ECE 421
PROJECT 2

Qi Zhou
Jakob Lau
Tianyuan Fang

Design Rationale

A red-black tree provides a way to balance a binary search tree. It adds logic to rebalance after inserts. By enforcing a set of rules, a tree can be programmatically verified to be balanced. The rules are:
The root node is always black
Each other node is either red or black
All leaves are considered black
A red node can only have black children
Any path from the root to its leaves has the same number of black nodes
By enforcing these rules, a tree can be programmatically verified to be balanced. Rules 4 and 5 provide the answer: if each branch must have the same number of black nodes, neither side can be significantly longer than the other unless there were lots of red nodes. This makes red-black trees more balanced than ordinary binary search trees.

Another method to balancing a binary search tree is by using an AVL tree. It has one main rule where the difference between the depth of the right and left subtrees of any node cannot be more than one. This difference is called the balance factor. Every insertion and deletion will check if the tree has a balance factor of more than one and will rotate the tree based on the pattern of the height imbalance using 4 different possible types of rotations: right rotation, left rotation, right-left rotation and left-right rotation.

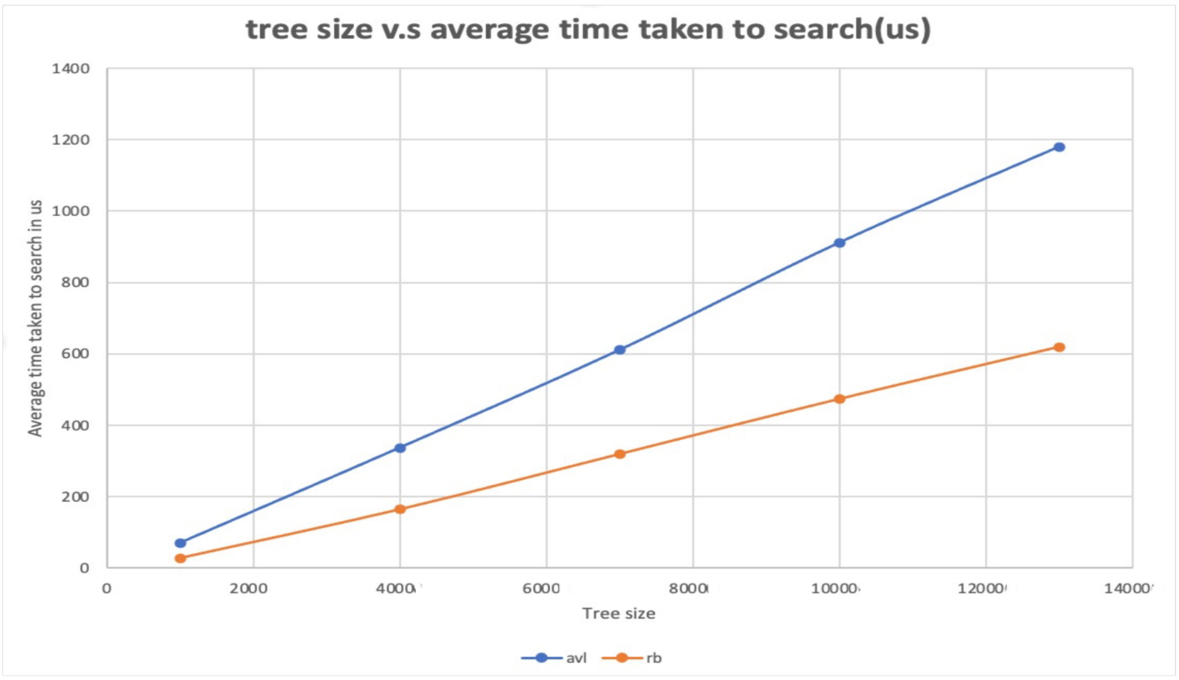
Both Red Black Tree and AVL tree require careful handling of edge cases and null values when performing insertion, deletion, and rotation operations. To achieve robust error handling, we used the Option<T> type and the 'if let' statement in Rust. The Option<T> type represents a value that can be either Some(T) or None. The 'if let' statement allows us to match a pattern and execute a block of code only if the pattern matches. This way, we can handle the cases where a node is None gracefully without panicking or crashing the program.

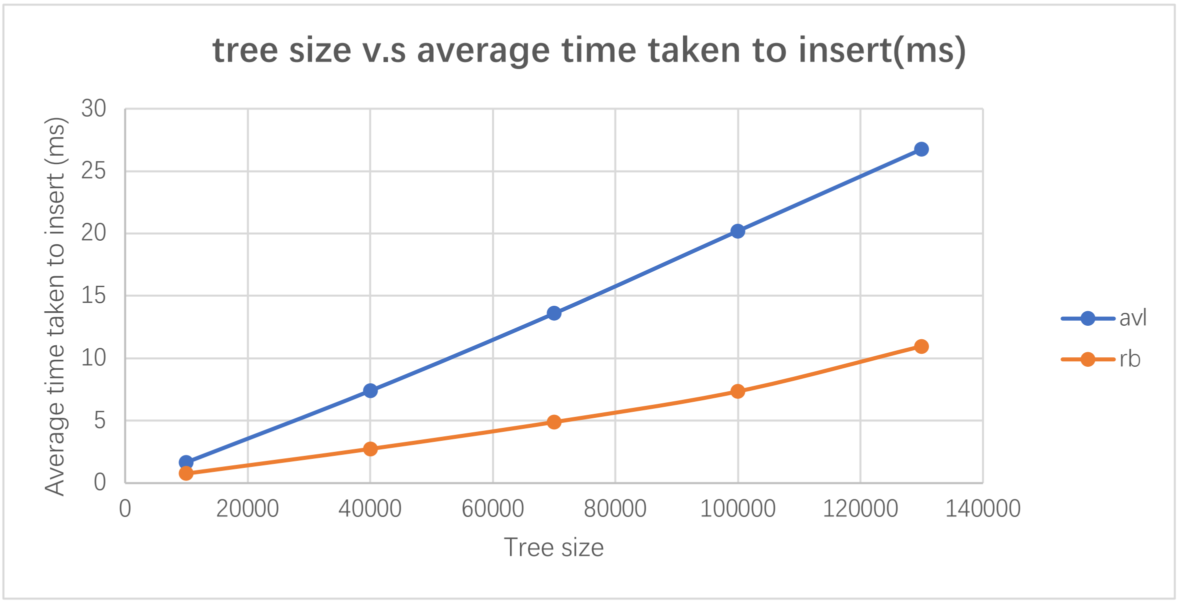
Both red-black trees and AVL trees are self-balancing binary trees. They both have the following in common:

They are both binary search trees.
They both use rotations to balance the tree.
They both have a height of O(log n) where n is the number of nodes in the tree.

One of the challenges of implementing self-balancing binary search trees such as AVL tree and Red Black tree is to manage the memory allocation and deallocation of the nodes. In Rust, a systems programming language that guarantees memory safety, we used smart pointers to define the tree's node structure. Smart pointers are types that implement the `Deref` and `Drop` traits, which allow them to behave like references and automatically free the memory when they go out of scope. By using smart pointers such as `Rc` and `RefCell`, we were able to efficiently and effectively extend the node structure to support different kinds of trees. For example, we could reuse the same code to build a 2-3-4 tree or a B-tree by changing the number of keys and children in each node.

Benchmark:





From benchmarking result, we could see that red black tree is generally faster in insertion and search than avl tree, therefore red black tree has better performance in insertion and search time. A detailed benchmark report can be found within folder Project2/target/criterion.

User Manual

Getting started:

```
cargo run
```

There are currently two packages in the project, One is avl and the order is for redblack, each one containing their respective trees. When cargo run is called, a command line interface will be shown that prompts the user to either select a red-black tree or AVL tree. Afterwards, the user will be given a list of commands that can:

1. insert a node,
2. delete a node,
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 structure, or
8. exit the program.

Each command can be selected by inputting their respective number.

Benchmarking:

```
cargo bench
```

When a benchmark is run for the program, it will return the results for inserting 10000, 40000, 70000, 100000 and 130000 node entries and searching 1000, 4000, 7000, 10000, 13000 node entries for both a red-black and AVL tree.