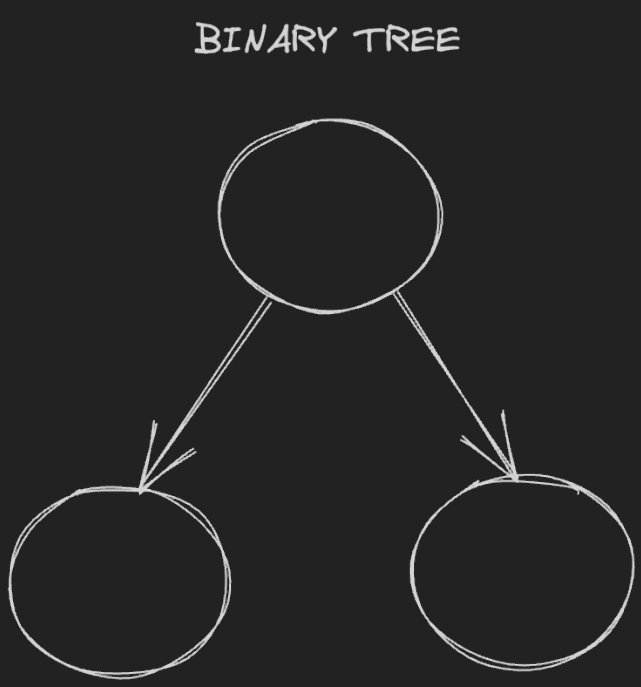
Operation	Tree
Access Root	O(1)
Traversal	O(n)
Parent	O(1)
Child	O(1)
Height	O(log n)

Definition

A tree is a hierarchical data structure consisting of nodes connected by edges. It starts with a **root** node, which is the topmost node in the tree. Each node can have zero or more **child** nodes, except for the leaf nodes, which are nodes with no children. Every node, except the root, has a **parent** node that is connected to it through an edge.

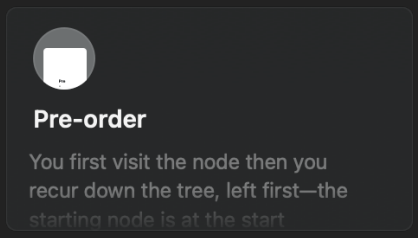


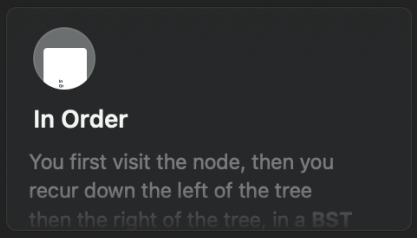
Terminology

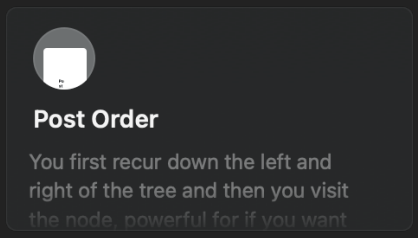
Root	The first node in a tree
Height	The longest path from the root, to the most child node
Binary Tree	A tree in which has at most 2 children, at least 0 children
General Tree	A tree with 0 or more children
Binary Search Tree	A tree in which has a specific odering to the ndoes and at most 2 children
Leaf Node	A node without children, they are the end nodes of a tree
Child Node	If it is connected to another node below it
Parent Node	The node a child node is connected to
Balanced	A tree is *perfectly* balanced when any node's left and right children have the same height
Branching Factor	The amount of children a tree has

Traversal

Tree traversal is the process of visiting each node in a tree exactly once. There are different traversal algorithms, such as **pre-order**, **in-order**, and **post-order**, which define the order in which the nodes are visited.





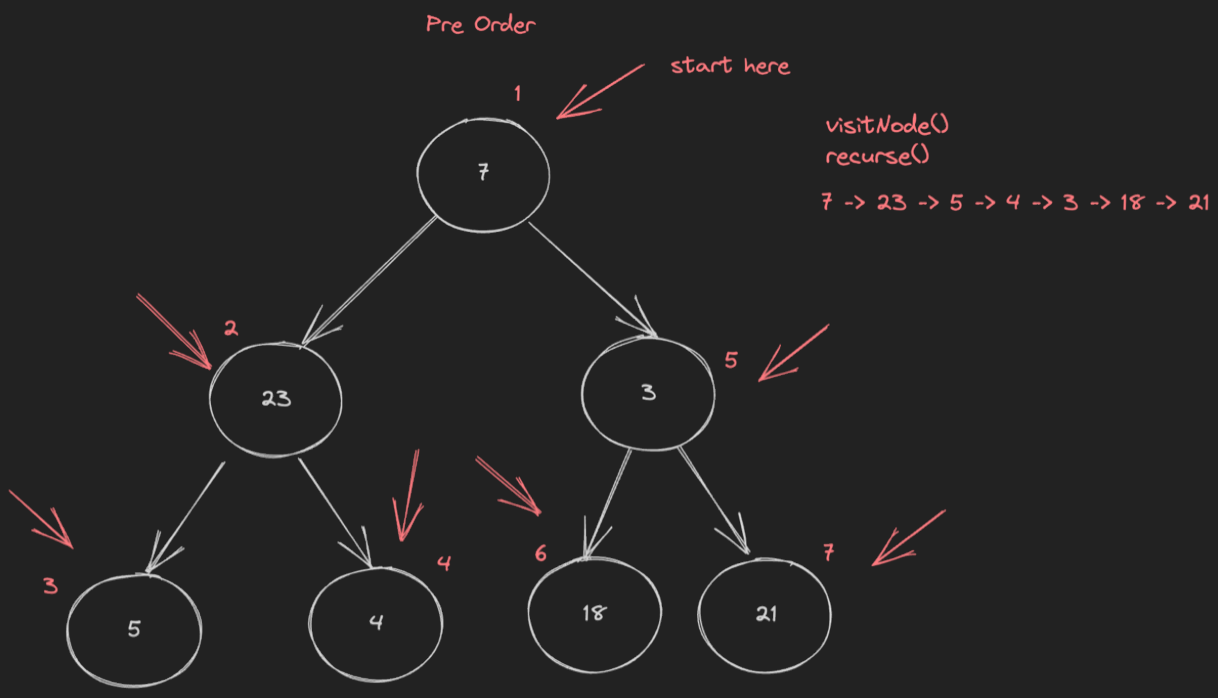


Use Cases

- Representing hierarchical data, such as file systems or organization charts
- Implementing search algorithms like binary search trees
- Parsing expressions or representing arithmetic expressions

↑ Trees

Pre-order

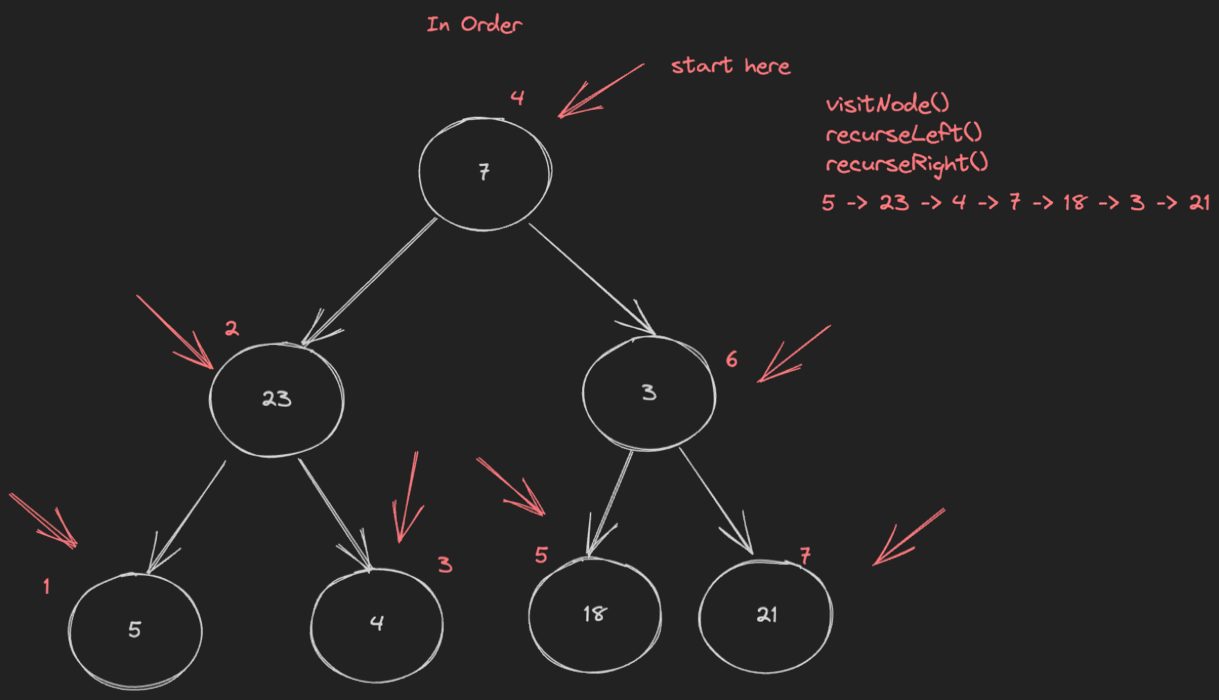


You first visit the node then you recur down the tree, left first—the starting node is at the start

```python
class BinaryNode:
    def __init__(self, value: int):
        self.value = value
        self.left = None
        self.right = None

def walk(curr: Union[BinaryNode, None], path: List[int]) -> List[int]:
    if curr is None:
        return path

    path.append(curr.value)

    # Recursive calls
    walk(curr.left, path)
    walk(curr.right, path)

    return path

def pre_order_search(head: BinaryNode) -> List[int]:
    return walk(head, [])
```

↑ Trees

In Order

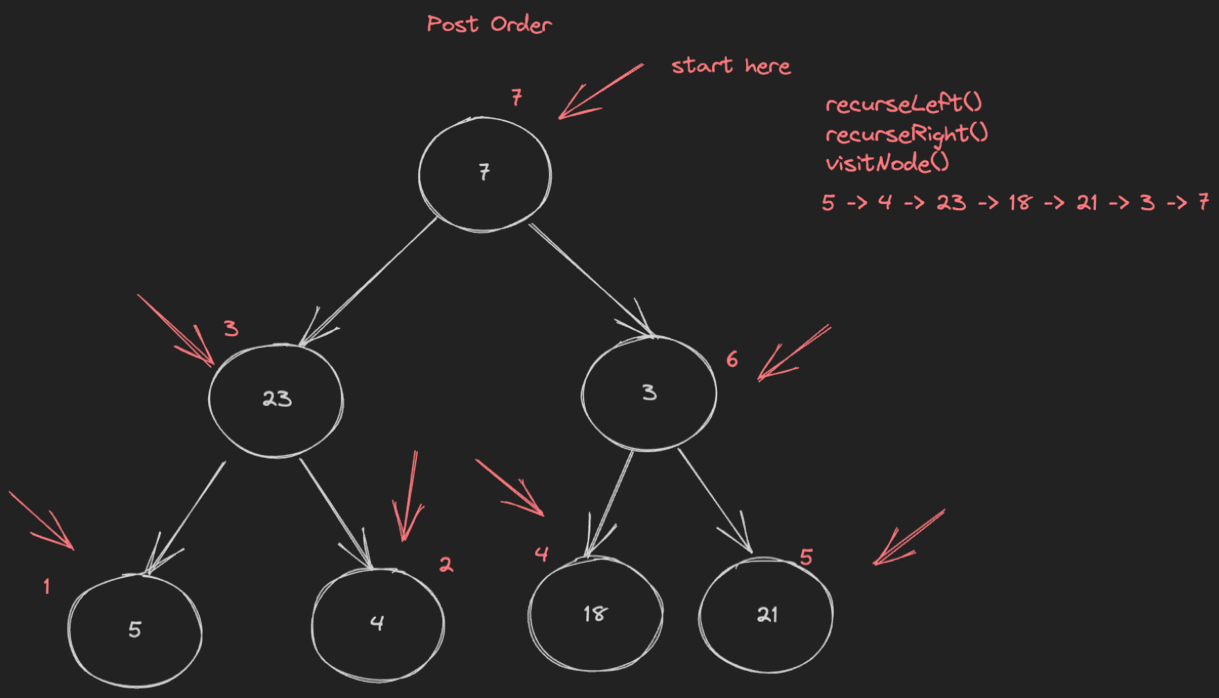


You first visit the node, then you recur down the left of the tree then the right of the tree, in a **BST** this will lead to an ordered array—the starting node is at the middle

```
class BinaryNode:
    def __init__(self, value: int):
        self.value = value
        self.left = None
        self.right = None

def walk(curr: Union[BinaryNode, None], path: List[int]) -> None:
    if not curr:
        return path

    # Recursive calls
    walk(curr.left, path)
    path.append(curr.value)
    walk(curr.right, path)

    return path

def in_order_search(head: BinaryNode) -> List[int]:
    return walk(head, [])
```

↑ Trees

Post Order



You first recur down the left and right of the tree and then you visit the node, powerful for if you want to deallocate the node—the starting node is at the end

```
class BinaryNode:
    def __init__(self, value: int):
        self.value = value
        self.left = None
        self.right = None

def walk(curr: Union[BinaryNode, None], path: List[int]) -> None:
    if not curr:
        return path

    # Recursive calls
    walk(curr.left, path)
    walk(curr.right, path)
    path.append(curr.value)

    return path

def po_order_search(head: BinaryNode) -> List[int]:
    return walk(head, [])
```