Tree Structure & Binary Tree

layvathna@gmail.com

Linear to Hierarchical

Recap of what we have so far

- Arrays: fast index access; layout is strictly linear; resizing/insert-in-middle is costly.
- Linked lists: flexible inserts/removes; still a single chain; no random access.
- Stacks: LIFO discipline — always work at one end (top).
- Queues: FIFO discipline — always work at two ends (front/back).

From Linear to Hierarchical

Some datasets are not a single line of items. They form multi-level categories where one item has multiple children and operations care about an item with all of its descendants.

- University org: university → faculties → departments → programs → courses; we often need “all courses under Department X”.
- File system: folders containing subfolders/files; move a whole folder (reparent a subtree) efficiently.
- E-commerce navigation: category → subcategory → brand → product; render level-by-level menus fast.

From Linear to Hierarchical

Requirements that force a new structure

- Branching (degree > 1): a node can have many children.
- Level awareness: show data by levels (root, level-1, level-2...).
- Reparenting: move a whole group from one parent to another without touching unrelated items.

Core element

- Root: the single node with no parent.
- Parent / Child / Siblings: family relations.
- Degree of a node: number of children.
- Depth (level): distance from the root.
- Height: longest path from a node down to a leaf; tree height = height(root).
- Leaf: node with degree 0 (no children).
- Subtree: a node and all of its descendants.
- Forest: a set of trees (multiple roots).

Tree (Design Thinking)

- Choose the root perspective: What is the topmost entity? (e.g., / (root folder) or University.)
- Partition into levels: For each node, list its immediate components—not all descendants at once.
- Decide node payload: What minimal info does each node store? (name/id, size, path, etc.)
- Prioritize operations: What must be fast? (e.g., “iterate children of X”, “find parent of Y”, “count subtree size”.)

Array vs Linked-List (Decision Guide)

Operation/ Concern	Array-Based Children	Linked-List Children
Iterate all children of X	Very fast, cache-friendly	OK but pointer-chasing
Access k-th child	O(1)	O(k)
Append at end	Amortized O(1) (with capacity)	O(1) with tail pointer
Insert/remove in middle	O(d) shifts	O(1) if position known
Memory overhead per child	Low (contiguous)	Higher (next/prev pointers)
Repeated structural edits	Less friendly	Very friendly
Typical use	Read-heavy, stable order	Update-heavy, flexible order

Traversal (Conceptual Only)

- Preorder (root-first): Visit node, then process each child subtree.
  - Good for: exporting a tree to a linear outline, copying structure.
- Postorder (root-last): Process all children first, then the node.
  - Good for: safe deletion, computing aggregated properties from leaves upward.
- Level-order (BFS): Visit nodes level by level using a queue.
  - Good for: shortest-level insights, UI rendering by levels.

Conclusion

- Fit to scenario: Choose trees when you need branching, subtree operations, level awareness, or reparenting (move a whole group at once).
- Hierarchy ≠ sequence: trees are for branching structure, not just before/after.
- Pick by workload: let your common operations dictate array vs. linked-list children.
- Postorder for teardown/roll-ups: compute or delete from leaves up to avoid mistakes.
- Representation follows operations:
  - Array-based children → read-heavy, order matters, quick k-th child, fast scans.
  - Linked-list children → update-heavy, frequent inserts/removes/moves, stable pointers.

General Tree

- General Tree: Nodes can have any number of children.
- It is useful in real-world hierarchies (like file systems, organizational charts, or family trees).
- However, when it comes to programming, especially in languages like C, general trees pose some practical challenges.

General Trees Problem

- While general trees are conceptually powerful, they are not always ideal for programming because:
  - Variable Number of Children
  - Pointer Management
  - Algorithm Complexity

Binary Trees: A Practical Tree

- A binary tree is a fundamental type of tree where each node has at most two children. Binary trees are widely used because they provide a simple yet powerful structure for many algorithms.
- There are two common ways to represent binary trees:
  - Linked List Representation (Dynamic)
  - Array Representation (Static)

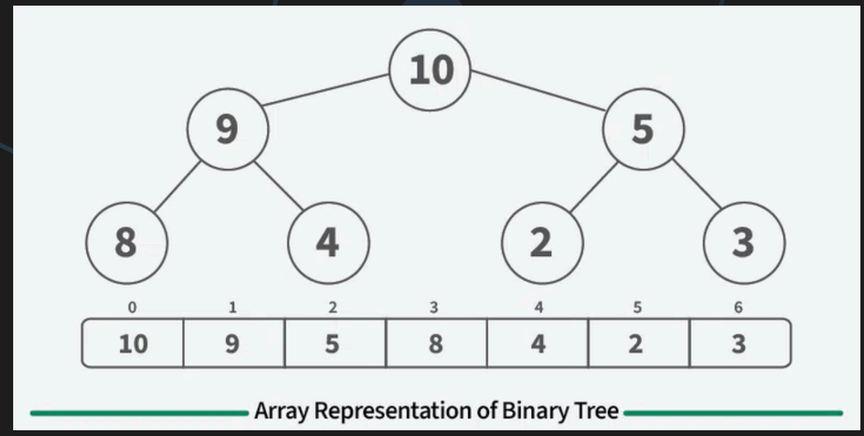
Binary Trees: Linked List

- Each node is defined with:
  - A data field.
  - A pointer (or reference) to its left child.
  - A pointer to its right child.

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

Binary Trees: Array

- Suitable for complete or nearly complete binary trees.
- The root is stored at index 0. For any node at index i:
- Left child is at index 2*i + 1
- Right child is at index 2*i + 2
- This method is space-efficient when the tree is complete but can waste space if the tree is sparse.



Tree Traversals in Binary Trees

- Traversal is the process of visiting each node in a tree in a specific order. For binary trees, three primary methods are used:
  - Inorder Traversal
  - Preorder Traversal
  - Postorder Traversal

Inorder Traversal

- Process:
  - Recursively traverse the left subtree.
  - Visit the root node.
  - Recursively traverse the right subtree.
- Characteristic: In a binary search tree (covered later), inorder traversal retrieves data in sorted order.

```c
void inorderTraversal(Node *root) {
    if (root == NULL) return;
    inorderTraversal(root->left);
    printf("%d ", root->data);
    inorderTraversal(root->right);
}
```

Preorder Traversal

- Process:
  - Visit the root node.
  - Recursively traverse the left subtree.
  - Recursively traverse the right subtree.
- Usage: Often used to copy a tree or to get a prefix expression of an expression tree.

```
void preorderTraversal(Node *root) {
    if (root == NULL) return;
    printf("%d ", root->data);
    preorderTraversal(root->left);
    preorderTraversal(root->right);
}
```

Postorder Traversal

- Process:
  - Recursively traverse the left subtree.
  - Recursively traverse the right subtree.
  - Visit the root node.
- Usage: Useful for deleting the tree (freeing nodes) or evaluating postfix expressions in expression trees.

```c
void postorderTraversal(Node *root) {
    if (root == NULL) return;
    postorderTraversal(root->left);
    postorderTraversal(root->right);
    printf("%d ", root->data);
}
```

Summary

- Trees are non-linear, hierarchical structures composed of nodes.
- Essential terminology includes root, child, parent, leaf, subtree, height, and depth.
- Binary trees limit nodes to at most two children, and they can be implemented using either linked or array representations.
- The three common tree traversal techniques are:
  - Inorder: Left, Root, Right.
  - Preorder: Root, Left, Right.
  - Postorder: Left, Right, Root.