Assignment no. 6

Title: Tree

- Task 1
  1. **Binary Tree Creation**: Implement a function to create a binary tree. You can take input in the form of a list or any other suitable data structure.
  2. **Traversal**: Implement functions for pre-order, in-order, and post-order traversals of the binary tree.
  3. **Height of Binary Tree**: Write a function to find the height (or maximum depth) of the binary tree.
  4. **Count Leaf Nodes**: Create a function to count the number of leaf nodes in the binary tree.
- Task 2
  1. **BST Insertion**: Implement a function to insert a new node into a Binary Search Tree (BST).
  2. **BST Search**: Write a function to search for a given value in the BST.
  3. **BST Deletion**: Implement a function to delete a node from the BST. Handle different cases such as node with no children, one child, and two children.
  4. **Find Lowest Common Ancestor**: Write a function to find the Lowest Common Ancestor (LCA) of two nodes in the BST.
  5. **Level Order Traversal**: Implement a function to perform level-order traversal (also known as breadth-first traversal) of a binary tree. Print the nodes at each level from left to right.
- Task 3
  1. **Check if Binary Tree is Balanced**: Implement a function to check if a binary tree is balanced. A tree is balanced if the height of the left and right subtrees of every node differs by at most one.
  2. **Check if Binary Tree is a Binary Search Tree**: Write a function to check if a binary tree is a valid Binary Search Tree.
  3. **Diameter of Binary Tree**: Find the diameter (longest path between any two nodes) of the binary tree.
- ***Optional*** Task 4
  1. **Serialize and Deserialize a Binary Tree**: Implement functions to serialize a binary tree into a string and then deserialize it back into a tree. This is useful for storing and reconstructing binary trees.
  2. **Max Path Sum in Binary Tree**: Create a function to find the maximum path sum between any two nodes in a binary tree. The path can start and end at any node in the tree.
  3. **Construct Binary Tree from Inorder and Preorder Traversals**: Write a function that constructs a binary tree given its inorder and preorder traversal sequences.
  4. **Check if a Binary Tree is a Subtree**: Implement a function to check if a given binary tree is a subtree of another binary tree.
  5. **General Tree Creation**:
     a. Create a general tree where each node can have any number of children.
     b. Implement traversals for the general tree (e.g., DFS, BFS)
     c. Create a function to count the number of nodes at level k in the general tree.

- ***Optional*** Part (Applications)
    1. **File System Implementation**: Create a simplified file system using a tree structure. Nodes represent directories and files. Implement operations like creating files, deleting files, navigating directories, and listing contents.
    2. **Binary Expression Tree Evaluation**: Implement a program that builds a binary expression tree from an infix expression and then evaluates it. The nodes of the tree represent operators and operands.
    3. **Huffman Coding Compression**: Implement the Huffman coding algorithm using a tree data structure. Build a Huffman tree based on character frequencies and use it to compress and decompress text.
    4. **Genealogy/Family Tree**: Design a program that allows users to create and manipulate a family tree. Nodes represent family members, and edges represent parent-child relationships. Implement operations like adding new members, finding ancestors, and listing descendants.
    5. **Hierarchical Data Representation**: Use a tree to represent hierarchical data such as organizational structures, family trees, or product category hierarchies. Implement operations like adding new nodes, finding parent/child nodes, and traversing the hierarchy.
    6. **Disjoint Set Data Structure (Union-Find)**: Implement the disjoint set data structure using a forest of trees. Provide operations for merging sets and finding the representative element of a set.
    7. **Spell Checker Using Trie**: Build a spell checker using a trie data structure to efficiently store and search for words. Implement functions for adding words, checking if a word is valid, and suggesting corrections.
    8. **Multiway Search Trees (B-trees**): Design and implement a B-tree data structure. Include operations like insertion, deletion, and searching. Test the efficiency of B-trees for large datasets.
    9. **Balanced Search Trees (AVL Trees)**: Implement an AVL tree, a self-balancing binary search tree. Include operations like insertion, deletion, and searching. Test the efficiency of AVL trees for various operations.
    10. **Merkle Trees for Data Integrity**: Build a Merkle tree to ensure data integrity in a distributed system. Implement functions for creating the tree, verifying data, and handling updates.
    11. **XML/HTML Parsing Using DOM Tree**: Develop a program to parse XML or HTML documents using a Document Object Model (DOM) tree. Implement operations to traverse and manipulate the tree.
    12. **Database Indexing with B+ Trees**: Simulate a database indexing system using B+ trees. Implement operations for adding, deleting, and searching for records in the database.