Major Innovations:

One of the major innovations that we implemented in our project is the changes to the Node implementation for both trees. Instead of the default implementation, we made each node into an enumerated class, that is either an instance of Node or Empty. We also implemented the use of generics in our code, which allows the tree to be used to store any sort of data type. We also implemented our command-line interface in a common tree crate, which allows it to be used in both tree implementations without defining it twice. Another major innovation we implemented was the inclusion of random testing. By explicitly defining monkey tests (random numbers to check validity of the tree), we could adequately test our implementation at a scale that would be impossible manually.

Rationale:

Using an enumerated class instead of just the Node struct makes our implementation safer. This prevents programming errors that could occur if we used the basic Node struct. Runtime errors that result from accessing nonexistent Nodes are prevented by using an enum class. This also allows us to write more concise and efficient code that takes advantage of Rust's pattern matching syntax. Using generics in our implementation allows us to write code that can be reused with different types of data. This is important in the context of trees because a Node can store any data type. Generics are also a zero-level abstraction that will maintain performance while increasing code flexibility.

Limitations: None that we know of.

Manual:

Red Black Tree:

Functions:

- `RBTree::default()`: Initializes an empty Red-black tree.
- `insert_node(&mut self, _key: u32)`: Takes an unsigned 32-bit integer as input and inserts a node with that key into the tree. If the root of the tree is empty, a new node is created and assigned to the root. Otherwise, the tree is navigated through until a suitable position for the new node is found.
- `delete_node(&mut self, _key: u32)`: Takes an unsigned 32-bit integer as input and attempts to delete the relevant node from the tree. If the node is not found, the method returns. Otherwise, the function navigates to the correct node and deletes it from the tree.
- `count_leaves(&self) -> u32`: Counts the number of leaves in the tree and returns as an unsigned 32-bit integer.
- `height(&self) -> u32`: Determines the tree's height and returns as an unsigned 32-bit integer.
- `in_order(&self) -> Vec<u32>`: Returns a vector containing the keys of each node using in-order traversal.

- `is_empty(&self) -> bool`: Returns a boolean indicating whether the tree is empty.
- `print_tree(&self)`: Prints the tree out showing the structure and color of each node.

Structs/Enums:
- `RBTree`: One field containing the root node (Tree).
- `RBNode`: Enumerated class with two options: Node (key, color, left, right, parent) and Empty.
- `Color`: Enumerated class with two options: Red and Black.

Type Definitions:
- `Tree`: Typedef for children of a node
- `WeakTree`: Typedef for the parent of a node

AVL Tree:

Functions:
- `AVLTree::default()`: Initializes an empty AVL tree.
- `insert_node(&mut self, _key: u32)`: Takes an unsigned 32-bit integer as input and inserts a node with that key into the tree. If the root of the tree is empty, a new node is created and assigned to the root. Otherwise, the tree is navigated through until a suitable position for the new node is found.
- `delete_node(&mut self, _key: u32)`: Takes an unsigned 32-bit integer as input and attempts to delete the relevant node from the tree. If the node is not found, the method returns. Otherwise, the function navigates to the correct node and deletes it from the tree.
- `count_leaves(&self) -> u32`: Counts the number of leaves in the tree and returns as an unsigned 32-bit integer.
- `height(&self) -> u32`: Determines the tree's height and returns as an unsigned 32-bit integer.
- `in_order(&self) -> Vec<u32>`: Returns a vector containing the keys of each node using in-order traversal.
- `is_empty(&self) -> bool`: Returns a boolean indicating whether the tree is empty.
- `print_tree(&self)`: Prints the tree out showing the structure and color of each node.

Structs/Enums:
- `AVLTree`: One field containing the root node (Tree).
- `AVLNode`: Enumerated class with two options: Node (key, left, right, parent, height) and Empty.

Type Definitions:
- `Tree`: Typedef for children of a node
- `WeakTree`: Typedef for the parent of a node

Command Line Interface:

Each tree is implemented in its own folder. To run the AVL tree implementation, one would first change directories into `avl_tree/`, then enter the command `cargo run` in the terminal. The same can be done for the Red-black tree by entering `cargo run` in `red_black_tree/`. A menu will appear with numbered options as follows:

```
1: Insert a node
2: Delete a node
3: Count the number of leaves
4: Height of the tree
5: Print In-order traversal of the tree
6: Check if the tree is empty
7: Print the tree
8: Quit
```

Upon running the program, an empty AVL tree is initialized. Based on the chosen operation, either the tree is modified or information about the tree is displayed. After every modification (insert/delete), the entire tree is printed. After the menu is printed, the user is prompted to select an option by entering the associated number. For example, if one wants to create an RBTree by inserting the values [20, 30, 40, 50] in order then, deleting [30], they would execute the following commands:

```
...
1 (insert)
Enter the value of the node: 20
...
1 (insert)
Enter the value of the node: 30
...
1 (insert)
Enter the value of the node: 40
...
1 (insert)
Enter the value of the node: 50
...
2 (delete)
Enter the value of the node: 30
...
3 (count leaves)
Number of leaves: 2
...
4 (height)
Height of the tree: 2
...
```

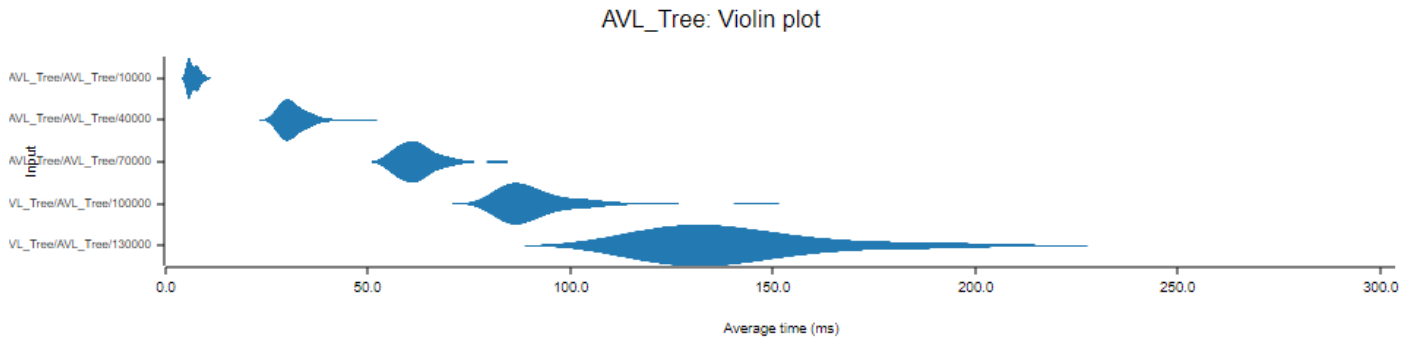
```
5 (in-order)
In-order traversal of the tree: [20, 40, 50]
```

Benchmark Results:

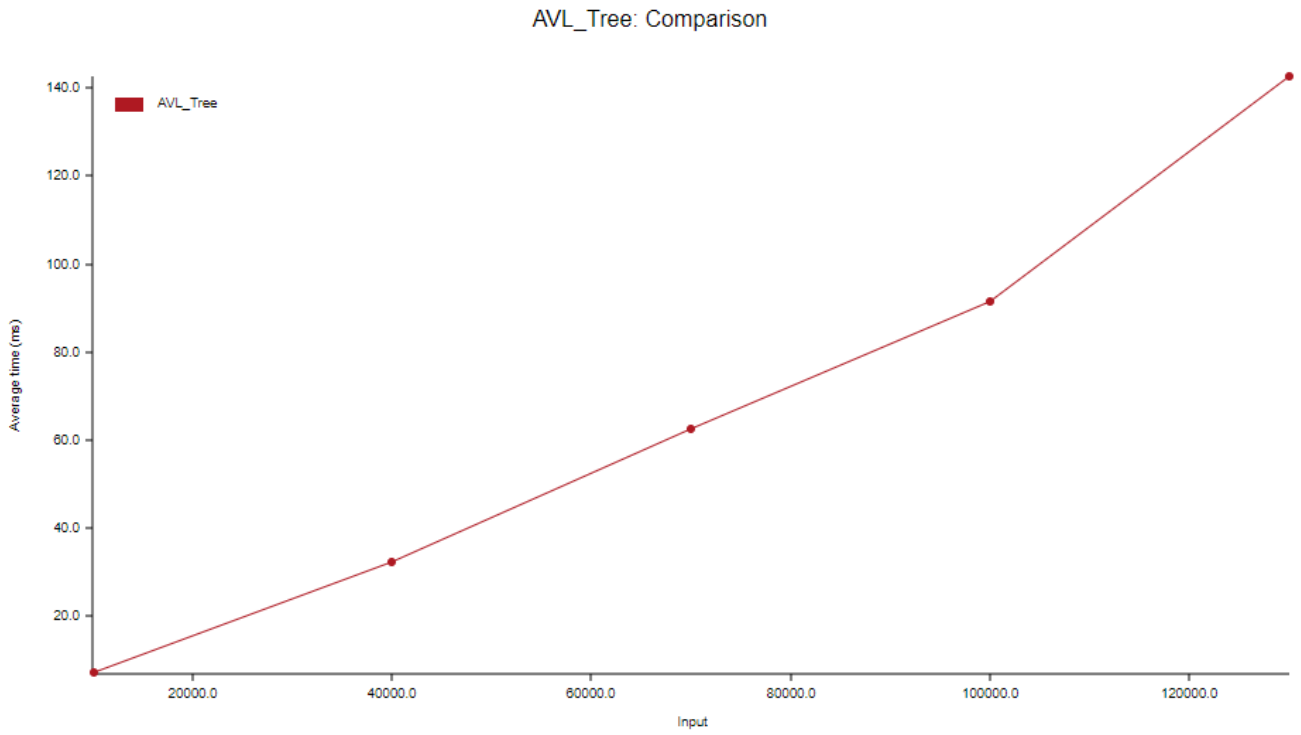
The following charts and graphs display the benchmark results for insert and search operations for both trees. We tested the worst-case insert (sequential: 1, 2, ..) and the standard search operation on the lowest 10% of values. A violin plot shows the average time taken to complete the benchmark operation, divided by test case. The thickness of each section indicates the percentage of samples that finished execution in that time interval. The following line graph simply shows the mean execution time against the size of input. Each following chart is specific to a test case (tree size), and shows a distribution as well as scatter plot indicating the execution times of the 100 samples.

AVL Tree:

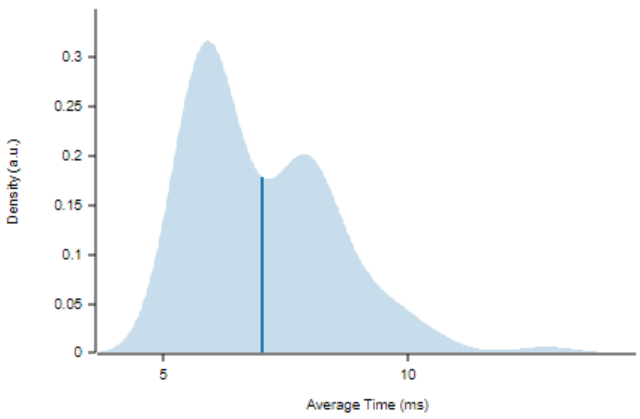
Violin Plot

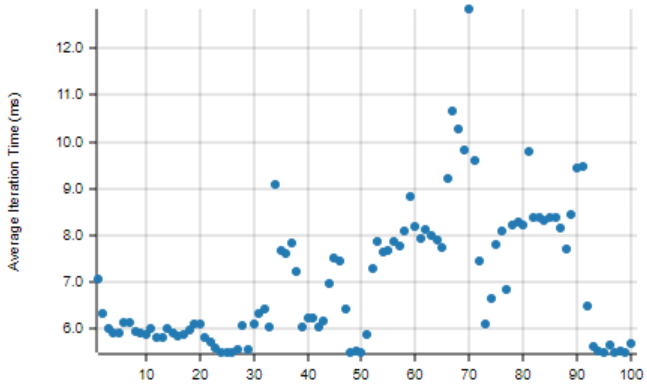


Line Chart

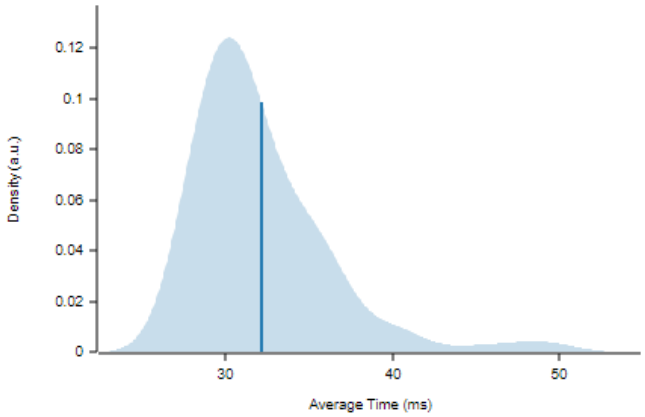


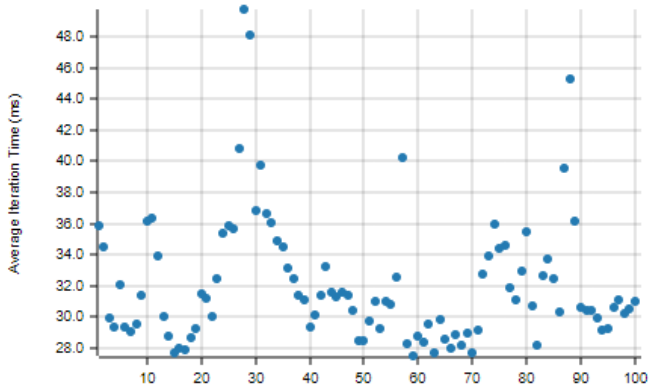
AVL_Tree/AVL_Tree/10000



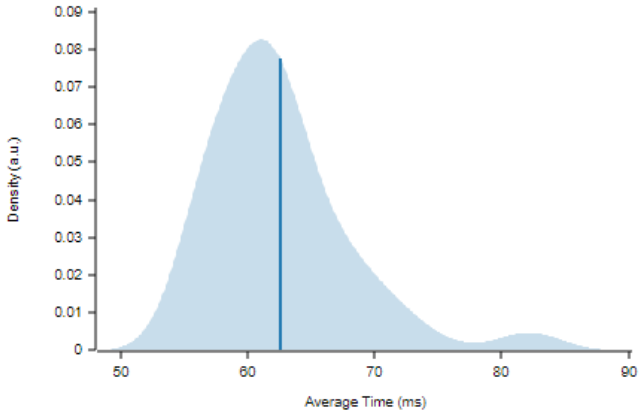


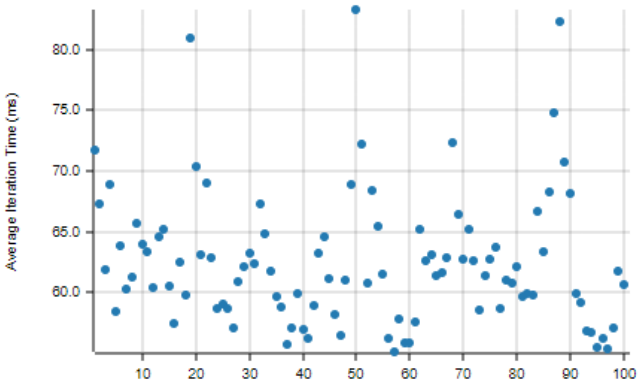
AVL_Tree/AVL_Tree/40000



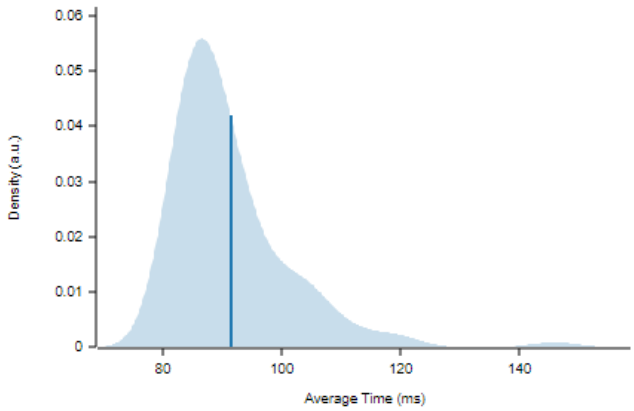


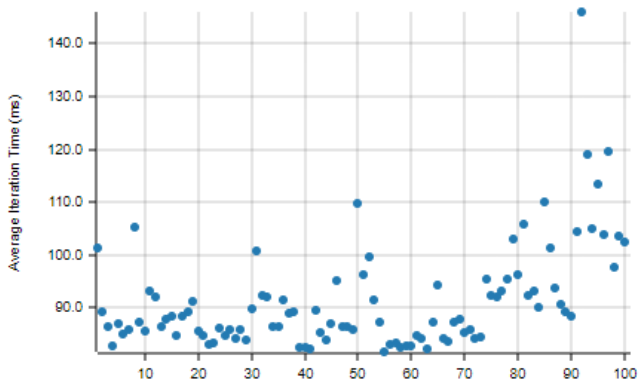
AVL_Tree/AVL_Tree/70000



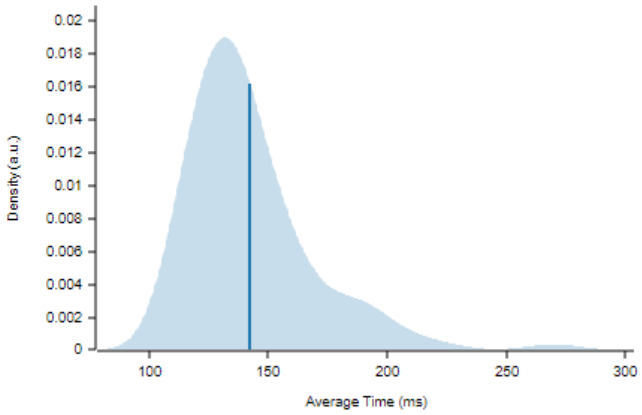


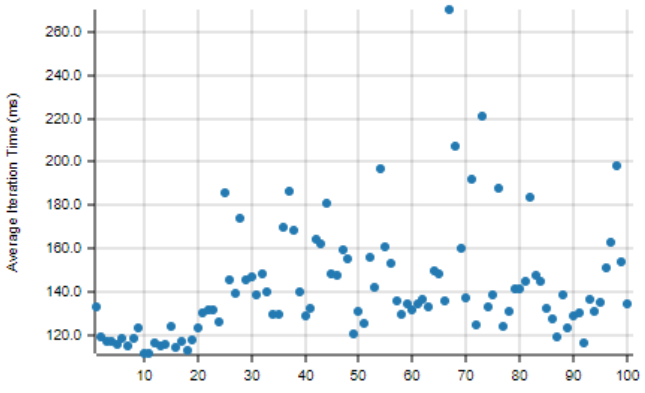
AVL_Tree/AVL_Tree/100000





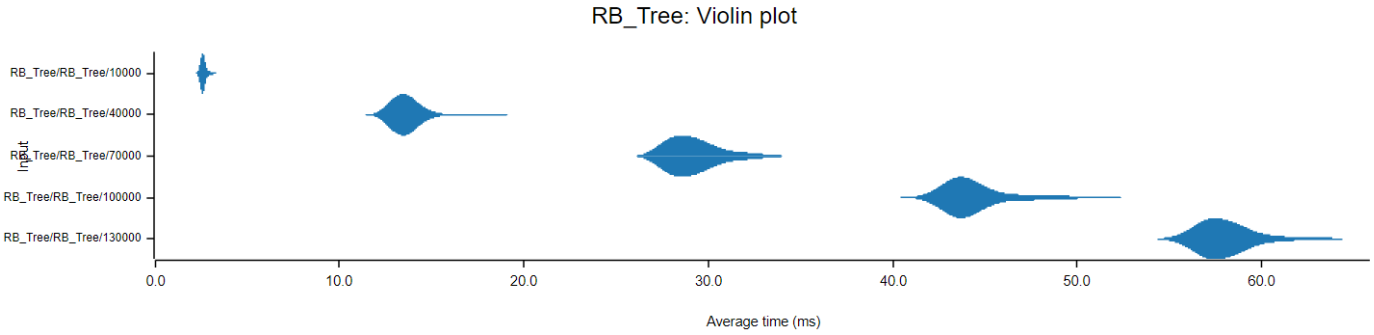
AVL_Tree/AVL_Tree/130000

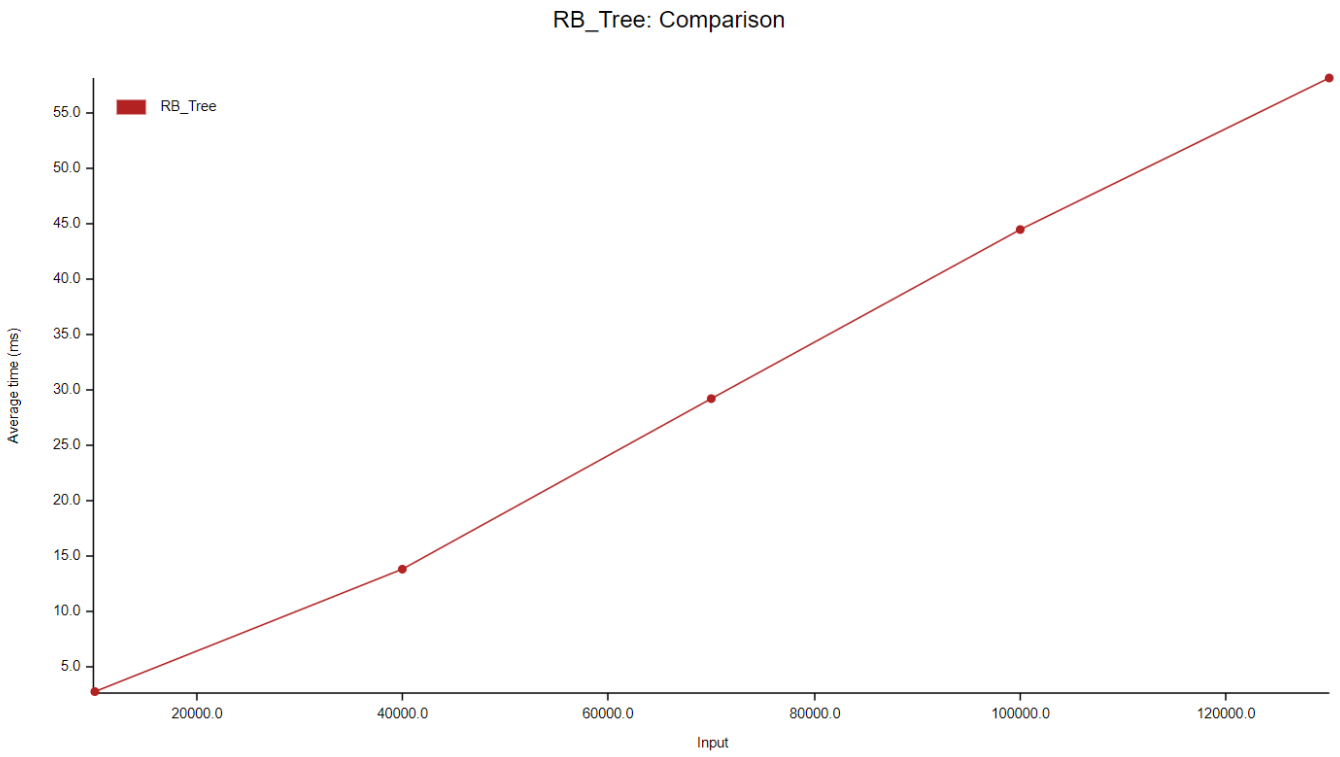




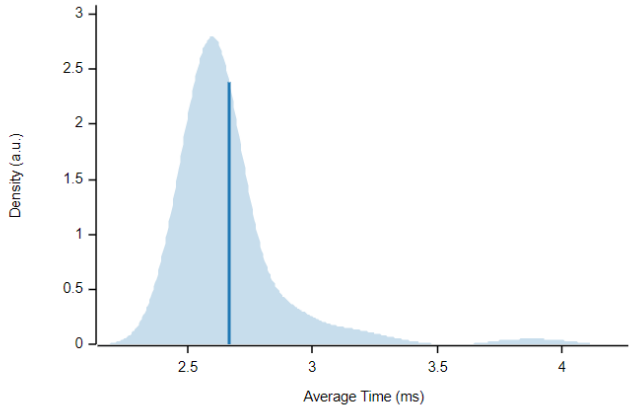
Red-Black Tree:

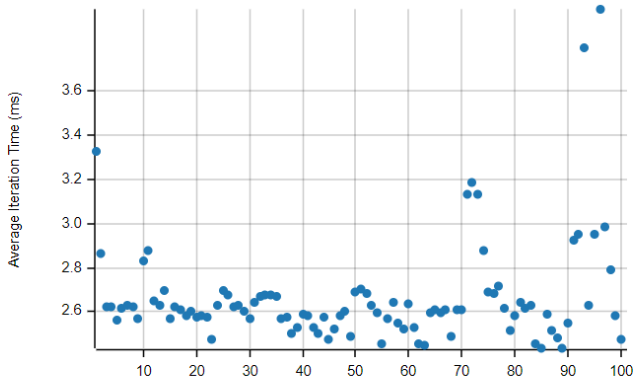
Violin Plot



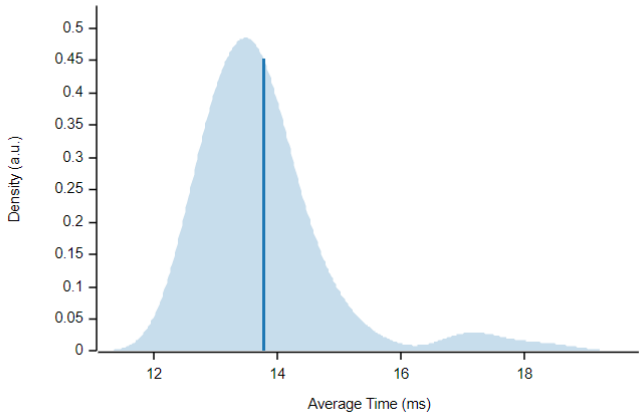


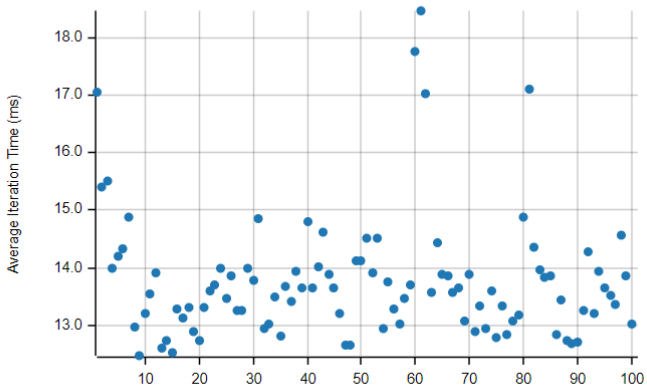
RB_Tree/RB_Tree/10000



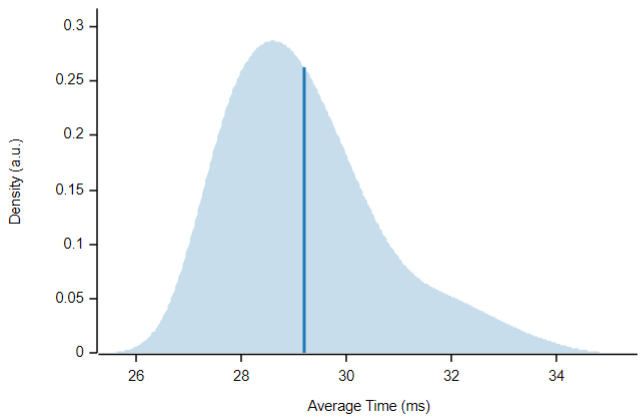


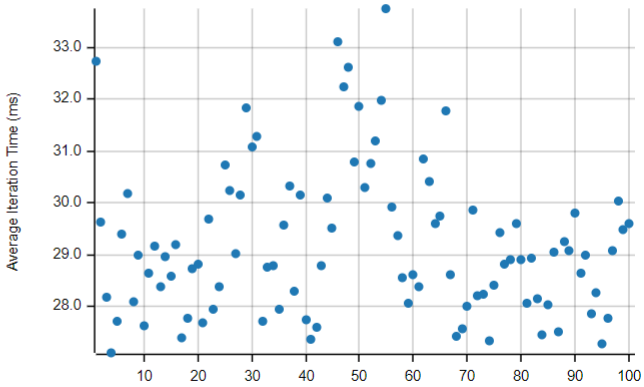
RB_Tree/RB_Tree/40000



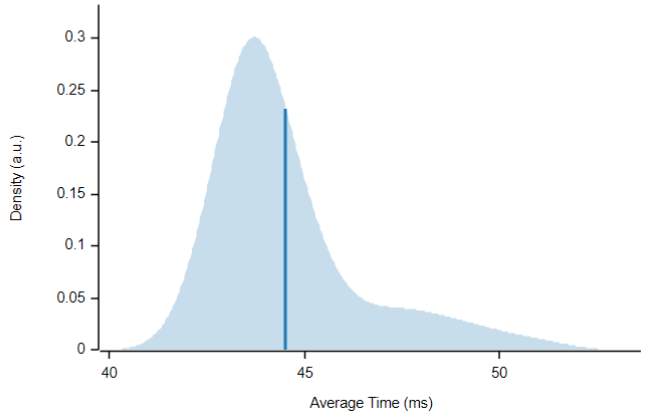


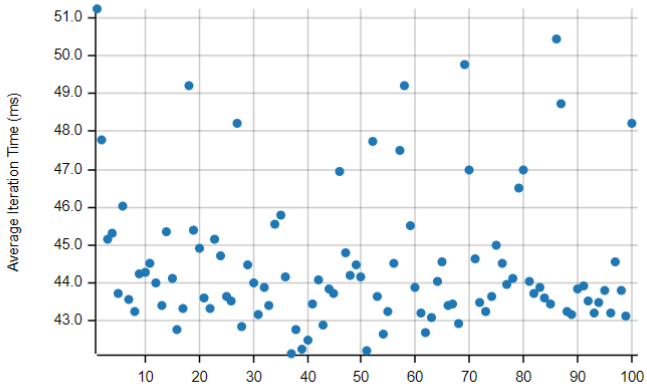
RB_Tree/RB_Tree/70000



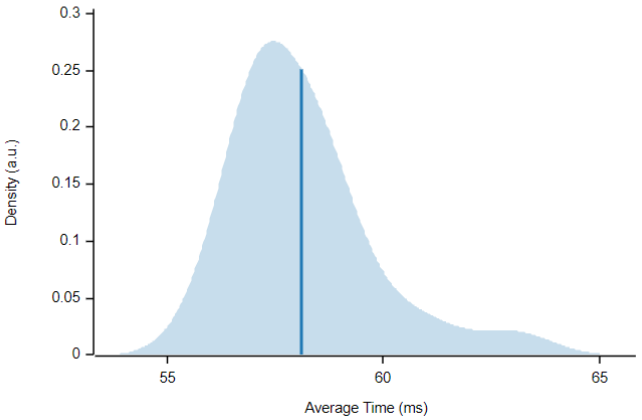


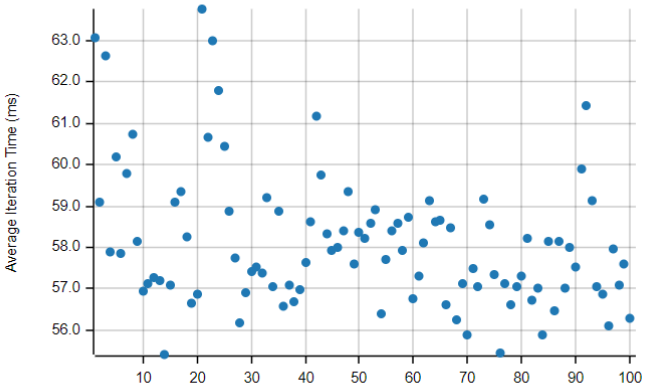
RB_Tree/RB_Tree/100000





RB_Tree/RB_Tree/130000





Comparison:

Both trees follow a linear relationship between the input size and the execution time. It is clear from the results that the red-black tree performs significantly better than the AVL tree in inserting and searching. On average, the red-black tree completes execution in less than half the time for the same input tree size. For example, for a tree size of 100000, the AVL tree's average execution time is approximately 90 ms, while the red-black tree's time is 45 ms.

It might be useful to benchmark both trees against a simple binary search tree. This will allow us to see the benefits of using a more specialized tree (red-black, AVL) against the basic version (BST). Our current benchmarking protocol allows us to compare the two specialized trees but does not compare against the base case. Worst-case, with no balancing, a binary search tree will have an insert and search time complexity of O(n). With the worst-case inputs we are using to benchmark the two trees, it is not feasible to test a binary search tree as it will occupy significant CPU resources for an extended period of time. To further examine the relationship between the two trees, other operations such as deletion can be benchmarked. It is also possible to test average case time complexity to see how both trees would perform in a normal working environment.

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

A red-black tree guarantees a logarithmic time complexity (O(log n)) for major operations such as search, insertion, and deletion. The worst case time complexity for a simple binary search tree is linear (O(n)) if the tree is unbalanced. A red-black tree also is self-balancing as it rebalances itself after every insertion and deletion. This minimizes the number of rotations needed to keep balance.

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.

At the most basic level, both trees are built off a binary search tree, meaning each node has at most two child nodes. Both trees also need rotations to maintain balance after each insertion and deletion. The time complexity of most operations is also the same for both trees. These similarities allowed us to share a significant amount of code between the two implementations.

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?

Our implementation can be adapted to accommodate 2-3-4 trees and B trees with some challenges. 2-3-4 and B trees all can have more than 2 children per node, which contradicts many assumptions made in our implementation. This would require extensive changes to the code, as most operations (search, delete, insert) depend on a simple value comparison. For example, to search for a node, one needs to simply compare the value against the current root. This will indicate whether the node has been reached, or which

subtree to search next (left: less than, right: greater than). In 2-3-4 and B trees, this comparison will not be sufficient. One would need to compare against each value in the node to determine the course of action, which is more expensive than our current implementation. There are also many specific cases to 2-3-4 and B trees that do not apply for Red-black and AVL trees. These additional cases would need to be applied in our implementation, making changes more involved and specific.