Project I:

Efficiency of basic operations based on BST, AVL and Splay Trees

Group 1: Lin Zhiyu, Hu Junyu, Huang Zhenhua

2025-09-XX

1 Introduction

In this project, we implement three types of binary search trees: BST, AVL Tree, and Splay Tree as well as some basic operations on them including insertion, deletion and find. We then conduct experiments to compare the efficiency of a series of these operations on the three types of trees.

2 Algorithm Specification

This section details the implementation of the three tree structures: Binary Search Tree (BST), AVL Tree, and Splay Tree.

2.1 Binary Search Tree (BST)

BST is a binary tree data structure which has the following properties: - The left subtree of a node contains only nodes with keys lesser than the node's key. - The right subtree of a node contains only nodes with keys greater than the node's key. - The left and right subtree each must also be a binary search tree. - There must be no duplicate nodes.

Our implementation is a straightforward recursive approach.

2.1.1 BST Insertion The insertion operation recursively finds the correct position for the new key while maintaining the BST property.

```
function BST_INSERT(node, key):
  // If the current node is null, we've found the insertion point.
  if node is null:
    return CREATE_NODE(key)

  // Recur down the tree.
  if key < node.key:
    node.left = BST_INSERT(node.left, key)
  else if key > node.key:
    node.right = BST_INSERT(node.right, key)

  // Return the (unchanged) node pointer.
  return node
```

2.1.2 BST Deletion Deletion handles three cases for the node to be deleted: no children, one child, or two children. For a node with two children, it is replaced by its in-order successor (the smallest key in its right subtree).

```
function BST_DELETE(node, key):
  if node is null: return null

  // Find the node to be deleted.
  if key < node.key:
    node.left = BST_DELETE(node.left, key)
  else if key > node.key:
    node.right = BST_DELETE(node.right, key)
  else: // Key found, this is the node to be deleted.
    // Case 1 & 2: Node with one or no child.
    if node.left is null:
      temp = node.right
      free(node)
      return temp
    else if node.right is null:
      temp = node.left
      free(node)
      return temp

    // Case 3: Node with two children.
```

```
    // Find the in-order successor (smallest in the right subtree).
    temp = FIND_MIN(node.right)
    // Copy the successor's key to this node.
    node.key = temp.key
    // Delete the in-order successor from the right subtree.
    node.right = BST_DELETE(node.right, temp.key)

  return node
```

2.2 AVL Tree

An AVL tree is a self-balancing Binary Search Tree. The heights of the two child subtrees of any node differ by at most one. If at any time they differ by more than one, rebalancing is done to restore this property.

2.2.1 AVL Insertion Insertion begins like a standard BST insertion. After inserting the new node, we traverse back up the tree, updating the height of each ancestor node. At each node, we check the balance factor and perform rotations if the tree has become unbalanced.

```
function AVL_INSERT(node, key):
  // 1. Perform standard BST insertion.
  if node is null:
    return CREATE_NODE(key)
  if key < node.key:
    node.left = AVL_INSERT(node.left, key)
  else if key > node.key:
    node.right = AVL_INSERT(node.right, key)
  else: // Duplicate keys are not inserted.
    return node

  // 2. Update height of the current ancestor node.
  UPDATE_HEIGHT(node)

  // 3. Get the balance factor to check if this node became unbalanced.
  balance = GET_BALANCE(node)

  // 4. If unbalanced, perform rotations.
  // Left Left Case
  if balance > 1 and key < node.left.key:
    return ROTATE_RIGHT(node)

  // Right Right Case
  if balance < -1 and key > node.right.key:
    return ROTATE_LEFT(node)

  // Left Right Case
  if balance > 1 and key > node.left.key:
    node.left = ROTATE_LEFT(node.left)
    return ROTATE_RIGHT(node)

  // Right Left Case
  if balance < -1 and key < node.right.key:
    node.right = ROTATE_RIGHT(node.right)
    return ROTATE_LEFT(node)

  // 5. Return the (possibly new) root of the subtree.
  return node
```

2.2.2 AVL Deletion Deletion also starts as a standard BST deletion. After the node is removed, the algorithm retraces the path upwards to the root, updating heights and rebalancing each node along the path.

```
function AVL_DELETE(node, key):
  // 1. Perform standard BST delete.
  if node is null: return null
  if key < node.key:
    node.left = AVL_DELETE(node.left, key)
  else if key > node.key:
    node.right = AVL_DELETE(node.right, key)
  else:
    // Deletion logic for 0, 1, or 2 children (same as BST).
    ...

  // If the tree had only one node then return.
  if node is null: return null

  // 2. Update height of the current node.
  UPDATE_HEIGHT(node)

  // 3. Get balance factor and perform rotations to rebalance the tree.
  // The logic is similar to insertion but checks the balance of children subtrees.
  balance = GET_BALANCE(node)
  // Left Heavy
  if balance > 1:
    if GET_BALANCE(node.left) >= 0: // LL Case
      return ROTATE_RIGHT(node)
    else: // LR Case
      node.left = ROTATE_LEFT(node.left)
      return ROTATE_RIGHT(node)
  // Right Heavy
  if balance < -1:
    if GET_BALANCE(node.right) <= 0: // RR Case
      return ROTATE_LEFT(node)
    else: // RL Case
      node.right = ROTATE_RIGHT(node.right)
      return ROTATE_LEFT(node)

  return node
```

2.3 Splay Tree

A Splay Tree is a self-adjusting binary search tree with the additional property that recently accessed elements are quick to access again. It performs basic operations such as insertion, look-up and removal in O(log n) amortized time. All operations on a splay tree are based on the `splay` operation.

2.3.1 Splay Operation The splay operation brings a target key to the root of the tree through a series of rotations. Our implementation is a recursive, top-down splay.

- **Zig:** A single rotation (left or right) when the target's parent is the root.
- **Zig-Zig:** Two rotations in the same direction (e.g., two right rotations).
- **Zig-Zag:** Two rotations in opposite directions (e.g., a left then a right rotation).

```
function SPLAY(node, key):
  // Base case: node is null or key is at root.
  if node is null or node.key == key:
    return node

  // Key lies in left subtree.
  if key < node.key:
    if node.left is null: return node // Key not found.
    // Zig-Zig (Left Left)
    if key < node.left.key:
```

```
      node.left.left = SPLAY(node.left.left, key)
      node = ROTATE_RIGHT(node)
    // Zig-Zag (Left Right)
    else if key > node.left.key:
      node.left.right = SPLAY(node.left.right, key)
      if node.left.right is not null:
        node.left = ROTATE_LEFT(node.left)

    // Final Zig rotation for the node.
    if node.left is null: return node
    else: return ROTATE_RIGHT(node)

  // Key lies in right subtree (symmetric to the left case).
  else:
    if node.right is null: return node // Key not found.
    // Zig-Zag (Right Left)
    if key < node.right.key:
      node.right.left = SPLAY(node.right.left, key)
      if node.right.left is not null:
        node.right = ROTATE_RIGHT(node.right)
    // Zig-Zig (Right Right)
    else if key > node.right.key:
      node.right.right = SPLAY(node.right.right, key)
      node = ROTATE_LEFT(node)

    // Final Zig rotation for the node.
    if node.right is null: return node
    else: return ROTATE_LEFT(node)
```

2.3.2 Splay Insertion To insert a key, we first perform a standard BST insertion. Then, we splay the newly inserted key to the root. This makes the new item quick to access again.

```
function SPLAY_INSERT(node, key):
  // Insert the key as in a normal BST.
  node = BST_INSERT(node, key)
  // Splay the newly inserted key to the root.
  return SPLAY(node, key)
```

2.3.3 Splay Deletion To delete a key, we first splay the tree on that key. If the key is found, it becomes the root. We then remove the root, which leaves two subtrees (left and right). We then join these two subtrees by splaying on the largest key in the left subtree (making it the new root of the left part) and then attaching the right subtree as its right child.

```
function SPLAY_DELETE(node, key):
  if node is null: return null

  // Splay the key to the root.
  node = SPLAY(node, key)

  // If key is not in the tree, do nothing.
  if node.key != key:
    return node

  // Key is now at the root.
  // If there is no left subtree, the right subtree becomes the new tree.
  if node.left is null:
    new_root = node.right
  else:
    // Splay the largest element in the left subtree to its root.
    new_root = SPLAY(node.left, key) // Note: key is not in left subtree, so this brings max element to root.
```

```
    // Attach the original right subtree.
    new_root.right = node.right

free(node)
return new_root
```

3 Test Design

4 Test Results

5 Analysis and Comments

Declaration

We hereby declare that all the work done in this project is of our own independent effort.