Design and Analysis of Modified Binary Search Trees for Duplicate Keys

Manan Patel

A. Design and Analysis

To modify the BST from the textbook, the following adjustments are required:

- Each node should include fields `left`, `right`, and `key` but exclude a parent pointer (`p`).
- Keys are represented as words, ordered lexicographically.
- Duplicate keys should be recorded.

Two Approaches

1. **Keep Duplicates in Separate Nodes:**

- **Search:**
  With duplicates stored in separate nodes, searching may require traversing more nodes and performing additional comparisons, which increases search time. In the worst case, time complexity is $O(m)$, where m is the total number of nodes (including duplicates).
- **Insert:**
  Each insertion (including duplicates) requires traversing to the appropriate position in the tree, increasing tree size and potentially height. This can lead to unbalanced growth, resulting in a time complexity of $O(m^2)$ for inserting all duplicates.
- **Delete:**
  Deleting duplicates requires locating the specific node to be deleted. This process is less efficient as it may require traversing all occurrences. Time complexity is $O(m)$ per deletion, as tree height and node count increase with duplicates.

2. **Keep Duplicates in the Same Node with a Counter:**

- **Search:**
  By storing duplicates in a single node with a counter, the search process is

simplified since each unique key is stored in one place. This reduces traversal and improves efficiency, with a time complexity of $O(\log n)$, where n is the number of unique keys.

- **Insert:**
  Inserting a duplicate only requires incrementing the counter of the existing node, thus avoiding the need to create new nodes or modify the tree structure. For m total insertions (including duplicates), this approach has a time complexity of $O(m \log n)$.

- **Delete:**
  Deleting duplicates is straightforward: decrement the counter if it is greater than one, or remove the node if the counter reaches zero. Decrementing is $O(1)$, and removing the node (if needed) is $O(\log n)$, as restructuring is minimal.

Conclusion

Using a counter within each node for duplicates (Approach 2) is more efficient for search, insert, and delete operations compared to using separate nodes (Approach 1). This method:

- Maintains a smaller tree, reducing traversal time.
- Simplifies operations by avoiding additional tree restructuring and node creation.

Approach 2 thus provides better performance and resource management, especially with frequent duplicate keys.

Pseudocode for Operations

Tree-Search

Algorithm 1 Tree-Search

```
procedure Tree-Search(root, word)
    node ← root
    while node ≠ NULL do
        if word < node.key then
            node ← node.left
        else if word > node.key then
            node ← node.right
        elsereturn node.count
        end if
    end whilereturn 0
end procedure
```

Tree-Insert

Algorithm 2 Tree-Insert

```
procedure Tree-Insert(root, word)
    count_before ← Tree-Search(root, word)
    root ← Insert-Helper(root, word) return count_before
end procedure
function Insert-Helper(node, word)
    if node = NULL then
        return New-Node(word)
    end if
    if word < node.key then
        node.left ← Insert-Helper(node.left, word)
    else if word > node.key then
        node.right ← Insert-Helper(node.right, word)
    else
        node.count ← node.count + 1
    end ifreturn node
end function
```

Tree-Delete

Algorithm 3 Tree-Delete

```
procedure Tree-Delete(root, word)
    count_before ← Tree-Search(root, word)
    if count_before > 0 then
        root ← Delete-Helper(root, word)
    end ifreturn count_before
end procedure
function Delete-Helper(node, word)
    if node = NULL then
        return NULL
    end if
    if word < node.key then
        node.left ← Delete-Helper(node.left, word)
    else if word > node.key then
        node.right ← Delete-Helper(node.right, word)
    else
        if node.count > 1 then
            node.count ← node.count - 1
        else
            if node.left = NULL then return node.right
            else if node.right = NULL then return node.left
            else
                min_larger_node ← Find-Min(node.right)
                node.key ← min_larger_node.key
                node.count ← min_larger_node.count
                min_larger_node.count ← 1
                node.right ← Delete-Helper(node.right, min_larger_node.key)
            end if
        end if
    end ifreturn node
end function
function Find-Min(node)
    while node.left ≠ NULL do
        node ← node.left
    end whilereturn node
end function
```