Lecture 6.41

Lecture 41: Search and Deletion in Binary Search Trees

Agenda

- Searching a node in a Binary Search Tree
- Algorithm and Example for BST Search
- Overview of Deletion Cases in BST
- Case 1: Deleting a Leaf Node
- Case 2: Deleting a Node with One Child
- Case 3: Deleting a Node with Two Children
- Finding In-order Successor/Predecessor
- Time Complexity Analysis

Searching a Node in BST: Concept

- The BST property ensures that left descendants are strictly less than the root, and right descendants are greater.
- Search operation leverages this property to eliminate half the tree at each step.
- If target key == current node's key, return True.
- If target key ¡ current node's key, recurse/traverse the left subtree.
- If target key ¿ current node's key, recurse/traverse the right subtree.

Algorithm and Code: Searching in BST

- Iterative vs. Recursive Approach
- Recursive Approach in Python:
- def search(root, key):
- if root is None or root.val == key:
- return root
- if root.val ¡ key:
- return search(root.right, key)
- return search(root.left, key)
- Base cases: Node is found, or we reach a None pointer (node not in tree).

Overview of Deletion in a BST

- Deletion is the most complex standard operation on a BST.
- The primary challenge is maintaining the BST property after a node is removed.
- We divide the problem into three mutually exclusive cases based on the target node's children:
- 1. The node has no children (Leaf Node).
- 2. The node has exactly one child.
- 3. The node has two children.

Deletion Case 1: Leaf Node

- A leaf node has no left or right children (both are None).
- This is the simplest case to handle.
- Algorithm: Simply remove the node from the tree.
- In Python/memory terms: update the parent's pointer (left or right) to point to None instead of the deleted node.
- The BST property remains perfectly intact.

Deletion Case 2: Node with One Child

- The target node has either a left child OR a right child, but not both.
- Algorithm: 'Bypass' the node being deleted.
- Link the parent of the node directly to the single child of the node.
- If the node is a left child of its parent, the parent's left pointer takes the node's child.
- If the node is a right child, the parent's right pointer takes the node's child.

Deletion Case 3: Node with Two Children

- The target node has both left and right children.
- We cannot simply bypass it, as a parent node can only hold two pointers, and we have two subtrees to attach.
- Solution: Replace the node's value with its In-order Successor or In-order Predecessor.
- After replacing the value, recursively delete the In-order Successor (which is guaranteed to have at most one child).

Finding the In-order Successor

- The In-order Successor is the node with the smallest key strictly greater than the current node.
- To find it: Go to the right child, then travel down the left pointers as far as possible.
- Algorithm: current = node.right; while current.left is not None: current = current.left
- Crucial property: The In-order Successor will NEVER have a left child (otherwise, that child would be the successor).

Algorithm and Code: Deletion Case 3

- 1. Find In-order Successor (min node in right subtree).
- 2. Copy successor's value to the target node.
- 3. Recursively call delete() on the right child to remove the duplicate successor node.
- Python Implementation snippet:
- temp = minValueNode(root.right)
- root.val = temp.val
- root.right = deleteNode(root.right, temp.val)

Complexity Analysis of BST Operations

- Time Complexity (Search and Delete):
- Average Case: O(h) = O(log N), where h is the height of the tree and N is the number of nodes.
- Worst Case: O(N) when the tree becomes skewed (e.g., essentially a linked list).
- Space Complexity:
- Recursive Approach: O(h) due to the call stack overhead.
- Iterative Approach: O(1) auxiliary space.
- Importance of balanced trees (like AVL or Red-Black trees) to guarantee O(log N) operations.

Summary and Next Lecture Preview

- Summary:
- - BST search takes advantage of the ordered property to find nodes efficiently.
- - Deletion requires handling three distinct topological cases.
- - In-order successors bridge the gap for deleting nodes with two children.
- Next Lecture Preview:
- - Tree Traversals (Inorder, Preorder, Postorder)
- - Implementing traversal algorithms in Python