ABBOTTABAD UNIVERSITY OF SCIENCE AND TECHNOLOGY

NAME	**SAAD SHAH**
ROLL NO	**12373**
SUBJECT	**DSA**
ASSIGNMENT	**06**
SUBMITTED TO	**MR JAMAL ABDUL AHAD**
DATED	**11-01-2024**

Q NO 1 How can you efficiently search for a specific element in a binary search tree (BST)? What is the worst-case time complexity?

BST Search Operation Overview:

1. Start at the Root:

Initiate the search from the root of the Binary Search Tree (BST).

2. Compare with Current Node:

Equal: Target found; search successful.

Less than: Move to the left subtree.

Greater than: Move to the right subtree.

3. Repeat in Sub tree:

Iterate steps 2 and 3 in the respective subtree until the element is found or a leaf node is reached.

Time Complexity:

Worst-case: O(h) where h is the tree height.

Balanced BST: O(log n) (logarithmic in the number of nodes).

Worst-case (skewed tree): O(n), less efficient.

Balancing for Efficiency:

Maintain balance using AVL trees or Red-Black trees to ensure logarithmic height, optimizing search, insertion, and deletion operations.

Q2: What is the process of deleting a node from a BST? How can you handle edge cases like leaf nodes and nodes with two children?

Deleting a Node in a Binary Search Tree (BST):

1. Find the Node:

Locate the node containing the element to delete, following the search process.

2. Identify Case:

Case 1: Leaf Node (No Children):

Simply remove the node.

Case 2: Node with One Child:

Replace the node with its child.

Case 3: Node with Two Children:

Find the node's in-order successor (smallest node in the right subtree), replace the node's value with the successor's value, and then recursively delete the successor.

Handling Edge Cases:

1. Leaf Node (No Children):

Directly remove the leaf node.

2. Node with One Child: Replace the node with its child

3. Node with Two Children:

Find in-order successor (smallest in the right sub tree).

Replace node's value with the successor's value.

Recursively delete the successor

These steps ensure that the BST properties are maintained after deletion.

Time Complexity:

Similar to search, O(h) in the worst case.

Balancing: After deletion, check and rebalance the tree if necessary using rotations (for AVL trees) or recoloring (for Red-Black trees).

Handling these cases ensures a consistent and balanced BST after node deletion.

Q3: Implement an algorithm to find the minimum and maximum values stored in a BST.

#Function to find minimum value in a BST

def find_min(node): while node.left is not

None:

node = node.left

return node.value

Function to find maximum value in a BST

def find_max(node): while node.right is

not None:

node = node.right

return node.value

Q4: Explain how traversals can be used to solve common problems like counting leaves, finding the number of internal nodes, or identifying full subtrees.

Counting Leaves:

Use In-order traversal and count nodes with no children.

Finding Internal Nodes:

Traverse the tree and count nodes with at least one child.

Identifying Full Subtrees:

Post-order traversal to check if a subtree is full.

Q5: What are different types of non-binary trees, like N-ary trees or tries? What are their specific advantages and applications?

Types of Non-Binary Trees:

N-ary Trees:

Nodes can have more than two children.

Tries:

Tree-like structure used for dynamic dictionary keys.

Advantages and Applications:

N-ary Trees:

More natural representation for hierarchical data (family trees, organizational structures).

Tries:

Efficient for dynamic dictionary implementations, like autocomplete systems.

Q6: Explain the purpose of self-balancing trees like AVL trees and Red-Black trees. How do they maintain balance and achieve optimal performance?

Self-Balancing Trees (AVL and Red-Black):

Purpose:

Maintain balance in BST to ensure optimal performance.

How:

AVL uses rotations; Red-Black uses recoloring and rotations.

Q7: How are tree data structures used in real-world applications like file systems, routing algorithms, or decision trees?

Tree Data Structures in Real-world Applications:

File Systems: Represent directory structures efficiently.

Routing Algorithms: Optimize routing decisions in computer networks.

Decision Trees: Used in machine learning for decision-making.

Q8: Compare and contrast trees with other data structures like arrays, linked lists, and graphs. When would you choose a tree over another option?

Comparison with Other Data Structures:

Arrays: Fast random access, but not dynamic.

Linked Lists: Efficient for insertion and deletion, but not for searching.

Graphs: More general, but may lack hierarchical organization.

Q9: Discuss the limitations of tree data structures and situations where other data structures might be more suitable.

Limitations of Tree Data Structures:

Memory Usage: Require more memory compared to arrays and linked lists.

Complexity: Operations may be complex compared to simpler structures.

Q10: Imagine you have a large dataset of employee records. How would you design a tree structure to efficiently perform queries like finding employees by department, salary range, or hire date?

Designing a Tree for Employee Records:

Structure: Root: Department nodes.

Each department node has subtrees for salary ranges or hire dates.

Efficient Queries:

In-order traversal for salary range.

Pre-order traversal for hire date.

This design allows for efficient querying based on department, salary range, or hire date.