1. What does it mean if a binary search tree is a balanced tree?

Answer:
A Binary Search Tree (BST) is a special binary tree in which each node has a key (or value) associated with it, and for each node in the tree, all nodes in its left subtree have keys smaller than that node's key, and all nodes in its right subtree have keys larger than that node's key. This makes operations such as finding, inserting, and removing nodes in a binary search tree relatively efficient.

A Balanced Tree is a self-balancing tree that automatically adjusts its structure every time a node is inserted or removed to keep it balanced. In this way, the height of the tree is always kept to the lowest possible value, ensuring that the time complexity of operations such as search, insert, and delete is close to O(log n).

If the binary search tree is a balanced tree, then this means that the tree will self-balance as nodes are inserted and removed, thus maintaining its good search performance. In concrete terms, this means that regardless of the shape of the tree, the longest possible path from the root node to any leaf node is never more than twice as long as the shortest possible path. This helps ensure efficient search, insert, and delete operations, especially in worst-case scenarios.

Common balanced binary search trees include AVL trees, red-black trees and so on. These data structures optimize performance by self-balancing while maintaining the characteristics of binary search trees.

2. What is the big-Oh search time for a balanced binary tree? Give a logical argument for your response. You may assume that the binary tree is perfectly balanced and full.

Answer:
In a fully balanced and full balanced binary tree (such as an AVL tree or a red-black tree), the time complexity of the search is O(log n), where n is the number of nodes in the tree. This is because in such a tree, the height of the tree h satisfies h = O(log n). The longest path length from the root node to any leaf node is no more than 2h, and the shortest path length is no less than h. Therefore, to search from the root node, the worst case path length to traverse is O(log n).

The logical arguments are as follows:

Balanced property: The key property of a balanced binary tree is that the height difference between its left and right subtrees does not exceed 1, which means that every part of the tree is as balanced as possible. This ensures that the depth of the tree is proportional to the logarithm of the total number of nodes.

Perfectly balanced: If the tree is perfectly balanced, then each node has two child nodes (except the leaf nodes), which results in the tree taking on a perfect binary tree shape. In this case, the relationship between the height of the tree h and the total number of nodes n is h = log_2(n + 1), where log_2 represents the logarithm of base 2.

Search process: In a binary search tree, the search for a specific value begins at the root node. If the value is less than the value of the root node, the search is performed in the left subtree; If the value is greater than the value of the root node, the search takes place in the right subtree. This process proceeds recursively along the structure of the tree until the target value is found or it is determined that the value does not exist in the tree.

Time complexity: Since the height of the balanced binary tree is O(log n), the longest path from the root node to the leaf node is also O(log n). Therefore, the worst case path length that the search operation needs to traverse is also O(log n). This means that no matter how many nodes there are in the tree, the time complexity of the search operation remains at a logarithmic level, making the search very efficient.

To sum up, the time complexity of the search operation is O(log n) for both fully balanced and full balanced binary trees, which provides fast search performance.

3. Now think about a binary search tree in general, and that it is basically a linked list with up to two paths from each node. Could a binary tree ever exhibit an O(n) worst-case search time? Explain why or why not. It may be helpful to think of an example of operations that could exhibit worst-case behavior if you believe it is so.

Answer:
Indeed, a general binary search tree (non-equilibrium) can exhibit O(n) search time complexity in the worst case. This is because in extreme cases, a binary search tree can degenerate into a linked list, where all nodes are arranged in a straight line in key-value order, or all nodes are clustered on one side.

In this case, the height of the tree becomes n (n is the total number of nodes) because there is only one path to reach each node (either the left or right subtree). When searching, if the target value is at the end of the linked list, then the search operation may need to traverse the entire linked list, which is O(n) time complexity.

For example, consider the following sequence of operations, which can cause a binary search tree to degenerate into a linked list:

When a node is inserted, the left or right subtree of the current node is always selected for insertion, rather than the left or right subtree according to the rules of binary search trees.

Initially, the tree has only one root node.

Insert nodes sequentially and always select the left or right subtree for insertion. For example, if the left subtree is always selected for insertion, the newly inserted nodes will be placed to the left of the root node in key-value order.

In this case, the search operation would have to traverse the entire linked list to find the target value, with a time complexity of O(n). This is because the height of the tree is equal to the total number of nodes n, and the search operation may need to traverse the entire tree in the worst case.

Thus, while a balanced binary search tree is able to maintain O(log n) search time complexity, an average binary search tree may exhibit O(n) search time complexity in the worst case, which is usually due to an imbalance in the tree.