Great! Here's your complete **Unit 1: Introduction to Data Structures** notes in **Hindi + English** with **deep explanation** and examples:

📘 UNIT 1 – Introduction to Data Structures

🔷 1.1 Array, Types of Array and Its Representation

Array kya hota hai?
Array ek linear data structure hai jisme **same type ke data elements** continuous memory locations mein store hote hain.

🔷 Types of Arrays:

1. **One Dimensional (1D)** – ek single row ya list (Ex: `int a[5]`)
2. **Two Dimensional (2D)** – row-column matrix form (Ex: `int a[2][3]`)
3. **Multi Dimensional** – 2D se zyada depth (Ex: `int a[2][2][2]`)

🔷 Representation in Memory:

- **1D Array Address:**
  `Address(A[i]) = Base Address + i * Size_of_Datatype`

◇ Example:

```
int A[5] = {10, 20, 30, 40, 50};  // Memory: A[0] to A[4]
```

◇ 1.2 Self Referential Structure

Definition:
Ek structure jisme **ek member pointer** hota hai jo **usi structure type** ko point karta hai.

◇ Example:

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

☑ Ye structure **linked list** banane ke liye use hota hai.

◇ 1.3 Pointer and Dynamic Memory Allocation

Pointer:
Pointer ek variable hai jo kisi dusre variable ka memory address store karta hai.

◇ Syntax:

```
int a = 5;
int *p = &a;  // p holds address of a
```

Dynamic Memory Allocation:
Memory run-time pe allocate hoti hai using these functions:

Function	Purpose
`malloc()`	Memory allocate karta hai
`calloc()`	Contiguous memory allocate karta hai
`realloc()`	Resize karta hai existing memory
`free()`	Memory ko release karta hai

◇ Example:

```
int *ptr;
ptr = (int*) malloc(sizeof(int));
```

◇ 1.4 Data Types, Data Objects, Abstract Data Type (ADT)

Data Types:

- **Primitive:** int, float, char

- **Derived:** array, pointer
- **User-defined:** struct, union

Data Objects:
Wo variables ya constants jo memory mein allocate hote hain.

Abstract Data Type (ADT):
ADT logical view provide karta hai data aur operations ka **without implementation details**.

◇ **Example:**

- **Stack ADT:** push(), pop(), peek()
- **Queue ADT:** enqueue(), dequeue()

◇ 1.5 Data Structure and Its Types

Definition:
Data structure ek method hai jisse data ko organize, store aur access kiya jata hai efficiently.

◇ **Types of Data Structures:**

Category	Examples
Linear	Array, Linked List, Stack, Queue
Non-Linear	Tree, Graph
Static	Array

Category	Examples
Dynamic	Linked List, Tree

➡️ **Linear:** Sequential arrangement
➡️ **Non-Linear:** Hierarchical (Tree) ya Network (Graph)

◇ 1.6 Algorithm Analysis: Space & Time Complexity, Asymptotic Notations

◇ Time Complexity:

Algorithm ko chalane mein lagne wale **time** ko represent karta hai (based on input size `n`).

◇ Space Complexity:

Algorithm ko execute karne mein lagne wali **memory**.

◇ Asymptotic Notations:

Notation	Case	Meaning
O(n)	Worst Case	Maximum time/steps taken
Ω(n)	Best Case	Minimum time/steps taken
Θ(n)	Average Case	Balanced/middle performance

◇ **Examples:**

- Linear Search: **O(n)**
- Binary Search: **O(log n)**
- Bubble Sort: **O(n²)**

☑ **Unit 1 Summary:**

- Arrays are foundational structures
- Pointers enable dynamic memory
- ADTs define logical data handling
- Complexity defines efficiency

Perfect! That's a great idea — you can use this clean text to create your own PDF. Here's a simple copy-paste version for your file:

▧ UNIT 1 – Introduction to Data Structures

◇ 1.1 Array, Types of Array and Its Representation

Array kya hota hai?
Array ek linear data structure hai jisme same type ke data elements continuous memory locations mein store hote hain.

Types of Arrays:

1. One Dimensional (1D) – ek single row ya list (Ex: int a[5])
2. Two Dimensional (2D) – row-column matrix (Ex: int a[2][3])
3. Multi Dimensional – 2D se zyada depth (Ex: int a[2][2][2])

Memory Representation:
Address(A[i]) = Base Address + i * Size_of_DataType

Example:

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

◇ 1.2 Self Referential Structure

Definition:
Structure jisme ek member pointer hota hai jo usi structure ko point karta hai.

Example:

```
struct Node {
   int data;
```

```
   struct Node* next;
};
```

Use: Linked Lists mein hota hai.

◇ 1.3 Pointer and Dynamic Memory Allocation

Pointer:
Variable that stores address of another variable.

Syntax:

```
int a = 5;
int *p = &a;
```

Dynamic Memory Allocation Functions:

- malloc()
- calloc()
- realloc()
- free()

Example:

```
int *ptr;
ptr = (int*) malloc(sizeof(int));
```

◇ 1.4 Data Types, Data Objects and ADT

- **Data Types:** Primitive (int, float), Derived (array, struct), User-defined
- **Data Object:** Variable ya constant stored in memory
- **ADT:** Logical model (e.g., Stack ADT → push, pop, peek)

◇ 1.5 Data Structure and Its Types

Definition:
Method to store, organize, and access data efficiently.

Types:

Category	Examples
Linear	Array, Stack, Queue, Linked List
Non-Linear	Tree, Graph
Static	Array
Dynamic	Linked List

◇ 1.6 Algorithm Analysis

Time Complexity: Time taken based on input size
Space Complexity: Memory required

Asymptotic Notations:

Notation	Meaning
O(n)	Worst Case
Ω(n)	Best Case
Θ(n)	Average Case

Examples:

- Linear Search: O(n)
- Binary Search: O(log n)

☑ Summary:

- Arrays store similar data types sequentially
- Self-referential structures form linked data
- Pointers manage dynamic memory
- ADT helps define behavior without internal details
- Time and space complexity tell us algorithm efficiency

Let me know when you're ready for **Unit 2: Linear Data Structures** and I'll start preparing that next!

Awesome! Let's now begin with ▧ **Unit 2: Linear Data Structures**
This unit covers **sorting, searching, and basic concepts of linear data structures**.

☑ UNIT 2 – Linear Data Structures

💡 **(Hindi + English Mix | Deep Explanation)**

◇ 2.1 Introduction to Linear Data Structure

Definition:
Linear data structure wo hota hai jisme data elements **sequence (line)** mein arranged hote hain — ek ke baad ek.

Characteristics:

- Elements ko sequentially access kiya jata hai.
- Har element ka ek predecessor aur ek successor hota hai (except first & last).
- Memory allocation continuous ya linked ho sakti hai.

Examples:

- Array
- Stack
- Queue
- Linked List

◇ 2.2 Sorting Algorithms with Time Complexity

Sorting:
Sorting ka matlab hota hai data ko kisi specific order mein arrange karna (ascending ya descending).

◇ (a) Bubble Sort

Working:
Adjacent elements ko compare karo, aur swap karo agar out of order ho. Ye process repeatedly hota hai.

```
for(i=0; i<n-1; i++)
  for(j=0; j<n-i-1; j++)
    if(arr[j] > arr[j+1])
      swap(arr[j], arr[j+1]);
```

Time Complexity:

- Best: O(n) (already sorted)
- Worst: $O(n^2)$

◇ (b) Insertion Sort

Working:
Ek ek karke element ko uthao aur usse correct position mein insert karo (like cards in hand).

```
for(i=1; i<n; i++) {
  key = arr[i];
  j = i - 1;
```

```
  while(j >= 0 && arr[j] > key) {
    arr[j+1] = arr[j];
    j--;
  }
  arr[j+1] = key;
}
```

Time Complexity:

- Best: O(n)
- Worst: $O(n^2)$

◇ (c) Selection Sort

Working:
Har baar smallest element find karo aur usko uski correct position pe daal do.

```
for(i=0; i<n-1; i++) {
  min = i;
  for(j=i+1; j<n; j++)
    if(arr[j] < arr[min])
      min = j;
  swap(arr[i], arr[min]);
}
```

Time Complexity: $O(n^2)$ (in all cases)

◇ (d) Merge Sort (Divide & Conquer)

Working:

- Array ko half-half divide karo
- Har half ko sort karo (recursively)
- Merge sorted halves

```
mergeSort(arr[], l, r)
if (l < r)
  mid = (l+r)/2;
  mergeSort(arr, l, mid);
  mergeSort(arr, mid+1, r);
  merge(arr, l, mid, r);
```

Time Complexity:

- Always O(n log n)
- Space: O(n)

◇ (e) Quick Sort

Working:

- Ek pivot select karo
- Smaller elements pivot ke left, greater right
- Recursively sort sub-arrays

```
quickSort(arr, low, high)
{
  if(low < high)
  {
    pi = partition(arr, low, high);
    quickSort(arr, low, pi-1);
```

```
    quickSort(arr, pi+1, high);
  }
}
```

Time Complexity:

- Best/Average: O(n log n)
- Worst: O(n^2) (already sorted in reverse)

◇ 2.3 Searching Techniques

◇ (a) Linear Search

Working:
Ek ek element ko check karo jab tak match na mile.

```
for(i=0; i<n; i++)
  if(arr[i] == key)
    return i;
```

Time Complexity:

- Best: O(1)
- Worst: O(n)

◇ (b) Binary Search (Only sorted arrays)

Working:
Middle element ko check karo. Agar match nahi mila toh decide left half ya right half check karna hai.

```
while(low <= high) {
  mid = (low + high)/2;
  if(arr[mid] == key) return mid;
  else if(arr[mid] < key) low = mid + 1;
  else high = mid - 1;
}
```

Time Complexity:

- Best: O(1)
- Worst: O(log n)

☑ Unit 2 Summary:

Algorithm	Best Case	Worst Case	Avg. Case	Type
Bubble Sort	O(n)	$O(n^2)$	$O(n^2)$	Comparison
Insertion Sort	O(n)	$O(n^2)$	$O(n^2)$	Comparison
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	Comparison
Merge Sort	O(n log n)	O(n log n)	O(n log n)	Divide & Conquer
Quick Sort	O(n log n)	$O(n^2)$	O(n log n)	Divide & Conquer

Algorithm	Best Case	Worst Case	Avg. Case	Type
Linear Search	O(1)	O(n)	O(n)	Sequential
Binary Search	O(1)	O(log n)	O(log n)	Efficient (Sorted Only)

Shall I continue to **Unit 3: Linked List** next?

Also, do you want to copy this text now and save as PDF, or want the final PDF bundle at the end of all 5 units?

Below is the complete **Unit 3: Linked List** notes in a mix of Hindi + English. You can copy this text and later save it as a PDF along with your other units.

📄 UNIT 3 – Linked List

◇ 3.1 Introduction to Linked List

Definition:
A linked list is a dynamic data structure where elements,

called nodes, are stored in non-contiguous memory. Each node contains data and a pointer (or reference) to the next node in the list.

Linked List ka Fayda & Nuksaan:

- **Advantages (Fayde):**
  - **Dynamic Memory Allocation:** Memory allocation is done at runtime, so size is flexible.
  - **Ease of Insertion/Deletion:** New nodes can be inserted or removed without reorganizing the entire structure.
- **Disadvantages (Nuksaan):**
  - **Extra Memory:** Each node requires additional storage for pointer(s).
  - **Sequential Access:** To access a node, you must traverse from the head (starting node).

◇ 3.2 Implementation of Linked List

Node Structure (C Language Example):

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

This simple structure shows a single node containing an integer `data` and a pointer `next` to the next node.

Static vs. Dynamic Representation:

- **Static Representation:** Fixed-size memory allocation (rarely used for linked lists since their key strength is dynamic allocation).
- **Dynamic Representation:** Memory is allocated on the fly using functions like `malloc()` in C, creating nodes when needed.

◇ 3.3 Types of Linked List

◇ 3.3.1 Singly Linked List

Definition:
A singly linked list is a collection of nodes where each node points only to the next node.

Operations:

1. **Creation:**
    - Create a head node and continuously add new nodes at the end.
2. **Printing:**
    - Traverse from the head and print each node's data.
3. **Insertion:**

- **At Beginning:**
  Insert new node and point its `next` to the current head, then update head.
- **In the Middle:**
  Find the required position by traversing the list, then adjust pointers to insert the new node.
- **At End:**
  Traverse to the last node (where `next` is `NULL`) and insert the new node.

4. **Deletion:**
   - **From Beginning:**
     Update head to next node and free the old head.
   - **From Middle:**
     Find the node before the one to delete, adjust its pointer to skip the removed node.
   - **From End:**
     Traverse to the node just before the last, set its pointer to `NULL`.
5. **Reverse:**
   - Reverse the pointers so that the list is traversed in reverse order.
6. **Searching:**
   - Traverse from the head to find the node with the desired data.

◇ 3.3.2 Doubly Linked List

Definition:
In a doubly linked list, each node contains two pointers: one pointing to the next node and another to the previous node. This allows traversal in both directions.

Operations:

1. **Creation & Printing:**
    - Start with a head node; each node points forward and backward.
2. **Insertion:**
    - **At Beginning:**
      Create new node, adjust the previous pointer of the current head, update head.
    - **In the Middle & End:**
      Traverse to the desired position, adjust `prev` and `next` pointers accordingly.
3. **Deletion:**
    - Similar to singly linked list but need to adjust both `prev` and `next` of adjacent nodes.

◇ 3.3.3 Circularly Singly Linked List

Definition:
A circularly singly linked list is similar to a singly linked list except that the last node's pointer points to the head rather than `NULL`.

Operations:

1. **Creation:**
    - Create nodes and adjust the last node's pointer to point to the head.
2. **Printing:**
    - Start from the head and continue until you reach the head again.

◇ 3.3.4 Circularly Doubly Linked List

Definition:
A circularly doubly linked list is a doubly linked list where the `next` pointer of the last node points to the head and the `prev` pointer of the head points to the last node.

Operations:

1. **Creation & Printing:**
    - Similar principles as the doubly linked list with circular pointers connecting head and tail.

☑ Unit 3 Summary:

- **Linked Lists** provide dynamic memory management and flexible insertion/deletion.
- **Singly Linked List:** Single pointer, traversal in one direction.

- **Doubly Linked List:** Two pointers per node, allowing bidirectional traversal.
- **Circular Linked Lists:** The last node connects to the head, forming a circle.

This detailed note covers the fundamental concepts, types, and operations of linked lists with explanations in both Hindi and English.

Let me know if you want any modifications or if you're ready for the **next unit (Unit 4: Stack and Queue)**!

Perfect! Let's now move on to ▧ **Unit 3: Linked Lists** — **deeply explained** in **Hindi + English mix**.

☑ UNIT 3 – Linked Lists

💡 **(Hindi + English Mix | Deep Explanation)**

◇ 3.1 Introduction to Linked List

Definition:
Linked List ek linear data structure hai jisme elements ko sequentially **linked** kiya jata hai, har element apne **next element** ka address rakhta hai.

Linked List ka Difference from Array:

- **Array:** Fixed size, contiguous memory allocation
- **Linked List:** Dynamic size, non-contiguous memory allocation

◇ Structure:

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

◇ 3.2 Implementation of Linked List

Static Representation:
Ek array ya structure mein nodes ko link karna.

Dynamic Representation:
Memory dynamically allocate ki jati hai jab bhi node add hoti hai, jo `malloc()` ke through hota hai.

◇ 3.3 Types of Linked Lists

◇ 3.3.1 Singly Linked List (SLL)

Definition:
Isme har node ka ek pointer hota hai jo **next node** ko point karta hai. Last node ka **next pointer** `NULL` hota hai.

Operations in SLL:

- **Create:** Initializing the first node.
- **Print:** Traversing the list and printing nodes.
- **Insertion:** Add new node at:
  - Beginning
  - Middle
  - End
- **Deletion:** Remove node from:
  - Beginning
  - Middle
  - End
- **Reverse:** Reverse the order of nodes.
- **Search:** Search for a particular element.

```
// Node creation
struct Node* head = NULL;
struct Node* new_node = (struct
Node*)malloc(sizeof(struct Node));
new_node->data = 10;
new_node->next = NULL;
head = new_node;
```

◇ 3.3.2 Doubly Linked List (DLL)

Definition:
Har node ka **do pointers** hote hain — ek **next** (next node) aur ek **prev** (previous node).

Advantages:

- Traversal both directions (forward & backward)
- Deletion in middle is easier (can directly access previous node)

Operations in DLL:

- **Create**
- **Print**
- **Insertion** at:
  - Beginning
  - Middle
  - End
- **Deletion** from:
  - Beginning
  - Middle
  - End

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

◇ 3.3.3 Circular Singly Linked List (CSLL)

Definition:
Isme last node ka **next pointer first node** ko point karta hai, making the list circular.

Operations:

- **Create**
- **Print** (Traverse Circularly)

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

◇ 3.3.4 Circular Doubly Linked List (CDLL)

Definition:
Har node ka **next** aur **prev pointer** dono circularly connected hote hain, where the last node points to the first and vice-versa.

Operations:

- **Create**
- **Print**

```
last_node->next = head;
head->prev = last_node;
```

◇ 3.4 Linked List Operations (Detailed)

◇ Insertion Operations:

1. **At the Beginning:**

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

2. **At the End:**

```
last_node->next = new_node;
new_node->next = NULL;
```

3. **At the Middle:**

- Traverse to the desired position
- Insert the new node

◇ **Deletion Operations:**

1. **From the Beginning:**

```
temp = head;
head = head->next;
free(temp);
```

2. **From the End:**

- Traverse to the second last node
- Set second last node's `next` to `NULL`

☑ Unit 3 Summary:

Type	Description	Operations
Singly Linked List (SLL)	One direction traversal, node points to next only	Insert, Delete, Reverse, Print, Search
Doubly Linked List (DLL)	Two direction traversal, node points to both next & prev	Insert, Delete, Traverse in both directions
Circular Singly Linked List	Last node points to first node (circular)	Insert, Delete, Traverse in circular manner
Circular Doubly Linked List	Circular, two direction traversal	Insert, Delete, Traverse in circular manner

---

Shall I now proceed with **Unit 4: Stack and Queue** next?

Let me know if you want me to modify anything!

Great! Let's dive into ▧ **Unit 4: Stack and Queue** — in **Hindi + English** with deep explanations, real-life examples, and operations.

☑ UNIT 4 – Stack and Queue

◇ 4.1 Introduction: Concept and Characteristics (LIFO)

📌 Stack:

- Stack ek **Linear Data Structure** hai.
- Principle: **LIFO** (Last In First Out) → jo last insert hua, wahi pehle niklega.
- Real-world Examples:
  - Browser history (last page visited pops first)
  - Undo operations (last action undone first)

◇ 4.2 Stack Implementation

📍 Static Representation (Using Array)

```
#define SIZE 100
int stack[SIZE];
int top = -1;
```

📍 Dynamic Representation (Using Linked List)

```
struct Node {
    int data;
```

```
    struct Node* next;
};
```

◇ 4.3 Stack Operations

Operation	Description
Push	Insert element on top
Pop	Remove top element
Peek/Top	Show top element without removing
isEmpty()	Check if stack is empty
isFull()	Check if stack is full (only for array implementation)

Example:

```
// Push operation
top++;
stack[top] = value;

// Pop operation
int value = stack[top];
top--;
```

◇ 4.4 Applications of Stack

- **Expression evaluation**
- **Backtracking algorithms**
- **Function call stack (recursion)**
- **Undo/redo features**
- **Parentheses checking in expressions**

◇ 4.5 Infix, Prefix, Postfix

□ Expression Notations:

Type	Example
Infix	A + B
Prefix	+AB
Postfix	AB+

💡 Conversion Rules:

- **Infix → Postfix**
- **Infix → Prefix**
- Use **Stack** to convert expressions using **precedence** and **associativity**

Example:
Infix: `(A + B) * C`
Postfix: `AB+C*`
Prefix: `*+ABC`

Postfix Evaluation Example:

```
Input: "23*54*+9-"
Stack: Evaluate from left to right using
stack
```

◇ 4.6 Queue Introduction

📌 Queue:

- Linear data structure
- Principle: **FIFO** (First In First Out) → jo pehle aaya, wahi pehle niklega

Real-world Examples:

- Ticket counter
- Call center
- Printer queue

◇ 4.7 Queue Representation

📍 Static (Array)

```
#define SIZE 100
int queue[SIZE];
int front = 0;
int rear = -1;
```

📍 Dynamic (Linked List)

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

◇ 4.8 Primitive Operations on Queue

Operation	Description
Insert (Enqueue)	Add element at rear
Delete (Dequeue)	Remove element from front
Display	Print elements from front to rear

```
// Enqueue
rear++;
queue[rear] = value;

// Dequeue
value = queue[front];
front++;
```

◇ 4.9 Circular Queue

Problem in Simple Queue:

- Rear reaches end of array even if front has space → **wasted space**

Solution: Circular Queue

- After rear reaches end, it circles back to beginning (if empty)

Formula:

```
rear = (rear + 1) % SIZE;
```

Operations in Circular Queue:

- Insert

- Delete
- Display

☑ Summary Table

Data Structure	Concept	Order	Applications
Stack	LIFO	Last In First Out	Undo, Recursion, Expression eval
Queue	FIFO	First In First Out	Ticket counter, CPU scheduling
Circular Queue	Reuse space	Circular memory usage	Printer queue, Real-time systems

---

💬 **Next: Unit 5 – Tree and Graph**
Shall I start Unit 5 now?

Let me know if you want any **diagrams, examples, or visuals** added to this before that.

Awesome! Let's now move on to ▧ **Unit 5: Tree and Graph** — explained deeply in **Hindi + English** mix for better understanding.

☑ UNIT 5 – Tree and Graph

◇ 5.1 Tree Concept & Terminologies

📌 Tree:

- Tree ek **non-linear hierarchical data structure** hai.
- Consists of **nodes** and **edges**.

- Topmost node: **Root**
- Nodes connected to a node: **Children**
- Node without children: **Leaf**

🕮 Important Terminologies:

Term	Meaning
Root	First/top node
Leaf	Node with no children
Edge	Connection between two nodes
Parent	A node having child nodes
Child	A node connected to parent
Siblings	Nodes with same parent
Depth	Levels from root to that node
Height	Longest path from node to leaf

◇ 5.2 Binary Tree (BT) & Binary Search Tree (BST)

📌 Binary Tree (BT):

- Har node max **2 children**: left & right
- Not ordered necessarily

📌 Binary Search Tree (BST):

- Left child < root < Right child
- All left subtree nodes < root

- All right subtree nodes > root

Example:

```
      10
    /    \
   5      20
  / \     / \
 2   7   15  25
```

◇ 5.3 Operations on BT and BST

🛠 Create Node

```
struct Node {
    int data;
    struct Node *left, *right;
};
```

🛠 Insertion

```
if (data < root->data)
    insert(root->left, data);
else
    insert(root->right, data);
```

🛠 Deletion (3 Cases):

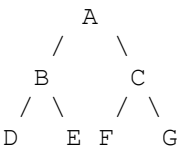
1. **Node with no child** → delete directly
2. **One child** → replace with child
3. **Two children** → replace with inorder successor/predecessor

---

◇ 5.4 Tree Traversals (Very Important)

Type	Order
Preorder	Root → Left → Right
Inorder	Left → Root → Right
Postorder	Left → Right → Root

Example Tree:



Traversal	Result
Inorder	D B E A F C G
Preorder	A B D E C F G
Postorder	D E B F G C A

◇ 5.5 AVL Tree (Height Balanced Tree)

📌 AVL Tree:

- Self-balancing BST
- **Balance Factor (BF) = Height(left) - Height(right)**
- BF must be **-1, 0, or 1**

Rotations (for balancing):

1. LL Rotation
2. RR Rotation
3. LR Rotation
4. RL Rotation

◇ 5.6 Introduction to Graph

📌 Graph:

- Non-linear structure of **nodes (vertices)** and **edges (connections)**

Types:

- **Directed Graph (Digraph)**
- **Undirected Graph**

◇ 5.7 Graph Representation

📍 Adjacency Matrix

- 2D array
- `matrix[i][j] = 1` if edge exists from `i` to `j`

📍 Adjacency List

- Each node has a list of connected nodes
- Efficient for sparse graphs

◇ 5.8 Graph Traversals

◇ BFS (Breadth First Search)

- Level by level
- Use Queue

```
// Steps
1. Start from root
2. Visit and mark
3. Add neighbors to queue
```

◇ DFS (Depth First Search)

- Go deep first
- Use Stack or Recursion

```
// Steps
1. Start from root
2. Visit and mark
3. Visit unvisited neighbors recursively
```

◇ 5.9 Degree of Graph

- **Degree:** Number of edges connected to a node
- **Indegree:** Number of edges coming **into** a node

- **Outdegree:** Number of edges going **out from** a node

◇ 5.10 Spanning Tree

📌 Spanning Tree:

- Subset of a graph
- Contains all vertices
- No cycles
- Minimum edges

For `n` vertices → Spanning Tree has `n-1` edges

☑ Minimum Spanning Tree Algorithms:

- **Prim's Algorithm**
- **Kruskal's Algorithm**

☑ Summary Table

Concept	Key Point	Example Use
Binary Tree	Max 2 children	Hierarchical data
BST	Ordered binary tree	Searching
AVL Tree	Height balanced	Fast operations
Graph	Nodes + Edges	Maps, Networks

Concept	Key Point	Example Use
BFS	Level traversal	Social networks
DFS	Deep traversal	Maze solving
Spanning Tree	Minimum edges	Network design

🎯 **Unit 5 Complete!**
You can now create the PDF for this unit too.

Shall I now give you **all units combined in one DOC or text format** so you can easily convert it into a full study PDF?

Or would you prefer **separate PDFs unit-wise**?

Let me know what works best for you.