ECE 421 Project 2: Trees, Trees, and More Trees

Team 1:
Prabh Kooner,
Jiannan Lu,
Brandon Hoynick

March 20, 2024

Overview:
This document is the summary of design considerations for our crate library that produces the binary tree types of AVL tree and Red-Black tree. This document serves as a user manual for the crate and an insight into the development design for it (including considerations prompted from the requestors of the crate).

User Manual:
Get Started – How to add crate to your project:
1. Copy the crate file to your project: Copy 'lib.rs' to project's 'src' folder.
2. In your project's 'main.rs' (for example), add 'extern crate trees;' to top;
then add 'use trees::tree::*;' for RB trees or for AVL trees.
3. Begin using supplied structs and associated commands. The example snippet below contains all high level commands and some print out results; note that self-printing commands begin with print (removing these will return functional result instead of print).

```
    let mut myrbtree = RBTree::new();
    myrbtree.print_find(30); // should be: Cannot find the 30 node in the tree.
    myrbtree.print_is_tree_empty(); // should be: true
    myrbtree.insert(20);
    myrbtree.print_is_tree_empty(); // should be: false
    myrbtree.insert(10);
    myrbtree.insert(30);
    myrbtree.insert(40);
    myrbtree.insert(50);
    myrbtree.print_count_number_of_leaves(); // should be: 6
    myrbtree.print_get_height_of_tree(); // should be: 3
    myrbtree.print_in_order_traversal(); // should be: 10 20 30 40 50
    myrbtree.print_is_tree_empty(); // should be: false
    myrbtree.print_pre_order_traversal(); // should be: 20 10 40 30 50
    myrbtree.print_tree();
    // should be:
            ┌── 50 (Red)
        ┌── 40 (Black)
        │   └── 30 (Red)
    ┌── 20 (Black)
    │   └── 10 (Black)
    myrbtree.print_find(30); // should be: Found node: 30
    myrbtree.print_find(22); // should be: Cannot find the 22 node in the RBTree.
    myrbtree.print_delete(50); // should be: Found node: 50, deleting.
    myrbtree.print_tree();
    // should be:
        ┌── 40 (Black)
        │   └── 30 (Red)
    ┌── 20 (Black)
    │   └── 10 (Black)
    myrbtree.print_delete(50); // should be: Cannot find the 50 node in the tree.
```

```rust
    let mut mytree = AVLTree::new();
    mytree.print_find(30); // should be: Cannot find the 30 node in the tree.
    mytree.print_is_tree_empty(); // should be: true
    mytree.insert(20);
    mytree.print_is_tree_empty(); // should be: false
    mytree.insert(10);
    mytree.insert(30);
    mytree.insert(40);
    mytree.insert(50);
    mytree.print_count_number_of_leaves(); // should be: 6
    mytree.print_get_height_of_tree(); // should be: 3
    mytree.print_in_order_traversal(); // should be: 10 20 30 40 50
    mytree.print_is_tree_empty(); // should be: false
    mytree.print_pre_order_traversal(); // should be: 20 10 40 30 50
    mytree.print_tree();
    // should be:
    /*
            ┌── 50(1)
        ┌── 40(2)
        │   └── 30(1)
    ┌── 20(3)
    │   └── 10(1)
        */
    mytree.print_find(30); // should be: Found node: 30
    mytree.print_find(22); // should be: Cannot find the 22 node in the tree.
    mytree.print_delete(50); // should be: Found node: 50, deleting.
    mytree.print_tree();
    // should be:
    /*
        ┌── 40(2)
        │   └── 30(1)
    ┌── 20(3)
    │   └── 10(1)
        */
    mytree.print_delete(50); // should be: Cannot find the 50 node in the tree.
```

Program-based Tree Tester – Users can optionally execute our '**trees_tester.exe**' file to get program to test both trees. The following will show how to test crate structs and functions through a demo executable:
1. Copy the crate executable file ('**trees_ tester.exe**') to folder location of your choice.
2. Navigate/Open a terminal to location of '**trees_tester.exe**' and execute file by running '**./trees_ tester**' or '**trees_ tester**' (whatever command your OS uses to run executables).
3. The program will prompt you to chose to build a Red-Black tree or AVL tree.
4. After selecting, you can build/modify your tree with various commands (and there are prompts for exiting too). The commands are:

- **insert <value>** : Inserts the <value> into the tree; (duplicates are skipped).
- **find <value>** : Finds the <value> from tree; (queried values that were not present in the tree will print a message stating so).
- **delete <value>** : Deletes the <value> from tree; (queried values that were not present in the tree will print a message stating so).
- **leaves** : Counts the number of leaves (NULL nodes) in the tree.
- **height** : Counts the longest strip of nodes (from root to ends).
- **inorder** : Prints the in-order traversal (from left most child of whole tree, to right most child of whole tree) of tree's node values; this print out is essentially the same as printing all tree's values in an ascending sort.
- **preorder** : Prints the pre-order traversal, from root downward to left child to right child, of tree's node values; this print out is essentially the same as printing all tree's values from top row downward (grabbing the left most subtrees first).
- **ifempty** : This checks if tree is empty (i.e. has no nodes; is just a root pointer).
- **print** : Prints tree in structured format, where the printout shows the root pointer, the connected node's values (and other attributes like colour and parent value), and line connections between nodes.
- **exit** : Exits the program.

Major Innovations – Additional to the requested project specifications:

The user manual specifies standards options for the tree's functions, but we also decided on other specifications:

- Duplicates are not allowed, instead when duplicate node is inserted, the pointer to the in-tree-original node (of duplicate) is returned.
- We added 'pre-order traversal' function (It was not specified in the initial crate request, but it was similar to implements as the in-order version, so we added it; as seen in the user manual).
- We added 'find' function (it was used for benchmark testing, but not specified in the initial crate request, so we added it; as seen in the user manual).

Design rationales – Considerations based on requestor's design questions:

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

Both AVL and Red-Black tree provide self balancing mechanisms that allow its total structure to remain in such a way that worst case searching for an element would be O(logN); the self balancing spreads out elements so that the longest tree leg height is only a couple nodes difference from the shortest tree leg height; this provides evenness in search locations so that value areas are not too far to reach (hence the O(logN) guarantee). Red-Black does this by have interchanging red and black colors on nodes (see it's Wikipedia page for all the rules) with the basic intention being that all paths have the same black node lengths (which results in sharing heights), while the AVLE tree does this by having a balance-factor calculated on node's children to be within a [-1,0,1] difference range, and anything more than that difference would constitute imbalance and correction (this also results in sharing heights).

Refs: https://en.wikipedia.org/wiki/Red%E2%80%93black_tree
https://en.wikipedia.org/wiki/AVL_tree

2- Please add a command-line interface (function main) to your crate to allow users to test it. As state in the above User Manual, we have included a 'Program-based Tree Tester' executable.

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

We found that Option<T> are very good types to have for code that contains 'null' cases (especially empty trees or null nodes i.e. leaves) and changes or queries that could result with not finding values (another None case), otherwise we try to setup the program to not rely on panics nor Result::Err.

4- What components do the Red-black tree and AVL tree have in common? Don't Repeat Yourself! Never, ever repeat yourself – a fundamental idea in programming.

We ended up reusing the Node structs to allow us to reused node specific functions easily, but didn't find a good solution for sharing functions as the input/outputs in Rust need to be type safe, so having different tree types requires more of the same function signatures.

5- How do we construct our design to "allow it to be efficiently and effectively extended"? For example. Could your code be reused to build a 2-3-4 tree or B tree?

Code functions should be built in a modular format to allow sharing/reusing of components and easy extension/upgrading.

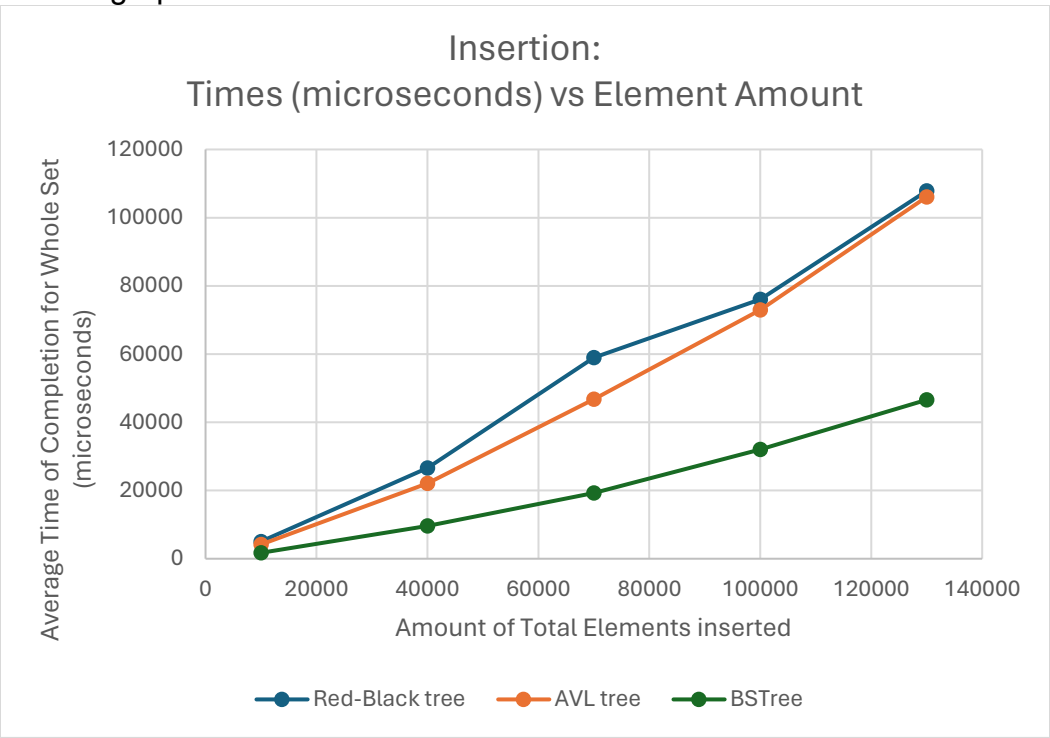
Details of any known errors, faults, defects, missing functionality, etc.:

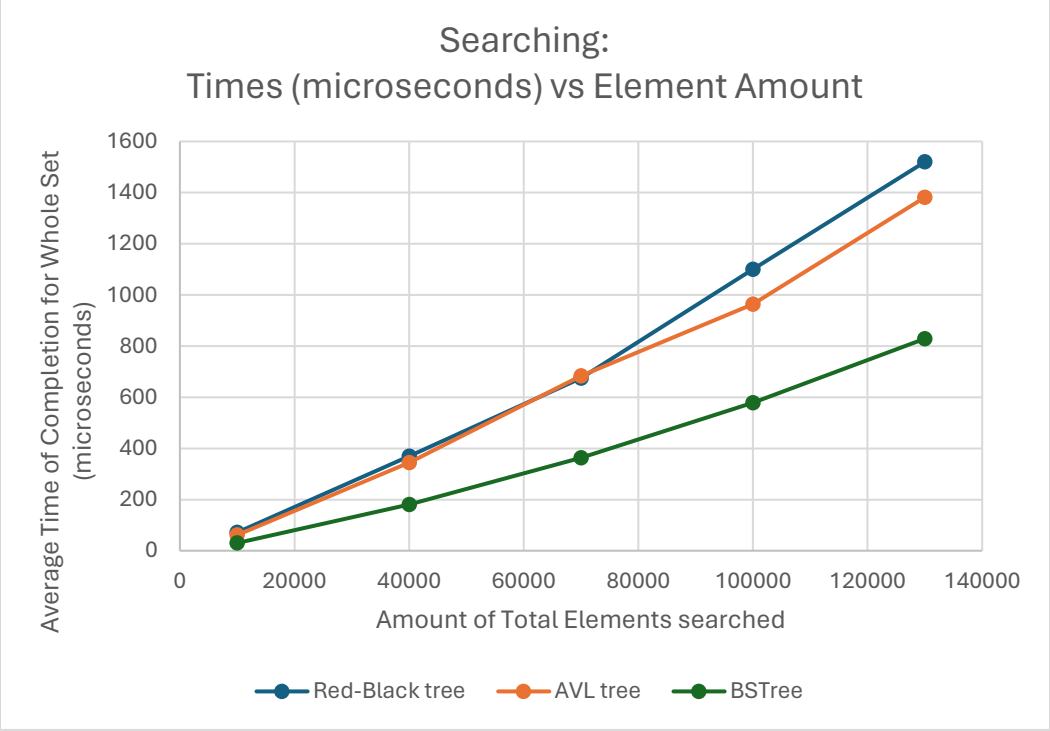
- We tested all user functions and simple cases of tree modification (rotates, color changes), but are unaware of any issues on complex large trees with edge cases.
- Our trees could ideally be set up with any data types that have inherit sortable values, but we only set ours up for integers.
- We would get unpredictable issues with deletion functions, in our testing it seems to work with small tree, but unsure of this for large trees with edge case deletions.

A 2-minute video highlighting the new system:
https://drive.google.com/file/d/19jhF1-PGX7ekcJyJvHTaHldyr3a-VMVT/view?usp=sharing

Benchmark testing:

We used the criterion crate to benchmark our tree using the requested specifications (10000 to 130000 entries, insert and search separately). We also compared it to a binary search tee crate (Ref: https://crates.io/crates/binary_search_tree). These were plotted on two graphs of insert and search.





Both graphs have comparable speeds between graph type, for both operation types, but ours are unfortunately slower than the simple binary search tree crate (maybe its not simple and is optimized).
The AVL tree seems to be marginally better than the Red-Black tree for both insertion and searching.

Which data structure is more efficient?
Supposedly the RB vs AVL has its own strengths over the other (but over regular binary trees, they both are faster O(logN) processing); Supposedly: Red-black has faster insertion, but AVL has faster searching (Ref: https://www.geeksforgeeks.org/red-black-tree-vs-avl-tree/). In our results, the AVL tree seems to perform better than the Red-Black tree.

Do you think we need to accommodate other test cases?
Instead of just searching lower numbers, you could do middle numbers, and random all numbers (but not upper numbers as that would be same as lower numbers, though you could do that just to confirm).

Do you think we need to include additional data structures in the benchmarking to perform as the baseline (i.e., binary search tree)?
We tested with a crate for binary search tree, but our performance was worse.

Project References / Helper Tools
https://www.cs.usfca.edu/~galles/visualization/AVLtree.html
https://www.cs.usfca.edu/~galles/visualization/RedBlack.html
https://crates.io/crates/tree_collections
https://en.wikipedia.org/wiki/Red%E2%80%93black_tree
https://en.wikipedia.org/wiki/AVL_tree
https://www.geeksforgeeks.org/introduction-to-red-black-tree/?ref=lbp
https://www.geeksforgeeks.org/introduction-to-avl-tree/?ref=lbp
https://www.geeksforgeeks.org/red-black-tree-vs-avl-tree/
https://crates.io/crates/binary_search_tree
{
Video's picture of tree: https://www.britannica.com/plant/tree
Video's intro music (by tamesu): https://opengameart.org/content/level-theme
}