ECE 522 Project 2 Final Report: AVL and Red-Black Tree Implementation

0. Group Members

Name	ID
Yicheng Yang	1647241
Jiale Cai	1815497
Zhouyiyang Yang	1868452
Yulong Zhang	1823084

1. Introduction

This report provides a comprehensive description of the development and testing of AVL (Adelson-Velsky and Landis) and Red-Black trees as part of the **ECE 522 - Group Project 2**. The goal was to design, implement, and benchmark these self-balancing binary search trees with a focus on insertion, deletion, searching, and other related operations.

2. Project Overview

The main objectives of this project were:

1. **Develop two self-balancing binary search trees:**
    - AVL Tree
    - Red-Black Tree
2. **Implement functionalities:**
    - Insertion
    - Deletion
    - Search
    - Count leaves
    - Compute the height of the tree
    - In-order traversal
    - Print the tree structure
3. **Benchmark and compare performance** between AVL and Red-Black trees under various workloads.

3. Design

3.1 Tree Data Structure Comparison

The AVL and Red-Black trees are self-balancing binary search trees that maintain O(log n) worst-case time complexity for operations such as insertion, deletion, and search.

Key Differences

Feature	AVL Tree	Red-Black Tree
Balancing Strategy	Height-balanced (height difference of left and right subtrees ≤ 1).	Color-based balancing with red and black properties.
Rotations	May require multiple rotations.	Fewer rotations required during insertion and deletion.
Search Time	Typically faster due to stricter balancing.	Slightly slower due to looser balancing.
Implementation Complexity	Higher complexity due to strict rebalancing.	Simpler and faster in rebalancing operations.

3.2 Implementation Plan

Components of the Project:

1. **AVL Tree Implementation**:
   - Core methods: `insert`, `delete`, `search`, `count_leaves`, `height`, and tree visualization methods.
2. **Red-Black Tree Implementation**:
   - Red-Black properties were implemented to ensure tree balancing with color markers.
3. **CLI Interface**:
   - A command-line interface was implemented to allow users to interact with the AVL and Red-Black trees.
4. **Performance Benchmarking**:
   - Used the **Criterion** benchmarking library to measure the insertion and search performance of AVL and Red-Black trees under varying input sizes.

---

4. Implementation

4.1 AVL Tree

The AVL Tree was implemented with the following core features:

1. **Insert Operation**:
   - Inserts a node and ensures the balance property (height balance) is maintained.

- Uses left and right rotations for rebalancing when necessary.

2. **Delete Operation**:
   - Deletes a node and maintains the AVL properties by performing rotations.
3. **Search Operation**:
   - Search complexity is O(log n) due to the AVL properties.
4. **Count Leaves**:
   - Recursively computes the number of leaf nodes in the AVL tree.
5. **Tree Visualization**:
   - Prints the structure and properties (height) of nodes for debugging and visualization purposes.

4.2 Red-Black Tree

The Red-Black Tree maintains balancing by enforcing five key rules:

1. The root node is always black.
2. Every other node is either red or black.
3. All leaves are considered black.
4. Red nodes only have black children.
5. Any path from the root to its leaves has the same number of black nodes.

Key features of the Red-Black tree implementation:

1. Insert and rebalancing with rotations.
2. Deletion with color rebalancing.
3. Search performance was ensured by maintaining log-depth properties.

4.3 Code Structure

The implementation is modularized into:

1. `avl_tree.rs`: Defines AVL Tree operations and structure.
2. `redblack_tree.rs`: Defines Red-Black Tree operations and balancing logic.
3. `main.rs`: Implements CLI interactions for the user to test AVL and Red-Black operations.
4. `benchmarks.rs`: Uses Criterion to compare insertion and search performance.

---

5. CLI Interface

5.1 Features

The command-line interface allows the user to:

- Choose between AVL Tree and Red-Black Tree.
- Perform insertion, deletion, count leaves, compute height, search, check empty, and in-order traversal.
- View the tree structure in a visualized format.
- Run performance tests or benchmarks.

5.2 Example CLI Interaction

```
=========Welcome for using AVL and Red-Black tree!=========
Please choose the type of tree you want to use
1. AVL Tree
2. Red-Black Tree

1
Please choose the type of value you want to add
1. Integer
2. Floating-point number

1
----------------------------------------
Welcome for using AVL Tree!
----------------------------------------
----------------------------------------
Please choose the operation you want to do: (input corresponding number)
1. Insert node(s) to the tree.
2. Delete node(s) from the tree.
3. Count the number of leaves in a tree.
4. Return the height of a tree.
5. Print in-order traversal of the tree.
6. Check if the tree is empty.
7. Print the tree showing its colors and structure.
8. Quit.
----------------------------------------
1
----------------------------------------
Please input the value(s) of the node(s) that you want to insert: Separate by one
whitespace. e.g. 1 2 3 4 5
3
Insert [3] successfully.
----------------------------------------
Please choose the operation you want to do: (input corresponding number)
1. Insert node(s) to the tree.
2. Delete node(s) from the tree.
```

```
3. Count the number of leaves in a tree.
4. Return the height of a tree.
5. Print in-order traversal of the tree.
6. Check if the tree is empty.
7. Print the tree showing its colors and structure.
8. Quit.
----------------------------------------
1
----------------------------------------
Please input the value(s) of the node(s) that you want to insert: Separate by one whitespace. e.g. 1 2 3 4 5
5
Insert [5] successfully.
----------------------------------------
Please choose the operation you want to do: (input corresponding number)
1. Insert node(s) to the tree.
2. Delete node(s) from the tree.
3. Count the number of leaves in a tree.
4. Return the height of a tree.
5. Print in-order traversal of the tree.
6. Check if the tree is empty.
7. Print the tree showing its colors and structure.
8. Quit.
----------------------------------------
1
----------------------------------------
Please input the value(s) of the node(s) that you want to insert: Separate by one whitespace. e.g. 1 2 3 4 5
8
Insert [8] successfully.
----------------------------------------
Please choose the operation you want to do: (input corresponding number)
1. Insert node(s) to the tree.
2. Delete node(s) from the tree.
3. Count the number of leaves in a tree.
4. Return the height of a tree.
5. Print in-order traversal of the tree.
6. Check if the tree is empty.
7. Print the tree showing its colors and structure.
8. Quit.
----------------------------------------
5
----------------------------------------
```

```
In-order traversal:
in-order: 3 5 8
----------------------------------------
Please choose the operation you want to do: (input corresponding number)
1. Insert node(s) to the tree.
2. Delete node(s) from the tree.
3. Count the number of leaves in a tree.
4. Return the height of a tree.
5. Print in-order traversal of the tree.
6. Check if the tree is empty.
7. Print the tree showing its colors and structure.
8. Quit.
----------------------------------------
7
----------------------------------------
The tree structure is:
AVL Tree Structure:
Root: 5 (Height: 2)
  L: 3 (Height: 1)
  R: 8 (Height: 1)
----------------------------------------
Please choose the operation you want to do: (input corresponding number)
1. Insert node(s) to the tree.
2. Delete node(s) from the tree.
3. Count the number of leaves in a tree.
4. Return the height of a tree.
5. Print in-order traversal of the tree.
6. Check if the tree is empty.
7. Print the tree showing its colors and structure.
8. Quit.
----------------------------------------
6
----------------------------------------
This tree is not empty
----------------------------------------
Please choose the operation you want to do: (input corresponding number)
1. Insert node(s) to the tree.
2. Delete node(s) from the tree.
3. Count the number of leaves in a tree.
4. Return the height of a tree.
5. Print in-order traversal of the tree.
6. Check if the tree is empty.
7. Print the tree showing its colors and structure.
```

```
8. Quit.
---------------------------------------
4
---------------------------------------
The height of the tree is: 2
---------------------------------------
Please choose the operation you want to do: (input corresponding number)
1. Insert node(s) to the tree.
2. Delete node(s) from the tree.
3. Count the number of leaves in a tree.
4. Return the height of a tree.
5. Print in-order traversal of the tree.
6. Check if the tree is empty.
7. Print the tree showing its colors and structure.
8. Quit.
---------------------------------------
8
---------------------------------------
Thank you for using!
```

---

6. Performance Results

We benchmarked the performance of the AVL tree and RB tree against various input sizes.

Tested Tree Sizes for Insertion

Input Sizes: `[10,000, 40,000, 70,000, 100,000, 130,000]`

Tested Tree Sizes for Search

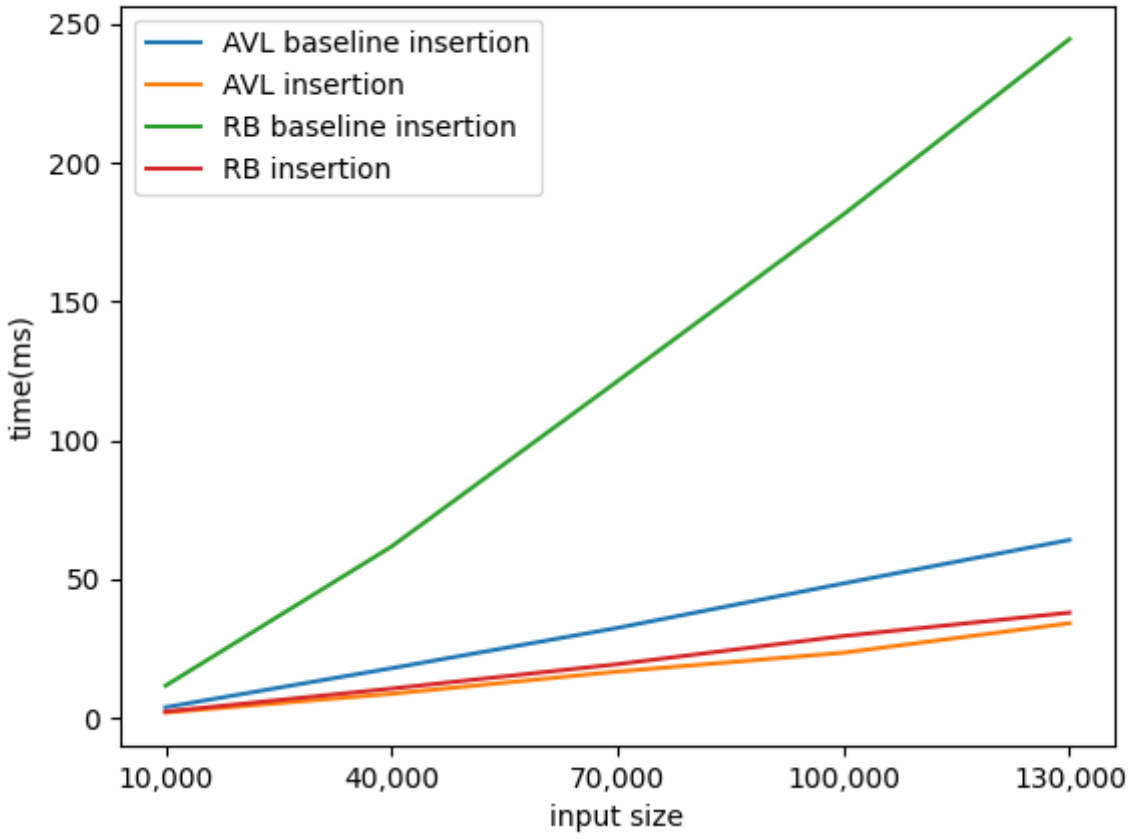
Input Sizes: `[1,000, 4,000, 7,000, 10,000, 13,000]`

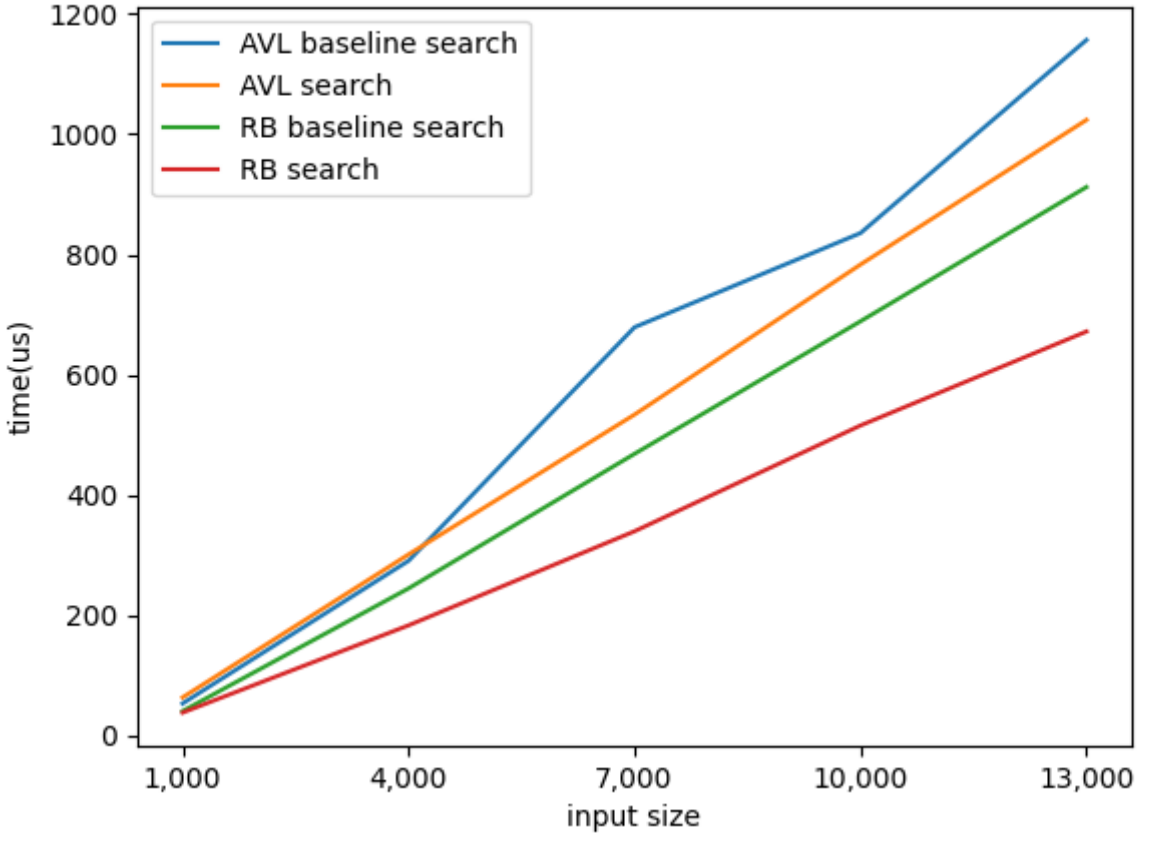
Results

Using the **Criterion** benchmarking library, the performance measurements were made for:

1. **Insertion Time**:
   - AVL insertion was fast but showed higher overhead for very large tree sizes.
2. **Search Time**:
   - AVL trees maintained consistent O(log n) search time.

Comparison Line Graph





7. Challenges and Lessons Learned

Challenges

1. Managing smart pointers like `Rc<RefCell>` while ensuring no borrowing issues.
2. Balancing logic complexity for AVL and Red-Black trees under heavy insert/delete workloads.
3. Benchmark initialization costs and ensuring stable performance measurement using Criterion.

Lessons Learned

1. Understanding how tree balancing impacts insertion and search performance.
2. Learning the difference in complexity between AVL and Red-Black trees and their respective rebalancing strategies.
3. Debugging recursive pointer-based structures in Rust is critical for reliability.

---

8. Future Work

1. Implement better visualization capabilities for AVL and Red-Black trees.
2. Explore additional self-balancing trees like 2-3-4 trees or B-trees for comparison.
3. Expand benchmarks to include edge cases with skewed tree scenarios and concurrent insert/delete operations.

---

9. Project Design Questions

1. What does a red-black tree provide that cannot be accomplished with ordinary binary search trees?

Red-black trees provide **self-balancing properties** that ordinary binary search trees (BSTs) lack. These properties ensure that the tree's height remains logarithmic relative to the number of nodes, which is critical for maintaining efficient operations.

2. Do you need to apply any kind of error handling in your system (e.g., panic macro, Option<T>, Result<T, E>, etc..)?

Yes, error handling is important to ensure the robustness of the system. Many tree operations (e.g., `search`, `delete`) might return `None` if the desired element or node is not found. `Option<T>` is used to handle these cases without panicking. For critical operations that might fail due to logical errors (e.g., failed to convert input), `Result<T, E>` is used to propagate error information to the caller.

3. What components do the Red-black tree and AVL tree have in common?

Core Common Components:

- **Search:** Both trees use the same binary search logic.
- **In-order Traversal:** Traversal logic is identical as it depends on the BST property.
- **Self-rebalancing:** Both trees maintain balance after insertion and deletion operations to ensure logarithmic height.

- **Tree Rotations:** Both require left and right rotations for rebalancing.

---

4. How do we construct our design to “allow it to be efficiently and effectively extended”?

A modular and extensible design ensures the tree implementations can serve as a foundation for more complex data structures. There are some design implemented in this project:

1. **Generic Data Types:**
   - Use generics to allow flexibility in the type of data stored in the tree.
   - Example:

```
pub struct TreeNode<T> {
    pub key: T,
    pub height: i32,
    pub parent: Option<Link<T>>,
    pub left: Option<Link<T>>,
    pub right: Option<Link<T>>,
}
```

2. **Focus on Interoperability:**
   - Ensure the tree implementation can integrate with other data structures or systems by adhering to common Rust traits like `Debug`, `Display`, and `Iterator`.

Future Extensions:

Due to time constraints, some optimizations have not yet been implemented. We plan to apply the following principles in the future to optimize the code and enhance the project's flexibility.

1. **Layered Functionality:**
   - Separate the balancing logic from the basic BST operations.
   - Example:
     - Base layer: Binary Search Tree (`BinaryTree<T>`)
     - Derived layer: AVL Tree or Red-Black Tree.
2. **Pluggable Balancing Mechanisms:**
   - Design the balancing logic as modular units that can be swapped for different tree types.
   - Example:
     - Red-Black balancing with color rules.
     - AVL balancing with height rules.
3. **Focus on Interoperability:**

- Ensure the tree implementation can integrate with other data structures or systems by adhering to common Rust traits like `Debug`, `Display`, and `Iterator`.

---

9. Conclusion

We successfully implemented AVL and Red-Black trees with their respective features, provided a functional command-line interface, and benchmarked their insertion and search performance. Through this, we gained insights into the design considerations of these two self-balancing trees and their trade-offs in terms of complexity, speed, and use cases.

---

10. References

1. **AVL Tree Wikipedia Article**: https://en.wikipedia.org/wiki/AVL_tree
2. **Red-Black Tree Wikipedia Article**: https://en.wikipedia.org/wiki/Red-black_tree
3. **Criterion Benchmark Documentation**: https://docs.rs/criterion