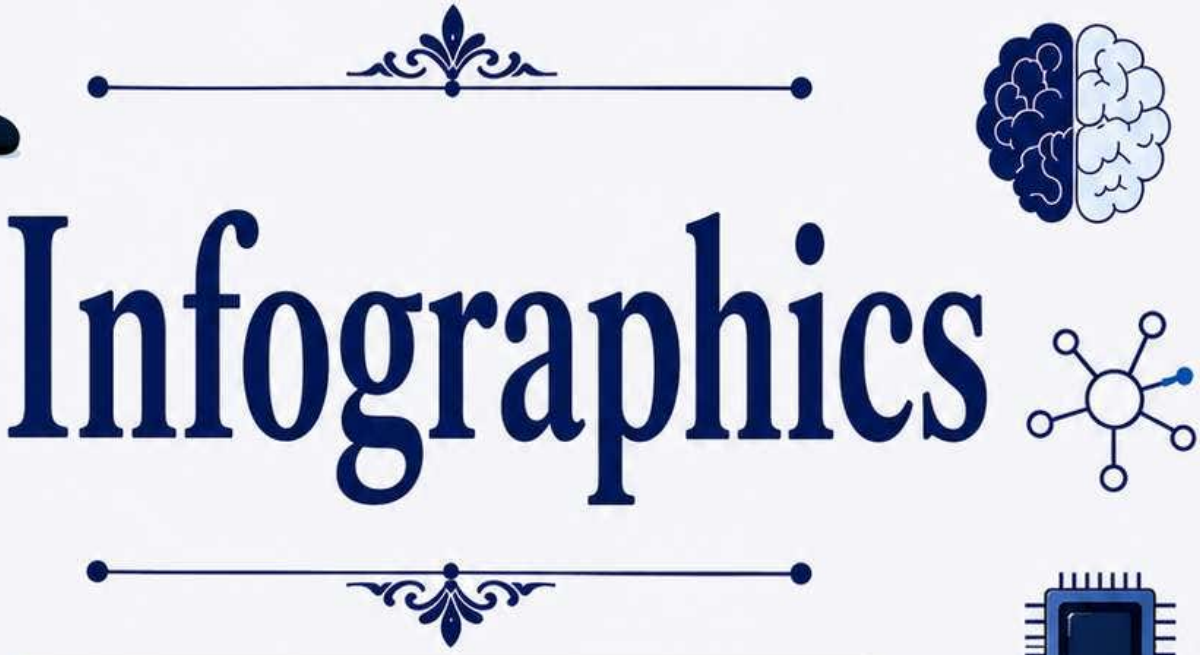
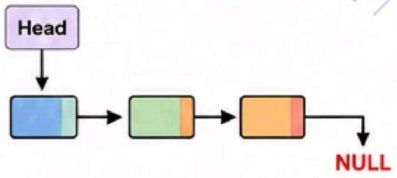
Infographics



Unit - III


DATA STRUCTURES
ALGORITHMS
COMPUTER NETWORKS
DATABASE SYSTEMS
#include <iostream>
using namespace std;
int main()
{
cout << "Hello, World!";
return 0;
}



LINKED LIST

Definition and Concepts, Memory Representations

What is Linked List?

A linked list is a linear data structure in which elements (nodes) are not stored in contiguous memory locations.
Each node stores data and a reference (address) to the next node in the sequence.

1. DEFINITION

A linked list is a dynamic data structure consisting of a collection of nodes. Each node contains two parts:

- DATA: Stores the actual value
- LINK (REFERENCE): Stores the address of the next node in the list

Unlike arrays, elements in a linked list are not stored in contiguous memory locations.

2. CONCEPTS

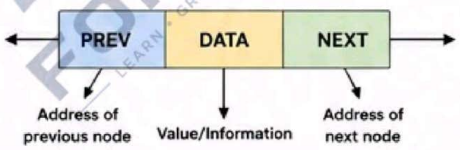
- Dynamic Size: Can grow or shrink at runtime.
- Non-contiguous Memory Allocation: Nodes are scattered in memory.
- Efficient Insertions/Deletions: No need to shift elements; only links (addresses) are updated.
- Access: Sequential access only (no random access like arrays).
- Requires Extra Memory: Additional memory is needed to store addresses.

3. NODE STRUCTURE

A node is the basic building block.

Singly Linked List Node

Doubly Linked List Node

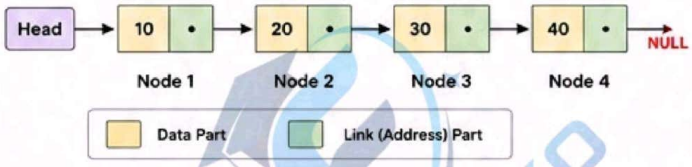


4. MEMORY REPRESENTATIONS

4.1 SINGLY LINKED LIST

Each node points to the next node. The last node's link part contains NULL.

Memory Representation Diagram



Example in Memory

Node	Address	Data	Link (Address of Next Node)
Node 1	1000	10	2000
Node 2	2000	20	3000
Node 3	3000	30	4000
Node 4	4000	40	NULL

Actual Memory Layout (NON-CONTIGUOUS)

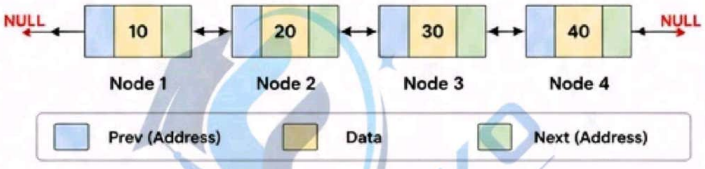
1000	10	2000
2000	20	3000
3000	30	4000
4000	40	NULL

Nodes are stored at different memory locations.

4.2 DOUBLY LINKED LIST

Each node has two links: one to the next node and one to the previous node.

Memory Representation Diagram



Example in Memory

Node	Address	Prev (Address)	Data	Next (Address)
Node 1	1000	NULL	10	2000
Node 2	2000	1000	20	3000
Node 3	3000	2000	30	4000
Node 4	4000	3000	40	NULL

Traversal can be done in both forward (Next) and backward (Prev) directions.

5. ADDRESS CALCULATION (CONCEPTUAL)

In Linked List:

- Nodes are allocated dynamically using memory allocation functions (e.g., malloc() in C).
- Memory addresses are not consecutive.
- Links store addresses (references) of next/prev nodes.

Example:

- If address of Node 1 = 1000
- Node 1 points to Node 2 whose address = 2000
- Node 2 points to Node 3 whose address = 3000
- And so on...

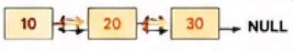
Advantage of Linked List Memory Representation

- Efficient memory utilization
- Easy insertions and deletions
- Dynamic size management

6. TYPES OF LINKED LIST

1. SINGLY LINKED LIST

Each node contains data and address of next node.

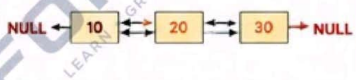


Features

- Traversal in one direction only
- Less memory (one link field)
- Simple and efficient

2. DOUBLY LINKED LIST

Each node contains address of previous node and next node.

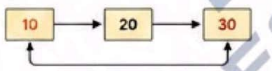


Features

- Traversal in both directions
- More memory (two link fields)
- Easy deletion of a given node

3. CIRCULAR LINKED LIST

Last node points back to the first node.

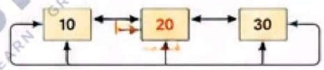


Features

- No NULL at the end
- Useful in cyclic data processing
- Can start traversal from any node

4. DOUBLY CIRCULAR LINKED LIST

Doubly linked list in which last node connects to the first node and vice versa.



Features

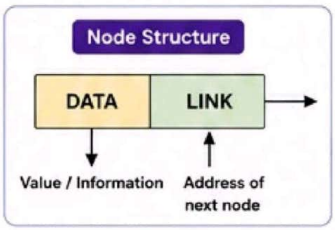
- Bidirectional circular traversal
- No NULL at either end
- Useful in advanced applications

KEY TAKEAWAY

Linked List is a dynamic and flexible data structure where elements are connected using links (addresses).
It allows efficient insertions and deletions, but sacrifices random access and requires extra memory for links.

TYPES OF LINKED LIST

A linked list is a linear data structure in which elements (nodes) are not stored in contiguous memory locations. Each node contains data and link(s) to other node(s).



MAIN TYPES OF LINKED LIST

1. SINGLY LINKED LIST

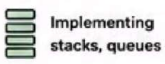
Each node contains data and address of next node.
The last node points to NULL.

Structure Diagram

Features

- Traversal is possible in one direction only (forward).
- Less memory (one link field per node).
- Simple to implement.
- Efficient for insertions and deletions.

Applications


Implementing stacks, queues

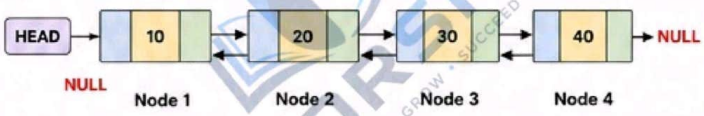
$a_n x^n + \ldots + a_0$


Memory management

2. DOUBLY LINKED LIST

Each node contains data and addresses of both next node and previous node.

Structure Diagram



Features

- Traversal possible in both forward and backward directions.
- More memory (two link fields per node).
- Easy deletion of a given node.
- Slower than singly linked list due to extra links.

Applications


Browser back/forward functionality

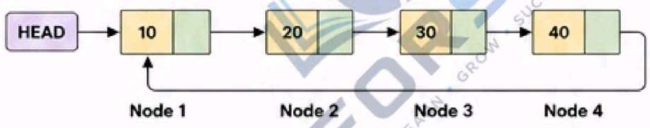

Music playlists


Undo / Redo operations

3. CIRCULAR LINKED LIST (SINGLY)

Last node points back to the first node. There is no NULL at the end.

Structure Diagram



Features

- Traversal can start from any node and continue in a circle.
- No NULL at the end.
- Efficient for circular data processing.
- Less memory (one link field per node).

Applications


Round Robin scheduling

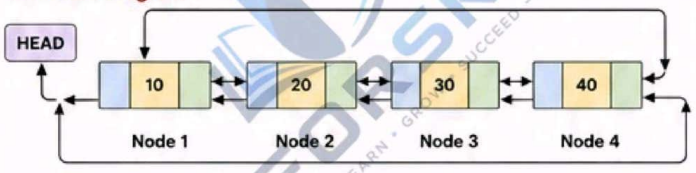

Circular queues

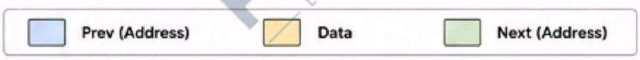

Multiplayer games

4. DOUBLY CIRCULAR LINKED LIST

Each node has two links (prev and next) and last node connects to first node and first node connects to last node.

Structure Diagram





Features

- Traversal in both directions in a circular manner.
- No NULL at both ends.
- More memory (two link fields per node).
- Most flexible among linked lists.

Applications


Circular deques


Advanced scheduling systems

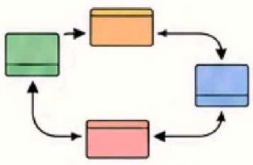

Operating system process lists

COMPARISON OF LINKED LIST TYPES

Type	Links per Node	Direction of Traversal	NULL at End	Memory Usage	Advantages	Disadvantages	Best Used For
Singly Linked List	1 (Next)	One (Forward)	Yes	Low	Simple, less memory, easy insertion	One-way traversal, hard to delete a given node	Stacks, Queues, Simple applications
Doubly Linked List	2 (Prev & Next)	Both	Yes	Medium	Two-way traversal, easy deletion	More memory, slightly complex	Browser history, Undo/Redo, Playlists
Circular Linked List (Singly)	1 (Next)	One (Circular)	No	Low	No NULL, can start from any node	One-way only, hard to delete	Round robin, Circular queues
Doubly Circular Linked List	2 (Prev & Next)	Both (Circular)	No	High	Most flexible, two-way circular	More memory, complex implementation	Deques, Advanced scheduling

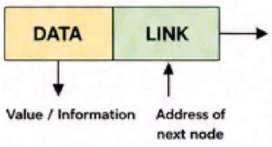
KEY TAKEAWAY

- ★ Linked lists are dynamic structures made of nodes connected by links.
- ★ Singly Linked List is simple and memory efficient but one-way.
- ★ Doubly Linked List allows two-way traversal and easy deletion.
- ★ Circular Linked List is useful for circular data processing and scheduling.
- ★ Doubly Circular Linked List is the most flexible but uses more memory.



LINKED LIST OPERATIONS
TRAVERSING, INSERTION, DELETION

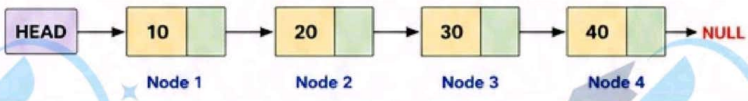
Node Structure



What is Linked List?

A linked list is a linear data structure in which elements (nodes) are not stored in contiguous memory locations. Each node contains data and a link (address) to the next node.

In a Singly Linked List, each node points to the next node and the last node points to NULL.



Legend

LINK (Address)

1. TRAVERSING (DISPLAY)

Traversing means visiting each node of the linked list from the beginning (HEAD) to the end (NULL) and processing (e.g., printing) the data.

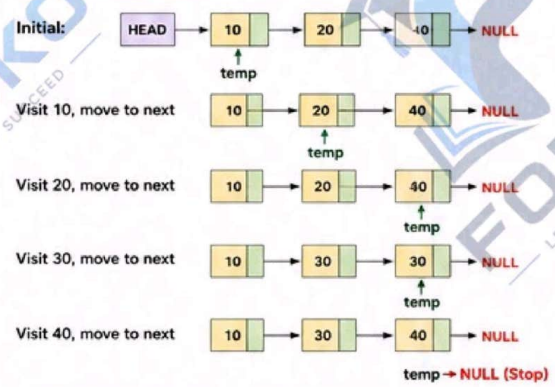
Algorithm

1. Start from HEAD.
2. Visit (process) the data part of the current node.
3. Move to the next node using the link part.
4. Repeat steps 2 and 3 until the next pointer is NULL.

Pseudocode

```
temp = HEAD
while (temp != NULL)
{
   visit(temp->data)
   temp = temp->link
}
```

Step-by-Step Example



Key Points

- Traversal is one-way (from HEAD to NULL).
- Time Complexity: O(n)
- Space Complexity: O(1)
- Used to display or search elements.

Example Output

10	20	30	40

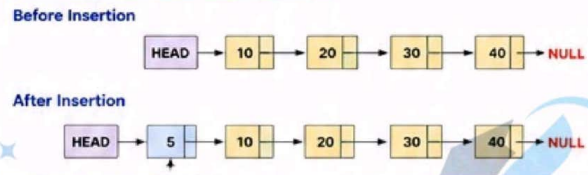
2. INSERTION

Insertion means adding a new node at a specified position.

Types of Insertion

1. At Beginning
2. At End
3. At Given Position (after a given node)

Example: Insert 5 at the beginning



2.1 INSERTION AT BEGINNING

Steps

1. Create a new node.
2. Set new node's link = HEAD.
3. Update HEAD = new node.

Pseudocode

```
new_node = (data)
new_node->link = HEAD
HEAD = new_node
```

2.2 INSERTION AT END

Steps

1. Create a new node.
2. If list is empty, set HEAD = new node.
3. Else, move to last node.
4. Set last node's link = new node.

Pseudocode

```
new_node = (data)
new_node->link = NULL
if (HEAD == NULL)
   HEAD = new_node
else
   temp = HEAD
   while (temp->link != NULL)
      temp = temp->link
   temp->link = new_node
```

2.3 INSERTION AFTER A GIVEN NODE (KEY)

Steps

1. Search for the node with data = KEY.
2. Create a new node.
3. Set new node's link = (node after KEY).
4. Set KEY node's link = new node.

Pseudocode

```
find node with data = KEY
if (node found)
{
   new_node = (data)
   new_node->link = key_node->link
   key_node->link = new_node
}
```

Key Points

- Insertion is easy and efficient in linked list.
- No need to shift elements (unlike arrays).
- Time Complexity:
  - At Beginning: O(1)
  - At End: O(n)
  - At Given Position: O(n)
- Space Complexity: O(1)

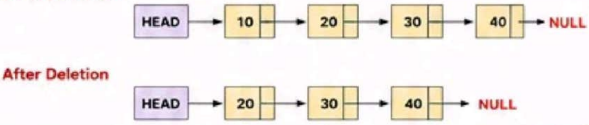
3. DELETION

Deletion means removing a node from the linked list.

Types of Deletion

1. From Beginning
2. From End
3. From Given Position (after a given node or by key)

Example: Delete 10 (first node)



3.1 DELETION FROM BEGINNING

Steps

1. If list is empty, Underflow.
2. Store HEAD in temp.
3. Move HEAD to next node.
4. Delete temp.

Pseudocode

```
if (HEAD == NULL)
   Underflow
else
{
   temp = HEAD
   HEAD = HEAD->link
   free(temp)
}
```

3.2 DELETION FROM END

Steps

1. If list empty or one node, handle separately.
2. Move to second last node.
3. Delete last node.

Pseudocode

```
if (HEAD == NULL) Underflow
else if (HEAD->link == NULL)
{
   free(HEAD)
   HEAD = NULL
}
else
{
   temp = HEAD
   while (temp->link->link != NULL)
      temp = temp->link
   free(temp->link)
   temp->link = NULL
}
```

3.3 DELETION OF A GIVEN NODE (KEY)

Steps

1. If first node is key, delete it.
2. Else, find node just before key node.
3. Change link to skip the key node.
4. Delete the key node.

Pseudocode

```
if (HEAD->data == KEY)
{
   temp = HEAD
   HEAD = HEAD->link
   free(temp)
}
else
{  prev = NULL
   curr = HEAD
   while (curr != NULL and curr->data != KEY)
   {
      prev = curr
      curr = curr->link
   }
   if (curr == NULL) Not Found
   else
   {  prev->link = curr->link
      free(curr)
   }
}
```

Key Points

- Deletion requires finding the previous node (except for first node).
- No need to shift elements.
- Time Complexity:
  - From Beginning: O(1)
  - From End: O(n)
  - At Given Position: O(n)
- Space Complexity: O(1)
- Handle special cases: empty list and single node list.

SUMMARY TABLE

Operation	Description	Time Complexity	Space Complexity
Traversing	Visit each node from HEAD to NULL	O(n)	O(1)
Insertion	Add new node	O(1) to O(n)	O(1)
Deletion	Remove a node	O(1) to O(n)	O(1)

IMPORTANT NOTES

- Linked list size is dynamic.
- Memory is allocated as per need.
- Efficient insertions and deletions.
- Extra memory is needed for links.
- No random access (sequential access only).

KEY TAKEAWAY

Linked list operations are efficient for insertions and deletions, but access (search/traversal) is slower compared to arrays.

TREE

Definition and Terminologies, Memory Representations

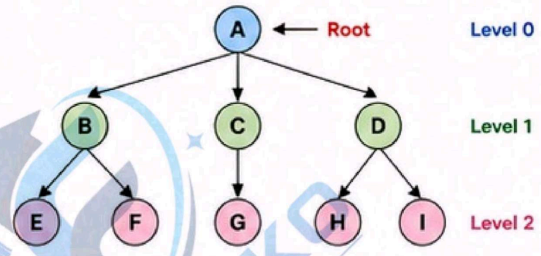
WHAT IS A TREE?

A tree is a non-linear hierarchical data structure consisting of nodes connected by edges.

It has a special node called Root, and the remaining nodes are partitioned into disjoint subtrees of the root.

- A tree with 'n' nodes has exactly (n – 1) edges.
- There is exactly one path between any two nodes.
- A tree does not contain any cycle.

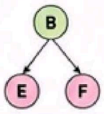
EXAMPLE OF TREE



1. DEFINITION AND TERMINOLOGIES

Root — The topmost node of the tree. It has no parent.

Parent — A node that has one or more children.

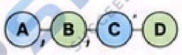


Child — A node that is directly connected below another node.

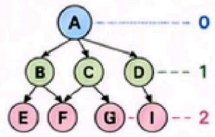
Siblings — Nodes that have the same parent.

Leaf Node / External Node — A node that does not have any children.

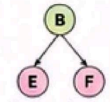
Internal Node / Non-leaf Node — A node that has at least one child.



Level — Level of a node is the number of edges from the root to that node. (Root is at level 0)



Height of Node — The height of a node is the number of edges on the longest path from that node to a leaf node.



Height(B) = 1

Height of Tree — The height of a tree is the height of the root node (i.e., longest path from root to a leaf).

Height of Tree = 2

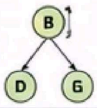
Degree of Node — The degree of a node is the number of children it has.

Degree(C) = 1
Degree(B) = 2

Degree of Tree — The degree of a tree is the maximum degree of any node in the tree.

Degree of Tree = 2 (because B has maximum 2 children)

Subtree — A subtree is a tree formed by a node and all its descendants.



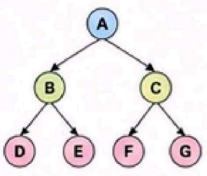
Subtree of A rooted at B

2. MEMORY REPRESENTATIONS OF TREE

2.1 SEQUENTIAL (ARRAY) REPRESENTATION

- Used mainly for Binary Trees (especially Complete Binary Trees).
- Nodes are stored level by level in an array.
- If a node is at index i (0-based indexing):
  - Left child → 2i + 1
  - Right child → 2i + 2
  - Parent → ⌊ (i – 1) / 2 ⌋

Example Tree



Array Representation (0-based index)

Index	0	1	2	3	4	5	6
Value	A	B	C	D	E	F	G

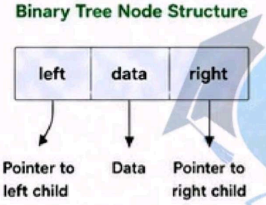
Note:

- Works efficiently when tree is complete or nearly complete.
- Not suitable for skewed (degenerate) trees as it wastes space.

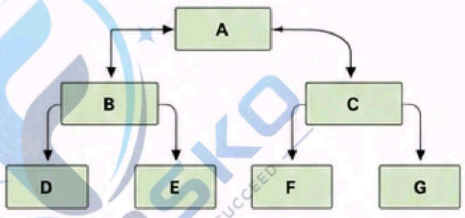
2.2 LINKED REPRESENTATION

- Each node is represented using a structure.
- Every node contains data and pointers to its children.
- For a binary tree, each node has two pointers: left and right.
- For a general tree, a node may contain multiple child pointers (using array or list).

Binary Tree Node Structure



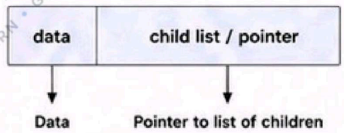
Example (Binary Tree)



Note:

- Efficient for skewed trees.
- Extra memory used for storing pointers.
- Easily supports dynamic size.

General Tree Node Structure (using list)



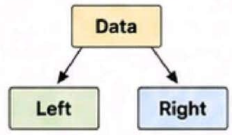
WHAT IS A TREE?

- A tree is a hierarchical data structure.
- It consists of nodes connected by edges.
- One special node is called **Root**.

TYPES OF TREES : BINARY TREES

A **Binary Tree** is a tree data structure in which each node has at most two children, referred to as the **Left child** and **Right child**.

BINARY TREE NODE



Left → Left Subtree
Right → Right Subtree

KEY PROPERTIES OF BINARY TREE

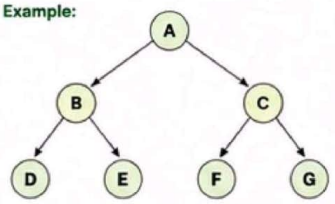
- Each node has at most 2 children.
- The left child is considered before the right child.
- The order of children matters (Left & Right).
- Binary Tree with 'n' nodes has at most (n − 1) edges.

TYPES OF BINARY TREES

1. FULL (STRICT) BINARY TREE

Every node has either 0 or 2 children.
No node has only one child.

Example:



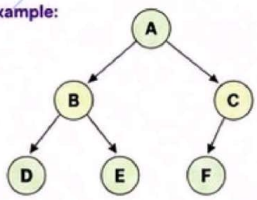
Properties:

- Every internal node has exactly 2 children.
- If the number of internal nodes = i, then number of leaves = i + 1.
- Max nodes at level l (root at level 0) = 2^l

2. COMPLETE BINARY TREE

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

Example:



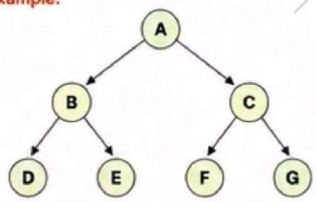
Properties:

- All levels except last are full.
- Last level nodes are as left as possible.
- For n nodes, height is minimum and ≤ $\lfloor \log_2 n \rfloor$.

3. PERFECT BINARY TREE

All internal nodes have 2 children and all leaves are at the same level.

Example:



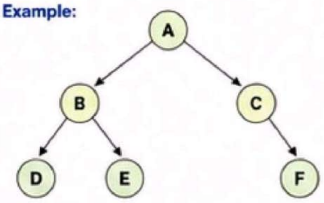
Properties:

- All levels are completely filled.
- Number of nodes = $2^{h+1} - 1$ (where h = height of tree, root at level 0)
- Number of leaves = 2^h

4. BALANCED BINARY TREE

For every node, the height difference between left and right subtree is at most 1.

Example:



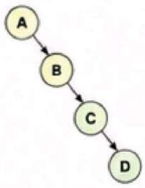
Properties:

- |Height(Left Subtree) − Height(Right Subtree)| ≤ 1 for every node.
- Used in Self-balancing Trees (AVL, Red-Black, etc.).

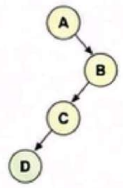
5. DEGENERATE (SKEWED) BINARY TREE

Every node has only one child.

Left Skewed Tree



Right Skewed Tree



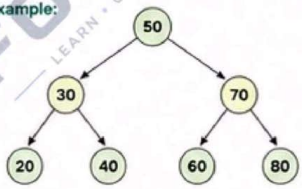
Properties:

- Each node has only one child.
- Number of nodes = height + 1.
- Behaves like a linked list.

6. BINARY SEARCH TREE (BST)

For every node:
Left subtree contains values < node data.
Right subtree contains valuen > node data.

Example:



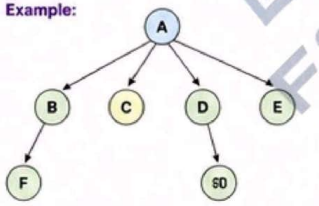
Properties:

- Left < Root < Right.
- Inorder traversal gives sorted order.
- Used in Searching, Sorting, Range queries.

7. EXTENDED / 2-ARY TREE

A generalization where a node can have more than 2 children.

Example:



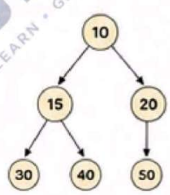
Properties:

- Node can have more than 2 children.
- Not a binary tree in strict sense.
- Used in file systems, organization charts.

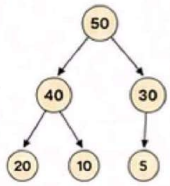
8. HEAP (MIN-HEAP / MAX-HEAP)

A Complete Binary Tree that satisfies Heap property.

Min-Heap Example



Max-Heap Example



Properties:

- Min-Heap: Parent ≤ Children.
- Max-Heap: Parent ≥ Children.
- Used in Priority Queues, Heap Sort.

COMPARISON OF BINARY TREE TYPES

Tree Type	Max Children per Node	All Levels Full?	Leaves at Same Level?	Height (h) for n nodes	Main Use / Feature
Full (Strict)	2	No	No	≤ n − 1	No node with only one child
Complete	2	Except Last	No	≤ $\lfloor \log_2 n \rfloor$	Efficient storage, used in Heaps
Perfect	2	Yes	Yes	$\log_2(n+1) - 1$	Ideal tree, all levels full
Balanced	2	No	No	≤ $\lfloor \log_2 n \rfloor$	Keeps tree height small
Degenerate (Skewed)	1	No	No	= n − 1	Worst case, like linked list
BST	2	No	No	Depends	Searching, Sorting operations
Extended (2-ary)	> 2	No	No	Depends	Multi-way representation
Heap	2	Complete	Not Necessarily	≤ $\lfloor \log_2 n \rfloor$	Priority Queue, Heap Sort

KEY TAKEAWAYS

- Binary Tree is the foundation of many advanced data structures.
- Full, Complete and Perfect trees are special forms of Binary Tree.
- Balanced trees keep operations efficient.
- BSTs maintain sorted data for fast search operations.
- Heaps are complete binary trees with special ordering.

NOTE:
All the above trees are types of Binary Trees either by structure (shape) or by property (ordering).

COMPLETE BINARY TREES

Definition, Properties, Examples, Representation and Uses

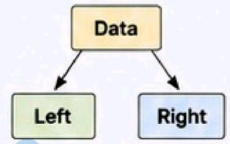
WHAT IS A TREE?

A tree is a non-linear data structure in which nodes are connected by edges.
One special node is called Root and the remaining nodes are partitioned into disjoint subtrees.

DEFINITION

A Complete Binary Tree is a binary tree in which all the levels are completely filled except possibly the last level, and all nodes in the last level are as far left as possible.

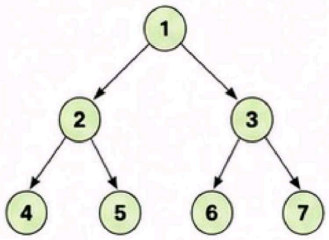
BINARY TREE NODE



Left → Left Subtree
Right → Right Subtree

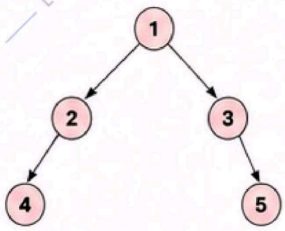
EXAMPLES

Complete Binary Tree (Valid)



- All levels are full except the last level.
- Last level nodes are as far left as possible.

Not a Complete Binary Tree (Invalid)



- Last level is not filled from the left.
- Node 5 is present but left position (under 2) is empty while a right position exists.

PROPERTIES

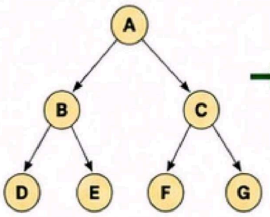
1. All levels are completely filled except possibly the last level.
2. All nodes of the last level are placed as far left as possible.
3. If a complete binary tree has 'n' nodes, the number of nodes in the last level can be from 1 to 2^h (where h = height of the tree, root at level 0).
4. A complete binary tree with h levels has at least 2^h nodes and at most $2^{h+1} - 1$ nodes.
5. Efficient for array representation and commonly used in Heap data structure.

NOTE

Every Perfect Binary Tree is a Complete Binary Tree, but every Complete Binary Tree is not necessarily Perfect.

ARRAY (SEQUENTIAL) REPRESENTATION

- In a complete binary tree, nodes can be stored in an array without wasting space.
- Nodes are stored level by level from left to right.



Array Representation (0-based Indexing)

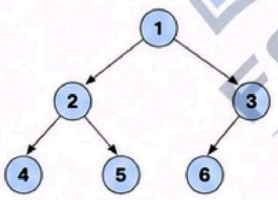
Index	0	1	2	3	4	5	6
Value	A	B	C	D	E	F	G

Index Relations (0-based indexing)

- Left child of node at index i → 2i + 1
- Right child of node at index i → 2i + 2
- Parent of node at index i → (i - 1) / 2
- Root is stored at index 0

REPRESENTATION EXAMPLE

- Consider a complete binary tree with 6 nodes.
- Height = 2 (levels 0, 1, 2)
- Last level is filled from the left.



Array (0-based indexing)

Index	0	1	2	3	4	5
Value	1	2	3	4	5	6

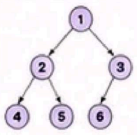
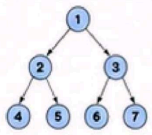
Why Use Array?

- No pointers needed.
- Fast access to parent/children.
- Less memory overhead.
- Ideal for heaps and priority queues.

APPLICATIONS

Heaps	Complete Binary Trees are used to implement Min-Heap and Max-Heap.
Priority Queues	Efficient insertion and deletion of elements.
Array Based Implementation	Enables efficient storage and memory utilization.
Used in Algorithms	Used in various algorithms like Heap Sort, Tree Traversals, etc.

COMPARISON

Feature	Complete Binary Tree	Perfect Binary Tree
All levels full?	Except possibly last	Yes
Last level nodes	As far left as possible	Completely filled
Number of nodes at level i (0-based)	At most 2^i	Exactly 2^i
Total nodes in h levels	2^h to $2^{h+1} - 1$	$2^{h+1} - 1$
Example		

KEY TAKEAWAYS

- All levels are full except possibly the last level.
- Last level nodes are as far left as possible.
- Efficient array representation.
- Used widely in heaps and priority queues.
- Number of nodes in h levels is between 2^h and $2^{h+1} - 1$.

Quick Fact

If a complete binary tree has n nodes, then height h = $\lfloor \log_2 n \rfloor$

BST OPERATIONS
INSERTION & DELETION

Maintain BST property by comparing keys and moving left or right

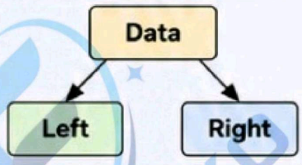
BST PROPERTY

- ✔ All keys in left subtree < node key
- ✔ All keys in right subtree > node key

WHAT IS BST?

A Binary Search Tree (BST) is a binary tree in which, for every node, all keys in the left subtree are smaller and all keys in the right subtree are greater.

NODE STRUCTURE



1. INSERTION IN BST

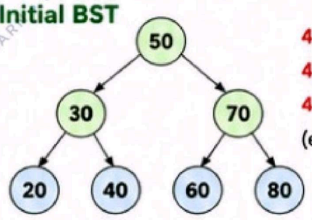
Insert a new key by finding the correct position using comparisons.

ALGORITHM

1. If the tree is empty, create the new node and make it root.
2. Compare the key with current node.
   - If key < node key → go to left subtree.
   - If key > node key → go to right subtree.
3. Repeat until an empty position is found.
4. Insert the new node at that position.

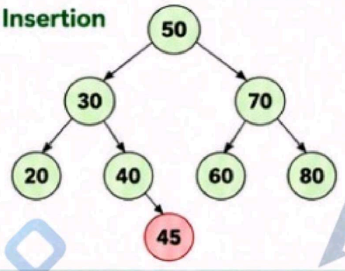
EXAMPLE: Insert 45

Initial BST



45 < 50 → go left
45 > 30 → go right
45 > 40 → go right
(empty) → insert here

BST after Insertion



TRACE TABLE

Step	Current Node	Comparison	Move
1	50	45 < 50	Left
2	30	45 > 30	Right
3	40	45 > 40	Right
4	NULL	Insert 45	Stop

TIME COMPLEXITY

- Best Case : **O(1)** (tree is empty)
- Average Case : **O(log n)** (balanced BST)
- Worst Case : **O(n)** (skewed BST)

2. DELETION IN BST

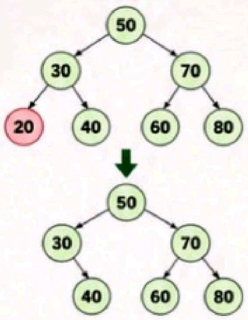
Delete a node and maintain BST property.

ALGORITHM

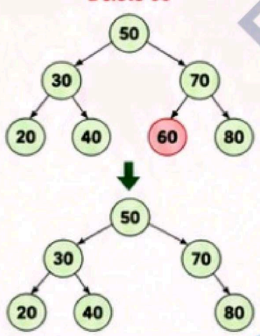
1. Find the node to be deleted using BST search.
2. There are three cases :
   - Case 1 : No child
   - Case 2 : One child
   - Case 3 : Two children
3. Perform the appropriate case and maintain BST property.

THREE CASES OF DELETION

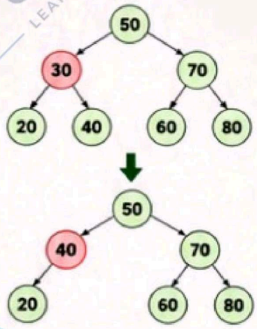
Case 1 : No Child (Leaf Node)
Delete 20



Case 2 : One Child
Delete 60



Case 3 : Two Children
Delete 30



How Case 3 Works?

Find inorder successor (smallest in right subtree) or inorder predecessor (largest in left subtree), replace the node, then delete that successor/predecessor.

Example (Case 3 using Inorder Successor)

1. Inorder successor of 30 is 40.
2. Replace 30 with 40.
3. Delete the successor node 40 from its original position.

Inorder Sequence

Before : 20, 30, 40, 50, 60, 70, 80
After : 20, 40, 50, 60, 70, 80
(Sorted order maintained)

TIME COMPLEXITY (Deletion)

- Best Case : O(1)
- Average Case : O(log n)
- Worst Case : O(n) (Same as search)

NOTE

After every insertion or deletion, the BST property must remain true for the tree.
(Left < Node < Right)

KEY TAKEAWAYS

- Insertion is done at a leaf position after comparisons.
- Deletion has 3 cases: No child, One child, Two children.
- BST operations are efficient for balanced trees (O(log n)).
- Skewed BST behaves like a linked list (O(n)).

Better Data Structures → Better Programs → A Better You!