Course Project
Software Construction Verification Evaluation

Group 8

Sajjad Alizadeh

Table of Content

Major Innovations

We expanded the capabilities of both the AVL tree and Red-Black tree beyond the original project specifications by introducing additional functionalities:

- **Printing Preorder and Postorder Traversals**: These are fundamental traversal methods within data structures, not limited to trees but also prevalent in linked lists. Preorder traversal, for instance, finds application in emulating a computer's file directory structure, while postorder traversal aids in determining memory size allocation for folders. Additionally, postorder traversal can serve as a reference for system process cleanup, ensuring child processes are handled before the main process is terminated.
- **Element Existence Check:** The core purpose of data structures revolves around data storage. To fulfill this purpose effectively, they require fundamental functions like addition, deletion, and search. Much like in a database such as MySQL, the ability to verify whether an element exists becomes crucial, prompting us to integrate this feature.
- **Counting Nodes:** Providing a macro summary of the number of nodes in the tree aids users in comprehending the volume of stored elements, offering a snapshot of the structure's current state.
- **Retrieving Maximum and Minimum Elements:** Understanding the distribution of data within the tree becomes more accessible when users have access to the maximum and minimum elements. This insight empowers users to manage forthcoming insertions by offering a grasp of the data distribution.

These additions not only enhance the usability of the trees but also provide users with valuable insights into the structure's contents and enable better control and manipulation of the data stored within.

Decisions and Questions

Q1. What does a red-black tree provide that cannot be accomplished with ordinary binary search trees?

Binary trees can be quite different in how they handle stuff depending on how they're set up. Red-Black trees are like a special kind of binary tree that tries hard to stay nice and balanced. Imagine each side of the tree is like a set of scales, and Red-Black trees try to keep both sides pretty close in weight.

When these Red-Black trees are nicely balanced—where one side isn't much heavier than the other—putting new elements into them is pretty fast. It always takes around log(n) time, which means even with lots of elemets, it won't take too long to add more.

But in a regular binary tree, nodes might not stay balanced. If they get all out of unbalanced, adding new elements could take a lot longer. Sometimes it might even take as much time as there are things already in the tree (that's O(n)), which could be a real slowdown if the tree grows big in the worst-case scenarios.

Q2. Please add a command-line interface (function main) to your crate to allow users to test it.

The project consists of both a lib.rs and a main.rs file, making it both a library and a runnable program. This dual setup allows users to not only utilize it as a library but also run and test it as an executable.

There's a Command Line Interface (CLI) built into the project, kicking in as soon as the project is run. This CLI acts as a guide for users, helping them select the type of tree and the kind of keys they want to store in it. Once these preferences are known, the CLI sets up the tree accordingly.

Then, the CLI prompts the user to choose operations to perform on the tree. After each operation, it asks what the next action should be, continuing this cycle until the user decides to exit the application. This interactive approach enables users to engage with the project and test different tree operations at their own pace.

Q3. Do you need to apply any kind of error handling in your system (e.g., panic macro, Option<T>, Result<T, E>, etc.)

In handling errors within our library package, it's important to be mindful of how we manage these situations. While panic! is a forceful way to stop the program, it's not suitable for a library since we can't predict how users will handle these abrupt terminations.

Instead, we lean on tools like Option and Result for error handling. Option is extensively used throughout the code, especially in every node of the tree. On the other hand, although

I haven't employed Result yet, it's a logical fit for functions like search. This is because the search might either find a result and want to return it or not find anything and need to pass an error message back to the caller.

By leveraging Option and planning to integrate Result in such scenarios, we can better manage errors within our library, ensuring more predictable and controlled handling of potential issues without abruptly stopping the program.

Q4. 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.

Both the red-black tree and AVL tree share a common foundation as types of binary trees. Despite their differences, they also have similarities. For instance, a red-black tree node remains a binary tree node but includes an additional attribute for color, while an AVL tree node, being a binary tree node, maintains a record of the height difference.

To capitalize on these shared traits, We utilize traits for both tree types and their respective nodes. This approach helps maintain a common ground between these trees. By employing traits, functionalities needed in both tree types can be written just once within this shared trait. This way, common operations or attributes can be implemented more efficiently and consistently across both trees, reducing redundancy and streamlining the overall code structure.

Q5. 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?

The library's design with a focus on binary trees lays a solid foundation for seamlessly incorporating additional variations of binary trees. The existing codebase, with its modular and reusable components, streamlines the process of integrating new types of binary trees.

Given the common functionality already present in the library, such as node structures, basic tree operations, and perhaps shared traits, introducing another variety of binary trees becomes a much more manageable task. Leveraging these existing structures and functionalities reduces the need to reinvent the wheel for common operations.

However, extending the library to encompass additional tree variations like B trees and 2-3-4 trees poses a different challenge. These tree types differ significantly from binary trees as they often contain multiple keys within each node and possess more than two children.

Challenges and Disadvantages

Testing

The project's unit tests currently have limitations, but there are numerous strategies available to enhance them, such as dividing tests into smaller units to assess specific functionalities rather than combining multiple operations, can greatly improve their effectiveness. As of now, certain segments of the application, such as the CLI, lack tests. This gap exists because the tests created so far focus on the library components rather than the user interaction aspects present in the binary sections of the application.

Data types

The trees exclusively handle integers and not other data types like characters or floating-point numbers. Adapting these trees to work with different data types can be straightforward by using generics to generalize their usage. However, it's essential to note that red-black trees and AVL trees are specifically designed to efficiently manage keys, and in most cases, these keys can be interpreted as integers. Thus, our current implementations remain practical and effective for their intended purpose, even though they are tailored to handle integer keys.

Documentation

Making the package publicly available involves providing comprehensive documentation to guide developers on utilizing the package effectively. A great way to achieve this is by leveraging Rust's **rustdoc** tool, which generates documentation directly from the code. By utilizing **rustdoc**, we can create clear, understandable, and easily accessible documentation that showcases the package's functionalities, usage instructions, provided APIs, and examples. This documentation serves as a valuable resource for developers, enabling them to understand, implement, and utilize the package's features efficiently.

User Manual

User CLI

First, we start by explaining how to use user CLI, the cli can easily run by this command:

```
cargo run
```

Then we can choose the type of the tree

```
→  code cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.19s
     Running `target/debug/tree`
Please select the desired tree (Insert 1 or 2):
1- RB Tree
2- AVL Tree
```

For this example, we choose red black tree, the commands for AVL tree is exactly the same. After choosing the red black tree, a menu consist of operations shows up.

```
RB tree is selected!
Please select an operation.
1 - Insert a node to the tree.
2 - Delete a node from the tree.
3 - Count the number of leaves in the tree.
4 - Return the height of the tree.
5 - Print inorder traversal of the tree.
6 - Print preorder traversal of the tree.
7 - Print postorder traversal of the tree.
8 - Check if the tree is empty.
9 - Print the tree, showing it's structure.
10 - Return the max element of the tree.
11 - Return the min element of the tree.
12 - Search the tree for the given key.
13 - Count the number of nodes.
14 - Exit
```

Now if we choose to insert a node, the program asks for the key, let’s insert key 1.

```
Please select an operation.
1 - Insert a node to the tree.
2 - Delete a node from the tree.
3 - Count the number of leaves in the tree.
4 - Return the height of the tree.
5 - Print inorder traversal of the tree.
6 - Print preorder traversal of the tree.
7 - Print postorder traversal of the tree.
8 - Check if the tree is empty.
9 - Print the tree, showing it's structure.
10 - Return the max element of the tree.
11 - Return the min element of the tree.
12 - Search the tree for the given key.
13 - Count the number of nodes.
14 - Exit
1
Please enter the key to insert
1
Inserted 1
```

We also add 2,3,4,5,6,7,8. Now if we print the tree we have this:

```
Please select an operation.
1 - Insert a node to the tree.
2 - Delete a node from the tree.
3 - Count the number of leaves in the tree.
4 - Return the height of the tree.
5 - Print inorder traversal of the tree.
6 - Print preorder traversal of the tree.
7 - Print postorder traversal of the tree.
8 - Check if the tree is empty.
9 - Print the tree, showing it's structure.
10 - Return the max element of the tree.
11 - Return the min element of the tree.
12 - Search the tree for the given key.
13 - Count the number of nodes.
14 - Exit
9

            L 1 Black
      L 2 Red
      |     R 3 Black
Root 4 Black
      |     L 5 Black
      R 6 Red
            R 7 Black
                  R 8 Red
```

We can also print different traverses of the tree (in-order, pre-order, post-order):

```
5
In order traversal of the tree: [1, 2, 3, 4, 5, 6, 7, 8]
```

```
6
Pre order traversal of the tree: [4, 2, 1, 3, 6, 5, 7, 8]
```

```
7
Post order traversal of the tree: [1, 3, 2, 5, 8, 7, 6, 4]
```

We can get max and min of the elements:

```
10
Max element of the tree: 8
```

```
11
Min element of the tree: 1
```

We can count the number of the leaves in the tree:

```
3
Number of leaves: 4
```

We can get height of the tree:

```
4
Height of the tree: 4
```

We can check if the tree is empty or not:

```
8
The tree is not empty.
```

We can check if an element exists in the tree (4 exists in the tree but 14 doesn't):

```
12
Please enter the key to search
4
Existance of the key in tree: true
```

```
12
Please enter the key to search
14
Existance of the key in tree: false
```

We can count the number of nodes in the tree:

```
13
Number of nodes in tree: 8
```

Also, we can delete an element, let's say delete 4 (the root):

```
2
Please enter the key to delete
4
Deleted 4
```

The tree would look like this after deletion:

```
9

            L 1 Red
      L 2 Black
Root 3 Black
      |     L 5 Black
      R 6 Red
            R 7 Black
                  R 8 Red
```

As you can see, the deletion was successful. Also, this flow can easily done for AVL tree.

Functions:

We implemented various functions for both trees that developers who use our crate could benefit of it. Here are the list of functions:

```
fn new() -> Self;
fn get_root(&self) -> &Option<Rc<RefCell<TN>>>;
fn insert(&mut self, key: i64);
fn delete(&mut self, key: i64);
fn print_tree(&self);
fn get_height(&self) -> u32;
fn get_min(&self) -> Option<i64>;
fn get_max(&self) -> Option<i64>;
fn count_leaves(&self) -> u32;
fn count_nodes(&self) -> u32;
fn contain(&self, key: i64) -> bool;
fn is_empty(&self) -> bool;
fn search(&self, key: i64) -> (bool, Option<Rc<RefCell<TN>>>);
fn in_order_traversal(&self) -> Vec<i64>;
fn pre_order_traversal(&self) -> Vec<i64>;
fn post_order_traversal(&self) -> Vec<i64>;
```

The usage of the crate is very simple. One just has to import libraries by using use keyword:

```
use tree::avlnode::AVLNode;
use tree::avltree::AVLTree;
use tree::rbtree::RBTree;
use tree::tree::Tree;
use tree::node::Node;
use tree::rbnode::RBNode;
```

After that, by using new function, one can create an instance of the tree:

```
let tree = AVLTree::new();
```

Now, one can call any desired function to this instance, for example:

```
tree.insert(key);
tree.delete(key);
tree.count_leaves();
tree.get_height();
tree.in_order_traversal();
tree.pre_order_traversal();
tree.post_order_traversal();
tree.is_empty();
tree.print_tree();
tree.get_max();
tree.get_min();
tree.contain(key);
tree.count_nodes();
```

Bench Marking

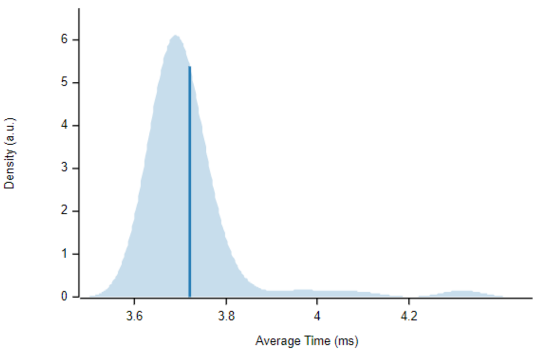
Results

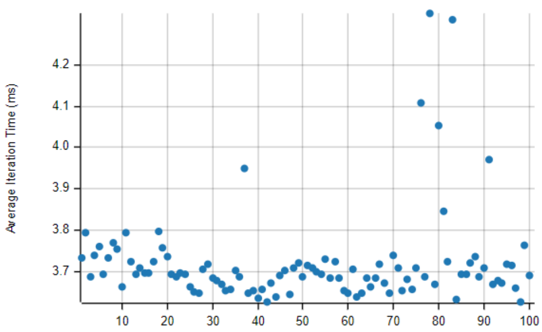
We have two data structures to test, red-black and AVL tree. We measure the insertion and searching performance with 5 different tree sizes (10000, 40000, 70000, 100000, 130000). All of the results can be found in bench_results folder.

In all the following results, the plot on the left displays the average time per iteration for this benchmark. The shaded region shows the estimated probability of an iteration taking a certain amount of time, while the line shows the mean. Click on the plot for a larger view showing the outliers. The plot on the right shows the average time per iteration for the samples. Each point represents one sample.

Red Black Tree Performance Result (insert)

rbtree_tests/insert:10000





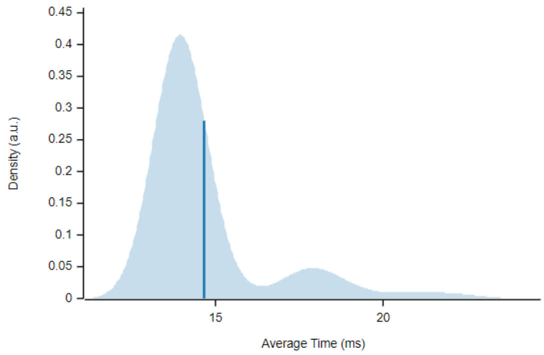
Additional Statistics:

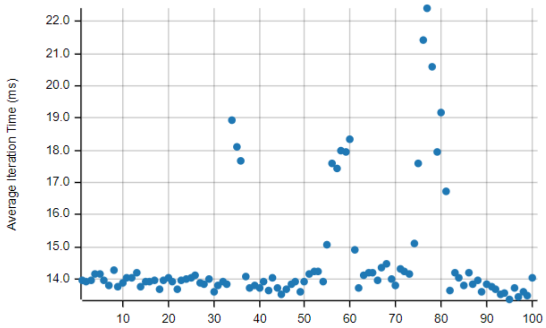
	Lower bound	Estimate	Upper bound
R^2	0.0006902	0.0007120	0.0006804
Mean	3.7012 ms	3.7214 ms	3.7459 ms
Std. Dev.	62.697 µs	114.33 µs	156.52 µs
Median	3.6881 ms	3.6941 ms	3.7048 ms
MAD	30.605 µs	42.097 µs	52.458 µs

Additional Plots:

- Typical
- Mean
- Std. Dev.
- Median
- MAD

rbtree_tests/insert:40000





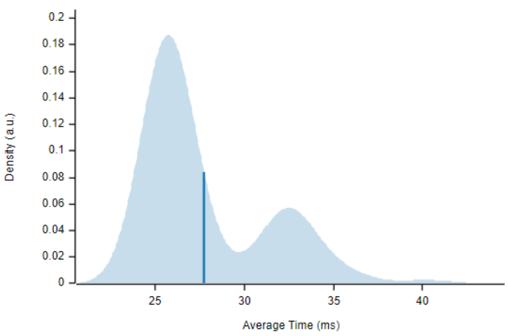
Additional Statistics:

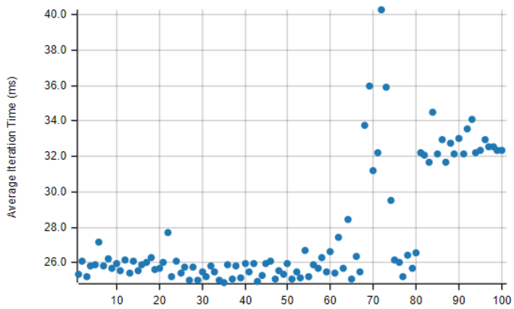
	Lower bound	Estimate	Upper bound
R^2	0.0011565	0.0011960	0.0011467
Mean	14.327 ms	14.661 ms	15.035 ms
Std. Dev.	1.2888 ms	1.8139 ms	2.2534 ms
Median	13.923 ms	13.971 ms	14.053 ms
MAD	231.44 µs	328.51 µs	399.22 µs

Additional Plots:

- Typical
- Mean
- Std. Dev.
- Median
- MAD

rbtree_tests/insert:70000





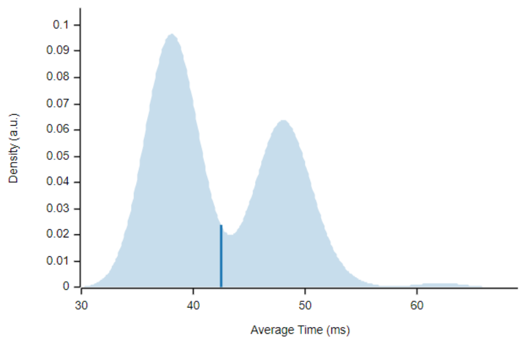
Additional Statistics:

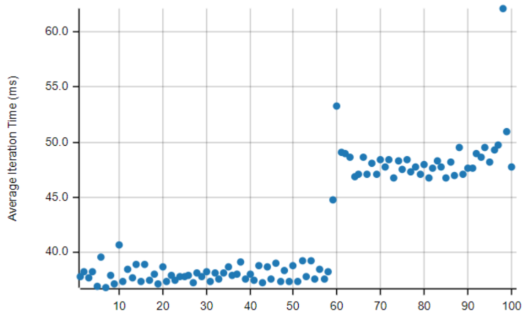
	Lower bound	Estimate	Upper bound
R^2	0.0174539	0.0180637	0.0173671
Mean	27.095 ms	27.737 ms	28.424 ms
Std. Dev.	2.8384 ms	3.4192 ms	3.9484 ms
Median	25.798 ms	25.958 ms	26.166 ms
MAD	643.80 µs	952.71 µs	1.4126 ms

Additional Plots:

- Typical
- Mean
- Std. Dev.
- Median
- MAD

rbtree_tests/insert:100000





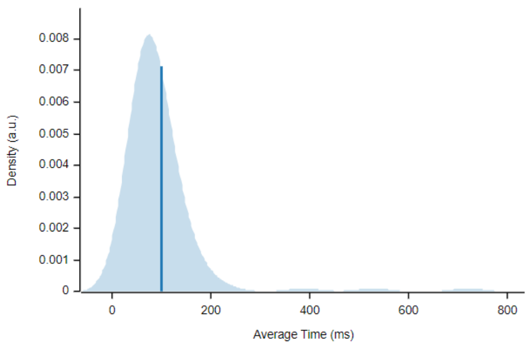
Additional Statistics:

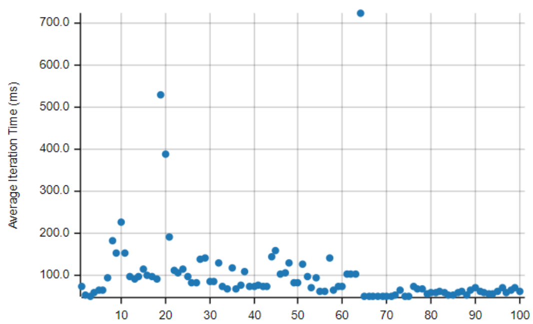
	Lower bound	Estimate	Upper bound
R^2	0.0094639	0.0098153	0.0094456
Mean	41.394 ms	42.441 ms	43.516 ms
Std. Dev.	4.8188 ms	5.4334 ms	6.1543 ms
Median	38.304 ms	38.871 ms	46.779 ms
MAD	1.2968 ms	2.2004 ms	7.6497 ms

Additional Plots:

- Typical
- Mean
- Std. Dev.
- Median
- MAD

rbtree_tests/insert:130000





Additional Statistics:

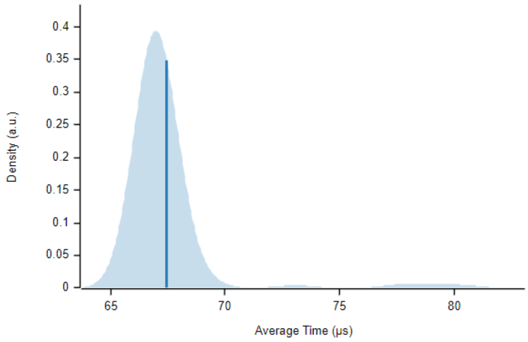
	Lower bound	Estimate	Upper bound
R^2	0.0015971	0.0016431	0.0015685
Mean	83.747 ms	98.851 ms	118.25 ms
Std. Dev.	34.450 ms	89.324 ms	132.54 ms
Median	67.446 ms	73.994 ms	84.196 ms
MAD	18.690 ms	28.079 ms	34.398 ms

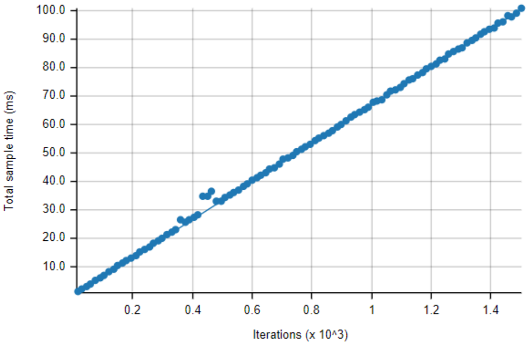
Additional Plots:

- Typical
- Mean
- Std. Dev.
- Median
- MAD

Red Black Tree Performance Result (search)

rbtree_tests/search:10000





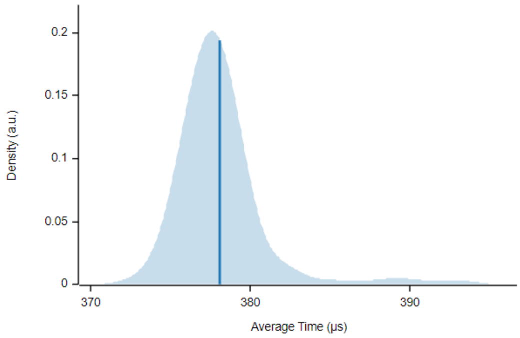
Additional Statistics:

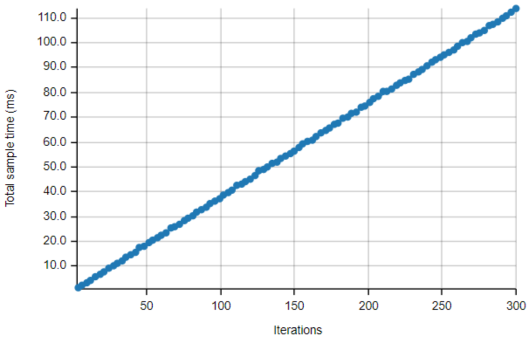
	Lower bound	Estimate	Upper bound
Slope	66.966 µs	67.077 µs	67.221 µs
R^2	0.9625568	0.9629089	0.9623134
Mean	67.083 µs	67.449 µs	67.916 µs
Std. Dev.	719.81 ns	2.1594 µs	3.0999 µs
Median	66.927 µs	66.967 µs	67.059 µs
MAD	192.76 ns	242.33 ns	315.76 ns

Additional Plots:

- Typical
- Mean
- Std. Dev.
- Median
- MAD
- Slope

rbtree_tests/search:40000





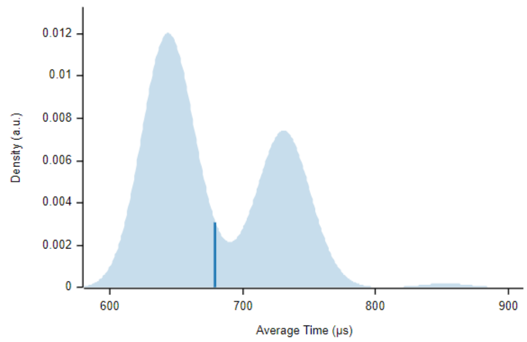
Additional Statistics:

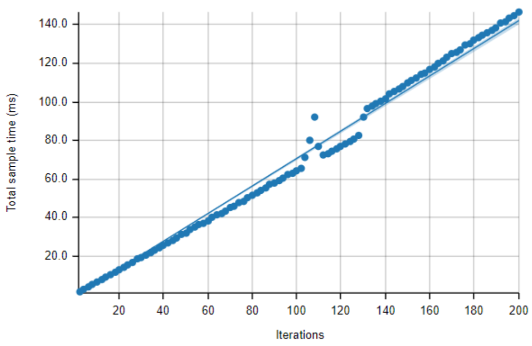
	Lower bound	Estimate	Upper bound
Slope	377.49 µs	377.81 µs	378.14 µs
R^2	0.9978098	0.9979082	0.9978023
Mean	377.61 µs	378.13 µs	378.72 µs
Std. Dev.	1.7760 µs	2.8370 µs	3.7573 µs
Median	377.39 µs	377.71 µs	378.13 µs
MAD	1.2221 µs	1.6312 µs	1.9608 µs

Additional Plots:

- Typical
- Mean
- Std. Dev.
- Median
- MAD
- Slope

rbtree_tests/search:70000





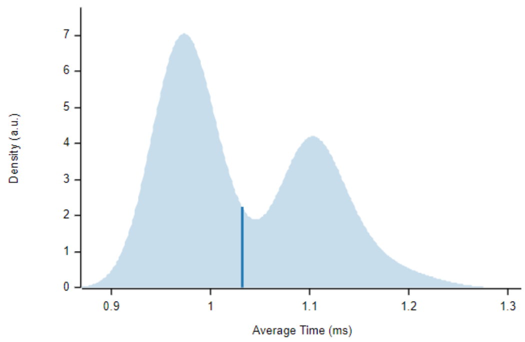
Additional Statistics:

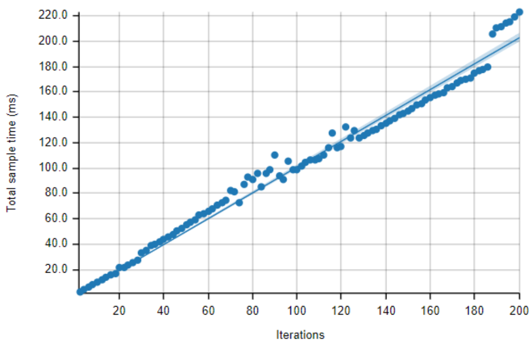
	Lower bound	Estimate	Upper bound
Slope	702.52 µs	711.28 µs	718.37 µs
R^2	0.7216676	0.7312621	0.7249491
Mean	670.63 µs	679.30 µs	688.50 µs
Std. Dev.	40.445 µs	45.897 µs	52.247 µs
Median	643.52 µs	646.21 µs	675.06 µs
MAD	4.3268 µs	8.9959 µs	42.198 µs

Additional Plots:

- Typical
- Mean
- Std. Dev.
- Median
- MAD
- Slope

rbtree_tests/search:100000





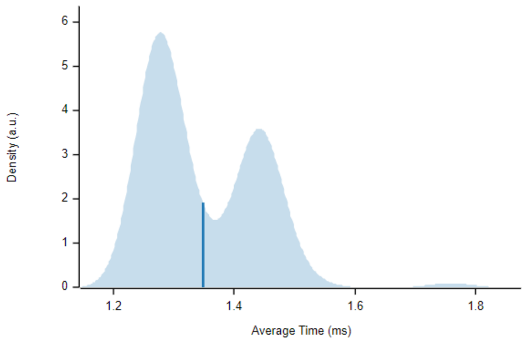
Additional Statistics:

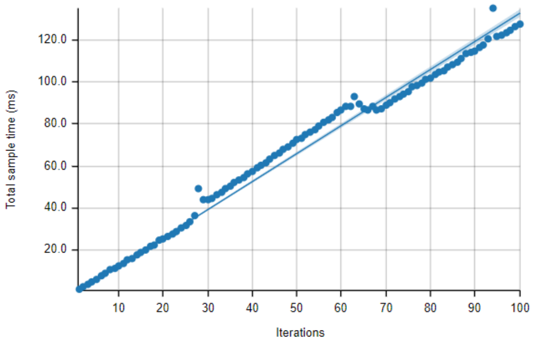
	Lower bound	Estimate	Upper bound
Slope	999.52 µs	1.0157 ms	1.0331 ms
R^2	0.6987811	0.7111855	0.6969353
Mean	1.0182 ms	1.0320 ms	1.0462 ms
Std. Dev.	64.432 µs	71.851 µs	78.390 µs
Median	973.90 µs	988.31 µs	1.0471 ms
MAD	15.199 µs	36.450 µs	96.572 µs

Additional Plots:

- Typical
- Mean
- Std. Dev.
- Median
- MAD
- Slope

rbtree_tests/search:130000





Additional Statistics:

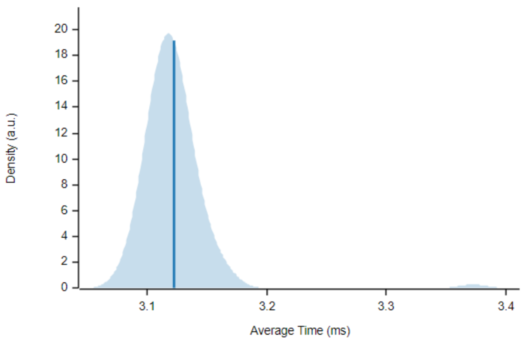
	Lower bound	Estimate	Upper bound
Slope	1.3084 ms	1.3233 ms	1.3414 ms
R^2	0.6367046	0.6453135	0.6327420
Mean	1.3298 ms	1.3472 ms	1.3654 ms
Std. Dev.	76.818 µs	91.267 µs	109.01 µs
Median	1.2772 ms	1.2860 ms	1.3505 ms
MAD	14.126 µs	33.611 µs	117.65 µs

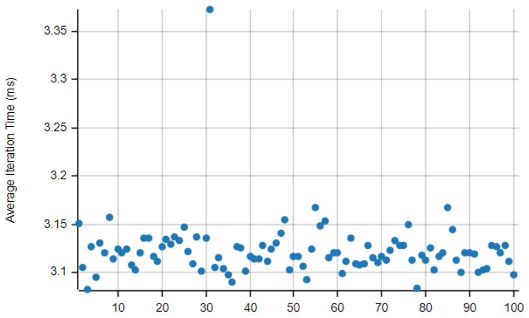
Additional Plots:

- Typical
- Mean
- Std. Dev.
- Median
- MAD
- Slope

AVL Tree Performance Result (insert)

avltree_tests/insert:10000





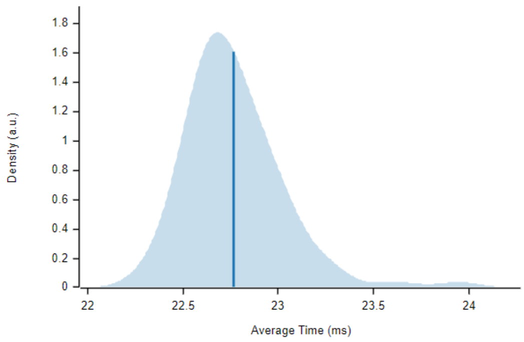
Additional Statistics:

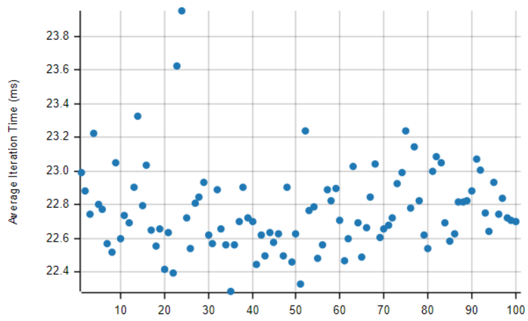
	Lower bound	Estimate	Upper bound
R^2	0.0067950	0.0069789	0.0066569
Mean	3.1176 ms	3.1226 ms	3.1293 ms
Std. Dev.	14.703 µs	30.232 µs	46.816 µs
Median	3.1158 ms	3.1200 ms	3.1230 ms
MAD	11.478 µs	13.903 µs	19.379 µs

Additional Plots:

- Typical
- Mean
- Std. Dev.
- Median
- MAD

avltree_tests/insert:40000





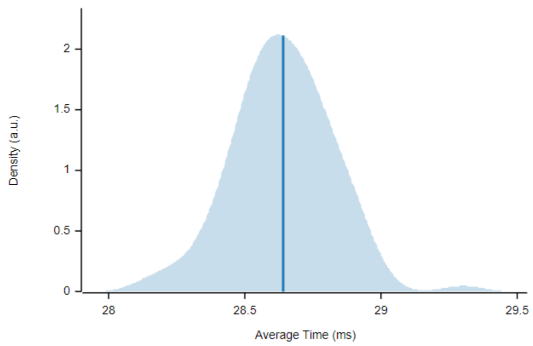
Additional Statistics:

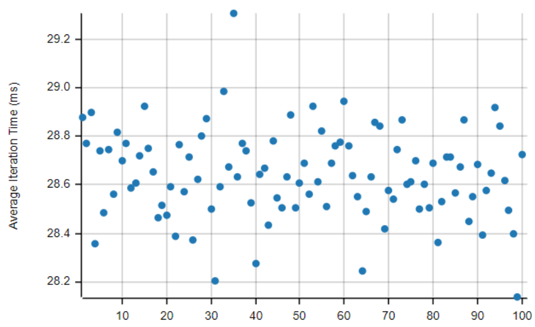
	Lower bound	Estimate	Upper bound
R^2	0.0007825	0.0008104	0.0007780
Mean	22.720 ms	22.768 ms	22.820 ms
Std. Dev.	194.58 µs	254.86 µs	316.05 µs
Median	22.682 ms	22.719 ms	22.788 ms
MAD	154.96 µs	210.72 µs	254.16 µs

Additional Plots:

- Typical
- Mean
- Std. Dev.
- Median
- MAD

avltree_tests/insert:70000





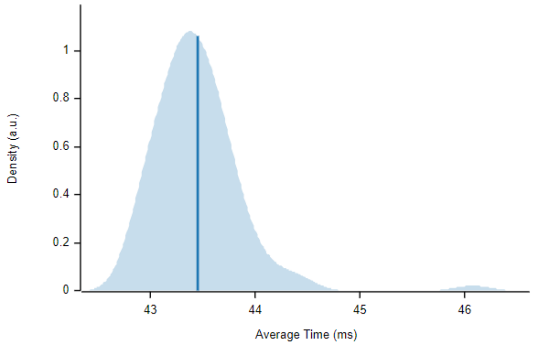
Additional Statistics:

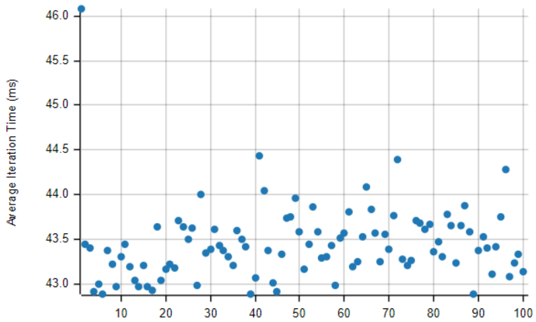
	Lower bound	Estimate	Upper bound
R^2	0.0021293	0.0022098	0.0021275
Mean	28.605 ms	28.641 ms	28.677 ms
Std. Dev.	151.41 µs	182.91 µs	213.76 µs
Median	28.601 ms	28.633 ms	28.689 ms
MAD	134.02 µs	179.95 µs	205.54 µs

Additional Plots:

- Typical
- Mean
- Std. Dev.
- Median
- MAD

avltree_tests/insert:100000





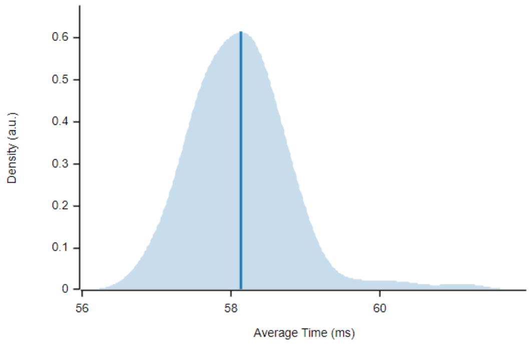
Additional Statistics:

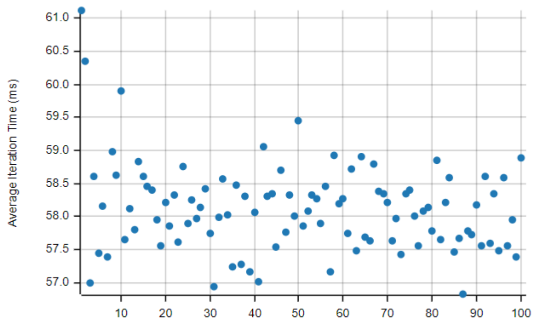
	Lower bound	Estimate	Upper bound
R^2	0.0054521	0.0056349	0.0054020
Mean	43.369 ms	43.446 ms	43.534 ms
Std. Dev.	290.84 µs	424.06 µs	577.28 µs
Median	43.321 ms	43.394 ms	43.474 ms
MAD	252.14 µs	318.46 µs	393.43 µs

Additional Plots:

- Typical
- Mean
- Std. Dev.
- Median
- MAD

avltree_tests/insert:130000





Additional Statistics:

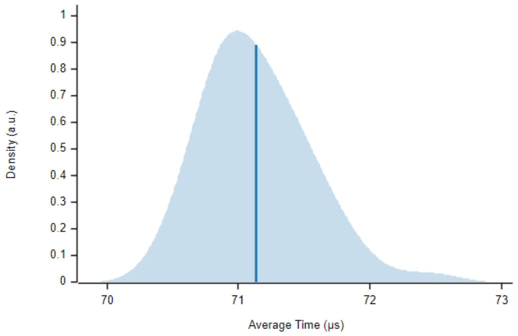
	Lower bound	Estimate	Upper bound
R^2	0.0125307	0.0129792	0.0124790
Mean	57.998 ms	58.124 ms	58.259 ms
Std. Dev.	514.78 µs	669.19 µs	823.10 µs
Median	57.953 ms	58.108 ms	58.249 ms
MAD	449.52 µs	589.09 µs	721.38 µs

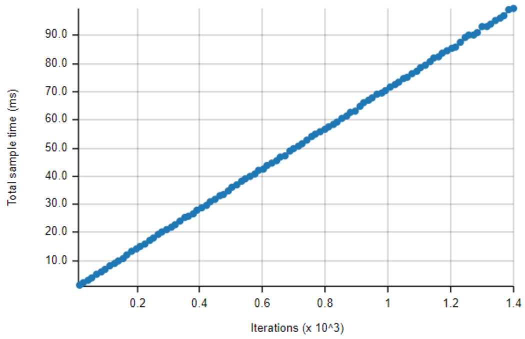
Additional Plots:

- Typical
- Mean
- Std. Dev.
- Median
- MAD

AVL Tree Performance Result (search)

avltree_tests/search:10000





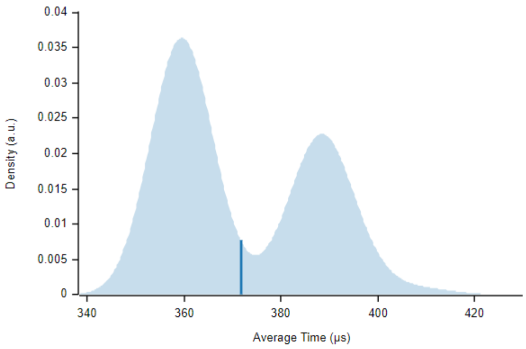
Additional Statistics:

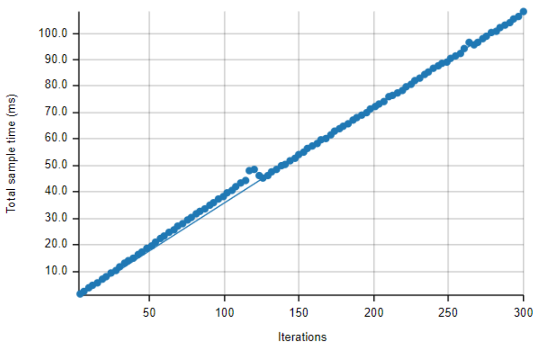
	Lower bound	Estimate	Upper bound
Slope	70.996 µs	71.076 µs	71.160 µs
R^2	0.9971126	0.9972830	0.9970902
Mean	71.058 µs	71.137 µs	71.218 µs
Std. Dev.	342.71 ns	411.06 ns	476.09 ns
Median	70.996 µs	71.091 µs	71.201 µs
MAD	309.64 ns	397.78 ns	478.48 ns

Additional Plots:

- Typical
- Mean
- Std. Dev.
- Median
- MAD
- Slope

avltree_tests/search:40000





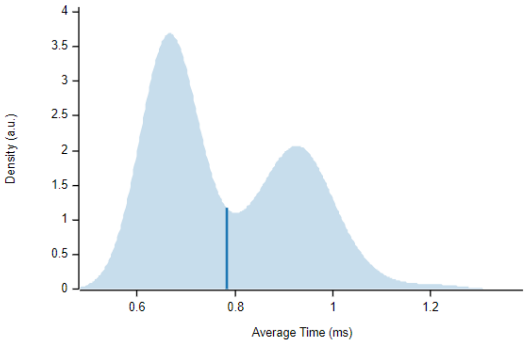
Additional Statistics:

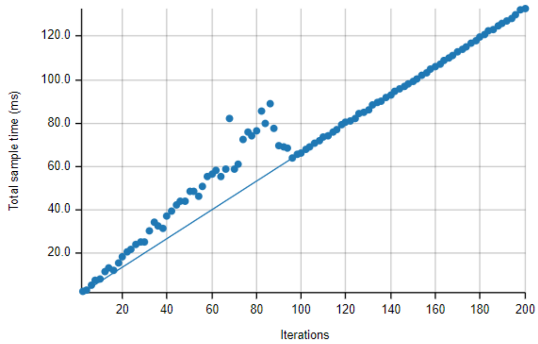
	Lower bound	Estimate	Upper bound
Slope	360.56 µs	361.56 µs	362.88 µs
R^2	0.9299679	0.9309260	0.9292710
Mean	368.78 µs	371.66 µs	374.66 µs
Std. Dev.	13.774 µs	15.030 µs	16.142 µs
Median	359.95 µs	360.66 µs	376.00 µs
MAD	2.2618 µs	3.8966 µs	19.982 µs

Additional Plots:

- Typical
- Mean
- Std. Dev.
- Median
- MAD
- Slope

avltree_tests/search:70000





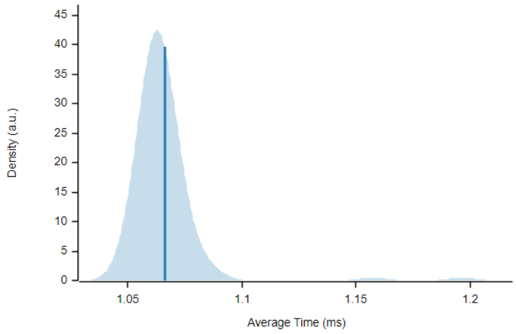
Additional Statistics:

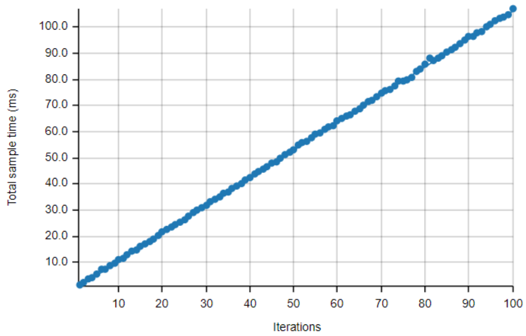
	Lower bound	Estimate	Upper bound
Slope	679.76 µs	689.95 µs	703.65 µs
R^2	0.2712275	0.2741584	0.2689024
Mean	756.16 µs	782.98 µs	810.65 µs
Std. Dev.	124.62 µs	139.73 µs	154.39 µs
Median	665.57 µs	670.01 µs	837.23 µs
MAD	5.9100 µs	14.726 µs	222.44 µs

Additional Plots:

- Typical
- Mean
- Std. Dev.
- Median
- MAD
- Slope

avltree_tests/search:100000





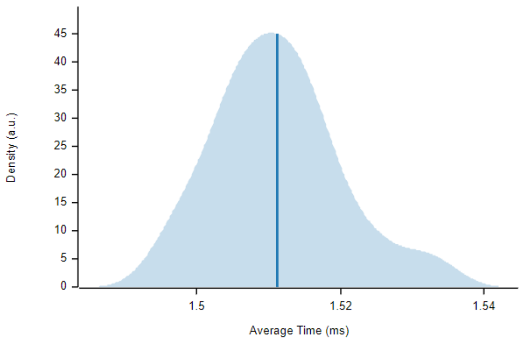
Additional Statistics:

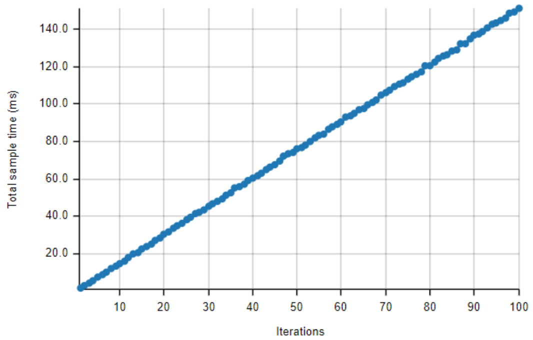
	Lower bound	Estimate	Upper bound
Slope	1.0621 ms	1.0636 ms	1.0651 ms
R^2	0.9956076	0.9958475	0.9955614
Mean	1.0633 ms	1.0661 ms	1.0699 ms
Std. Dev.	5.8417 µs	17.350 µs	26.729 µs
Median	1.0621 ms	1.0631 ms	1.0644 ms
MAD	3.9608 µs	4.9910 µs	6.4632 µs

Additional Plots:

- Typical
- Mean
- Std. Dev.
- Median
- MAD
- Slope

avltree_tests/search:130000





Additional Statistics:

	Lower bound	Estimate	Upper bound
Slope	1.5098 ms	1.5113 ms	1.5128 ms
R^2	0.9973417	0.9974735	0.9973298
Mean	1.5095 ms	1.5112 ms	1.5129 ms
Std. Dev.	7.3235 µs	8.6617 µs	9.8347 µs
Median	1.5085 ms	1.5111 ms	1.5124 ms
MAD	6.1798 µs	8.0943 µs	9.5867 µs

Additional Plots:

- Typical
- Mean
- Std. Dev.
- Median
- MAD
- Slope

For insertion, the mean time of RB tree is [3.7 14.3 27 41.3 82.7], and the mean time of AVL tree is [3.1 22.7 28 43 57]. Since the worst case time complexity of both data structures is O(n), the result makes sense. However, since the average time complexity of RB tree is O(1), we can see that at some tests RB tree has a better performance than AVL tree.

For search, the mean time of RB tree is [67 377 670 1018 1329], and the mean time of AVL tree is [71 368 756 1063 1509]. As you can see, RB tree has a slightly better performance than AVL tree. I think this is because AVL tree is strictly balanced compared to RB tree, and also we search only for the first size/10 items, which means the tree for both of these scenarios is balanced hence the results are close to each other.

About which data structure is more efficient, it really comes down to how we use it. AVL trees are strictly balanced, while red-black trees are more flexible in their balance. So, if we're doing tons of inserts, red-black trees are great. But if we don't insert much and really need top-notch search speed, then AVL trees are the way to go.

About running more tests, there's a lot we haven't explored yet. First, we need to add randomness to data to see how these trees work under unpredictable conditions. Moreover, we like to see how the tree behaves if it's stored on disk instead of memory.

About including more data structures, since we're not sure how the trees will be used, it could be helpful to include a basic option like a regular binary search tree. Sometimes, the extra effort of using AVL or Red-Black trees for adding things might not be worth the possible boost in searching. Adding a simple tree for comparison lets us see if the fancier trees are actually worth it in different situations.