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.

**User Manual: (can put this in this in README.md file too)**

**(explain functions, and have a “get started”, [see other crates for examples])**

~~-also crate.io: example ‘get started’~~ [~~https://crates.io/crates/tree\_collections~~](https://crates.io/crates/tree_collections)

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::rbtree::\*;’ for RB trees;

and/or use ‘trees:: rbtree::avltree;’ for AVL trees.

3. Begin using supplied structs and associated commands:

let mut rb\_tree = RBTree::new();

rb\_tree.insert(1);

rb\_tree.insert(2);

rb\_tree.insert(3);

rb\_tree.insert(4);

rb\_tree.insert(5);

rb\_tree.delete(3);

println!("Red-Black Tree:");

rb\_tree.print();

println!("Number of leaves: {:?}", rb\_tree.count\_leaves());

println!("Height of tree: {:?}", rb\_tree.height());

let mut avl\_tree = AVLTree::new();

avl\_tree.insert(1);

avl\_tree.insert(2);

avl\_tree.insert(3);

avl\_tree.insert(4);

avl\_tree.insert(5);

avl\_tree.delete(3);

println!("AVL Tree:");

avl\_tree.print();

println!("Number of leaves: {:?}", avl\_tree.count\_leaves());

println!("Height of tree: {:?}", avl\_tree.height());

Program-based Tree Tester – How to test crate structs and functions through a demo executable: (i.e. Users can optionally execute our main.exe (maybe call it ‘tree\_builder’) file to get program to test both trees.)

1. Copy the crate executable file to folder location of your choice: Copy ‘tree\_builder.exe’ to desired folder.

2. Navigate/Open a terminal to location of tree\_builder.exe and execute file by running ‘./tree\_builder’ or ‘tree\_builder’ (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).
* 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 (with left child to right child, per node) of tree’s node values; this print out is essentially the same as printing all tree’s values from top row to bottom row (with printing left node to right node, per row).
* 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 print out 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 project specification:**

Not sure what to say??? (maybe any bonus stuff); maybe shortly describe how our design is ‘easy’ to use? 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 of design based on decisions with regard to the above design questions:**

1. **Please ponder (and answer) the following questions as you go:**

**1- What does a red-black tree provide that cannot be accomplished with ordinary binary search trees? “**Compared to other self-balancing binary search trees, the nodes in a red-black tree hold an extra bit called "color" representing "red" and "black" which is used when re-organising the tree to ensure that it is always approximately balanced. The (re-)balancing is not perfect, but guarantees searching in Big O time of O(logN).” Ref: <https://en.wikipedia.org/wiki/Red%E2%80%93black_tree>

“ The [AVL tree](https://en.wikipedia.org/wiki/AVL_tree) is another structure supporting OlogN�(log⁡�) search, insertion, and removal. AVL trees can be colored red–black, thus are a subset of RB trees. Worst-case height is 0.720 times the worst-case height of RB trees, so AVL trees are more rigidly balanced. The performance measurements of Ben Pfaff with realistic test cases in 79 runs find AVL to RB ratios between 0.677 and 1.077, median at 0.947, and geometric mean 0.910.[[22]](https://en.wikipedia.org/wiki/Red%E2%80%93black_tree#cite_note-22) [WAVL trees](https://en.wikipedia.org/wiki/WAVL_tree) have a performance in between those two. “ Ref: <https://en.wikipedia.org/wiki/Red%E2%80%93black_tree>

“ VL trees are often compared with [red–black trees](https://en.wikipedia.org/wiki/Red%E2%80%93black_tree) because both support the same set of operations and take O(log⁡�)O(logN) time for the basic operations. For lookup-intensive applications, AVL trees are faster than red–black trees because they are more strictly balanced. Similar to red–black trees, AVL trees are height-balanced. Both are, in general, neither [weight-balanced](https://en.wikipedia.org/wiki/Weight-balanced_tree) nor mu�-balanced for any mu <= 0.5 �≤12; that is, sibling nodes can have hugely differing numbers of descendants.” Ref: <https://en.wikipedia.org/wiki/AVL_tree>

**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), 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.**

I think all the code can be easily copied (but some are ingrained not to be reused). \* we should try to make this work for both trees, if we have time.

**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 (\*and highly cohesive, low coupling format; this is OOD though) format to allow sharing/reusing of components and easy extension/upgrading.

**A list of known errors, faults, defects, missing functionality, etc. Telling us about your system’s limitations will score better than letting us find them!:**

Any known issues? 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.

**A 2-minute video highlighting the new system – marketing is everything ☺:**

(\*link to google drive of video)

**Benchmark testing:** whether a Red-black tree or AVL tree has a better performance in insertion and search time

**Part 3.** You had a discussion with your project manager about. To have the final say in this argument, you thought that you should do some benchmarking to test the two trees in some of the worst-cases. To decide on the test cases, you thought that the worst case for a binary search tree is when elements are inserted continuously in increasing or decreasing values (e.g., 3, 5, 7, 8, 11,… ). On the other hand, the worst case for searching would be when we search for elements that are located at the top or at the bottom of the tree. Hence, this is what you are planning to do to benchmark the two trees:

Show chart for:

for tree\_size in (10,000, 40,000, 70,000, 100,000, 130,000) do:

Start by creating an empty tree.

Values with tree\_size are inserted into the tree.

A search is conducted for the (tree\_size/10) lowest values.

end

For example, the first benchmark case would be inserting 10K elements in the tree and then search for the 1000 lowest elements.

Please benchmark insert and search separately!

Save the benchmark results and illustrate them using the appropriate graphs and charts. Provide your own comments concluded from the results:

**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/> ) But what was our actual results?

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

Could try comparing against: binary search tree, B+ tree, 2-3 trees

<https://www.geeksforgeeks.org/types-of-trees-in-data-structures/>

<https://www.geeksforgeeks.org/2-3-trees-search-and-insert/>

References???

Visualizers

Viewed crate srcs

Geekes for geeks