Trees

Introduction

A **tree** is a specialized data structure that organizes and stores data in a hierarchical manner. Here are some key points about trees:

1. **Definition**: A tree consists of nodes connected by edges. Unlike linear structures, trees arrange data in a way that mirrors natural hierarchies.
2. **Basic Terminology**:
   - **Root Node**: The topmost node in a tree.
   - **Child Nodes**: Nodes directly connected to a parent node.
   - **Leaf Nodes**: Nodes without any children.
   - **Ancestors**: Predecessor nodes on the path from the root to a specific node.
   - **Descendants**: Nodes reachable from a given node.
   - **Siblings**: Children of the same parent node.
   - **Level**: The count of edges from the root to a node.
   - **Internal Node**: A node with at least one child.
   - **Subtree**: A node along with its descendants.
3. **Hierarchical Structure**: Trees are non-linear because they don't store data sequentially. Instead, they arrange elements in multiple levels.
4. **Common Use Cases**:
   - Representing hierarchical relationships (e.g., file systems, organization charts).
   - Efficient searching and retrieval (e.g., binary search trees).
   - Parsing expressions (e.g., abstract syntax trees).
5. **Traversal Algorithms**:
   - Inorder: Visit left subtree, current node, right subtree.
   - Preorder: Visit current node, left subtree, right subtree.
   - Postorder: Visit left subtree, right subtree, current node.

Tree Traversals

1. **Inorder Traversal**:
   - Visits nodes in the order: Left Root Right.
   - Algorithm:
     1. Traverse the left subtree (call Inorder on the left subtree).
     2. Visit the root node.
     3. Traverse the right subtree (call Inorder on the right subtree).
   - **Uses**:
     - In binary search trees (BST), Inorder traversal gives nodes in non-decreasing order.

- Evaluating arithmetic expressions stored in expression trees.

- **Example (C++):**

```
void printInorder(struct Node* node) {
    if (node == NULL) return;
    printInorder(node->left);
    cout << node->data << " ";
    printInorder(node->right);
}
```

2. **Preorder Traversal**:
   - Visits nodes in the order: Root Left Right.
   - Algorithm:
     1. Visit the root node.
     2. Traverse the left subtree (call Preorder on the left subtree).
     3. Traverse the right subtree (call Preorder on the right subtree).
   - **Uses**:
     - Creating a copy of the tree.
     - Obtaining prefix expressions from an expression tree.
   - **Example (Python):**

```
def preorder(node):
    if node is None:
        return
    print(node.data, end=" ")
    preorder(node.left)
    preorder(node.right)
```

3. **Postorder Traversal**:
   - Visits nodes in the order: Left Right Root.
   - Algorithm:
     1. Traverse the left subtree (call Postorder on the left subtree).
     2. Traverse the right subtree (call Postorder on the right subtree).
     3. Visit the root node.
   - **Uses**:
     - Deleting nodes in a tree.
     - Evaluating postfix expressions.
   - **Example (Java):**

```
void postorder(Node node) {
    if (node == null) return;
    postorder(node.left);
    postorder(node.right);
    System.out.print(node.data + " ");
}
```

4. **Level Order Traversal (Breadth-First Search)**:
   - Visits nodes level by level, left to right.
   - Algorithm:
     1. Enqueue the root node.
     2. While the queue is not empty:
        - Dequeue a node.

    - Process it.
    - Enqueue its children (if any).
- **Uses**:
  - Level-wise processing (e.g., printing tree level by level).
  - Building heaps.
- **Example (JavaScript):**

```
function levelOrder(root) {
    const queue = [root];
    while (queue.length > 0) {
        const node = queue.shift();
        console.log(node.data);
        if (node.left) queue.push(node.left);
        if (node.right) queue.push(node.right);
    }
}
```

Binary Search Trees

What Is a Binary Search Tree (BST)?

A **Binary Search Tree** is a hierarchical data structure used in computer science for organizing and storing data in a sorted manner. Here are the key points about BSTs:

1. **Structure**:
   - Each node in a BST has at most two children: a left child and a right child.
   - The left child contains values less than the parent node, and the right child contains values greater than the parent node.
   - This hierarchical arrangement allows for efficient searching, insertion, and deletion operations.
2. **Basic Operations**:
   - **Insertion**: Adding a new value to the tree while maintaining the BST property.
   - **Searching**: Finding whether a specific value exists in the tree.
   - **Deletion**: Removing a value from the tree while preserving the BST structure.
3. **Traversal Algorithms**:
   - **Inorder Traversal**: Visits nodes in the order: Left Root Right.
   - **Preorder Traversal**: Visits nodes in the order: Root Left Right.
   - **Postorder Traversal**: Visits nodes in the order: Left Right Root.
4. **Applications**:
   - Representing hierarchical relationships (e.g., file systems, organization charts).
   - Efficient searching (binary search) and sorting.
   - Expression parsing (e.g., abstract syntax trees).

Example (In Python):

```python
class TreeNode:
    def __init__(self, val):
        self.val = val
        self.left = None
        self.right = None

def insert(root, val):
    if not root:
        return TreeNode(val)
    if val < root.val:
        root.left = insert(root.left, val)
    else:
        root.right = insert(root.right, val)
    return root

def inorder_traversal(root):
    if root:
        inorder_traversal(root.left)
        print(root.val, end=" ")
        inorder_traversal(root.right)

# Example usage:
root = None
values = [5, 3, 8, 2, 4, 7, 9]
for val in values:
    root = insert(root, val)
inorder_traversal(root)  # Output: 2 3 4 5 7 8 9
```