A[i] == key
3.         return i
4. return -1
```

Where:

- n = number of elements
- key = value to search

1. Best Case

Condition: The key is found at the **first position**.

Example:

```
Array = [10, 20, 30, 40, 50]  
Key = 10
```

Comparisons:

```
10 == 10 ✓
```

Only **1 comparison** is needed.

Time Complexity:

$O(1)$

2. Worst Case

Condition: The key is at the **last position** or **not present**.

Example:

```
Array = [10, 20, 30, 40, 50]  
Key = 50
```

Comparisons:

```
10 ✗  
20 ✗  
30 ✗  
40 ✗  
50 ✓
```

Total comparisons = **5**

If the key is **60** (not present):

```
10 ✗  
20 ✗  
30 ✗  
40 ✗  
50 ✗
```

Again, **5 comparisons**.

Time Complexity:

$$O(n)$$

3. Average Case

Condition: The key can be anywhere in the array.

Example:

Array = [10, 20, 30, 40, 50]

Possible comparisons:

Position Comparisons

1st 1

2nd 2

3rd 3

4th 4

5th 5

Average comparisons:

$$(1+2+3+4+5)/5=15/5=3$$

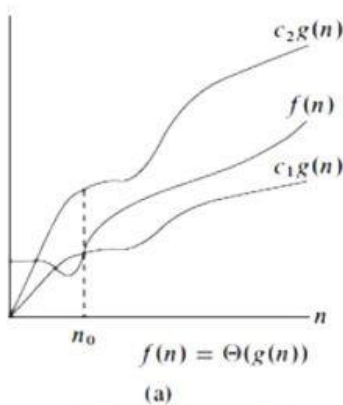
In general,

$$(n+1)/2$$

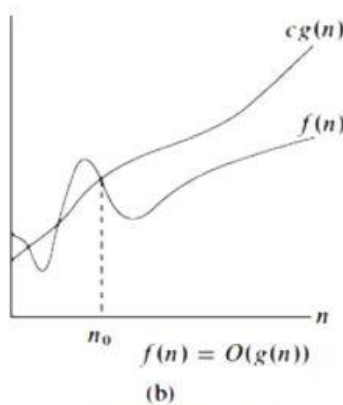
Ignoring constants, the time complexity is:

- $O(n)$

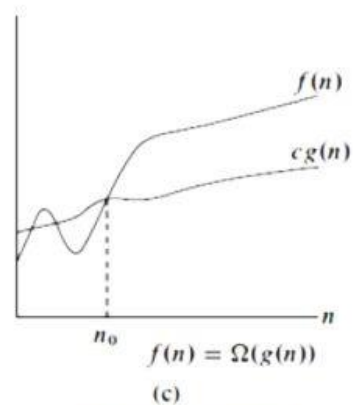
All Three Notations



Theta



Big O



Omega

Theta Notation: Let g and f be the function from the set of natural numbers to itself. The function f is said to be $\Theta(g)$, if there are constants $c_1, c_2 > 0$ and a natural number n_0 such that $c_1 * g(n) \leq f(n) \leq c_2 * g(n)$ for all $n \geq n_0$

Big-Oh Notation: If $f(n)$ describes the running time of an algorithm, $f(n)$ is $O(g(n))$ if there exist a positive constant C and n_0 such that, $0 \leq f(n) \leq c g(n)$ for all $n \geq n_0$

Theta Notation: Let g and f be the function from the set of natural numbers to itself. The function f is said to be $\Omega(g)$, if there is a constant $c > 0$ and a natural number n_0 such that $c * g(n) \leq f(n)$ for all $n \geq n_0$

CHAPTER-2

STACK:

- ❑ A **stack** is a **linear data structure** that follows the **LIFO (Last In First Out)** principle.
- ❑ It means the element that is **inserted last** is the **first to be removed**.
- ❑ Example: A stack of plates – the plate placed on top is removed first.
- ❑ **Linear structure**: Elements are arranged sequentially.
- ❑ **Restricted access**: Insertion and deletion are allowed only at **one end (top)**.
- ❑ **LIFO principle**: Last In, First Out.
- ❑ **Dynamic size**: Stack size can grow/shrink depending on implementation.

Basic Operations on Stack

1. **Push(x)**: Inserts an element x onto the stack.
2. **Pop()**: Removes the top element from the stack.

TERMINOLOGY USED IN STACK:

- **TOP**: Pointer points to the top of the stack.
- **Max_size/ size**: Maximum size of the stack.

Algorithms for Stack Operations

1. PUSH Operation (Insert an element)

- **Definition**: Adds a new element at the **top** of the stack.
- **Algorithm (using array representation)**:

Algorithm PUSH(stack, top, MAX, item)

1. If $top == MAX - 1$ then
 Print "Stack Overflow" // no more space
 Exit
 2. Else
 $top \leftarrow top + 1$
 $stack[top] \leftarrow item$
 Print "Item inserted"
 3. End If
- End Algorithm
-

2. POP Operation (Remove an element)

- **Definition:** Removes the element from the **top** of the stack.
- **Algorithm (using array representation):**

Algorithm POP(stack, top)

```
1. If top == -1 then
    Print "Stack Underflow" // stack is empty
    Exit
2. Else
    item ← stack[top]
    top ← top - 1
    Print "Deleted item is", item
3. End If
End Algorithm
```

Notes

- **Stack Overflow** → Trying to push an element when the stack is full.
- **Stack Underflow** → Trying to pop an element when the stack is empty.

ARRAY REPRESENTATION OF STACK:

```
#define SIZE 5
```

```
int stack[SIZE], top = -1;
```

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

```
int pop() {
    if (top == -1)
        printf("Stack Underflow\n");
    else
        return stack[top--];
}
```

Applications of Stacks:

- **Function calls:** Stacks are used to keep track of the return addresses of function calls, allowing the program to return to the correct location after a function has finished executing.
- **Recursion:** Stacks are used to store the local variables and return addresses of recursive function calls, allowing the program to keep track of the current state of the recursion.
- **Expression evaluation:** Stacks are used to evaluate expressions in postfix notation (Reverse Polish Notation).
- **Syntax parsing:** Stacks are used to check the validity of syntax in programming languages and other formal languages.
- **Memory management:** Stacks are used to allocate and manage memory in some operating systems and programming languages.

Arithmetic Expression Types

Arithmetic expressions are statement written using **operands (numbers, variables)** and **operators (+, -, ×, ÷, etc.)**.

They can be represented in **three main notations** depending on the position of the operator.

1. Infix Expression

- In this type, the **operator is written between operands**.
- This is the **standard form** of writing arithmetic expressions.
- **Examples:**
 - $A + B$
 - $(A + B) * C$
 - $X - Y / Z$

✓ **Advantage:** Easy to read and understand (like normal mathematics).

✗ **Disadvantage:** Requires parentheses and operator precedence rules to avoid ambiguity.

2. Prefix Expression (Polish Notation)

- In this type, the **operator is written before the operands**.
- No need for parentheses since the order of evaluation is fixed.
- **Examples:**
 - $+ A B$ (equivalent to $A + B$)
 - $* + A B C$ (equivalent to $(A + B) * C$)

✓ **Advantage:** Eliminates ambiguity (no precedence/parentheses required).

✗ **Disadvantage:** Difficult for humans to read and write.

Polish Notation is a parenthesis-free method of writing mathematical expressions where the operators precede their operands.

Invented by Polish logician Jan Łukasiewicz in 1924.

It is heavily used in computer science and data structures because it allows compilers to parse and evaluate expressions linearly without needing brackets or complex operator precedence rules.

3. Postfix Expression (Reverse Polish Notation)

- In this type, the **operator is written after the operands**.
- Like prefix, it does not need parentheses.
- **Examples:**
 - $A B +$ (equivalent to $A + B$)
 - $A B + C *$ (equivalent to $(A + B) * C$)

✓ **Advantage:** Easy to evaluate using a **stack** (commonly used in computers).

✗ **Disadvantage:** Less natural for humans to read.

Expression Type	Example (A=5, B=3, C=2)	Meaning
Infix	$(A + B) * C$	Normal mathematical form
Prefix	$* + A B C$	Operator before operands
Postfix	$A B + C *$	Operator after operands

- **Reverse Polish Notation (RPN)** In computer science, RPN is highly favored because it **eliminates the need for parentheses** and complex operator precedence rules (like PEMDAS/BODMAS). This allows computers to parse and execute expressions linearly using a **Stack** data

Algorithm: Infix to Postfix Conversion (Using Stack)

We use a **stack** to hold operators and parentheses while scanning the infix expression.

Steps

1. Initialize an empty stack for operators.
2. Scan the infix expression from **left to right**.

3. For each symbol:
 - **Operand (A–Z, 0–9):** Add it directly to the postfix expression.
 - **'(' (left parenthesis):** Push it onto the stack.
 - **')' (right parenthesis):** Pop from the stack and add to postfix until a left parenthesis (is encountered. Remove (from the stack.
 - **Operator (+, −, *, /, ^):**
 - While the stack is not empty and the **precedence** of the operator at the top of the stack is **greater than or equal to** the current operator, pop from the stack and add to postfix.
 - Push the current operator onto the stack.
 4. After scanning all symbols, pop and add all remaining operators from the stack to the postfix expression.
-

Operator Precedence & Associativity

1. $\wedge \rightarrow$ Highest precedence (Right to Left)
 2. $*, / \rightarrow$ Next (Left to Right)
 3. $+, - \rightarrow$ Lowest (Left to Right)
-

Example: Convert Infix to Postfix

Expression:

$(A + B) * C - D$

Step-by-Step Process:

1. Input ($\rightarrow$ Push (
 - Stack: (
 - Postfix:
2. Input A $\rightarrow$ Operand $\rightarrow$ Add to postfix
 - Stack: (
 - Postfix: A
3. Input + $\rightarrow$ Push +
 - Stack: (+
 - Postfix: A
4. Input B $\rightarrow$ Operand $\rightarrow$ Add to postfix
 - Stack: (+
 - Postfix: A B
5. Input) $\rightarrow$ Pop until (
 - Pop + $\rightarrow$ Add to postfix
 - Stack: (empty)
 - Postfix: A B +
6. Input * $\rightarrow$ Push *
 - Stack: *
 - Postfix: A B +

7. Input C $\rightarrow$ Operand $\rightarrow$ Add to postfix
 - Stack: *
 - Postfix: A B + C
8. Input - $\rightarrow$ Pop higher/equal precedence (*) $\rightarrow$ Add to postfix, then push -
 - Stack: -
 - Postfix: A B + C *
9. Input D $\rightarrow$ Operand $\rightarrow$ Add to postfix
 - Stack: -
 - Postfix: A B + C * D
10. End of expression $\rightarrow$ Pop remaining - $\rightarrow$ Add to postfix
 - Stack: (empty)
 - Postfix: A B + C * D -

Algorithm: Evaluate Postfix Expression

Algorithm Evaluate Postfix(Expression)

1. Initialize an empty stack
 2. For each symbol X in Expression (from left to right):
 - a. If X is an operand:
Push(X) onto the stack
 - b. Else if X is an operator:
Pop operand2 from stack
Pop operand1 from stack
result $\leftarrow$ operand1 operator operand2
Push(result) onto the stack
 3. After scanning all symbols:
The element left in the stack is the final result
- End Algorithm
-

Example: Evaluate Postfix Expression

Postfix Expression:

5 6 2 + * 12 4 / -

Step-by-Step Execution:

1. Read 5 $\rightarrow$ Operand $\rightarrow$ Push $\rightarrow$ Stack: [5]
2. Read 6 $\rightarrow$ Operand $\rightarrow$ Push $\rightarrow$ Stack: [5, 6]
3. Read 2 $\rightarrow$ Operand $\rightarrow$ Push $\rightarrow$ Stack: [5, 6, 2]
4. Read + $\rightarrow$ Operator $\rightarrow$ Pop 2, 6 $\rightarrow$ 6 + 2 = 8 $\rightarrow$ Push $\rightarrow$ Stack: [5, 8]
5. Read * $\rightarrow$ Operator $\rightarrow$ Pop 8, 5 $\rightarrow$ 5 * 8 = 40 $\rightarrow$ Push $\rightarrow$ Stack: [40]
6. Read 12 $\rightarrow$ Operand $\rightarrow$ Push $\rightarrow$ Stack: [40, 12]
7. Read 4 $\rightarrow$ Operand $\rightarrow$ Push $\rightarrow$ Stack: [40, 12, 4]
8. Read / $\rightarrow$ Operator $\rightarrow$ Pop 4, 12 $\rightarrow$ 12 / 4 = 3 $\rightarrow$ Push $\rightarrow$ Stack: [40, 3]
9. Read - $\rightarrow$ Operator $\rightarrow$ Pop 3, 40 $\rightarrow$ 40 - 3 = 37 $\rightarrow$ Push $\rightarrow$ Stack: [37]

Questions:

1. Evaluate the postfix expression: $23*54*+9-$
2. Convert the following infix to Postfix expression

$$A*(B+C)/D$$

3. Convert the following expression to Prefix Expression: $A+B*C-D$

QUEUE

- A **Queue** is a **linear data structure** that follows the **FIFO (First In, First Out)** principle.
- This means the **element inserted first** is the **element removed first**.
- Example: A queue at a ticket counter – the person who comes first gets served first.

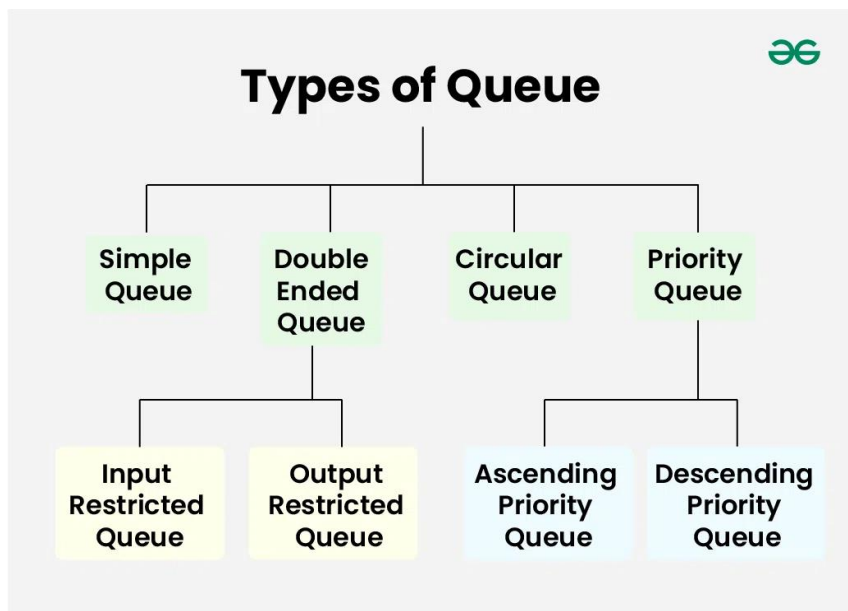
Characteristics of Queue

1. Linear data structure.
2. Follows **FIFO** principle.
3. Insertion happens at one end called **Rear**.
4. Deletion happens at the other end called **Front**.
5. Used when order of processing matters (first-come, first-served).

Types of Queues

Queue data structure can be classified into 4 types:

1. **Simple Queue:** Simple Queue simply follows **FIFO** Structure. We can only insert the element at the back and remove the element from the front of the queue. A simple queue is efficiently implemented either using a linked list or a circular array.
2. **Double-Ended Queue (Deque):** In a [double-ended queue](#) the insertion and deletion operations, both can be performed from both ends. They are of two types:
 - **Input Restricted Queue:** This is a simple queue. In this type of queue, the input can be taken from only one end but deletion can be done from any of the ends.
 - **Output Restricted Queue:** This is also a simple queue. In this type of queue, the input can be taken from both ends but deletion can be done from only one end.
3. **Priority Queue:** A [priority queue](#) is a special queue where the elements are accessed based on the priority assigned to them. They are of two types:
 - **Ascending Priority Queue:** In Ascending Priority Queue, the elements are arranged in increasing order of their priority values. Element with smallest priority value is popped first.
 - **Descending Priority Queue:** In Descending Priority Queue, the elements are arranged in decreasing order of their priority values. Element with largest priority value is popped first.



Applications of Queue

1. Scheduling processes in Operating Systems (CPU scheduling, Disk scheduling).
 2. Printer job management.
 3. Handling requests in a single shared resource (e.g., customer service systems).
 4. Data packet management in networks.
 5. Call center systems (managing customer calls in order).
-

Advantages

- Simple to implement.
- Maintains order (FIFO).
- Useful in real-world scenarios where fairness/order matters.

Disadvantages

- In a **simple queue (array-based)**, memory may be wasted.
- Requires more complex logic in circular and priority queues.

Algorithm Enqueue (queue, element)

Step 1: IF rear == MAX - 1

 PRINT "Queue is Full (Overflow)"

 EXIT

Step 2: IF front == -1 THEN

 front ← 0 // First element insertion

Step 3: rear ← rear + 1

Step 4: queue[rear] ← element

Step 5: PRINT "Element inserted successfully"

Algorithm Dequeue(Q)

Step 1: IF front = -1 OR front > rear THEN

 PRINT "Queue Underflow"

 EXIT

Step 2: item $\leftarrow$ Q[front]

Step 3: IF front = rear THEN

 front $\leftarrow$ -1

 rear $\leftarrow$ -1 // Queue becomes empty

ELSE

 front $\leftarrow$ front + 1

Step 4: PRINT "Deleted item is", item

Circular Queue

Introduction

- A **Queue** is a linear data structure that works on the **FIFO (First In First Out)** principle.
- In a **Simple Queue (Linear Queue)**, once the rear reaches the end of the array, we cannot insert more elements even if there is free space at the beginning.
- To overcome this limitation, we use a **Circular Queue**.
- **Definition:** A **Circular Queue** is a type of queue in which the last position is connected back to the first position, forming a **circle**.
- Insertion happens at the **rear**.
- Deletion happens at the **front**.
- Works on **FIFO principle**.
- Avoids **wastage of memory** compared to a simple queue.
- Uses **modulus (%) operator** to wrap around indices.
 - Example: rear = (rear + 1) % MAX

Operations on Circular Queue

a) Enqueue (Insert)

- Check if the queue is **full**.
- If not, insert the element at the rear position and update rear.

Condition for Full Queue (Overflow):

$(\text{front} == 0 \ \&\& \ \text{rear} == \text{MAX}-1) \text{ OR } (\text{front} == \text{rear}+1)$

b) Dequeue (Delete)

- Check if the queue is **empty**.
- If not, remove the element from the front and update front.

Condition for Empty Queue:

$\text{front} == -1$

c) Display

- Traverse the queue from front to rear in circular manner using modulus %.

Advantages

- Efficient utilization of memory.
- No wastage of space as in simple queue.
- Better performance in real-time systems.

6. Applications

- **Operating System** → CPU scheduling (Round Robin).
- **Traffic Systems** → Managing traffic lights in sequence.
- **Computer Networks** → Buffer management in data transfer.
- **Embedded Systems** → Handling multiple tasks in circular order.

Algorithm Enqueue(Q, item) (Insertion in circular queue)

Step 1: IF (front == 0 AND rear == MAX-1) OR (front == rear + 1) THEN

 PRINT "Queue Overflow"

 EXIT

Step 2: IF front == -1 THEN

 front $\leftarrow$ 0

 rear $\leftarrow$ 0

ELSE

 rear $\leftarrow$ (rear + 1) % MAX

Step 3: Q[rear] $\leftarrow$ item

Step 4: PRINT "Item inserted successfully"

Algorithm Dequeue(Q) (Deletion in Circular queue)

Step 1: IF front == -1 THEN

 PRINT "Queue Underflow"

 EXIT

Step 2: item $\leftarrow$ Q[front]

Step 3: IF front == rear THEN

 front $\leftarrow$ -1

 rear $\leftarrow$ -1 // Queue becomes empty

ELSE

 front $\leftarrow$ (front + 1) % MAX

Step 4: PRINT "Deleted item is", item

Double Ended Queue (Deque)

Introduction

- A **Queue** is a linear data structure that follows the **FIFO (First In First Out)** principle.
 - In a **Simple Queue**, insertion happens only at the **rear** and deletion only at the **front**.
 - In a **Double Ended Queue (Deque)**, insertion and deletion can be performed at **both ends** (front and rear).
-

2. Definition

A **Double Ended Queue (Deque)** is a linear data structure in which elements can be **inserted and deleted from both ends** (front and rear).

It provides more flexibility compared to a simple queue or circular queue.

Types of Deque

1. **Input-Restricted Deque**
 - Insertion is allowed **only at the rear**.
 - Deletion can be done from **both front and rear**.
 2. **Output-Restricted Deque**
 - Deletion is allowed **only at the front**.
 - Insertion can be done from **both front and rear**.
-

Basic Operations

1. **InsertFront(item)** → Insert an element at the **front** end.
 2. **InsertRear(item)** → Insert an element at the **rear** end.
 3. **DeleteFront()** → Remove an element from the **front** end.
 4. **DeleteRear()** → Remove an element from the **rear** end.
 5. **Display()** → Show all elements in the deque.
 6. **IsFull()** → Check if deque is full.
 7. **IsEmpty()** → Check if deque is empty.
-

Applications of Deque

- **Operating Systems** → Job scheduling, task management.
- **Memory Management** → Handling multiple memory blocks.
- **Palindrome Checking** → Used in algorithms for checking palindromes.
- **Sliding Window Problems** → Used in competitive programming and real-time data handling.
- **Undo/Redo Operations** → In text editors and software.

Advantages

- More flexible than simple queue and circular queue.
- Efficient for algorithms that require operations from both ends.
- Supports both **stack** and **queue** operations.

Algorithms of Deque Operations

Let the deque be implemented using an **array** with variables:

- front $\rightarrow$ index of the front element
- rear $\rightarrow$ index of the rear element
- MAX $\rightarrow$ maximum size of deque

1. Insert at Front (InsertFront)

Algorithm InsertFront(Q, item)

```
Step 1: IF (front == 0 AND rear == MAX-1) OR (front == rear+1) THEN
    PRINT "Deque Overflow"
    EXIT
Step 2: IF front == -1 THEN
    front  $\leftarrow$  0
    rear  $\leftarrow$  0
    ELSE IF front == 0 THEN
        front  $\leftarrow$  MAX-1
    ELSE
        front  $\leftarrow$  front - 1
Step 3: Q[front]  $\leftarrow$  item
Step 4: PRINT "Item inserted at front"
```

2. Insert at Rear (InsertRear)

Algorithm InsertRear(Q, item)

```
Step 1: IF (front == 0 AND rear == MAX-1) OR (front == rear+1) THEN
    PRINT "Deque Overflow"
    EXIT
Step 2: IF front == -1 THEN
    front  $\leftarrow$  0
    rear  $\leftarrow$  0
    ELSE IF rear == MAX-1 THEN
        rear  $\leftarrow$  0
    ELSE
        rear  $\leftarrow$  rear + 1
Step 3: Q[rear]  $\leftarrow$  item
```

Step 4: PRINT "Item inserted at rear"

3. Delete from Front (DeleteFront)

Algorithm DeleteFront(Q)

Step 1: IF front == -1 THEN

 PRINT "Deque Underflow"

 EXIT

Step 2: item $\leftarrow$ Q[front]

Step 3: IF front == rear THEN

 front $\leftarrow$ -1

 rear $\leftarrow$ -1

ELSE IF front == MAX-1 THEN

 front $\leftarrow$ 0

ELSE

 front $\leftarrow$ front + 1

Step 4: PRINT "Deleted item from front:", item

4. Delete from Rear (DeleteRear)

Algorithm DeleteRear(Q)

Step 1: IF front == -1 THEN

 PRINT "Deque Underflow"

 EXIT

Step 2: item $\leftarrow$ Q[rear]

Step 3: IF front == rear THEN

 front $\leftarrow$ -1

 rear $\leftarrow$ -1

ELSE IF rear == 0 THEN

 rear $\leftarrow$ MAX-1

ELSE

 rear $\leftarrow$ rear - 1

Step 4: PRINT "Deleted item from rear:", item

Round Robin Scheduling

- **Definition:**
Round Robin is a **CPU scheduling algorithm** where each process is assigned a fixed time slice (called **time quantum**).
 - Processes are kept in a **Ready Queue** (implemented using a Circular Queue).
 - The CPU executes each process for a time quantum in **cyclic order**.
-

Working of Round Robin

1. All processes are placed in a **ready queue**.
 2. The **CPU picks the first process** from the queue and executes it for **one time quantum**.
 3.
 - If the process **finishes within the time quantum**, it is removed from the queue.
 - If not finished, it is **moved to the rear** of the queue.
 4. The **CPU scheduler picks the next process** in the queue.
 5. The cycle continues until all processes are completed.
-

Example

Suppose **Time Quantum = 4 units**

Process Burst Time

P1 10 units

P2 4 units

P3 6 units

Execution order (Gantt Chart):

P1 (0–4) → P2 (4–8) → P3 (8–12) → P1 (12–16) → P3 (16–18) → P1 (18–20)

Advantages of Round Robin

- **Fair scheduling** → Every process gets equal share of CPU.
- **No starvation** → Each process is guaranteed execution.

- **Simple and easy** to implement using a queue.
 - **Responsive system** → Suitable for **time-sharing systems**.
-

Disadvantages

- **Context switching overhead** is high if time quantum is too small.
 - If time quantum is too large, it behaves like **First-Come First-Served (FCFS)** scheduling.
 - Average waiting time can be longer compared to other algorithms.
-

Applications of Round Robin

- **Operating Systems** → CPU scheduling in time-sharing systems.
- **Networking** → Data packet scheduling in routers.
- **Embedded Systems** → Managing tasks in real-time environments.
- **Multitasking Systems** → Ensures fairness among running processes.

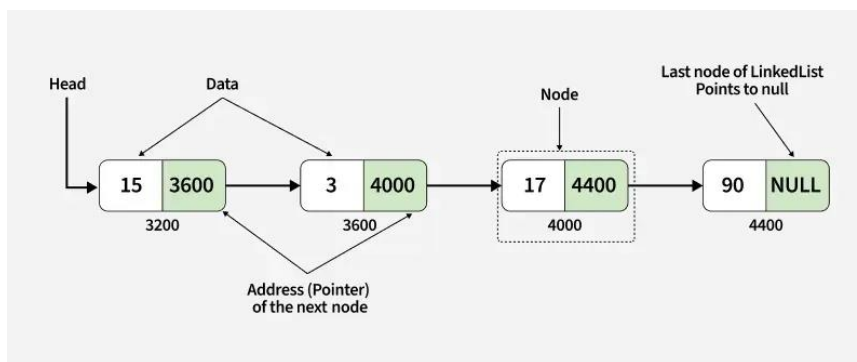
CHAPTER-3

LINKED LIST

.....

.....

- A linked list is a linear data structure, in which the elements are not stored at contiguous memory locations. The elements in a linked list are linked using pointers. In simple words, a linked list consists of nodes where each node contains a data field and a **reference(link)** to the next node in the list.
- It mainly allows efficient **insertion** and **deletion** operations compared to arrays. Like arrays, it is also used to implement other data structures like stack, queue and deque.



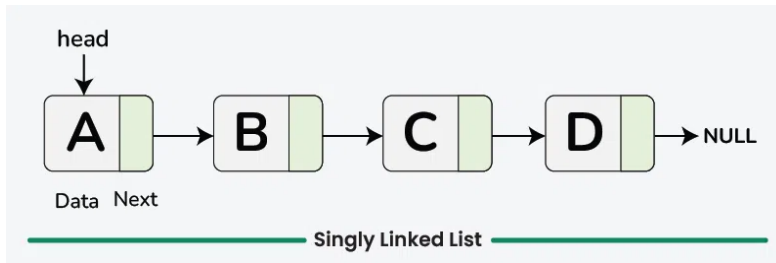
Linked List:

- **Data Structure:** Non-contiguous
- **Memory Allocation:** Typically allocated one by one to individual elements
- **Insertion/Deletion:** Efficient
- **Access:** Sequential

Types:

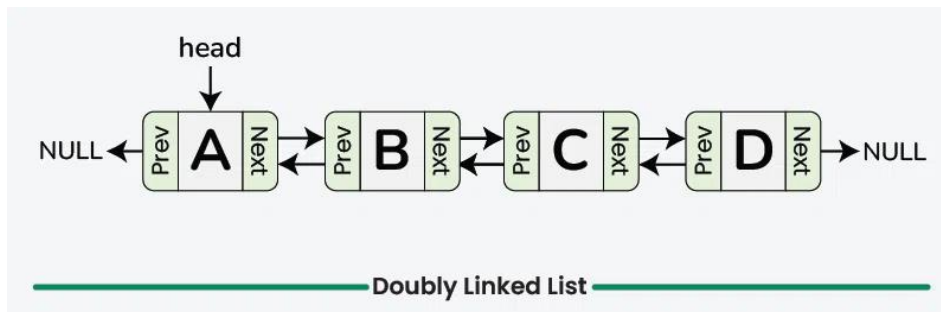
1. Singly Linked List

- Singly linked list is the simplest type of linked list in which every node contains some data and a pointer to the next node of the same data type.
- The node contains a pointer to the next node means that the node stores the address of the next node in the sequence.
- A single linked list allows the traversal of data only in one way.



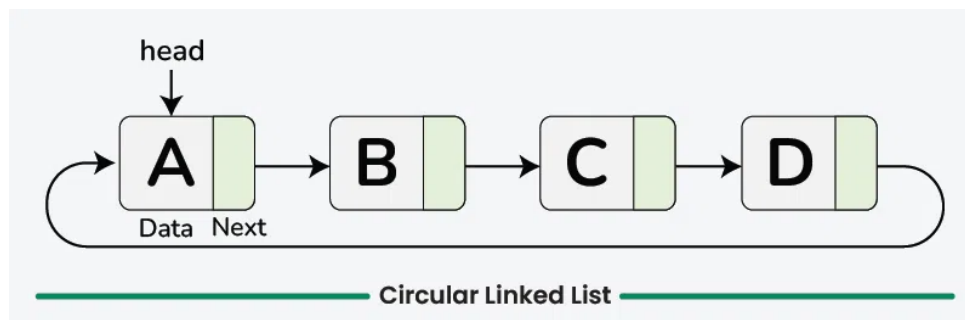
2. Doubly Linked List

- A doubly linked list or a two-way linked list is a more complex type of linked list that contains a pointer to the **next** as well as the **previous** node in sequence.
- Therefore, it contains three parts of data, a pointer to the **next** node, and a pointer to the **previous** node.
- This would enable us to traverse the list in the backward direction as well.



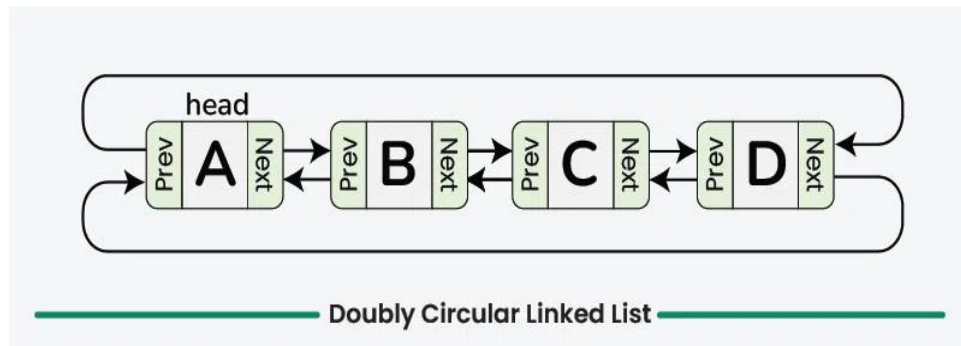
3. Circular Linked List

- A circular linked list is a type of linked list in which the last node's next pointer points back to the first node of the list, creating a circular structure.
- This design allows for continuous traversal of the list, as there is no null to end the list.
- While traversing a circular linked list, we can begin at any node and traverse the list in any direction forward and backward until we reach the same node we started. Thus, a circular linked list has no beginning and no end.



4. Doubly Circular linked list

- Doubly Circular linked list or a circular **two-way** linked list is a complex type of linked list that contains a pointer to the **next** as well as the **previous** node in the sequence.
- The difference between the **doubly linked** and **circular doubly list** is the same as that between a **singly linked list** and a **circular linked list**.
- The circular doubly linked list does not contain **null** in the **previous** field of the first node.



Algorithm for Creation of Linked List

Algorithm Steps

1. Start
2. Initialize head = NULL
3. Repeat the following steps for each new node to be created:
 - Allocate memory for the new node → newNode = (struct Node*)malloc(sizeof(struct Node))
 - Read the data → newNode->data = value
 - Set newNode->next = NULL
 - If head == NULL, then
 - Set head = newNode
 - Else
 - Traverse the list using a pointer temp until temp->next == NULL
 - Set temp->next = newNode
4. Repeat until the user wants to add more nodes
5. Stop

Result:

A linked list is created with all nodes connected sequentially.

? 2. Algorithm for Traversal of Linked List

Algorithm Steps

1. Start
2. If head == NULL
 - Print “List is empty” and stop
3. Initialize pointer temp = head
4. Repeat while temp != NULL
 - Print temp->data
 - Move temp = temp->next
5. Stop

Result:

All nodes' data are displayed one by one.

Algorithm: Insertion in a Singly Linked List

Types of insertion:

1. **At the beginning**
 2. **At the end**
 3. **At any given position**
-

□ Structure of a Node

Each node contains:

- **Data** – the value stored in the node
- **Next** – pointer to the next node

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

1. Insertion at the Beginning

Algorithm

1. Create a new node newNode.
2. Set newNode->data = value.
3. Set newNode->next = head.
4. Set head = newNode.

Result

The new node becomes the first node in the list.

2. Insertion at the End

Algorithm

1. Create a new node newNode.
2. Set newNode->data = value and newNode->next = NULL.
3. If head == NULL, set head = newNode.
4. Else,
 - o Traverse the list using a pointer temp until temp->next == NULL.
 - o Set temp->next = newNode.

Result

The new node is added after the last node.

Insertion at Any Position (Middle)

Algorithm

1. Create a new node newNode.
2. Set newNode->data = value.
3. Traverse the list using a pointer temp to reach the **(pos-1)th node**.
4. Set newNode->next = temp->next.
5. Set temp->next = newNode.

If the position is 1, it's the same as insertion at the beginning.

Result

The new node is inserted at the desired position.

Algorithm: Deletion in a Singly Linked List

Types of deletion:

1. **At the beginning**
 2. **At the end**
 3. **At any given position (middle)**
-

Structure of a Node

```
struct Node
{
```

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

1. Deletion at the Beginning

Algorithm

1. If head == NULL, print “List is empty” and stop.
2. Create a pointer temp = head.
3. Move head = head->next.
4. Free the memory of temp.

Result:

The first node is deleted, and the second node becomes the new head.

2. Deletion at the End

Algorithm

1. If head == NULL, print “List is empty” and stop.
2. If head->next == NULL,
 - o Free head and set head = NULL.
3. Else
 - o Create a pointer temp = head.
 - o Traverse the list until temp->next->next == NULL.
 - o Free temp->next.
 - o Set temp->next = NULL.

Result:

The last node is removed.

3. Deletion at Any Position (Middle)Algorithm

1. If head == NULL, print “List is empty” and stop.
2. If position = 1, delete the first node (same as beginning case).
3. Create a pointer temp = head.
4. Traverse the list to the **(pos-1)th node**.
 - o For i = 1 to pos-2, move temp = temp->next.
5. Create another pointer del = temp->next.
6. Set temp->next = del->next.
7. Free the memory of del.

Result:

The node at the specified position is deleted.

Stack Operation Using Linked List:

A **stack** is a linear data structure that follows the **LIFO (Last In, First Out)** principle. In a **linked list implementation**, the **top of the stack** is represented by the **head pointer** of the linked list.

Basic Stack Operations

1. **PUSH** → Insert an element at the top
2. **POP** → Delete an element from the top
3. **PEEK / TOP** → Display the top element
4. **DISPLAY** → Traverse and print all elements

Structure of a Node

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

1. PUSH Operation (Insert at Beginning)

Algorithm

1. Start
2. Create a new node newNode
3. Read value
4. Set newNode->data = value
5. Set newNode->next = top
6. Set top = newNode
7. Stop

Result:

The new element becomes the top of the stack.

2. POP Operation (Delete from Beginning)

Algorithm

1. Start
2. If top == NULL, print "Stack Underflow" and stop
3. Create pointer temp = top
4. Set top = top->next
5. Free memory of temp
6. Stop

Result:

The top element is removed from the stack.

? 3. PEEK Operation**Algorithm**

1. Start
2. If top == NULL, print “Stack is empty”
3. Else print top->data
4. Stop

Result:

Shows the value of the topmost element.

? 4. DISPLAY Operation**Algorithm**

1. Start
2. If top == NULL, print “Stack is empty”
3. Else
 - Create a pointer temp = top
 - Repeat while temp != NULL
 - Print temp->data
 - Move temp = temp->next
4. Stop

Result:

Displays all elements from top to bottom.

Queue Operation Using Linked List:

A **queue** is a linear data structure that follows the
□ **FIFO (First In, First Out)** principle.

In the **linked list implementation**,

- **Front** pointer represents the **first element (for deletion)**
- **Rear** pointer represents the **last element (for insertion)**

Structure of a Node

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

Two pointers are maintained:

```
struct Node *front = NULL;  
struct Node *rear = NULL;
```

Basic Queue Operations

1. **ENQUEUE (Insertion)** → Add an element at the rear
2. **DEQUEUE (Deletion)** → Remove an element from the front
3. **DISPLAY** → Show all elements from front to rear

🔍 ENQUEUE Operation (Insert at Rear)

Algorithm

1. Start
2. Create a new node newNode
3. Read the data value
4. Set newNode->data = value and newNode->next = NULL
5. If front == NULL and rear == NULL
 - Set front = rear = newNode
6. Else
 - Set rear->next = newNode
 - Set rear = newNode
7. Stop

Result:

A new node is added at the end of the queue.

DEQUEUE Operation (Delete from Front)

Algorithm

1. Start
2. If front == NULL
 - Print “Queue Underflow” and stop
3. Create a pointer temp = front
4. Set front = front->next
5. If front == NULL
 - Set rear = NULL
6. Free memory of temp
7. Stop

Result:

The first element is removed from the queue.

? 3. DISPLAY Operation

Algorithm

1. Start
2. If front == NULL
 - Print “Queue is empty” and stop
3. Create pointer temp = front
4. Repeat while temp != NULL
 - Print temp->data
 - Move temp = temp->next
5. Stop

Result:

All elements are displayed from front to rear.

CHAPTER-3

NON LINEAR DATA STRUCTURE

.....

.....

TREE

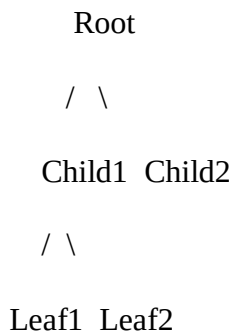
A **Tree** is a **non-linear data structure** that represents hierarchical relationships between elements.

It consists of nodes, where the topmost node is called the **root**, and every other node can have one or more **child nodes**.

A tree is an **acyclic connected graph**.

It consists of **nodes** connected by **edges**.

- The **topmost node** is called the **root**.
- Each node may have **child nodes**.
- Nodes without children are called **leaf nodes**.



Tree Terminology:

Term	Description
Node	Each element in a tree
Root	The topmost node of a tree
Parent	A node that has child nodes
Child	Node directly connected below a parent
Leaf Node	Node with no children
Edge	Link between parent and child
Siblings	Nodes having the same parent
Level	Depth of a node (root is level 0)
Height	Longest path from root to a leaf
Subtree	Any node and its descendants form a subtree

Types of Trees

a. General Tree

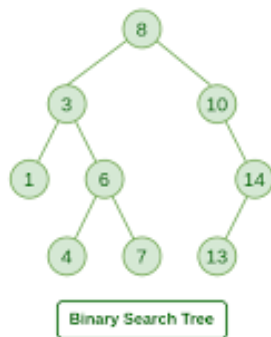
- Each node can have **any number of children**.

b. Binary Tree

- Each node can have **at most two children** — **left child** and **right child**.

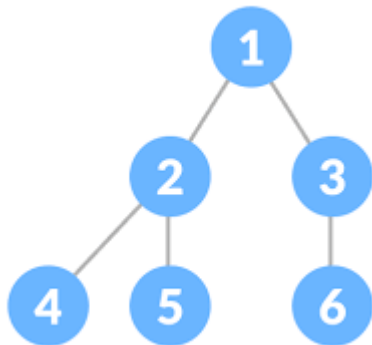
c. Binary Search Tree (BST)

- A **binary tree** where:
 - Left child < Parent node
 - Right child > Parent node
- Used for **efficient searching and sorting**.



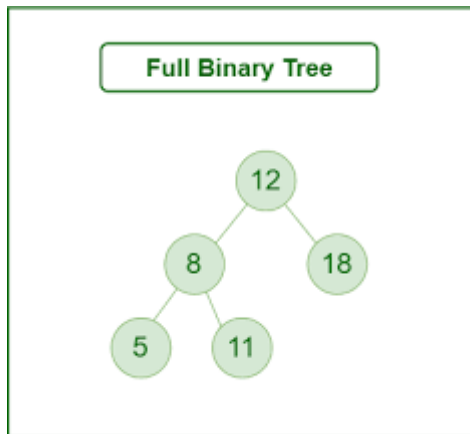
d. Complete Binary Tree

- All levels are completely filled except possibly the last level, filled from left to right.



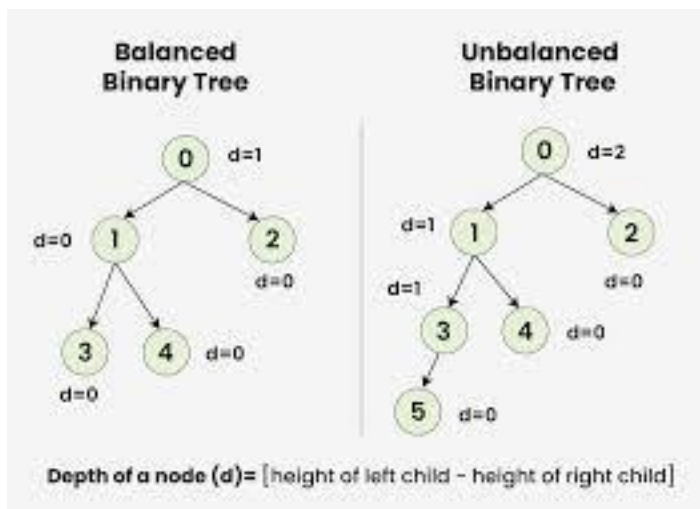
e. Full Binary Tree

- Every node has **0 or 2 children**.



f. Balanced Binary Tree

- The height difference between left and right subtrees is **at most 1**.

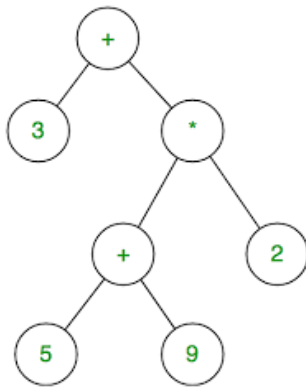


g. AVL Tree

- A **self-balancing** binary search tree.

h. Expression Tree

- Used to represent **arithmetic expressions**.
 - Leaves → Operands (numbers)
 - Internal nodes → Operators (+, -, ×, ÷)



Applications of Trees

- Represent **hierarchical data** (e.g., file system, organization chart)
- Used in **search operations** (BST, AVL tree)
- **Expression evaluation** (Expression Tree)
- **Routing algorithms** in networks
- **Decision-making systems** (Decision Trees)
- **Database indexing** (B-Trees, B+ Trees)

Properties of Tree Data Structure:

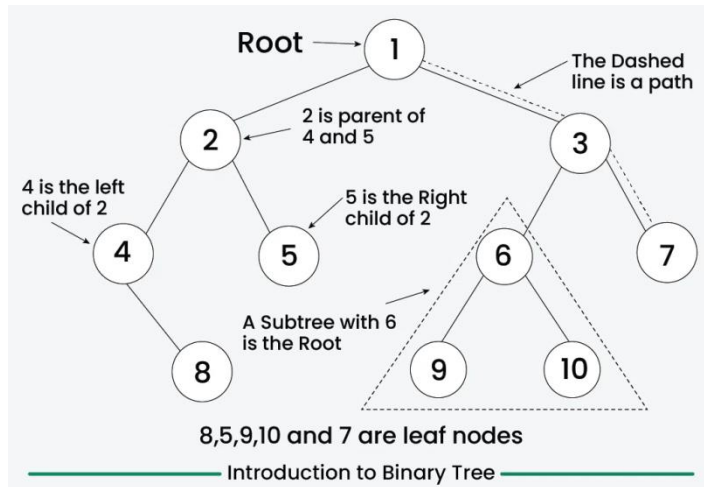
- **Number of edges:** An edge is the connection between two nodes. A tree with N nodes will always have $N - 1$ edges. There is exactly one path from any node to any other node in the tree.
- **Depth of a node:** The depth of a node is the length of the path from the root to that node. Each edge in the path adds 1 unit to the length. Equivalently, it is the number of edges from the root to the node.
- **Height of the tree:** The height of the tree is the length of the longest path from the root to any leaf node.
- **Degree of a node:** The degree of a node is the number of subtrees attached to it (i.e., the number of children it has).
A leaf node has a degree of 0.
The degree of the tree is the maximum degree among all nodes in the tree.

Definition and Concepts of Binary Trees:

A **Binary Tree** is a **non-linear data structure** in which each node has at most **two children** — called the **left child** and the **right child**.

Properties of Binary Tree

1. Each node can have **0, 1, or 2 children**.
2. The **maximum number of nodes** at level $L = 2^L$
3. The **maximum number of nodes** in a binary tree of height $h = 2^{(h+1)} - 1$
4. For a binary tree with n nodes, there are always $(n - 1)$ edges.
5. The **minimum height** of a binary tree with n nodes $= \lceil \log_2(n + 1) \rceil - 1$



Representations of a Binary Tree using Arrays and Linked Lists:

Binary Tree Representation in Memory

Binary trees can be represented in two ways:

a. Linked Representation

Each node has three fields:

- data — stores value
- left — pointer to left child
- right — pointer to right child

```
struct node
{
    int data;
    struct node *left;
    struct node *right;
};
```

Array Representation

- Root is stored at index 1.
- For any node at index i:
 - Left child = $2 * i$
 - Right child = $2 * i + 1$

```

      A
     /\
    B  C
   /\
  D  E
```

Stored as:

Index → 1 2 3 4 5

Value $\rightarrow$ A B C D E

Types of Binary Trees:

1. Binary Tree (Proper Binary Tree or Strictly Binary Tree)

- Every node in the tree has **either 0 or 2 children**.
- No node has only one child.
- All leaf nodes may or may not be at the same level.
- A full binary tree with n internal nodes has $(n + 1)$ leaf nodes.

A

/ \

B C

/ \

D E

2. Complete Binary Tree

- All levels are **completely filled**, except possibly the **last level**.
- The last level's nodes are filled **from left to right**.
- Efficiently represented using arrays.
- Commonly used in **heaps**.

A

/ \

B C

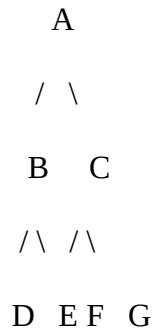
/ \ /

D E F

3. Perfect Binary Tree

- All internal nodes have **two children**.
- All **leaf nodes** are at the **same level**.
- Total number of nodes = $2^{(h+1)} - 1$, where h is height.

- Total number of leaf nodes = 2^h .



Operations on a Binary Tree-

Insertion in Binary Tree

Insertion in a binary tree can vary depending on the type (simple binary tree or binary search tree).

(a) Insertion in a Simple Binary Tree

- Nodes are filled **level by level** from **left to right**.
- A new node is added to the **first available position**.

Algorithm (Level Order Insertion):

1. If the tree is empty, create a new node as the root.
2. Otherwise, insert the new node in the first available left or right position while traversing level-wise.

(b) Insertion in a Binary Search Tree (BST)

- The **left child** contains a smaller value than the parent.
- The **right child** contains a greater value than the parent.

Algorithm (Insertion in BST):

1. Start from the root node.
2. If the new key < root → go to the **left subtree**.
3. If the new key > root → go to the **right subtree**.
4. Repeat until you find an empty position and insert the new node.

2. Deletion in Binary Tree

Deletion involves removing a node while maintaining the binary tree property.

Cases in Deletion:

1. **Node has no child (Leaf Node):**
 - Simply delete the node.
 2. **Node has one child:**
 - Remove the node and connect its child directly to the parent.
 3. **Node has two children:**
 - Find the **inorder successor** (smallest node in the right sub tree) or **inorder predecessor** (largest node in the left subtree).
 - Replace the value of the node with that of the successor/predecessor, and then delete that successor/predecessor node.
-

3. Traversal of Binary Tree

Traversal means visiting all nodes of the tree in a particular order.

There are **two main types** of traversals:

(A) Depth-First Traversal (DFT)

1. **Inorder Traversal (Left – Root – Right)**
 - Visit the left subtree
 - Visit the root
 - Visit the right subtree

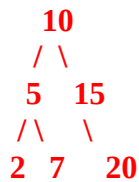
Example: Output is nodes in **sorted order** for BST.
 2. **Preorder Traversal (Root – Left – Right)**
 - Visit the root
 - Visit the left subtree
 - Visit the right subtree

Example: Used to **copy** or **create** the tree.
 3. **Postorder Traversal (Left – Right – Root)**
 - Visit the left subtree
 - Visit the right subtree
 - Visit the root

Example: Used to **delete** or **free memory** of the tree.
-

(B) Breadth-First Traversal (Level Order Traversal)

- Visit nodes **level by level**, from left to right.
- Implemented using a **queue**.



Traversals:

- **Inorder:** 2, 5, 7, 10, 15, 20
- **Preorder:** 10, 5, 2, 7, 15, 20
- **Postorder:** 2, 7, 5, 20, 15, 10
- **Level order:** 10, 5, 15, 2, 7, 20

GRAPHS:

Definition:

A **graph** is a non-linear data structure that consists of a set of **vertices (nodes)** and **edges (connections)** that link pairs of vertices.

Formally, a graph $G = (V, E)$ where:

- $V \rightarrow$ Set of vertices (nodes)
- $E \rightarrow$ Set of edges (connections between vertices)

1. Vertex (Node): A Vertex, often referred to as a **Node**, is a fundamental unit of a graph. It represents an **entity** within the graph. In applications like social networks, vertices can represent individuals, while in road networks, they can represent intersections or locations.

2.Edge: An Edge is a **connection between two vertices** in a graph. It can be either directed or undirected. In a directed graph, edges have a specific direction, indicating a one-way connection between vertices. In contrast, undirected graphs have edges that do not have a direction and represent bidirectional connections.

3. Degree of a Vertex: The Degree of a Vertex in a graph is the **number of edges incident** to that vertex. In a directed graph, the degree is further categorized into the in-degree (number of incoming edges) and out-degree (number of outgoing edges) of the vertex.

4.Path: A Path in a graph is a **sequence of vertices** where each adjacent pair is connected by an edge.

Example: Path $A \rightarrow B \rightarrow C$ means A is connected to C through B.

5.Cycle: A Cycle in a graph is a **path that starts and ends at the same vertex**, with no repetitions of vertices (except the starting and ending vertex, which are the same).

6.Connected Graph:

- A graph is **connected** if there is a **path between every pair of vertices**.
- In an **undirected graph**, all nodes are reachable from each other.

7.Disconnected Graph:

- A graph where **some vertices are not connected** by any path.

8. Subgraph:

- A **subgraph** is a portion of a graph that includes a **subset of its vertices and edges**.

9.Complete Graph:

- A **complete graph** is one in which **every vertex is connected to every other vertex** by a unique edge.
- For **n** vertices, number of edges = $n(n - 1) / 2$

10. Tree:

- A **tree** is a **special type of graph** that is **connected** and has **no cycles**.

11. Sparse and Dense Graphs:

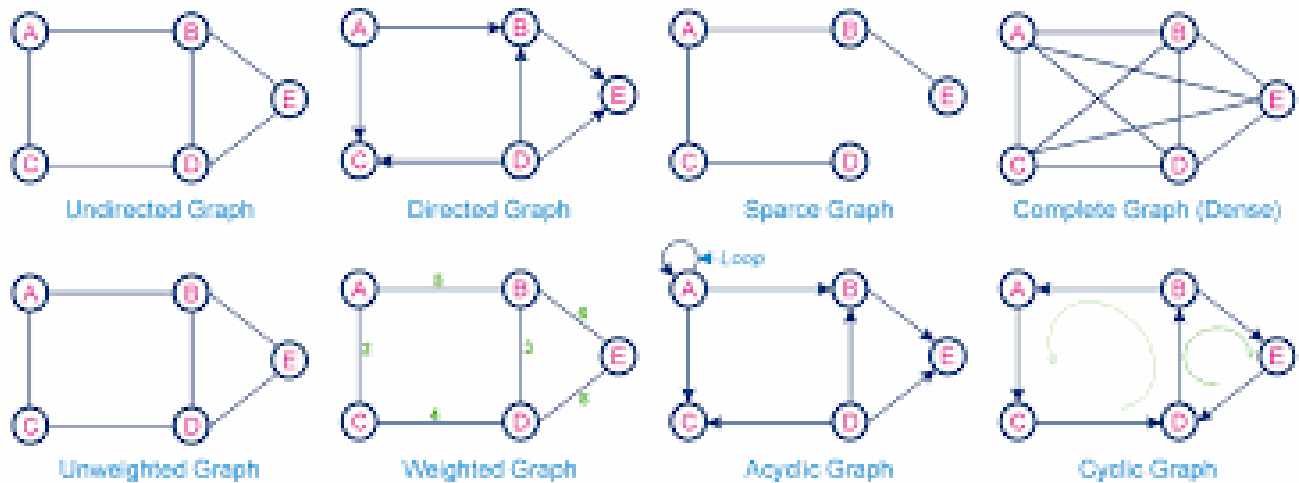
- **Sparse Graph:** Has **few edges** compared to the number of vertices.
- **Dense Graph:** Has **many edges**, close to the maximum possible.

12. Adjacency Matrix:

- A **2D array** used to represent a graph.
- If there is an edge between vertex i and j , then $A[i][j] = 1$, else 0.

13. Adjacency List:

- A **list-based representation** where each vertex stores a list of its adjacent vertices.
- It is **memory efficient** for sparse graphs.



Graph Representation Using Set:

A **graph** can be represented mathematically using **sets**.
It consists of two sets:

- A **set of vertices (V)**
- A **set of edges (E)**

Formally,

$$G = (V, E)$$

Where

- $V = \{\text{set of vertices (nodes)}\}$
- $E = \{\text{set of edges (connections between vertices)}\}$

1. Representation of an Undirected Graph

In an **undirected graph**, edges have **no direction**.

If there is an edge between vertex A and vertex B, it is written as **(A, B)** or **(B, A)**.

Example:

Let a graph have 4 vertices: A, B, C, D
and edges between A–B, A–C, B–D, and C–D.

Then,

$$V = \{A, B, C, D\} \quad E = \{(A, B), (A, C), (B, D), (C, D)\}$$

This representation tells which nodes are connected.

2. Representation of a Directed Graph (Digraph)

In a **directed graph**, edges have a **direction**.

If there is an edge from vertex A to B, it is written as $\langle A, B \rangle$, meaning from $A \rightarrow B$ (not necessarily from $B \rightarrow A$).

Example:

Let

$$V = \{A, B, C\} \quad V = \{A, B, C\} \quad V = \{A, B, C\}$$

and edges $A \rightarrow B$, $B \rightarrow C$, $C \rightarrow A$.

Then,

$$E = \{\langle A, B \rangle, \langle B, C \rangle, \langle C, A \rangle\} \quad E = \{\langle A, B \rangle, \langle B, C \rangle, \langle C, A \rangle\} \quad E = \{\langle A, B \rangle, \langle B, C \rangle, \langle C, A \rangle\}$$

3. Weighted Graph Representation Using Set

In a **weighted graph**, each edge has a **weight (cost)** associated with it.

Each edge is represented as a **triple (start, end, weight)**.

Example:

$$V = \{A, B, C\} \quad V = \{A, B, C\} \quad V = \{A, B, C\} \quad E = \{(A, B, 5), (B, C, 3), (C, A, 2)\} \quad E = \{(A, B, 5), (B, C, 3), (C, A, 2)\} \quad E = \{(A, B, 5), (B, C, 3), (C, A, 2)\}$$

Here, the edge from A to B has a **weight of 5**, etc.

4. Advantages of Set Representation

- Simple and mathematical way to represent a graph.
 - Useful for **theoretical analysis**.
 - Provides a **foundation** for adjacency list and adjacency matrix representations.
-

5. Limitations

- Not convenient for **computer implementation**, especially for large graphs.
- Difficult to **search or traverse** edges directly.

Graph Representation Using Linked List:

Definition:

A **graph** can be represented in a computer's memory using **linked lists**, which is an efficient and flexible way — especially for **sparse graphs** (graphs with fewer edges).

This method is called the **Adjacency List Representation**.

- Each **vertex** of the graph has a **linked list** of all the **vertices adjacent to it**.
- The **head node** represents the **vertex**, and the **linked nodes** represent the vertices directly connected to it by an edge.

So,

- For an **undirected graph**, if vertex A is connected to B, then B also has A in its adjacency list.
- For a **directed graph**, if there is an edge from $A \rightarrow B$, then B appears only in A's list.

Structure of Adjacency List

Each node in the adjacency list contains:

- **Data (vertex name)**
- **Pointer** to the next adjacent vertex

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

Example: Undirected Graph

```
  A
 /\
B---C
```

Vertices: A, B, C

Edges: (A, B), (A, C), (B, C)

Adjacency List Representation:

Vertex Adjacency List

```
A    → B → C
B    → A → C
```

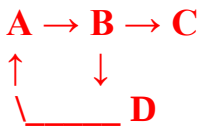
Vertex Adjacency List

C → A → B

Explanation:

- A is connected to B and C
 - B is connected to A and C
 - C is connected to A and B
-

4. Example: Directed Graph



Edges: (A,B), (B,C), (C,D), (D,A)

Adjacency List Representation:

Vertex Adjacency List

A → B

B → C

C → D

D → A

Advantages

- **Space efficient** for sparse graphs (less number of edges).
 - Easy to **traverse all neighbors** of a vertex.
 - Easy to **add or remove** edges or vertices dynamically.
-

6. Disadvantages

- **Slower edge lookup** between two specific vertices compared to adjacency matrix.
 - Requires **more complex programming** using pointers.
-

7. Applications

- Used in **graph algorithms** like BFS, DFS, Dijkstra's shortest path, etc.
- Suitable for **network routing**, **social networks**, and **web link analysis**.

Graph Traversal

Definition:

Graph Traversal is the process of **visiting all the vertices** of a graph **in a specific order**, starting from a selected node (called the **source** or **starting vertex**).

Traversal is essential for searching, analyzing connections, and performing operations like shortest path, spanning tree, or network flow.

Types of Graph Traversal

There are **two main methods** of graph traversal:

1. **Breadth-First Search (BFS)**
2. **Depth-First Search (DFS)**

Breadth-First Search (BFS)

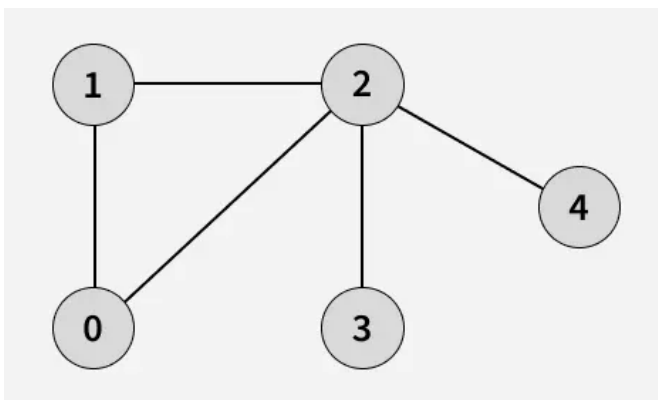
Definition:

BFS is a **level-order traversal** method.

It starts from a source vertex and **visits all its adjacent vertices first**, before moving to the next level.

Algorithm (Using Queue):

1. Start from the **source vertex**, mark it as **visited**, and insert it into a **queue**.
2. While the queue is **not empty**:
 - Remove a vertex from the queue and **visit all its unvisited adjacent vertices**.
 - Mark them as **visited** and **insert** them into the queue.
3. Repeat until all vertices are visited.



Explanation: Starting from 0, the BFS traversal proceeds as follows:

Visit 0 - Print: 0

Visit 1 (neighbor of 0) - Print: 1

Visit 2 (next neighbor of 0) - Print: 2

Visit 3 (first neighbor of 2 that hasn't been visited yet) - Print: 3

Visit 4 (next neighbor of 2) - Print: 4

Applications:

- Finding **shortest path** in an unweighted graph.
- **Level order traversal** in trees.
- Used in **network broadcasting** and **web crawling**

Depth-First Search (DFS)

Definition:

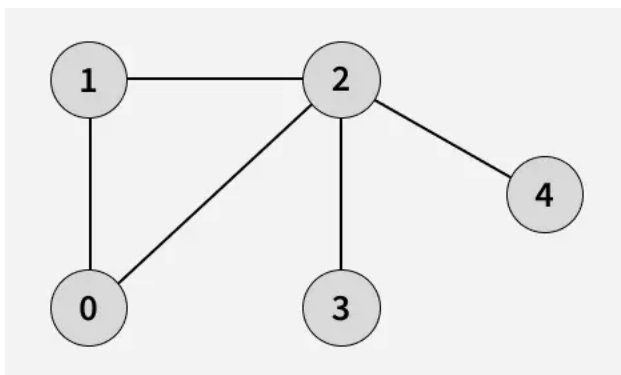
DFS explores a graph **as deep as possible** before backtracking.

It starts at a vertex and moves along **one path** until it reaches the end, then it **backtracks** to explore other paths.

Algorithm (Using Stack or Recursion):

1. Start from the **source vertex**, mark it as **visited**.
2. Visit an **unvisited adjacent vertex**, mark it as visited, and **move deeper**.
3. If no adjacent unvisited vertex remains, **backtrack** to the previous vertex.
4. Repeat until all vertices are visited.

Example:



Explanation: The source vertex is 0. We visit it first, then we visit its adjacent.

Start at 0: Mark as visited. Print 0.

Move to 1: Mark as visited. Print 1.

Move to 2: Mark as visited. Print 2.

Move to 3: Mark as visited. Print 3.(backtrack to 2)

Move to 4: Mark as visited. Print 4(backtrack to 2, then backtrack to 1, then to 0).

Note that there can be more than one DFS Traversals of a Graph. For example, after 1, we may pick adjacent 2 instead of 0 and get a different DFS.

Input: $adj[][] = [[2, 3], [2], [0, 1], [0], [5], [4]]$

Applications:

- **Detecting cycles** in graphs.
- **Topological sorting**.
- **Path finding** and **maze solving**.
- Used in **artificial intelligence** and **game trees**

CHAPTER-VI SORTING

BUBBLE SORT:

Bubble Sort is one of the simplest sorting algorithms in computer science.

It is often used for teaching the basic concept of **comparison and swapping** because it is easy to understand and implement.

The basic idea is:

- Repeatedly **compare adjacent elements** in a list.
- If they are in the wrong order, **swap them**.
- Continue doing this until the entire list is sorted.

Each pass “bubbles” the largest (or smallest) element to its correct position — just like bubbles rising to the top in water.

This algorithm is not suitable for large data sets as its average and worst-case time complexity are quite high.

Working Principle (Step-by-Step Example)

Let's take an example:

Unsorted List: [5, 3, 4, 1, 2]

Pass 1:

- Compare 5 and 3 → swap → [3, 5, 4, 1, 2]
- Compare 5 and 4 → swap → [3, 4, 5, 1, 2]
- Compare 5 and 1 → swap → [3, 4, 1, 5, 2]
- Compare 5 and 2 → swap → [3, 4, 1, 2, 5]

→ **Largest element (5)** moves to the end.

Pass 2:

- Compare 3 and 4 → OK
- Compare 4 and 1 → swap → [3, 1, 4, 2, 5]
- Compare 4 and 2 → swap → [3, 1, 2, 4, 5]

→ **Second largest (4)** is in place.

Pass 3:

- Compare 3 and 1 → swap → [1, 3, 2, 4, 5]
- Compare 3 and 2 → swap → [1, 2, 3, 4, 5]

Now the list is **sorted**.

4. Algorithm

Step 1:

Start

Step 2:

Read the number of elements, **n**

Step 3:

Read the array elements **A[1..n]**

Step 4:

Repeat for **i = 1 to n-1**

Repeat for **j = 1 to n-i**

If **A[j] > A[j+1]**, then

Swap **A[j]** and **A[j+1]**

End If

End for

End for

Step 5:

Print the sorted array **A[1..n]**

Step 6:

Stop

Property	Description
Type	Comparison-based sorting

Property	Description
Best Case	$O(n)$ (when list is already sorted)
Average Case	$O(n^2)$
Worst Case	$O(n^2)$

Advantages

- Very simple and easy to understand.
- Good for **small datasets**.
- Helps beginners learn sorting concepts.

Disadvantages

- Very **slow** for large lists ($O(n^2)$ time complexity).
- Performs **unnecessary comparisons** even if the list is nearly sorted.

SELECTION SORT:

Selection Sort is a simple **comparison-based sorting algorithm**.

It repeatedly selects the **smallest (or largest)** element from the **unsorted part** of the list and puts it in its correct position in the **sorted part**.

Concept

- Divide the list into two parts:
 - **Sorted part** (initially empty)
 - **Unsorted part** (initially all elements)
- In each pass:
 - Find the **minimum element** from the unsorted part.
 - Swap it with the **first element** of the unsorted part.
- Continue until all elements are sorted.

3. Example

Unsorted List: [5, 3, 1, 4, 2]

Pass	Operation	Result
1	Smallest = 1 → swap with 5	[1, 3, 5, 4, 2]
2	Smallest in remaining [3,5,4,2] = 2 → swap with 3	[1, 2, 5, 4, 3]
3	Smallest in [5,4,3] = 3 → swap with 5	[1, 2, 3, 4, 5]

Pass	Operation	Result
4	Already sorted	[1, 2, 3, 4, 5]

Property	Description
Type	Comparison-based sorting
Time Complexity	$O(n^2)$

Algorithm: Selection Sort

Step 1:

Start

Step 2:

Read the number of elements, **n**

Step 3:

Read the array elements **A[1..n]**

Step 4:

```

Repeat for i = 1 to n - 1
Set    min = i
For    j = i + 1 to n
If      A[j] < A[min] then
Set      min = j
End If
End For
If      min ≠ i then
Swap    A[i] and A[min]
End If
End For

```

Step 5:

Print the sorted array **A[1..n]**

Step 6:

Stop

Advantages

- Simple to understand and implement.
- Performs well on **small lists**.

8. Disadvantages

- Inefficient for large lists ($O(n^2)$).
- Always makes the same number of comparisons, even if the array is nearly sorted.

Insertion Sort:

Insertion Sort is a simple sorting algorithm that works the same way we **sort playing cards in our hands**.

We take one element at a time and insert it into its correct position among the previously sorted elements.

The list is divided into two parts:

- **Sorted part** (initially contains the first element)
- **Unsorted part** (contains the remaining elements)

☐ Take the first element from the unsorted part and **insert it into the correct position** in the sorted part.

☐ Repeat until all elements are placed correctly.

Unsorted List: [5, 3, 4, 1, 2]

Pass	Operation	Result
1	Insert 3 in [5] $\rightarrow$ [3, 5]	[3, 5, 4, 1, 2]
2	Insert 4 in [3,5] $\rightarrow$ [3,4,5]	[3, 4, 5, 1, 2]
3	Insert 1 in [3,4,5] $\rightarrow$ [1,3,4,5]	[1, 3, 4, 5, 2]
4	Insert 2 in [1,3,4,5] $\rightarrow$ [1,2,3,4,5]	✓ Sorted

Property	Description
Type	Comparison-based sorting
Time Complexity	$O(n^2)$
Best Case	$O(n)$ (when array is already sorted)
Space Complexity	$O(1)$

Advantages

- Simple and easy to understand.

- Efficient for **small or nearly sorted** lists.
- Stable sorting algorithm.

8. Disadvantages

- Slow for large datasets ($O(n^2)$).
- Many shifts for elements when large values appear early.

5. Merge Sort:

Merge Sort is an efficient **divide and conquer** sorting algorithm.

It divides the array into smaller parts, sorts them individually, and then **merges** them back into a single sorted list.

It is faster than simple sorting algorithms like **Bubble Sort**, **Selection Sort**, and **Insertion Sort**, especially for large datasets.

Concept (Divide and Conquer Approach)

Step 1 – Divide:

Split the array into two halves.

Step 2 – Conquer:

Recursively sort each half.

Step 3 – Combine:

Merge the two sorted halves into a single sorted array.

Unsorted Array: [8, 3, 5, 4, 7, 6, 1, 2]

Step 1: Divide

→ [8, 3, 5, 4] and [7, 6, 1, 2]

Step 2: Divide again

→ [8, 3], [5, 4], [7, 6], [1, 2]

Step 3: Sort and Merge small arrays

→ [3, 8], [4, 5], [6, 7], [1, 2]

Step 4: Merge again

→ [3, 4, 5, 8] and [1, 2, 6, 7]

Step 5: Final merge

→ [1, 2, 3, 4, 5, 6, 7, 8]

✓ The array is now sorted!

Algorithm: Merge Sort

Step 1: Start

Step 2: If the array has more than one element

- a) Divide the array into two halves
- b) Recursively sort the left half
- c) Recursively sort the right half
- d) Merge the two sorted halves

Step 3: End

Property	Description
Type	Divide and Conquer
Time Complexity	$O(n \log n)$

Advantages

- Very efficient for large datasets.
- Performs well even on data that is already partially sorted.
- Stable sorting algorithm.

9. Disadvantages

- Uses additional memory (not in-place).
- A bit more complex to implement than simple sorts.

Quick SORT:

Quick Sort is one of the most popular and fastest sorting algorithms.

It works on the **divide and conquer** principle like Merge Sort, but it sorts **in place**, meaning it doesn't require extra memory.

Concept (Divide and Conquer Method)

Quick Sort divides the array into **two subarrays** based on a **pivot element**:

- Elements **less than the pivot** go to the **left** subarray.
- Elements **greater than the pivot** go to the **right** subarray.

- ❑ Choose a **pivot element** (commonly the first, last, or middle element).
- ❑ Rearrange (partition) the array so that:
 - All elements smaller than the pivot are on its **left**.
 - All elements larger than the pivot are on its **right**.
- ❑ Recursively apply Quick Sort to the left and right subarrays.
- ❑ Combine the results — the array is now sorted.

Example

Unsorted Array: [7, 2, 1, 6, 8, 5, 3, 4]

Step 1: Choose pivot = 4

→ After partition → [2, 1, 3] 4 [7, 6, 8, 5]

Pivot (4) is now at its correct position.

Step 2: Apply Quick Sort to left and right subarrays

→ Left part [2, 1, 3] becomes [1, 2, 3]

→ Right part [7, 6, 8, 5] becomes [5, 6, 7, 8]

✓✓Final Sorted Array → [1, 2, 3, 4, 5, 6, 7, 8]

Property	Description
Type	Divide and Conquer
Time Complexity (Best/Average)	$O(n \log n)$
Worst Case	$O(n^2)$ (when pivot is always smallest/largest)
Space Complexity	$O(\log n)$

Advantages

- Very fast for large datasets.
- Requires less memory than Merge Sort (in-place).
- Efficient for both average and best cases.

Disadvantages

- Performance degrades to $O(n^2)$ if pivot selection is poor.
- Not a stable sort.

SEARCHING

Symbol Table:

A **Symbol Table** is a **data structure** used by **compilers** and **interpreters** to **store and retrieve information about identifiers (symbols)** in a program — such as variables, functions, constants, and objects.

It is mainly used for **fast searching and updating** of symbols during compilation.

Purpose of Symbol Table

- To **store the names of identifiers** used in a program.
- To **store attributes** such as:
 - Data type
 - Scope (local/global)
 - Memory location
 - Line number or block number
- To **retrieve information quickly** during compilation (for type checking, code generation, etc.).

Operations on Symbol Table

Operation	Description
Insert	Add a new symbol and its attributes to the table.
Search/Lookup	Find and return the information associated with a given symbol.
Modify/Update	Change the attributes of a symbol (e.g., if its value or type changes).
Delete	Remove a symbol from the table (optional in some implementations).

Data Structures Used to Implement Symbol Table

Data Structure	Description
Linear List (Array/Linked List)	Simple to implement; slow search ($O(n)$)
Binary Search Tree (BST)	Faster search ($O(\log n)$) if balanced
Hash Table	Most efficient (Average $O(1)$ search and insert)

EXAMPLE:

```
int x, y;
```

```
float total;
```

```
x = 10;
```

total = x + y;

Symbol Data Type Scope Memory Address

x	int	global	1000
y	int	global	1004
total	float	global	1008

This table helps the compiler **quickly find** information about any symbol.

BINARY SEARCH TREE:

A **Binary Search Tree (BST)** is a special type of binary tree where:

- Each node contains a **key (data)**.
- The **left child** of a node contains keys **smaller** than the parent.
- The **right child** of a node contains keys **greater** than the parent.

This property allows us to perform **fast searching** — much faster than linear search in arrays or linked lists.

Concept

In BST search, at each step we:

1. **Compare** the key to be searched (item) with the current node's key.
2. If they are **equal**, the item is found.
3. If the item is **smaller**, move to the **left** subtree.
4. If the item is **greater**, move to the **right** subtree.
5. Repeat until the item is found or the subtree becomes empty.

50

/ \

30 70

/\ /\

20 40 60 80

Searching for 60:

- Compare 60 with 50 → $60 > 50$ → move right
- Compare 60 with 70 → $60 < 70$ → move left
- Compare 60 with 60 → ✓ Found

SEARCHING IN BST:

Step 1: Start

Step 2: Set PTR = Root

Step 3: Repeat Steps 4 to 6 while $\text{PTR} \neq \text{NULL}$ and $\text{ITEM} \neq \text{PTR} \rightarrow \text{DATA}$

Step 4: If $\text{ITEM} < \text{PTR} \rightarrow \text{DATA}$

PTR = PTR $\rightarrow$ LEFT

Else

PTR = PTR $\rightarrow$ RIGHT

Step 5: If PTR = NULL

Print "ITEM not found"

Else

Print "ITEM found"

Step 6: Stop

Case	Time Complexity
Best Case	$O(\log n)$ (balanced tree)
Worst Case	$O(n)$ (unbalanced tree)

Searching in **BALANCED SEARCH TREE**:

A **Balanced Search Tree** is a special type of **Binary Search Tree (BST)** in which the **height of the left and right subtrees** of every node differs by at most **one** (or within a fixed limit).

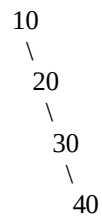
This balance property ensures that operations like **searching, insertion, and deletion** are always **efficient** — generally in **$O(\log n)$** time.

Common examples of balanced search trees are:

- **AVL Tree**
- **Red-Black Tree**
- **B-Tree** (used in databases and file systems)

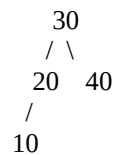
Need for Balance

In a **normal BST**, if data is inserted in sorted order (e.g., 10, 20, 30, 40...), the tree becomes **skewed** like a linked list:



This makes **searching slow ($O(n)$)**.

In a **balanced tree**, nodes are rearranged (rotated) so that both sides stay nearly equal in height:



Now the height is smaller, and **search becomes much faster ($O(\log n)$)**.

Searching Concept

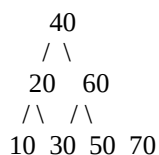
The **searching process** in a balanced tree is **the same as in a BST**, but faster because the tree height is smaller.

Steps:

1. Compare the key (item to be searched) with the **root** node.
 2. If the key matches → **Found**.
 3. If the key is **smaller** → search in the **left subtree**.
 4. If the key is **greater** → search in the **right subtree**.
 5. Continue until the key is found or a NULL pointer is reached.
-

Example

Balanced BST:



Search for 50:

- Compare 50 with 40 $\rightarrow 50 > 40 \rightarrow$ go right
 - Compare 50 with 60 $\rightarrow 50 < 60 \rightarrow$ go left
 - Compare 50 with 50 $\rightarrow \checkmark$ Found
-

Algorithm: Searching in a Balanced BST

Step 1: Start
Step 2: Set PTR = ROOT
Step 3: Repeat Steps 4–6 while PTR $\neq$ NULL and ITEM $\neq$ PTR->DATA
Step 4: If ITEM < PTR->DATA
 PTR = PTR->LEFT
Else
 PTR = PTR->RIGHT
Step 5: If PTR = NULL
 Print "ITEM not found"
Else
 Print "ITEM found"
Step 6: Stop

Case	Time Complexity
Best Case	$O(1)$
Average Case	$O(\log n)$
Worst Case	$O(\log n)$

Searching in Hash Tables:

A **Hash Table** is a **data structure** that stores data in the form of **(key, value)** pairs. It uses a **hash function** to convert the key into an **index** of an array, called a **hash table**, where the value is stored.

This allows **very fast searching** — generally **$O(1)$** time!

In structures like **arrays** or **linked lists**, searching takes time:

- Linear search $\rightarrow O(n)$
- Binary search $\rightarrow O(\log n)$

But in a **hash table**, we can directly find the location of an item using its **hash value**, making searching much faster.

Basic Idea

When a **key** is given:

1. Apply the **hash function** to the key.
2. The hash function returns an **index**.

3. Go to that index in the table.
 4. Check if the key matches.
 - If yes → item **found**.
 - If no → handle **collision** and continue searching.
-

Hash Function

A **hash function (h)** maps keys to indices:

$h(\text{key}) = \text{key} \% \text{table_size}$

For example:

If table size = 10 and key = 25

then $h(25) = 25 \% 10 = 5$

→ store key 25 at index 5.

Collision

Sometimes, **two keys may have the same hash value** (same index).

This is called a **collision**.

Collision handling techniques:

1. **Open Addressing (Linear Probing, Quadratic Probing)**
 2. **Chaining (Linked List at each index)**
-

Searching Process

Case 1: No Collision

- Compute the index using hash function.
- Check if the key exists at that index.
- If found → return the value.
- If not found → item doesn't exist.

Case 2: With Collision

- Check the index.
 - If it doesn't match → follow collision-handling rule (e.g., next index in linear probing or next node in chain).
 - Continue until the key is found or the list ends.
-

7. Example (Using Chaining)

Hash Table (size = 7)

$$h(\text{key}) = \text{key} \% 7$$

Index Elements (Linked List)

0 14 → 21 → 28

1 15

2 9

3 10

4 11

5 5

6 6

Search key = 21

→ $h(21) = 21 \% 7 = 0$

→ Check list at index 0 → found after comparing with 14 → ✓ **Found**

Case

Best Case

Average Case

Worst Case

Time

$O(1)$

$O(1)$

$O(n)$ (if all elements collide)

Advantages

- ✓ Very fast searching
- ✓ Efficient memory usage
- ✓ Ideal for dictionary-type data (key-value pairs)

CHAPTER-V

ALGORITHM PARADIGMS

Greedy Method

Concept:

The *Greedy method* builds up a solution step by step, always choosing the option that looks **best at the moment** (the “greedy choice”), without worrying about future consequences.

Key Idea:

- Make the **locally optimal choice** at each step, hoping it leads to a **globally optimal solution**.

Steps:

1. Identify the **optimal substructure** of the problem.
2. Make a **greedy choice**.
3. Check if the remaining problem can be solved with the same approach.

Examples:

- **Job Sequencing Problem**
- **Fractional Knapsack Problem**
- **Huffman Coding**
- **Prim’s Algorithm** (for Minimum Spanning Tree)
- **Kruskal’s Algorithm** (for Minimum Spanning Tree)
- **Dijkstra’s Shortest Path Algorithm**

Advantages:

- Simple and easy to understand
- Fast (less computation than other methods)

Disadvantages:

- Doesn’t always give the best solution for all problems.

Divide and Conquer

Concept:

This technique divides the problem into **smaller subproblems**, solves them **recursively**, and then **combines** the results to get the final solution.

Steps:

1. **Divide:** Break the problem into smaller subproblems.
2. **Conquer:** Solve the subproblems recursively.
3. **Combine:** Merge the subproblem solutions to form the overall solution.

Examples:

- Merge Sort
- Quick Sort
- Binary Search
- Strassen's Matrix Multiplication

Advantages:

- Efficient for problems that can be broken into independent parts
- Helps use recursion effectively

Disadvantages:

- Recursive calls may use extra memory (stack space)
- May be inefficient if subproblems overlap

Dynamic Programming (DP)

Concept:

Dynamic Programming solves problems by **breaking them into sub problems**, solving each **only once**, and **storing** their results (called **memorization**).

Used when sub problems **overlap** and have **optimal substructure**.

Key Idea:

- Solve **each sub problem once** and reuse its result.
- Often implemented using **tables or arrays** (bottom-up or top-down approach).

Steps:

1. Characterize the structure of the optimal solution.
2. Define the value of the optimal solution recursively.
3. Compute the value using table filling (bottom-up).
4. Construct the optimal solution (if needed).

Examples:

- **Fibonacci Sequence**
- **0/1 Knapsack Problem**
- **Matrix Chain Multiplication**
- **Shortest Path (Floyd–Warshall)**
- **Bellman–Ford Algorithm**

Advantages:

- Reduces computation time by avoiding repeated work
- Guarantees optimal results

Disadvantages:

- Requires extra memory for storing subproblem results
 - Problem must have overlapping subproblems and optimal substructure
-

Backtracking

Concept:

Backtracking is a **trial-and-error method**.

It builds a solution step by step, and **abandons** (backtracks) a path as soon as it determines that it **cannot lead to a valid solution**.

Key Idea:

- Explore all possible solutions.
- Use recursion to try options and backtrack when needed.

Steps:

1. Choose a possible option.
2. Check if it leads to a valid or complete solution.
3. If not, go back (backtrack) and try another option.

Examples:

- **N-Queens Problem**
- **Sudoku Solver**
- **Subset Sum Problem**
- **Graph Coloring**
- **Hamiltonian Cycle**

Advantages:

- Simple and systematic approach
- Can find all possible solutions

Disadvantages:

- May be slow (explores many paths)
 - Needs optimization to reduce search space
-

Branch and Bound

Concept:

Branch and Bound is used to find the **optimal solution** for **optimization problems**. It divides the problem into **branches (subproblems)** and uses **bounds** to eliminate subproblems that can't lead to a better solution.

Key Idea:

- “Branch” means to divide the problem into parts.
- “Bound” means to estimate the best possible solution in that part to **prune** unpromising branches.

Steps:

1. **Branch:** Generate subproblems.
2. **Bound:** Calculate the lower or upper bound.
3. **Prune:** Eliminate branches that cannot produce better results.
4. Continue until the optimal solution is found.

Examples:

- **0/1 Knapsack Problem (using B&B)**
- **Travelling Salesman Problem (TSP)**
- **Job Assignment Problem**

Advantages:

- Guarantees an optimal solution
- Reduces the number of branches to explore

Disadvantages:

- Complex to implement
- May require large memory

Technique	Problem Type	Approach	Example
-----------	--------------	----------	---------

Technique	Problem Type	Approach	Example
Greedy	Optimization	Local choice leads to global solution	Kruskal's, Dijkstra's
Divide & Conquer	Any (recursive)	Divide → Solve → Combine	Merge Sort, Binary Search
Dynamic Programming	Optimization with overlapping subproblems	Store and reuse results	0/1 Knapsack, Fibonacci
Backtracking	Constraint satisfaction	Try and backtrack if invalid	N-Queens, Sudoku
Branch & Bound	Optimization	Use bounds to prune search	TSP, Knapsack

CHAPTER-VII

GRAPHS

Paths and Cycles in Graphs

Path:

A **path** in a graph is a sequence of **vertices connected by edges**.

- If a path starts from vertex **A** and ends at vertex **B**, it is called a **path from A to B**.
- The **length of the path** = number of edges in the path.

Types of Paths:

- **Simple Path:** A path in which **no vertex is repeated**.
- **Closed Path:** A path where the **starting and ending vertices are the same**.
- **Cycle:** A **closed simple path** (no repetition of vertices except the start and end).

Example:

If the graph has edges (A–B), (B–C), (C–D), (D–A):

- Path: $A \rightarrow B \rightarrow C$
 - Cycle: $A \rightarrow B \rightarrow C \rightarrow D \rightarrow A$
-

Spanning Trees

Definition:

A **spanning tree** of a connected graph is a **subgraph** that:

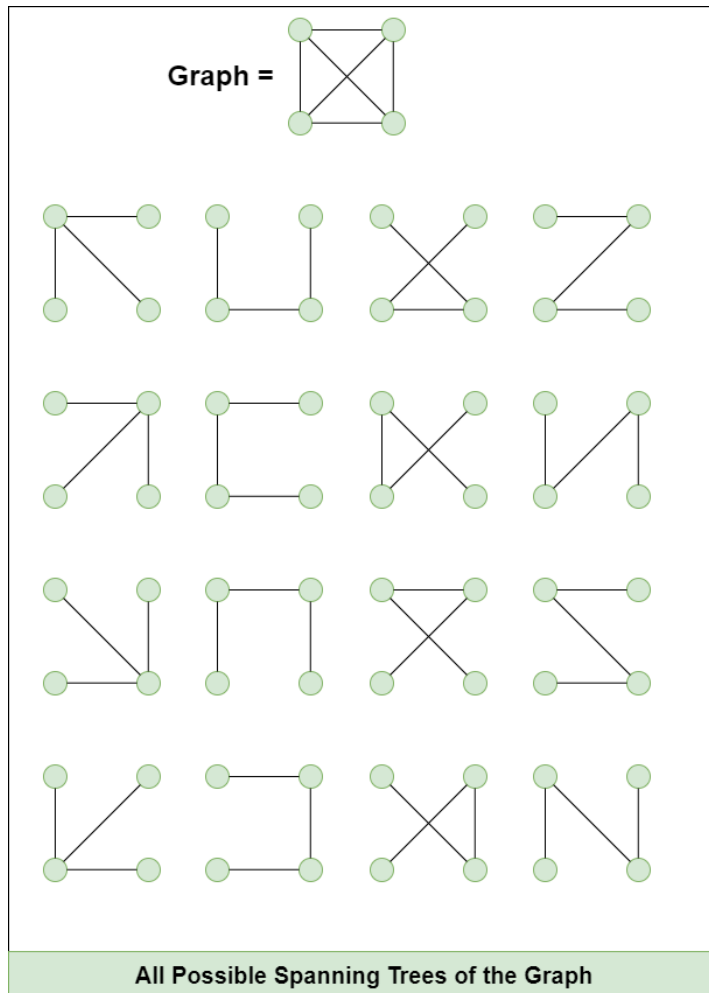
1. Includes **all the vertices** of the original graph.
2. Has **no cycles**.
3. Has **minimum possible number of edges** (for n vertices $\rightarrow n-1$ edges).

Key Points:

- A graph can have **many different spanning trees**.
- A **Minimum Spanning Tree (MST)** is a spanning tree with **minimum total edge weight**.

Applications:

- Designing efficient communication networks (least cost connection).
- Circuit design, clustering, etc.



Directed Acyclic Graphs (DAG)

Definition:

A **Directed Acyclic Graph (DAG)** is a **directed graph** that has **no cycles**.

That means it is impossible to start at a node and follow directed edges to return to the same node.

Properties:

- DAG represents **dependencies** between tasks.
- Has **topological ordering**.
- Commonly used in **scheduling, compiler design, and project management (PERT/CPM)**.

Example:

If $A \rightarrow B$ and $B \rightarrow C$, then A must occur before B, and B before C.

Topological Sorting

Definition:

Topological Sorting is a **linear ordering of vertices** in a **DAG** such that for every directed edge $u \rightarrow v$, vertex **u** comes **before** vertex **v** in the ordering.

Condition:

- Topological sort is **possible only in DAGs** (no cycles allowed).

Applications:

- Task scheduling (order of jobs with dependencies)
- Course prerequisite ordering
- Instruction scheduling in compilers

Minimum Spanning Tree (MST) Algorithms

Definition:

A **Minimum Spanning Tree** is a **spanning tree** with the **minimum total weight** of edges among all possible spanning trees.

Two Popular MST Algorithms:

Kruskal's Algorithm (Greedy Method)

Idea:

Build the MST by **adding edges in increasing order of weight**, ensuring no cycles are formed.

Steps:

1. Sort all edges in **increasing order of weight**.
2. Start with an empty tree.
3. Add edges one by one (smallest first), **skip if adding creates a cycle**.
4. Continue until the tree has **$(V - 1)$** edges.

Time Complexity: $O(E \log E)$

Example Use: Network design, road connections.

Prim's Algorithm (Greedy Method)

Idea:

Build the MST by **starting from a single vertex** and **growing the tree** one edge at a time.

Steps:

1. Start with any vertex.
2. At each step, add the **smallest edge** that connects a **new vertex** to the growing tree.
3. Repeat until all vertices are included.

Time Complexity: $O(V^2)$ or $O(E \log V)$ with priority queue.

Example Use: Communication networks, cable layout.

Shortest Path Algorithms

Definition:

A **shortest path algorithm** finds the **minimum distance (cost)** between vertices in a weighted graph.

Dijkstra's Algorithm (Greedy Method)

Used for:

Finding the **shortest path** from a **single source** to all other vertices in a **weighted graph** (with **non-negative weights**).

Steps:

1. Assign distance = 0 to the source vertex and ∞ to all others.
2. Mark all vertices as unvisited.
3. Select the unvisited vertex with the **smallest distance**.
4. Update the distances of its **adjacent vertices**.
 - o If new distance < current distance, update it.
5. Mark the vertex as **visited** (final shortest distance found).
6. Repeat until all vertices are visited.

Time Complexity: $O(V^2)$ or $O(E \log V)$ using a min-priority queue.

Example Application:

Shortest route in maps, network routing, logistics.

Other Shortest Path Algorithms (for knowledge)

Algorithm	Description
Bellman–Ford	Handles graphs with negative weights .
Floyd–Warshall	Finds shortest paths between all pairs of vertices.

Flow-Based Algorithms

Definition:

Flow-based algorithms are used to find the **maximum flow** possible in a **flow network** — a directed graph where each edge has a **capacity** and each edge can carry a certain **flow**.

Key Terms:

- **Source (s):** The node from which flow starts.
- **Sink (t):** The node where flow ends.
- **Capacity:** Maximum amount of flow allowed through an edge.
- **Residual Graph:** Shows available capacity after each flow.

Objective:

Maximize the flow from **source (s)** to **sink (t)** without exceeding capacity constraints.

Example Algorithm – Ford-Fulkerson Method

Steps:

1. Start with **0 flow** in all edges.
2. While there exists a path from **source** → **sink** with available capacity:
 - Find the **minimum residual capacity** on that path.
 - Add that value to the total flow.
 - Update residual capacities (reduce along path, increase on reverse edges).
3. When no more augmenting paths exist, the flow is **maximum**.

Applications:

- Network traffic routing
- Bipartite matching
- Airline scheduling
- Water distribution systems