Design and Analysis of Modified Binary Search Tree Handling Duplicate Keys

Introduction

In this report, we analyze two approaches for handling duplicate keys in a Binary Search Tree (BST) modified from the textbook design. The modifications include removing the parent pointer `p` from each node and storing keys as words in dictionary order. We compare the efficiency of two methods for managing duplicate keys:

1. Keeping duplicates in **separate nodes**, where the BST may contain multiple nodes with the same key.
2. Keeping duplicates in the **same node with a counter**, where each node represents a unique key and maintains a count of occurrences.

We analyze the major operations: search, insert, and delete, by solving the recurrence relations to determine the number of operations $T(n)$ required, providing mathematical expressions instead of Big O or Big Theta notations.

BST Modifications

- Each node contains:
  - `key`: a word.
  - `left`: reference to the left child.
  - `right`: reference to the right child.
- No parent pointer `p` is maintained.
- Keys are stored in dictionary order.
- Duplicate keys are remembered.

Approach 1: Duplicates in Separate Nodes

In this approach, each occurrence of a duplicate key is stored in a separate node. The BST may contain multiple nodes with the same key. Search operations can return any one of these nodes.

Analysis

Let m be the total number of nodes in the tree, including duplicates.

Tree Height

- **Duplicates increase the tree's height**, potentially degrading performance to linear time.
- If duplicates are inserted sequentially, the tree can become unbalanced, forming a degenerate tree (similar to a linked list) with height $h = m$.

Time Complexity for Operations

- **Search Operation**:

  The time $T_{\text{search}}(m)$ to search for a key is proportional to the height h of the tree.

$$T_{\text{search}}(m) = h = m$$

- **Insert Operation**:

  Each insertion operation involves traversing the tree to find the appropriate position for the new node. In the worst case, when the tree is completely unbalanced, inserting the k-th node requires k comparisons.

  The time $T_{\text{insert}}(k)$ for the k-th insertion is:

$$T_{\text{insert}}(k) = k$$

  Summing over all m insertions:

$$T_{\text{insert\_total}}(m) = \sum_{k=1}^{m} k = \frac{m(m+1)}{2}$$

- **Delete Operation**:

  Deletion requires searching for the node and possibly restructuring the tree. In the worst case, this involves traversing the entire height of the tree.

$$T_{\text{delete}}(m) = m$$

Total Time for m Operations

The total time $T_{\text{total}}(m)$ for all operations (assuming a mix of insertions, searches, and deletions) can be approximated by considering the dominant terms:

$$T_{\text{total}}(m) = T_{\text{insert\_total}}(m) + T_{\text{search\_total}}(m) + T_{\text{delete\_total}}(m)$$

Assuming that each insertion is followed by a search and possibly a deletion, the dominant term comes from the insertions:

$$T_{\text{total}}(m) = \frac{m(m+1)}{2} + m \times m + m \times m = \frac{m^2+m}{2} + 2m^2$$

Simplifying:

$$T_{\text{total}}(m) = \frac{5m^2+m}{2} = \frac{5m^2}{2} + \frac{m}{2}$$

Disadvantages of Approach 1

- **Duplicates increase the tree's height**, potentially degrading performance to linear time $T(n) = m$ per operation in the worst case.
- **Tree becomes unbalanced** with many duplicates, leading to inefficient searches and operations.
- **More memory overhead** due to additional nodes for duplicates.

Approach 2: Duplicates with Counters

In this approach, each node represents a unique key and maintains a counter for the number of occurrences.

Let n be the number of unique keys (nodes) in the tree, and $d = m - n$ be the number of duplicates.

Analysis

Tree Height

- **Tree remains balanced as duplicates do not add new nodes**.
- The height h of a balanced BST is approximately $\log_2 n$.

Time Complexity for Operations

- **Search Operation**:
  The time $T_{\text{search}}(n)$ to search for a key is proportional to the height h of the tree.

$$T_{\text{search}}(n) = \log_2 n$$

- **Insert Operation**:
  For each insertion, there are two cases:

  1. **New Key Insertion**:
     Inserting a new unique key requires traversing the tree to find the correct position.
     The time $T_{\text{insert\_unique}}(k)$ for the k-th unique insertion is:

$$T_{\text{insert\_unique}}(k) = \log_2 k$$

     Summing over all n unique insertions:

$$T_{\text{insert\_unique\_total}}(n) = \sum_{k=1}^{n} \log_2 k$$

     Using the property of logarithms:

$$\sum_{k=1}^{n} \log_2 k = \log_2 n!$$

     Approximating $\log_2 n!$ using Stirling's approximation:

$$\log_2 n! \approx n \log_2 n - n \log_2 e + \frac{1}{2} \log_2(2\pi n)$$

     For large n, the dominant term is $n \log_2 n$, so:

$$T_{\text{insert\_unique\_total}}(n) \approx n \log_2 n$$

  2. **Duplicate Key Insertion**:
     Inserting a duplicate key involves searching for the key and incrementing its counter.
     The time $T_{\text{insert\_duplicate}}$ for each duplicate insertion is:

$$T_{\text{insert\_duplicate}} = \log_2 n + c$$

     Where c is the constant time to increment the counter.
     For d duplicate insertions:

$$T_{\text{insert\_duplicate\_total}} = d(\log_2 n + c)$$

- **Delete Operation**:
  Deletion also involves two cases:
  1. **Decrementing Counter**:
     Deleting a duplicate key involves searching for the key and decrementing its counter.
     $$T_{\text{delete\_duplicate}} = \log_2 n + c$$
  2. **Removing Node**:
     If the counter reaches zero, the node must be removed, which involves restructuring the tree.
     $$T_{\text{delete\_unique}} = \log_2 n + d'$$
     Where d' is the time to restructure the tree, proportional to $\log_2 n$.

Total Time for m Operations

The total time $T_{\text{total}}(m)$ for all operations can be expressed as:

$$T_{\text{total}}(m) = T_{\text{insert\_unique\_total}}(n) + T_{\text{insert\_duplicate\_total}} + T_{\text{delete\_duplicate\_total}} + T_{\text{delete\_unique\_total}}$$

Assuming that deletion times are similar to insertion times, and focusing on the dominant terms:

$$T_{\text{total}}(m) \approx n \log_2 n + d(\log_2 n + c)$$

Given that $m = n + d$, we can substitute $d = m - n$:

$$T_{\text{total}}(m) \approx n \log_2 n + (m - n)(\log_2 n + c)$$

Simplifying:

$$T_{\text{total}}(m) = n \log_2 n + m \log_2 n - n \log_2 n + (m - n)c = m \log_2 n + (m - n)c$$

For large m and n, the term $m \log_2 n$ dominates, and the constant c becomes negligible in comparison. Therefore, we can approximate:

$$T_{\text{total}}(m) \approx m \log_2 n$$

Advantages of Approach 2

- **Tree remains balanced** as duplicates do not add new nodes.
- **Improved search efficiency** since duplicates do not increase tree height.
- **Less memory usage** due to fewer nodes.
- **Insertion and deletion often require only updating the counter**, not restructuring the tree.

Comparison and Conclusion

Total Time Comparison

1. **Approach 1**:
   $$T_{\text{total}}(m) = \frac{5m^2}{2} + \frac{m}{2}$$
   - The time grows quadratically with m.
2. **Approach 2**:
   $$T_{\text{total}}(m) \approx m \log_2 n$$
   - The time grows linearly with m and logarithmically with n.

Conclusion

Approach 2 is more efficient for the major operations (search, insert, delete) because:

- The tree maintains a smaller height due to fewer nodes, resulting in lower $T(n)$ for search and insert operations.
- **Duplicates do not increase the tree's height**, avoiding degradation to linear time in the worst case.
- Updating counters is a constant-time operation, adding minimal overhead.

Therefore, **Approach 2** provides better performance and resource utilization when handling duplicate keys in a BST.

Conclusion

Approach 2 is more efficient for the major operations (search, insert, delete) because:

- The tree maintains a smaller height due to fewer nodes, resulting in lower $T(n)$ for search and insert operations.
- **Duplicates do not increase the tree's height**, avoiding degradation to linear time in the worst case.
- Updating counters is a constant-time operation, adding minimal overhead.

Therefore, **Approach 2** provides better performance and resource utilization when handling duplicate keys in a BST.

Node and BST Classes Algorithm

My code implementation can be found here

Algorithm 1 Class Node Initialization

```
structure Node:
    members: key, count = 1, left = None, right = None
```

Algorithm 2 BST Class Constructor

```
structure BST:
    members: root = None
```

Algorithm 3 tree_search(key)

```
node ← root
while node ≠ None do
    if key = node.key then
        return node.count
    else if key < node.key then
        node ← node.left
    else
        node ← node.right
    end if
end while
return 0
```

Algorithm 4 tree_insert(key)

```
parent ← None
node ← root
while node ≠ None do
    parent ← node
    if key = node.key then
        node.count ← node.count + 1
        return node.count - 1
    else if key < node.key then
        node ← node.left
    else
        node ← node.right
    end if
end while
new_node ← new Node(key)
if parent = None then
    root ← new_node
else if key < parent.key then
    parent.left ← new_node
else
    parent.right ← new_node
end if
return 0
```

Algorithm 5 tree_delete(key)

```
call _delete_rec(root, key)
return occurrences
```

Algorithm 6 _delete_rec(node, key)

```
if node = None then
    return (None, 0)
else if key < node.key then
    node.left, occurrences ← _delete_rec(node.left, key)
else if key > node.key then
    node.right, occurrences ← _delete_rec(node.right, key)
else
    occurrences ← node.count
    if node.count > 1 then
        node.count ← node.count - 1
    else if node.left = None then
        return (node.right, occurrences)
    else if node.right = None then
        return (node.left, occurrences)
    else
        successor ← _min_value_node(node.right)        node.key ← successor.key
        node.count ← successor.count
        node.right, _ ← _delete_rec(node.right, successor.key)
    end if
end if
return (node, occurrences)
```

Algorithm 7 _min_value_node(node)

```
while node.left ≠ None do
    node ← node.left
end while
return node
```

Algorithm 8 Interactive User Interface

```
while True do
    cmd ← INPUT("Enter command (search, insert, delete, walk, print, exit): ")
    if cmd = "exit" then
        break
    else if cmd = "search" or cmd = "insert" or cmd = "delete" then
        word ← INPUT("Enter word: ")
        HANDLE_COMMAND(cmd, word) cmd = "walk" or cmd = "print"
        HANDLE_COMMAND(cmd)
    else
        PRINT("Invalid command.")
    end if
end while
```