Binary Tree Operations Project Documentation

Jude Eschete

March 11, 2025

1 Introduction

This document details the design, algorithm, and API of a C++ project that implements various binary tree operations. The functionalities include:

1. Building a binary tree with minimal height.
2. Validating if the tree is a Binary Search Tree (BST).
3. Printing all elements in the tree within a given range.
4. Finding the first common ancestor of two nodes.

2 Design and Algorithm

2.1 Building a Minimal Height Tree

To build a minimal height tree, the algorithm employs a divide-and-conquer strategy by choosing the middle element of a sorted array as the root. This ensures that the left and right subtrees have approximately the same number of nodes.

Pseudocode:

```
function buildMinimalHeightTree(arr, start, end):
  if start > end:
   return NULL
  mid = start + (end - start) / 2
  node = new Node(arr[mid])
  node.left  = buildMinimalHeightTree(arr, start, mid - 1)
  node.right = buildMinimalHeightTree(arr, mid + 1, end)
  return node
```

2.2 Validating the BST

Validation uses a recursive function that checks whether each node's value is within an acceptable range. This range is updated as the recursion traverses down the tree.

Pseudocode:

```
function validateBSTUtil(node, minVal, maxVal):
 if node is NULL:
  return true
 if node.data < minVal or node.data > maxVal:
  return false

 return validateBSTUtil(node.left,  minVal,     node.data - 1) AND
  validateBSTUtil(node.right, node.data + 1, maxVal)

function validateBST(root):
 return validateBSTUtil(root, -infinity, infinity)
```

2.3 Printing Elements in a Given Range

If the tree is a BST, an in-order traversal is used to print all nodes whose values lie within a specified range $[k_1, k_2]$.

Pseudocode:

```
function printRange(node, k1, k2):
 if node is NULL:
  return

 if node.data > k1:
  printRange(node.left, k1, k2)

 if k1 <= node.data <= k2:
  print node.data

 if node.data < k2:
  printRange(node.right, k1, k2)
```

2.4 Finding the First Common Ancestor

The function locates the first common ancestor of two nodes by recursively searching both left and right subtrees. If both sides return non-NULL, the current node is the common ancestor.

Pseudocode:

```
function firstCommonAncestor(node, k1, k2):
 if node is NULL:
  return NULL

 if node.data == k1 or node.data == k2:
  return node

 left  = firstCommonAncestor(node.left,  k1, k2)
```

```
right = firstCommonAncestor(node.right, k1, k2)

if left != NULL and right != NULL:
 return node

if left != NULL:
 return left
else:
 return right
```

3 API Specifications

3.1 buildMinimalHeightTree

Prototype:

```
Node* buildMinimalHeightTree(const vector<int>& arr, int start, int end)
```

Description: Recursively constructs a binary tree with minimal height by choosing the middle element of the array as the root.

3.2 validateBST

Prototype:

```
bool validateBST(Node* root)
```

Description: Uses a recursive helper function to verify that the tree meets BST constraints (left subtree ¡ root ¡ right subtree).

3.3 printRange

Prototype:

```
void printRange(Node* root, int k1, int k2)
```

Description: Prints all elements in the tree within the inclusive range $[k1, k2]$ by performing an in-order traversal optimized for BSTs.

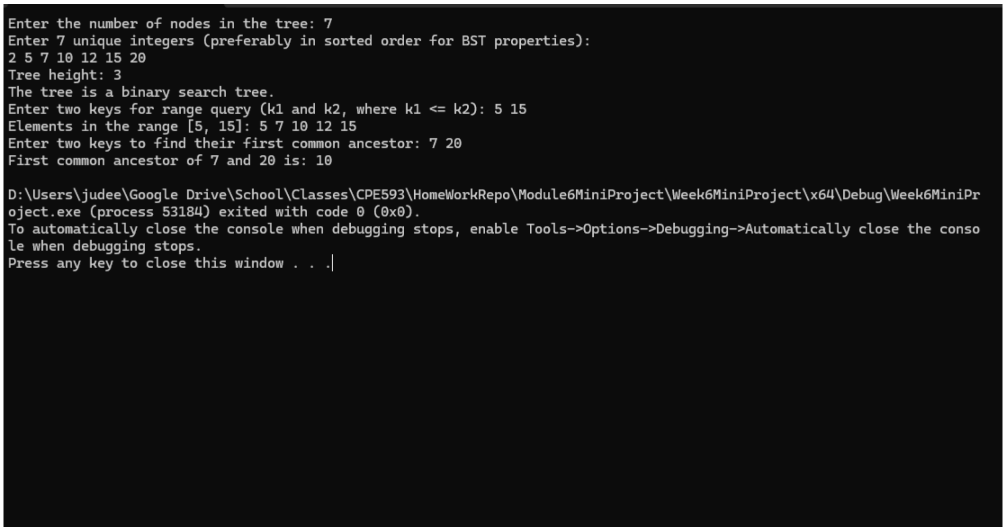
3.4 firstCommonAncestor

Prototype:

```
Node* firstCommonAncestor(Node* root, int k1, int k2)
```

Description: Finds and returns the node that is the first common ancestor of two specified node values.

4 Example Screenshot and Outputs



5 Additional Information

- **Input Assumptions:** The array contains unique integers; using a sorted array is recommended.
- **Efficiency:** The divide-and-conquer approach ensures minimal height. BST checks and in-order traversals are implemented recursively.
- **Potential Enhancements:** Additional error handling, iterative methods, and deletion or balancing routines.

6.1 Rebalance of the RB-Tree

In the intermediate status of the RB-Tree, there was a red-red conflict: node `P` was red and its child `N` was also red. This violates the Red-Black property that a red node cannot have a red child. To fix this, we performed the appropriate rotation around the grandparent (`G`) and recolored nodes to ensure the root remains black and all other RB-Tree properties are maintained (e.g., no consecutive red nodes, equal black height on all paths).

A possible final balanced version of the RB-Tree after rebalancing is shown in Figure 1. Node colors have been updated to remove any red-red violations, and rotations ensure that the tree is properly balanced.

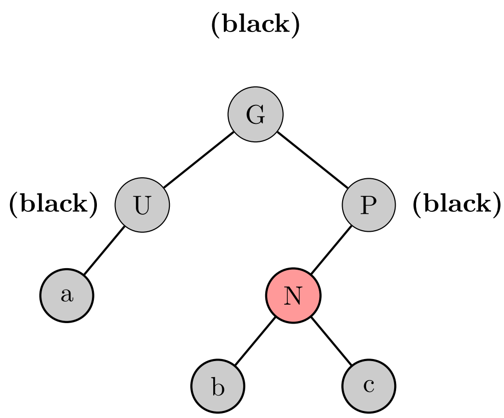


Figure 1: A rebalanced RB-Tree for 6.1, with corrected colors and structure.

After these adjustments:

- The root (`G`) is black, satisfying the property that the root of an RB-Tree is always black.
- No red node has a red parent, removing the original red-red conflict.
- Every path from the root to a leaf or NULL child has the same number of black nodes, preserving the black-height property.

6.2 Rebalance of the RB-Tree

In the intermediate status for 6.2, a red-red conflict arises because `P` is red and its child `N` is also red. This situation violates the Red-Black property that forbids a red node from having a red parent. Additionally, we must ensure that the resulting tree maintains a consistent black height on every path from the root to a leaf (or NULL) child.

To fix this violation, we identify `G` as the grandparent of `N`, note that the "uncle" node is black, and perform the appropriate rotation(s) around `G`. We then recolor the involved nodes to eliminate the red-red conflict while preserving the black-height property. Figure 2 shows a possible final arrangement of the tree after these adjustments, with `P` as the new root and no consecutive red nodes remaining.

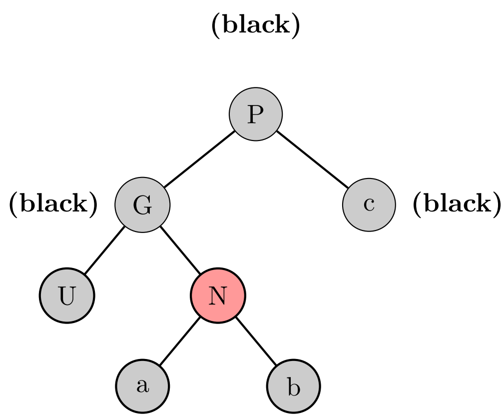


Figure 2: A possible rebalanced RB-Tree for 6.2, with `P` rotated up and recolored to eliminate the red-red conflict.

Why These Changes Were Necessary:

- *Red-Red Conflict:* Before rebalancing, both `P` and `N` were red. Rotations and recoloring were required so that a red node (`N`) would have a black parent (`P`).
- *Maintaining Black-Height:* By adjusting the tree structure and colors, we ensure each path from `P` to a leaf or NULL pointer has the same number of black nodes.
- *Root is Black:* In standard Red-Black Tree rules, the root should be black. Thus, `P` is recolored to black after the rotation.