Lecture 1: Introduction to Trees and Binary Trees

Authors: Refat Othman and Diaeddin Rimawi

What is a Tree Data Structure?

A **tree** is a hierarchical data structure composed of elements called **nodes**, with the following characteristics:

```
        A
       / \
      B   C
     / \   \
    D   E   F
```

- One special node called the **root** serves as the starting point. **Root**: A (the topmost node)
- One special node called the **root** serves as the starting point.
- Each node may have **zero or more child nodes**. **Child nodes**: B, C (directly connected to root)
- A node with **no children** is called a **leaf**. **Leaf nodes**: D, E, F (nodes with no children)
- A **subtree** is any node and all of its descendants. **Subtree**: B and its children form a subtree
- Nodes with the same parent are **siblings**. **Siblings**: B and C are siblings (share the same parent A)
- Every node (except the root) has exactly one **parent**. **Parent**: A is the parent of B and C; B is the parent of D and E
- The **depth** of a node is the number of edges from the root to the node. **Depth**: Node D has depth 2 (A → B → D)
- The **height** of a tree is the maximum depth among all nodes. **Height**: The tree has height 3 (levels A → B → D)

Real-World Examples of Trees

- **File systems** in operating systems
- **Organizational charts** in companies
- **Family trees**

Why Are Trees Important?

Trees are essential because they represent hierarchical relationships efficiently. Compared to:

- **Arrays**: Fast indexing, but inefficient for insertions/deletions.
- **Linked Lists**: Efficient sequential access, but no hierarchy.
- **Trees**: Efficient for representing and traversing hierarchical data.

Types of Trees

1. **General Tree**: No limit on the number of children.

```
        A
      / | \
     B  C  D
    / \
   E   F
```

2. **Binary Tree**: Each node has at most two children.

```
        A
       / \
      B   C
```

3. **Full Binary Tree**: Every internal node has exactly two children.

```
        A
       / \
      B   C
     / \ / \
    D  E F  G
```

4. **Complete Binary Tree**: All levels are filled except possibly the last, filled left to right.

```
        A
       / \
      B   C
     / \  /
    D  E F
```

5. **Balanced Tree**: Height difference between left and right subtree is small (e.g., AVL Tree).

```
        A
       / \
      B   C
     /
    D
```

6. **Binary Search Tree (BST)**: Left subtree < root < right subtree.

```
        8
       / \
      3   10
     / \    \
    1   6   14
```

7. **Expression Tree**: Represents mathematical expressions.

```
        *
       / \
      +   -
     / \ / \
    a  b c  d
```

In this week, we will focus on:

- **Binary Tree**
- **Binary Search Tree (BST)**

Python Implementation: Binary Tree

```python
class Node:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

class BinaryTree:
    def __init__(self):
        self.root = None

    def insert(self, data):
        if not self.root:
            self.root = Node(data)
        else:
            self._insert(self.root, data)

    def _insert(self, node, data):
        if not node.left:
            node.left = Node(data)
        elif not node.right:
            node.right = Node(data)
        else:
            self._insert(node.left, data)  # Simple left-first insertion

    def search(self, node, target):
        if node is None:
            return False
```

```
        if node.data == target:
            return True
        return self.search(node.left, target) or self.search(node.right, target)

    def delete(self, root, key):
        if root is None:
            return None
        if root.data == key:
            # Case 1: Node with no child
            if not root.left and not root.right:
                return None
            # Case 2: Node with only one child
            if not root.left:
                return root.right
            if not root.right:
                return root.left
            # Case 3: Node with two children - replace with inorder successor
            succ_parent = root
            succ = root.right
            while succ.left:
                succ_parent = succ
                succ = succ.left
            if succ_parent != root:
                succ_parent.left = succ.right
            else:
                succ_parent.right = succ.right
            root.data = succ.data
            return root
        root.left = self.delete(root.left, key)
        root.right = self.delete(root.right, key)
        return root

    def inorder(self, node):
        if node:
            self.inorder(node.left)
            print(node.data, end=' ')
            self.inorder(node.right)
```

```

```

Binary Tree vs. Binary Search Tree

- **Binary Tree**: No rules on how child nodes are organized.
- **Binary Search Tree (BST)**: Left child < parent < right child.

Python Implementation: BST

```
class BSTNode:
    def __init__(self, data):
        self.data = data
```

```
        self.left = None
        self.right = None

class BST:
    def __init__(self):
        self.root = None

    def insert(self, data):
        self.root = self._insert(self.root, data)

    def _insert(self, node, data):
        if node is None:
            return BSTNode(data)
        if data < node.data:
            node.left = self._insert(node.left, data)
        else:
            node.right = self._insert(node.right, data)
        return node

    def inorder(self, node):
        if node:
            self.inorder(node.left)
            print(node.data, end=' ')
            self.inorder(node.right)

    def search(self, node, target):
        if node is None:
            return False
        if node.data == target:
            return True
        elif target < node.data:
            return self.search(node.left, target)
        else:
            return self.search(node.right, target)

    def delete(self, node, key):
        if node is None:
            return node
        if key < node.data:
            node.left = self.delete(node.left, key)
        elif key > node.data:
            node.right = self.delete(node.right, key)
        else:
            if node.left is None:
                return node.right
            elif node.right is None:
                return node.left
            temp = self.findMin(node.right)
            node.data = temp.data
            node.right = self.delete(node.right, temp.data)
        return node

    def findMin(self, node):
        current = node
```

```
        while current.left:
            current = current.left
        return current
```

Lecture 2: Exercises and Assignment

Exercise 1: Tree Traversal

Given a binary tree, implement all three depth-first traversals:

- Inorder
- Preorder
- Postorder

Example Interview Question:

> Given a binary tree, return the inorder traversal of its nodes' values.

Exercise 2: Validate BST

Write a function to determine if a given binary tree is a valid BST.

Example Interview Question:

> Given a binary tree, check if it is a valid BST.

Exercise 3: Compute the Height of a Binary Tree

Write a function to compute the height of a binary tree.

Example Interview Question:

> Given the root of a binary tree, write a function to return its height (the number of nodes along the longest path from the root down to the farthest leaf node).

Assignment for Next Week

- Implement a `findMin` and `findMax` method in the BST class.
- Write a function that returns `True` if the binary tree is balanced, and `False` otherwise.