Trees

A **tree** is a hierarchical data structure composed of nodes connected by edges. Unlike arrays or linked lists which are **linear**, trees represent **non-linear** relationships. This makes them suitable for representing hierarchical models such as file systems, organization charts, and syntax trees.

1 Tree vs. Linear Structures

- **Arrays/Linked Lists:** Elements arranged linearly, one after another.
- **Trees:** Each node can have multiple children, forming a hierarchy.
- **Use Cases:** Trees are preferred where branching or hierarchy is essential.

2 Example: Directory Structure

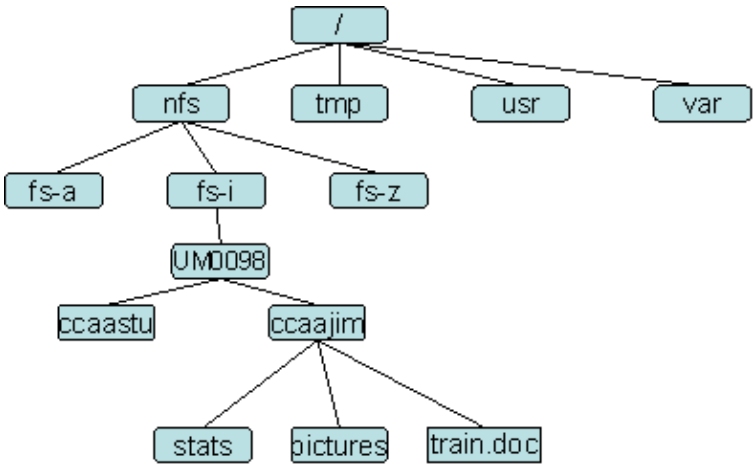


Figure 1: Tree representation of a directory structure

3 Terminology

- **Root:** The topmost node
- **Parent:** A node with child nodes
- **Child:** A node that descends from another node
- **Leaf:** A node with no children
- **Subtree:** A tree formed by a node and its descendants
- **Depth:** Distance from the root to a node

- **Height:** Length of the longest path to a leaf

4 Tree traversal

Preorder Traversal visits nodes in the order:

`Root → Children (Left to Right)`

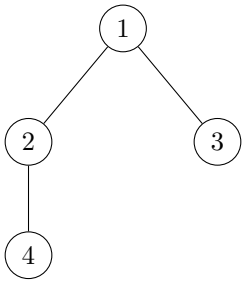
That is, it first processes the current node, then recursively visits each child subtree. This is useful when you want to copy or serialize the entire structure from the top down.

Postorder Traversal visits nodes in the order:

`Children (Left to Right) → Root`

It first traverses all child subtrees recursively, and then processes the current node. This is useful for tasks such as deleting the tree or evaluating expression trees (bottom-up computation).

Example Tree:



Preorder: $1 \to 2 \to 4 \to 3$
Postorder: $4 \to 2 \to 3 \to 1$

5 C++ Representation of a Tree

Below is a basic representation of a general tree where each node can have multiple children.

We define a `TreeNode` struct for individual nodes and a `Tree` class that manages the tree and includes traversal methods as member functions.

```cpp
#include <iostream>
#include <vector>
using namespace std;

struct TreeNode {
  int data;
  vector<TreeNode*> children;
  TreeNode(int val) : data(val) {}
};

class Tree {
public:
  TreeNode* root;

  Tree(int rootVal) {
    root = new TreeNode(rootVal);
  }

  // helper to add a new child under 'parent'
  TreeNode* addChild(TreeNode* parent, int val) {
    TreeNode* node = new TreeNode(val);
    parent->children.push_back(node);
    return node;
```

```
  }

  // preorder traversal
  void preorder(TreeNode* node) const {
    if (!node) return;
    cout << node->data << " ";
    for (TreeNode* c : node->children)
      preorder(c);
  }
  void preorder() const { preorder(root); }

  // postorder traversal
  void postorder(TreeNode* node) const {
    if (!node) return;
    for (TreeNode* c : node->children)
      postorder(c);
    cout << node->data << " ";
  }
  void postorder() const { postorder(root); }
};
```

Usage:

```
Tree t(1);
TreeNode* n2 = t.addChild(t.root, 2);
TreeNode* n3 = t.addChild(t.root, 3);
TreeNode* n4 = t.addChild(n2, 4);

t.preorder();   // prints: 1 2 4 3
t.postorder();  // prints: 4 2 3 1
```

Note

For binary trees (each node has at most 2 children), ‘inorder‘ traversal is also common (Left → Root → Right), but is not applicable to general trees.

6 Practice problem

Leetcode Problem 104: *Tree height*
`https://leetcode.com/problems/maximum-depth-of-binary-tree/description`