Analysis of Algorithms

BLG 335E

Project 3 Report

ÖMER YILDIZ
yildizom20@itu.edu.tr

Faculty of Computer and Informatics Engineering
Department of Computer Engineering
Date of submission: 29.12.2023

1. Implementation

1.1. Differences Between BSTs and RBTs

BSTs are unconstrained, resulting in potential height imbalances. In contrast, RBTs enforce rules regarding coloring and balance, maintaining logarithmic height for efficient operations.

1.2. Advantages of Red-Black Trees

The key advantage of RBTs lies in their guaranteed balanced height, ensuring consistent and efficient performance irrespective of the order of insertions or deletions.

1.3. Experimental Results

1.3.1. Population Data Versions

- **pop1 (Reverse Sorted):**
  - BST Height: 835
  - RBT Height: 21
- **pop2 (Sorted):**
  - BST Height: 13,806
  - RBT Height: 24
- **pop3 (Reverse Sorted):**
  - BST Height: 12,204
  - RBT Height: 24
- **pop4 (Random):**
  - BST Height: 65
  - RBT Height: 16

1.3.2. Analysis

- **pop1 and pop3 (Sorted and Reverse Sorted):** For both BST and RBT, the height is influenced by the order of insertion. RBT maintains logarithmic height, providing consistent performance.

- **pop2 (Sorted):** BST exhibits significantly higher height due to the lack of balancing, while RBT maintains a balanced structure.

- **pop4 (Random):** Even with a random dataset, RBT outperforms BST, demonstrating the resilience of RBTs to different data distributions.

1.4. Proof: Maximum Height of Red-Black Tree

Let h_{max} be the maximum height of a Red-Black Tree with n nodes.

1.4.1. Claim 1:

A Red-Black Tree with h black-height has at least $2^h - 1$ internal nodes.

Proof by induction:

Base Case ($h = 0$): A Red-Black Tree with black-height 0 has only one node (the nil leaf), and it satisfies $2^0 - 1 = 0$ internal nodes.

Inductive Step: Assume that for a black-height h, a Red-Black Tree has at least $2^h - 1$ internal nodes.

Consider a Red-Black Tree with black-height $h + 1$. Each path from the root to a nil leaf has black-height h. By the inductive assumption, there are at least $2^h - 1$ internal nodes along each path.

There are 2^h paths from the root to a nil leaf, and each path contributes at least one additional internal node. Therefore, the total number of internal nodes in the tree is at least $2^h + 2^h - 1 = 2^{h+1} - 1$.

This completes the inductive step, and we have shown that a Red-Black Tree with black-height $h + 1$ has at least $2^{h+1} - 1$ internal nodes.

1.4.2. Claim 2:

A Red-Black Tree with n internal nodes has black-height at most $\lfloor \log_2(n + 1) \rfloor$.

Proof by contradiction:

Assume there exists a Red-Black Tree with n internal nodes and black-height $h > \lfloor \log_2(n+1) \rfloor$.

By Claim 1, this tree must have at least $2^h - 1 > n$ internal nodes, which is a contradiction.

Therefore, the black-height of a Red-Black Tree with n internal nodes is at most $\lfloor \log_2(n+1) \rfloor$.

1.4.3. Maximum Height:

Since the black-height is at most $\lfloor \log_2(n+1) \rfloor$, the maximum height of the Red-Black Tree is twice the black-height, i.e., $2\lfloor \log_2(n+1) \rfloor$.

Red-Black Tree (RBTree)

Big-O Complexities and Explanations

Preorder Traversal (`preorder()`):

- **Complexity:** $O(n)$ (linear time complexity)
- **Explanation:** Preorder traversal visits each node exactly once, resulting in linear time complexity.

Inorder Traversal (`inorder()`):

- **Complexity:** $O(n)$ (linear time complexity)
- **Explanation:** Inorder traversal also visits each node exactly once. In the worst case, when the tree is degenerate, the time complexity is linear.

Postorder Traversal (`postorder()`):

- **Complexity:** $O(n)$ (linear time complexity)
- **Explanation:** Postorder traversal involves visiting each node exactly once, resulting in linear time complexity.

Search (`searchTree()`):

- **Complexity:** $O(\log n)$ (logarithmic time complexity)
- **Explanation:** Searching in a balanced Red-Black Tree has logarithmic time complexity due to the balanced nature of the tree.

Insertion (`insert()`):

- **Complexity:** $O(\log n)$ (logarithmic time complexity)
- **Explanation:** Insertion in a Red-Black Tree involves finding the correct position for the new node and adjusting the tree to maintain its properties. The worst-case time complexity is $O(\log n)$.

Deletion (`deleteNode()`):

- **Complexity:** $O(\log n)$ (logarithmic time complexity)
- **Explanation:** Deletion involves finding and removing a node, and then fixing the tree to preserve Red-Black Tree properties. The worst-case time complexity is $O(\log n)$.

Get Height (`getHeight()`):

- **Complexity:** $O(n)$ (linear time complexity)
- **Explanation:** getHeight function visiting each node exactly once, resulting in linear time complexity.

Get Maximum (`getMaximum()`):

- **Complexity:** $O(\log n)$ (logarithmic time complexity)
- **Explanation:** The maximum node is the rightmost node in the tree, so finding it involves traversing the right subtree, resulting in $O(\log n)$ time complexity in a balanced tree.

Get Minimum (`getMinimum()`):

- **Complexity:** $O(\log n)$ (logarithmic time complexity)
- **Explanation:** Similar to `getMaximum()`, finding the minimum node involves traversing the left subtree, resulting in $O(\log n)$ time complexity in a balanced tree.

Get Total Nodes (`getTotalNodes()`):

- **Complexity:** $O(n)$ (linear time complexity)
- **Explanation:** Counting all nodes in the tree requires visiting each node exactly once, leading to linear time complexity.

Binary Search Tree (BSTree)

Big-O Complexities and Explanations

Search (`searchTree()`):

- **Complexity:** $O(n)$ (linear time complexity in the worst case)
- **Explanation:** In a worst-case scenario where the BSTree is degenerate (skewed), the tree's height is $O(n)$, and searching will take $O(n)$ time.

Insertion (`insert()`):

- **Complexity:** $O(n)$ (linear time complexity in the worst case)
- **Explanation:** Similar to searching, insertion in a skewed BSTree involves adding nodes to the end of the tree, resulting in a linear height tree and a worst-case time complexity of $O(n)$.

Deletion (`deleteNode()`):

- **Complexity:** $O(n)$ (linear time complexity in the worst case)
- **Explanation:** Deletion in a skewed BSTree may involve removing a leaf node or a node with only one child, which does not balance the tree. In the worst case, the tree remains skewed, and deletion takes $O(n)$ time.

Get Height (`getHeight()`):

- **Complexity:** $O(n)$ (linear time complexity in the worst case)
- **Explanation:** In a worst-case scenario where the BSTree is degenerate, calculating the height requires traversing the entire tree, leading to linear time complexity.

Get Maximum (`getMaximum()`):

- **Complexity:** $O(n)$ (linear time complexity in the worst case)
- **Explanation:** In a skewed BSTree, finding the maximum node involves traversing the rightmost path, resulting in linear time complexity.

Get Minimum (`getMinimum()`):

- **Complexity:** $O(n)$ (linear time complexity in the worst case)
- **Explanation:** In a skewed BSTree, finding the minimum node involves traversing the leftmost path, resulting in linear time complexity.

Get Total Nodes (`getTotalNodes()`):

- **Complexity:** $O(n)$ (linear time complexity)
- **Explanation:** Counting all nodes in the tree requires visiting each node exactly once, leading to linear time complexity.

Ensuring Red-Black Tree (RBT) Rules

Insertion

When a new node is inserted into the Red-Black Tree (RBT), the following steps are taken to ensure that the RBT rules are satisfied:

1. **Standard BST Insertion:** The new node is inserted into the Red-Black Tree using standard Binary Search Tree (BST) insertion rules.

2. **Additional Steps for RBT:** After the standard BST insertion, the tree might violate Red-Black Tree properties, specifically the red-red violation. To address this, rotations and color changes are applied. Left-Left and Right-Right rotations are performed to balance the tree, and color changes are made to ensure that after the rotations, the Red-Black Tree properties are satisfied.

3. **Fixing Insertion:** The fixInsertion function is called to handle any violations introduced during the insertion process. This function performs rotations and color changes to bring the tree back into compliance with the Red-Black Tree properties.

Listing 1.1: Red-Black Tree Insertion

```
void insert(const std::string& name, int key) {
    // Insertion node
    RBT::Node* z = new RBT::Node();
    // Standard BST insertion
    // ...

    // Additional steps for RBT
    insertFixup(z);
}

void insertFixup(Node* newNode) {
    // Perform rotations and color changes
    // ...
}
```

Deletion

When a node is deleted from the Red-Black Tree, the following steps are taken to maintain the Red-Black Tree properties:

1. **Standard BST Deletion:** The node to be deleted is removed using standard Binary Search Tree (BST) deletion rules.
2. **Additional Steps for RBT:** After the standard BST deletion, the tree might violate Red-Black Tree properties. If the deleted node or its successor is black, additional adjustments are needed to handle the decrease in black height.
3. **Fixing Deletion:** The fixDeletion function is called to handle any violations introduced during the deletion process. This function performs rotations and color changes to redistribute the black height and maintain the Red-Black Tree properties.
4. **Traversing Upwards:** These adjustments are made while traversing up the tree towards the root, ensuring that the entire tree remains balanced and satisfies the Red-Black Tree rules.

Listing 1.2: Red-Black Tree Deletion

```
void deleteNode(int key) {
    // Search for the node with the given key in the Red-Black Tree
    RBT::Node* z = searchTree(key);
     // Initialize variables for
     replacement node, original color, and successor node
    RBT::Node* y = z;
    int yOriginalColor = y->color;
    RBT::Node* x;
    // Additional steps for RBT
    // If the original color of y was black,
    //fix the Red-Black Tree properties
    if (yOriginalColor == 0) {
        deleteFixup(x);
    }
}
```

```
void deleteFixupNode* deletedNode) {
    // Perform rotations and color changes
    // ...
}
```

Handling Different Cases in BST Deletion

The deletion operation in a Binary Search Tree (BST) involves different cases to preserve the binary search tree property:

Leaf Node Deletion

1. **Locate the Node:** The node to be deleted is located in the Binary Search Tree.
2. **Case 1: Node is a Leaf:** If the node to be deleted is a leaf (has no children), it is simply removed, and the tree remains a valid BST.

Listing 1.3: BST Leaf Node Deletion

```
BST::Node* deleteNodeHelper(BST::Node* root, int key) {
    // Recursive search for the node to delete
    //..

    // Node with only one child or no child
      if (root->left == nullptr) {
        BST::Node* temp = root->right;
        delete root;
        return temp;
      } else if (root->right == nullptr) {
        BST::Node* temp = root->left;
        delete root;
        return temp;
}
```

```
}
```

Node with One Child

1. **Locate the Node:** The node to be deleted is located in the Binary Search Tree.
2. **Case 2: Node has One Child:** If the node to be deleted has only one child, the node is replaced by its child. This ensures that the binary search tree property is preserved.

Listing 1.4: BST Node with One Child Deletion

```
BST::Node* deleteNodeHelper(BST::Node* root, int key) {
    // Recursive search for the node to delete
    //..

// Node with only one child or no child
      if (root->left == nullptr) {
        BST::Node* temp = root->right;
        delete root;
        return temp;
      } else if (root->right == nullptr) {
        BST::Node* temp = root->left;
        delete root;
        return temp;
}
}
```

Node with Two Children (Successor Deletion)

1. **Locate the Node:** The node to be deleted is located in the Binary Search Tree.

2. **Case 3: Node has Two Children:** If the node to be deleted has two children, its in-order successor is identified. The key of the successor is copied to the node to be deleted, and then the successor (which is now in the original node's position) is deleted recursively. This ensures that the binary search tree property is maintained.

Listing 1.5: BST Node with Two Children Deletion

```
BST::Node* deleteNodeHelper(BST::Node* root, int key) {
    // Recursive search for the node to delete
    //..

// Node with two children: Get the inorder successor (smallest
      // in the right subtree)
      BST::Node* temp = findMin(root->right);

      // Copy the inorder successor's content to this node
      root->data = temp->data;
      root->name = temp->name;

      // Delete the inorder successor
      root->right = deleteNodeHelper(root->right, temp->data);
    }
    return root;
```