0.1 Tree Traversal: Inorder, Preorder, Postorder

Unlike linear data structures such as arrays, linked lists, queues, and stacks, which have only one logical way to traverse them (typically from front to back or back to front), trees are non-linear, hierarchical structures. This non-linearity means that there are multiple distinct paths and ways to visit the nodes within a tree. The process of visiting every single node in a tree exactly once in a systematic way is referred to as **tree traversal**.

Tree Traversal

Tree traversal is a fundamental algorithmic process in which every node in a tree data structure is visited exactly once in a systematically defined sequence. The goal is to process or examine the data stored at each node, such as printing its value, updating its contents, evaluating an expression, or utilizing it in a broader computational operation.

Traversals are primarily categorized into two overarching approaches based on their exploration strategy: **Depth-First Search (DFS)** and **Breadth-First Search (BFS)**. In a Depth-First Search, the algorithm dives as deeply as possible down a single branch of the tree before it is forced to backtrack. Conversely, in a Breadth-First Search (often called Level-Order Traversal), the algorithm visits nodes level by level, moving from top to bottom and left to right across the same depth.

This section focuses entirely on the Depth-First Search methods specifically designed for binary trees. Depth-first approaches for binary trees include three distinct patterns based entirely on when the *root node* (or parent node) is processed relative to its left and right subtrees. These three traversal methods are **Preorder**, **Inorder**, and **Postorder**.

To illustrate these three traversals effectively and clearly, we will reference a standard binary tree as a running example throughout this section.

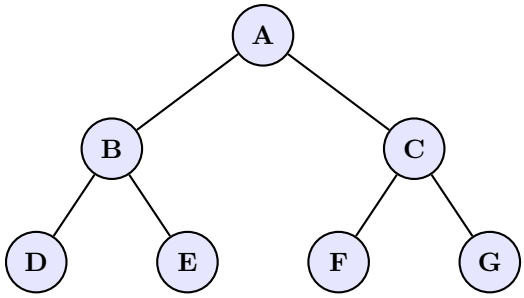

Figure 1: A sample binary tree used to illustrate Preorder, Inorder, and Postorder traversals.

0.1.1 Preorder Traversal (Node - Left - Right)

In a Preorder traversal, the algorithm processes the current **Node** (the root of the current subtree) first, before moving on to recursively explore the **Left** subtree, and finally recursively exploring the **Right** subtree. Because of this specific sequence, this method is often abbreviated as **NLR** (Node-Left-Right).

Key Takeaways

Preorder Traversal Rule:
① Visit the Root Node.
② Traverse the Left Subtree in Preorder.
③ Traverse the Right Subtree in Preorder.

Let us carefully trace the Preorder traversal algorithm on our sample binary tree:

1. We start our traversal at the absolute root node, **A**. We process and visit **A** first.
2. Following the NLR rule, we must now move to the left subtree of A, which is rooted at **B**. Since B is now the root of this new subtree, we visit **B**.
3. We then move to the left subtree of B, which is **D**. We visit **D**.
4. Node **D** is a leaf node and has no children, so we backtrack to B. B's left subtree is finished, so we move to B's right child, **E**. We visit **E**.
5. Now the entire left branch of A (consisting of B, D, and E) is fully traversed. We backtrack all the way up to A and move to its right subtree, rooted at **C**. We visit **C**.
6. We move to the left subtree of C, which is **F**. We visit **F**.
7. Finally, we backtrack to C and visit its right child, **G**.

Resulting Preorder Sequence: A, B, D, E, C, F, G

Application of Preorder Traversal

Preorder traversal is commonly used when you need to create a *replica* or exact clone of a given tree. Because the root is always visited first, you can easily instantiate the root node of the new tree before recursively instantiating and attaching its left and right subtrees. In compiler design, Preorder traversal is also used to generate a *prefix expression* (Polish notation) from an expression tree.

0.1.2 Inorder Traversal (Left - Node - Right)

In an Inorder traversal, the algorithm prioritizes exploring the **Left** subtree as far down as possible first, then it processes the current **Node** (root), and finally, it explores the **Right** subtree. This sequence is abbreviated as **LNR** (Left-Node-Right).

Key Takeaways

Inorder Traversal Rule:
① Traverse the Left Subtree in Inorder.
② Visit the Root Node.
③ Traverse the Right Subtree in Inorder.

Let us trace the Inorder traversal on our sample tree step-by-step:

1. Start at **A**. We cannot visit A yet; we must traverse its entire left subtree first. We move to **B**.
2. At **B**, we still cannot visit it. We must traverse its left subtree first. We move to **D**.
3. **D** has no left child to explore, so we finally visit **D**.
4. Having completely processed the left subtree of B (which was just D), we move up and can now visit the root of this subtree, **B**.
5. Next, we must traverse the right subtree of B. We move to **E**. Since E has no left child, we visit **E**.
6. At this point, the entire left subtree of A (nodes D, B, E) is complete. We can now visit the main root node, **A**.
7. We must now traverse the right subtree of A, rooted at **C**. We move to C, but first, we must go to its left child, **F**.
8. F has no left child, so we visit **F**. Then we move up and visit its parent, **C**.
9. Finally, we traverse the right child of C, which is **G**. We visit **G**.

Resulting Inorder Sequence: D, B, E, A, F, C, G

Important Property of Binary Search Trees

If you perform an inorder traversal on a specialized structure called a **Binary Search Tree (BST)**, the resulting output sequence will always be a list of the tree's elements sorted in strictly *ascending order*. This mathematical property is one of the primary reasons BSTs exist and makes Inorder traversal absolutely critical for searching and sorting algorithms.

0.1.3 Postorder Traversal (Left - Right - Node)

In a Postorder traversal, the algorithm takes a bottom-up approach. It explores both the **Left** and **Right** subtrees completely before finally returning to process the parent **Node** (root). This is abbreviated as **LRN** (Left-Right-Node).

Key Takeaways

Postorder Traversal Rule:
① Traverse the Left Subtree in Postorder.
② Traverse the Right Subtree in Postorder.
③ Visit the Root Node.

Let us carefully trace the Postorder traversal on our sample tree:

1. Start at **A**. We must traverse its left and right subtrees completely before we are allowed to visit A.
2. Go to **B**. We must traverse its subtrees completely. Go to its left child, **D**.

3. **D** has no children. Both of its non-existent subtrees are "finished," so we visit **D**.
4. Before visiting B, we must traverse its right child, **E**. **E** has no children, so we visit **E**.
5. Now that both the left and right subtrees of **B** have been processed, we can finally visit **B**.
6. We return to A. Its left subtree is done, but we must process its right subtree before visiting A. We move to **C**.
7. We traverse C's left child **F**. It has no children, so we visit **F**.
8. We traverse C's right child **G**. It has no children, so we visit **G**.
9. Both subtrees of **C** are completely processed, so we can now visit **C**.
10. Finally, both the left subtree (B, D, E) and the right subtree (C, F, G) of the main root **A** are complete. We can now visit **A**.

Resulting Postorder Sequence: D, E, B, F, G, C, A

Application of Postorder Traversal

Postorder traversal is the standard, safest algorithm used to **delete a tree** from a computer's RAM. Because the parent node is evaluated and deleted *only after* all its children have been deleted, the algorithm inherently guarantees that children nodes are freed first. This prevents severe memory leaks and "dangling pointers" from crashing a program. It is also utilized by compilers to generate *postfix expressions* (Reverse Polish notation).

0.2 Applications of Binary Trees

Binary trees are not just abstract mathematical concepts; they serve as the crucial structural backbone for numerous highly efficient data structures and algorithms in computer science. By organizing data hierarchically rather than sequentially, trees can break down complex computational problems, yielding vastly superior performance compared to flat structures like arrays or linked lists.

Below are the most prominent and practical real-world applications of binary trees:

0.2.1 1. Expression Trees in Compilers

An expression tree is a specific type of binary tree uniquely designed to represent mathematical, logical, and algebraic expressions. In these tree structures, the *leaf nodes* always contain operands (such as numerical constants or variables like 5, x, y), while the *internal nodes* (including the root) contain operators (such as $+$, $-$, $\times$, $\div$).

Compilers build expression trees automatically during the syntax analysis phase of compiling a program. The tree perfectly captures the precedence and associativity of operators in a mathematical formula written by a programmer. By traversing an expression tree using Inorder, Preorder, or Postorder traversals, a compiler can automatically translate human-readable code into postfix or prefix instructions that the computer's CPU can rapidly execute.

0.2.2 2. Binary Search Trees (BST) for Fast Lookups

A Binary Search Tree is a specialized binary tree where data is stored systematically: the left child of any given node contains only values strictly less than the node's value, and the right child contains only values strictly greater than the node's value.

This strict left/right organization creates an incredibly fast "divide and conquer" search environment. Every time a search query is performed, comparing the target value to a node allows the algorithm to instantly eliminate half of the remaining tree from consideration. This results in an average search, insertion, and deletion time complexity of $O(\log n)$, a massive speedup compared to an array's $O(n)$ time. BSTs form the foundational blueprint for more complex, self-balancing database tree architectures, such as AVL Trees and Red-Black Trees.

0.2.3 3. Huffman Coding for Data Compression

Huffman coding is an elegant algorithm used globally for lossless data compression, and it relies entirely on a specialized binary tree known as a **Huffman Tree**. Whenever you compress files into a ZIP archive, send data over a slow network, or save an image in the JPEG format, the computer uses Huffman trees under the hood. The algorithm evaluates the data and assigns shorter binary codes (fewer bits) to characters that appear very frequently, and longer binary codes to characters that appear rarely. The characters themselves are placed at the leaf nodes of a binary tree. The path taken from the root to the leaf dictates the binary sequence for that character (e.g., branching left appends a '0', branching right appends a '1'). This hierarchical mapping drastically reduces the overall disk size of the file.

0.2.4 4. Heaps and Priority Queues

A Heap is a complete binary tree that strictly satisfies the *heap property*. In a **Max-Heap**, the parent node is guaranteed to be greater than or equal to its children, meaning the absolute largest mathematical element is always located at the root. Conversely, in a **Min-Heap**, the absolute smallest element is always at the root.

Heaps are the preferred underlying data structure used to implement **Priority Queues**. A priority queue behaves like a normal line or queue, but elements are served based on their assigned priority level rather than the chronological order in which they arrived. Priority queues are deeply integrated into operating systems for CPU process scheduling (deciding which system task is most critical to run next) and are essential for advanced graph algorithms like Dijkstra's Shortest Path algorithm.

0.2.5 5. Routing Tables in Networking

In modern computer networks and the global Internet, hardware routers use specialized variations of binary trees—such as **Tries** (also known as Prefix Trees)—to store massive routing tables. When a packet of data arrives at a network router, the router must instantaneously determine the most efficient physical path to forward the packet based on its destination IP address.

By traversing a binary structure where each branch represents a single bit (0 or 1) of the target IP address, routers can perform "longest-prefix matching" in fractions of a millisecond. This hierarchical routing prevents bottlenecking, allowing immense volumes of internet traffic to flow smoothly across the globe.

Tree Type	**Primary Application Area**
Expression Tree	Compiler design, Evaluating mathematical formulas, Syntax trees.
Binary Search Tree	Fast database indexing, Dynamic dictionaries, High-speed searching algorithms.
Huffman Tree	Lossless data compression algorithms (ZIP, JPEG, MP3, PNG formats).
Heap Tree	Priority queues, CPU process scheduling, Heap Sort algorithm.
Trie (Prefix Tree)	High-speed IP Routing, Network forwarding, Autocomplete features in search engines.

Table 1: Summary of Key Binary Tree Applications in Computing