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

Team 1:

Prabh Kooner,

Jiannan Lu,

Brandon Hoynick

March 20, 2024

Your performance will be measured on:

● The accuracy of the delivered code with the requirements above.

● The performance (efficiency) of the code.

● Bonus credits will be given for new additional features beyond the old version.

By the deadline, you must deliver:

● A copy of the source code of the two libraries.

● A design document outlining:

o Major innovations – Additional to the project specification.

o A detailed rationale for your augmented decisions with regard to the above design questions.

o 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!

o A user manual.

o A 2-minute video highlighting the new system – marketing is everything ☺.

**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?

-duplicates are not allowed, instead when duplicate node is inserted, the pointer to the in-tree-original node (of duplicate) is returned

-added pre-oder traversal function (to see order of root to edges)

-added find function (it was used for benchmarking but not specified in user operations, so we added it)

**A detailed rationale for your augmented decisions with regard to the above design questions:**

1. **Your library must allow the user to perform the following operation:**

**1- Insert a node to the red-black tree.**

**2- Delete a node from the red-black 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. (Using println!(“{:#?}”,tree); is NOT sufficient)**(might not actually have to give rationale for these required specifications, but anything said here could be used in the ‘user manual’.)

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 �(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>

**2- Please add a command-line interface (function main) to your crate to allow users to test it.**

*~~Not needed, but may be insightful:~~* [*~~https://doc.rust-lang.org/book/ch14-02-publishing-to-crates-io.html~~*](https://doc.rust-lang.org/book/ch14-02-publishing-to-crates-io.html)

~~Could use this:~~ [~~https://doc.rust-lang.org/book/ch14-04-installing-binaries.html~~](https://doc.rust-lang.org/book/ch14-04-installing-binaries.html)

~~--involves creating a runnable program (within main.rs) that uses ‘cargo install’ (since our program isn’t on crate.io, I think we can just forgo the cargo install and just simulate a cargo release of a program that includes both trees and user prompts of : new, insert, find, delete, get length, print all, etc.)~~

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

We added an executable of a simulator program of both trees (see user manual for instructions).

**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 user manual:**

**(can use this for README.md file)**

**(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 with how to added crate to project:

-have lib.rs in src folder; 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::avltree::\*;’ for AVL trees

-users can optionally execute our main.exe (maybe call it ‘tree\_builder’) file to get program to test both trees:

--navigate terminal to main.exe and execute file (./main.exe or main.exe, whatever your OS)

--the program will prompt you to chose to build an RB tree or AVL tree

--after selecting, you can build/modify your tree with various commands of:

(insert <value>, delete <value> …list and explain briefly)

--the program also has prompts for exiting

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

(\*link to google drive of video)

**Part 3.** You had a discussion with your project manager about whether a Red-black tree or AVL tree has a better performance in insertion and search time. 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:

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);

Red-black has faster insertion, but AVL has faster searching

Ref: https://www.geeksforgeeks.org/red-black-tree-vs-avl-tree/

**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/>