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

Our project is dedicated to developing Rust libraries for Red-black trees and AVL trees, two self-balancing binary trees renowned for their efficiency in data structure operations. These trees maintain balance through rotations and recoloring (for Red-black trees) or by ensuring minimal height differences between branches (for AVL trees). Our goals include providing easy-to-use libraries with comprehensive documentation, focusing on simplicity, efficiency, and correctness, to empower developers to build scalable and high-performance software solutions.

First	Last	ccid
Adib	Nasreddine	nasreddi
Dhruv	Kumar	dkumar2
Mahmud	Jamal	majamal
Nam Son	Pham	npham

2024-03-20

Major Innovations

No major innovations were made outside of the project specifications.

Rationale

1. What does a red-black tree provide that cannot be accomplished with ordinary binary search trees?
   Red-black trees are a specialized form of self-balancing binary search trees that offer significant improvements over ordinary binary search trees, especially in ensuring better performance guarantees for insertion, deletion, and lookup operations. Unlike ordinary binary search trees, which can degrade to linked lists in the worst case—leading to linear time complexity for basic operations—red-black trees maintain a balanced structure, ensuring operations are performed in logarithmic time (O(logn)). This is achieved through a set of rules and rotations that keep the tree balanced, thus providing guaranteed logarithmic time complexity and improved worst-case performance. The self-balancing property of red-black trees comes from efficient rebalancing operations, which are simpler than those required for more strictly balanced trees like AVL trees, making them a good compromise between maintaining balance and minimizing overhead. The use of coloring nodes red or black and adhering to specific properties ensures the tree remains balanced, avoiding extensive restructuring. Therefore, red-black trees are particularly valuable in applications where predictable performance is crucial, offering a balanced binary search tree structure that efficiently supports dynamic set operations.

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

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

   For our implementation, we only used options to provide error handling on the return functions which would return None if nodes to delete were not found or if insertion was not successful. However, it is entirely posible to add Results for handling of run time error panics, along with type and format checking for input to handle uncontrolled behavior upon unexpected input.

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.

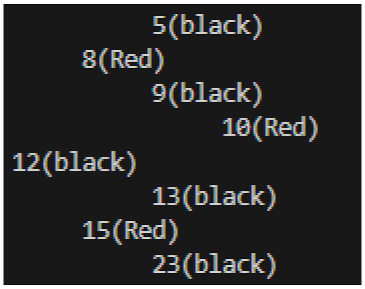
   Both the Red&Black tree and AVL tree are very similar in that both are binary trees with special properties. Thus any functionality which is inherent to a binary tree is reflected in both. Some of these functions would be, traversal, leaf count, and search. Where they differ are the way they are organized upon insertion and deletion which give them their respective logarithmic complexities. Red&Black trees have a better worst case search time than AVL trees, while AVL trees have a better search time on average.

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?

   A way to achieve this would be to have these different trees be under a same enum called binary tree, this will allow generalization of the tree types and which will make it so all the main binary tree functionalities can be inherited by the newer trees while adding their own specific organization techniques.

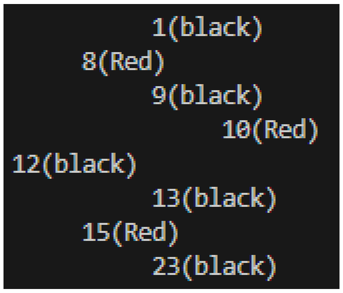
Limitations

Occasionally, the deletion operation in a Red-Black Tree may result in an incorrect output. To illustrate, consider a sequence where we insert the numbers 12, 8, 15, 5, 9, 13, 19, 10, and 23 into the tree. If we then proceed to delete the number 19, the tree correctly updates and displays as follows:



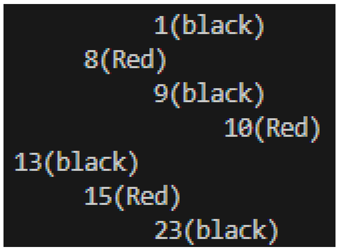
```
          5(black)
     8(Red)
          9(black)
               10(Red)
12(black)
          13(black)
     15(Red)
          23(black)
```

Continuing this process, we insert the number 1 and delete the number 5 from the tree, which still yields a correct output as shown below:



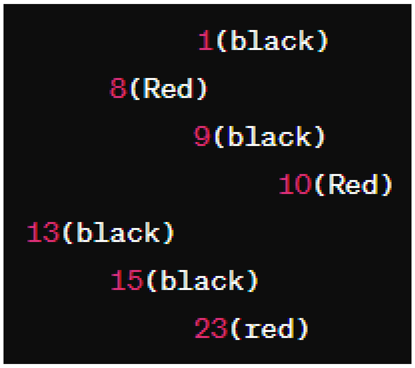
```
          1(black)
     8(Red)
          9(black)
               10(Red)
12(black)
          13(black)
     15(Red)
          23(black)
```

However, an issue arises when we attempt to delete 12 from the tree. The operation leads to an incorrect output:



```
          1(black)
     8(Red)
          9(black)
               10(Red)
13(black)
     15(Red)
          23(black)
```

This output needs to be corrected. The correct configuration of the tree after deleting the number 12 should be:



```
          1(black)
     8(Red)
          9(black)
               10(Red)
13(black)
     15(black)
          23(red)
```

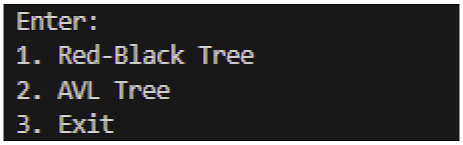
The insertion speed for the AVL tree is significantly slower than the Red-Black tree. It should only have a marginally slower insertion speed than the Red-Black tree due to the higher number of rotations required to balance.

User Manual

The program may be invoked using *$cargo run* while inside the project domain, this will then display a main menu as showcased below.

Main Menu

After starting the program, you will be presented with the main menu, which allows you to select between working with a Red-Black Tree, an AVL Tree, or exiting the program. The options are:

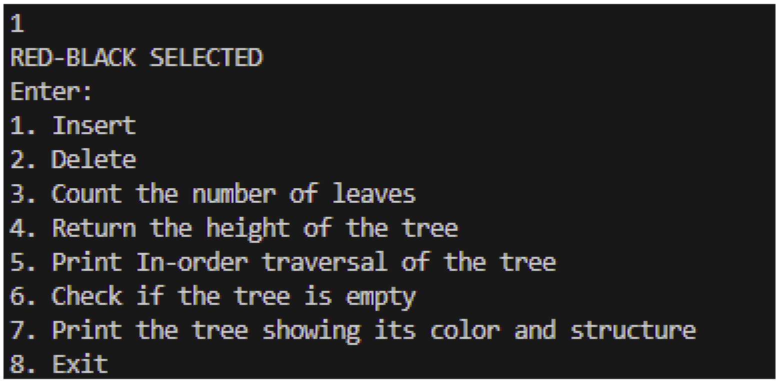


```
Enter:
1. Red-Black Tree
2. AVL Tree
3. Exit
```

Enter 1 to interact with a Red-Black Tree, 2 for an AVL Tree, or 3 to exit the program.

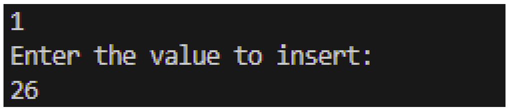
Red-Black Tree Operations

If you choose to work with a Red-Black Tree, you will see a new menu with the following options:



```
1
RED-BLACK SELECTED
Enter:
1. Insert
2. Delete
3. Count the number of leaves
4. Return the height of the tree
5. Print In-order traversal of the tree
6. Check if the tree is empty
7. Print the tree showing its color and structure
8. Exit
```

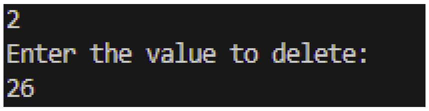
1. Insert
   - Enter 1 to insert a new value into the tree.
   - You will be prompted to enter the value you wish to insert.
   - After insertion, you will return to the Red-Black Tree menu.



```
1
Enter the value to insert:
26
```

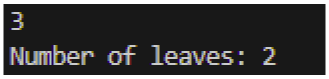
2. Delete
   - Enter 2 to delete a value from the tree.
   - You will be prompted to enter the value you wish to delete.

   - After deletion, you will return to the Red-Black Tree menu.



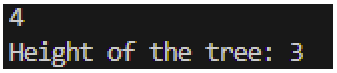
```
2
Enter the value to delete:
26
```

3. Count the Number of Leaves
   - Enter 3 to display the number of leaves in the tree.
   - The count will be displayed, and you will return to the Red-Black Tree menu.



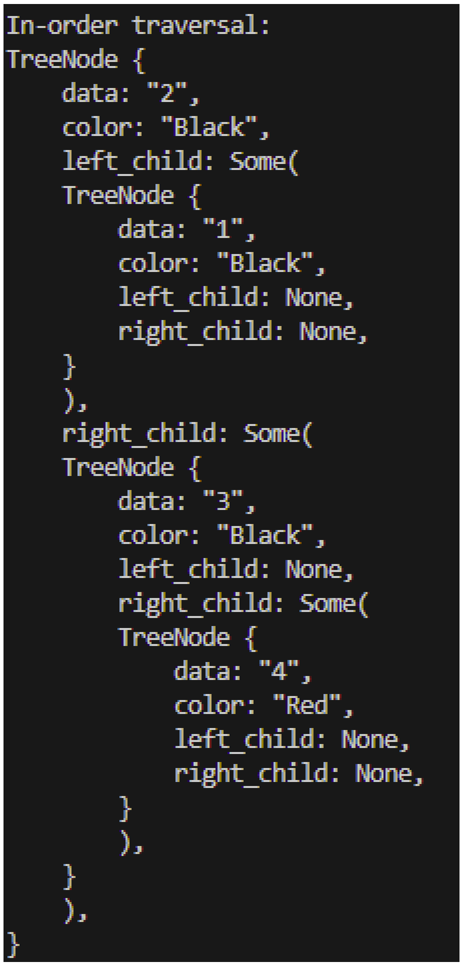
```
3
Number of leaves: 2
```

4. Return the Height of the Tree
   - Enter 4 to display the height of the tree.
   - The height will be displayed, and you will return to the Red-Black Tree menu.



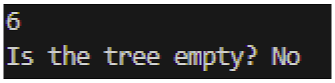
```
4
Height of the tree: 3
```

5. Print In-Order Traversal of the Tree
   - Enter 5 to print an in-order traversal of the tree.
   - The traversal will be displayed, and you will return to the Red-Black Tree menu.

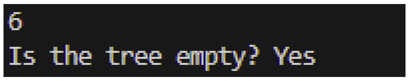


```
In-order traversal:
TreeNode {
    data: "2",
    color: "Black",
    left_child: Some(
    TreeNode {
        data: "1",
        color: "Black",
        left_child: None,
        right_child: None,
    }
    ),
    right_child: Some(
    TreeNode {
        data: "3",
        color: "Black",
        left_child: None,
        right_child: Some(
        TreeNode {
            data: "4",
            color: "Red",
            left_child: None,
            right_child: None,
        }
        ),
    }
    ),
}
```

6. Check if the Tree is Empty
    - Enter 6 to check if the tree is empty.
    - The result will be displayed as "Yes" or "No", and you will return to the Red-Black Tree menu.

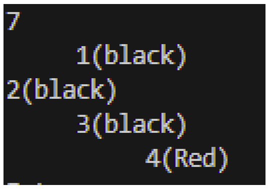


```
6
Is the tree empty? No
```



```
6
Is the tree empty? Yes
```

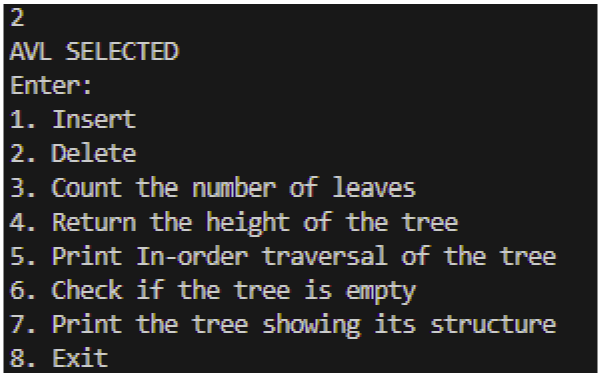
7. Print the Tree Showing Its Color and Structure
    - Enter 7 to print the tree along with its color and structure. The tree structure will be displayed, and you will return to the Red-Black Tree menu.



```
7
     1(black)
2(black)
     3(black)
          4(Red)
```

8. Exit
    - Enter 8 to exit the Red-Black Tree menu and return to the main menu.

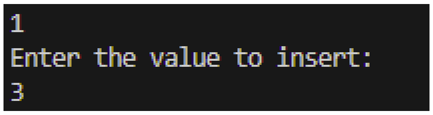
AVL Tree Operations



```
2
AVL SELECTED
Enter:
1. Insert
2. Delete
3. Count the number of leaves
4. Return the height of the tree
5. Print In-order traversal of the tree
6. Check if the tree is empty
7. Print the tree showing its structure
8. Exit
```

The AVL tree offer's an instruction set similar to that of the red and black tree, with different implementation to the insertion, deletion and balancing.

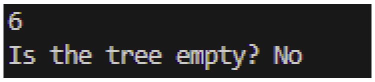
1. Insert



```
1
Enter the value to insert:
3
```

- Choosing the insert option allows for you to enter a number value into the tree
- If it is the first value inserted into a tree it becomes the root
- Value 3 is inserted into the tree

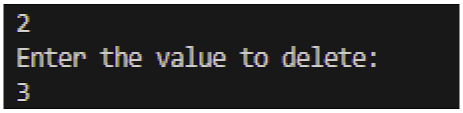
2. Check if the tree is empty



```
6
Is the tree empty? No
```

- Choosing this option returns whether the status of the tree is empty or not
- Since value 3 is inserted the tree is not empty

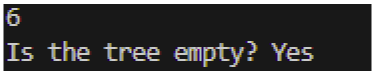
3. Delete



```
2
Enter the value to delete:
3
```

- Choosing delete will delete the given value from the AVL tree if it exists
- We can delete our inserted value 3 from the tree with this command

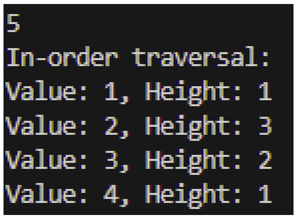
4. Check if the tree is empty



```
6
Is the tree empty? Yes
```

- Since we deleted the only value in the tree it is empty once again

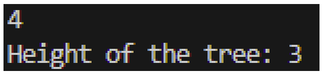
5. Print in-order traversal of the tree



```
5
In-order traversal:
Value: 1, Height: 1
Value: 2, Height: 3
Value: 3, Height: 2
Value: 4, Height: 1
```

- Prints the values and heights of each node from least to greatest value within the tree by visiting each node from the very left to the right
- In this case we have inserted the values 1, 2, 3 and 4 in order into the tree

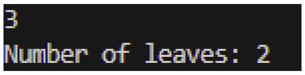
6. Return the height of the tree



```
4
Height of the tree: 3
```

- Returns the height of the tree from the root node to the lowest leaf node
- The height of the tree by inserting values from (5) is 3

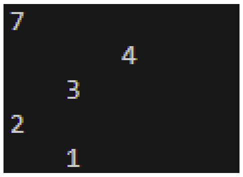
7. Count the number of leaves



```
3
Number of leaves: 2
```

- Returns the number of leaf nodes in the tree
- In the case from (5), we have 2 leaf nodes

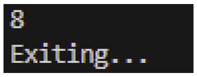
8. Print the tree showing it’s structure



```
7
        4
    3
2
    1
```

- Shows the structure of the tree from left to right, the left most value being the root. Each horizontal spacing defines the level of the tree node, left being higher than right.
- In the case from (5), we see that the root node is 2, and the two leaf nodes are 4 and 1, with the total height of the tree being 3.

9. Exit



```
8
Exiting...
```

- This command will allow you to exit from the program back to the terminal

Video

Project 2 Video.mp4

Testing

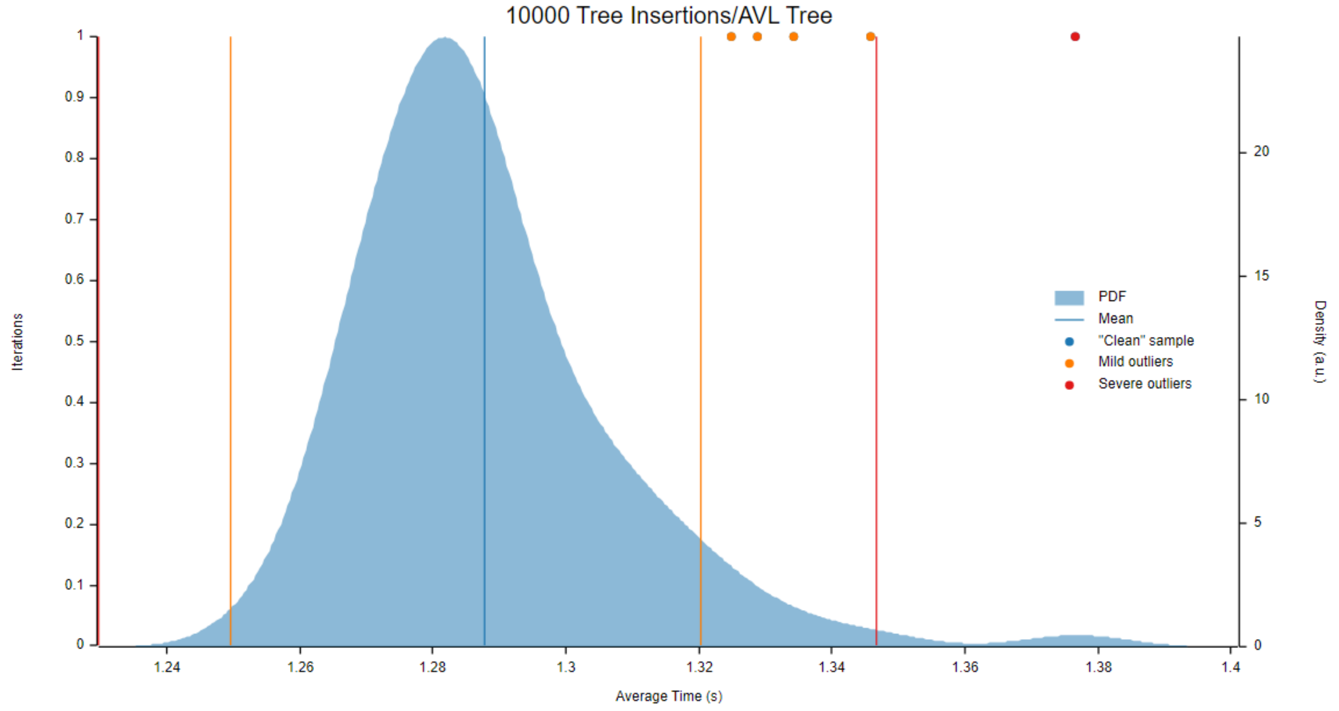
This section will be dedicated to the results of the benchmarks acquired while testing the program. Specifically we will be investigating how each tree type, Red-Black and AVL handle a large amount of insertions and a worst case scenario search for each tree respectively.

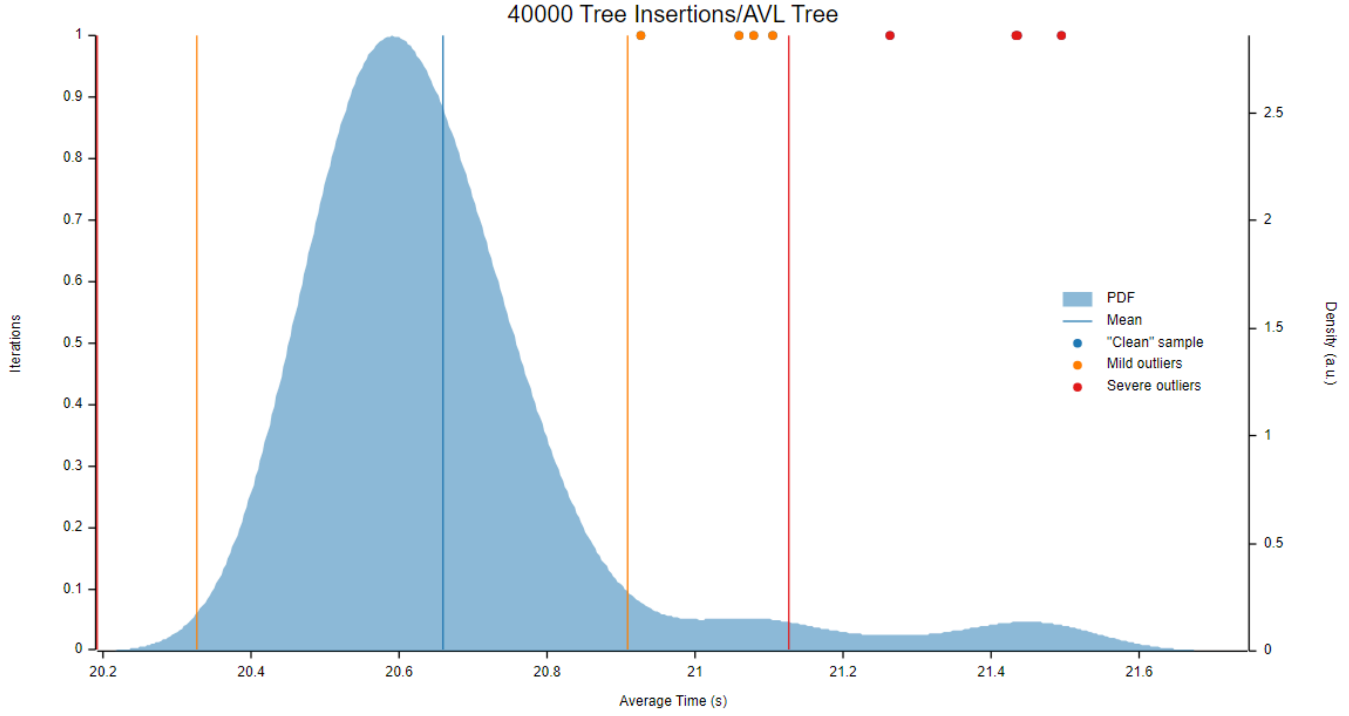
Insertions Charts

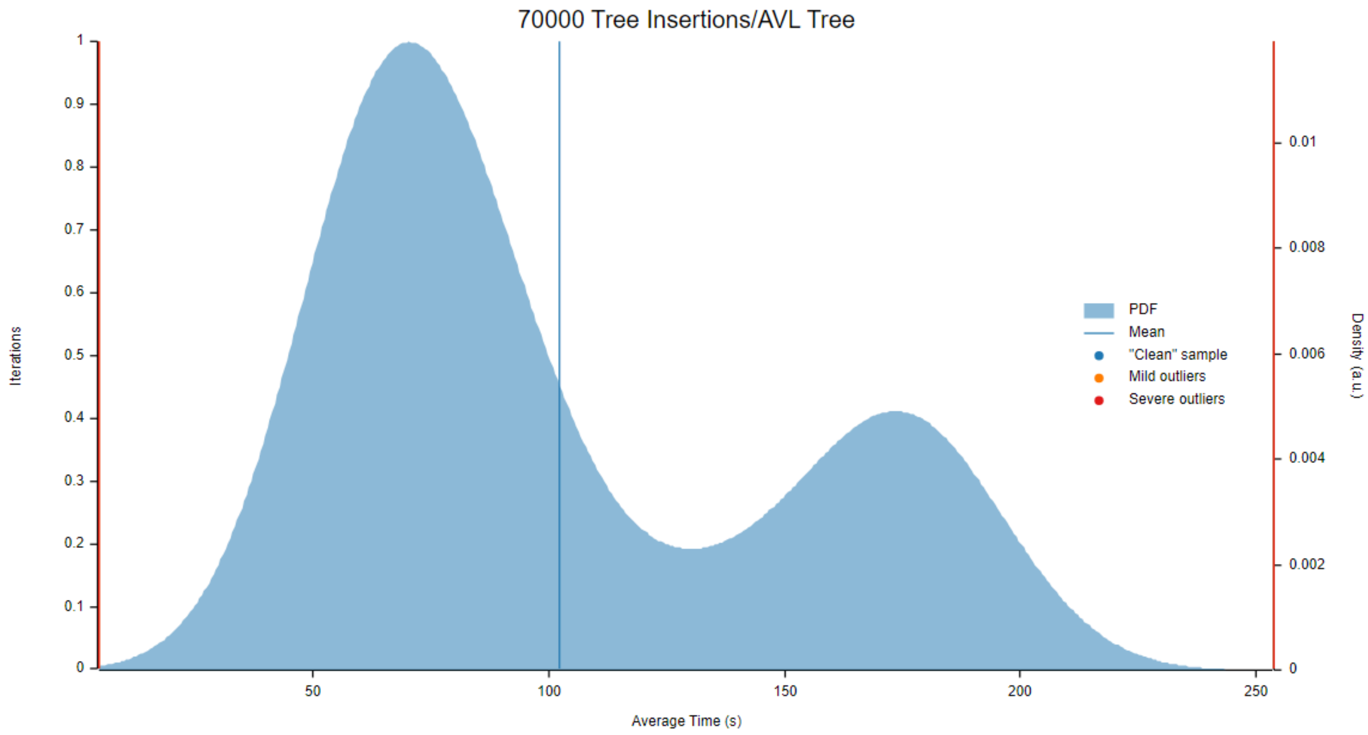
The charts below shows the relationship between function/parameter and iteration time. The thickness of the shaded region indicates the probability that a measurement of the given function/parameter would take a particular length of time.

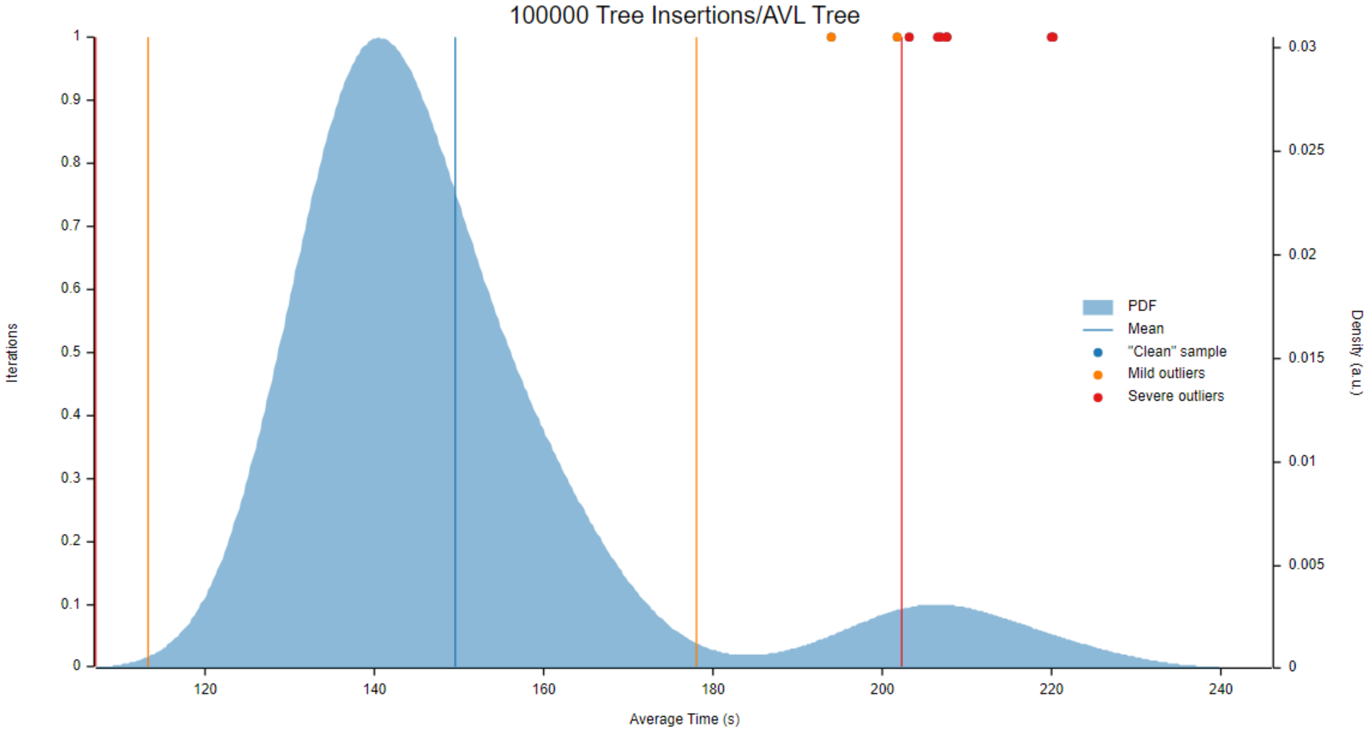
The insertions are simply inputted into the tree as [1, x] with x being the amount of insertions. To achieve this behavior we simply iterate over [1, x] and insert into the tree for each iteration. If x is 5 we will insert 1,2,..,5.

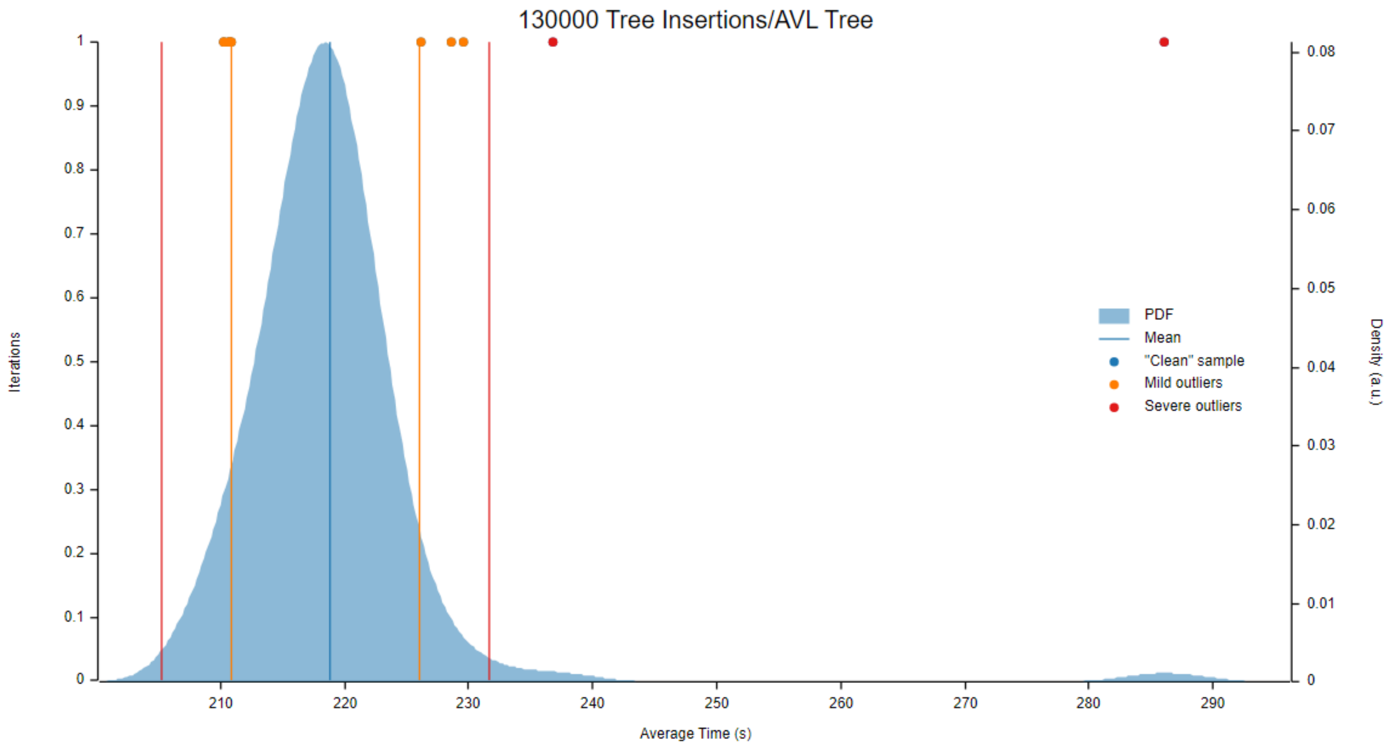
AVL Tree


10000 Tree Insertions/AVL Tree

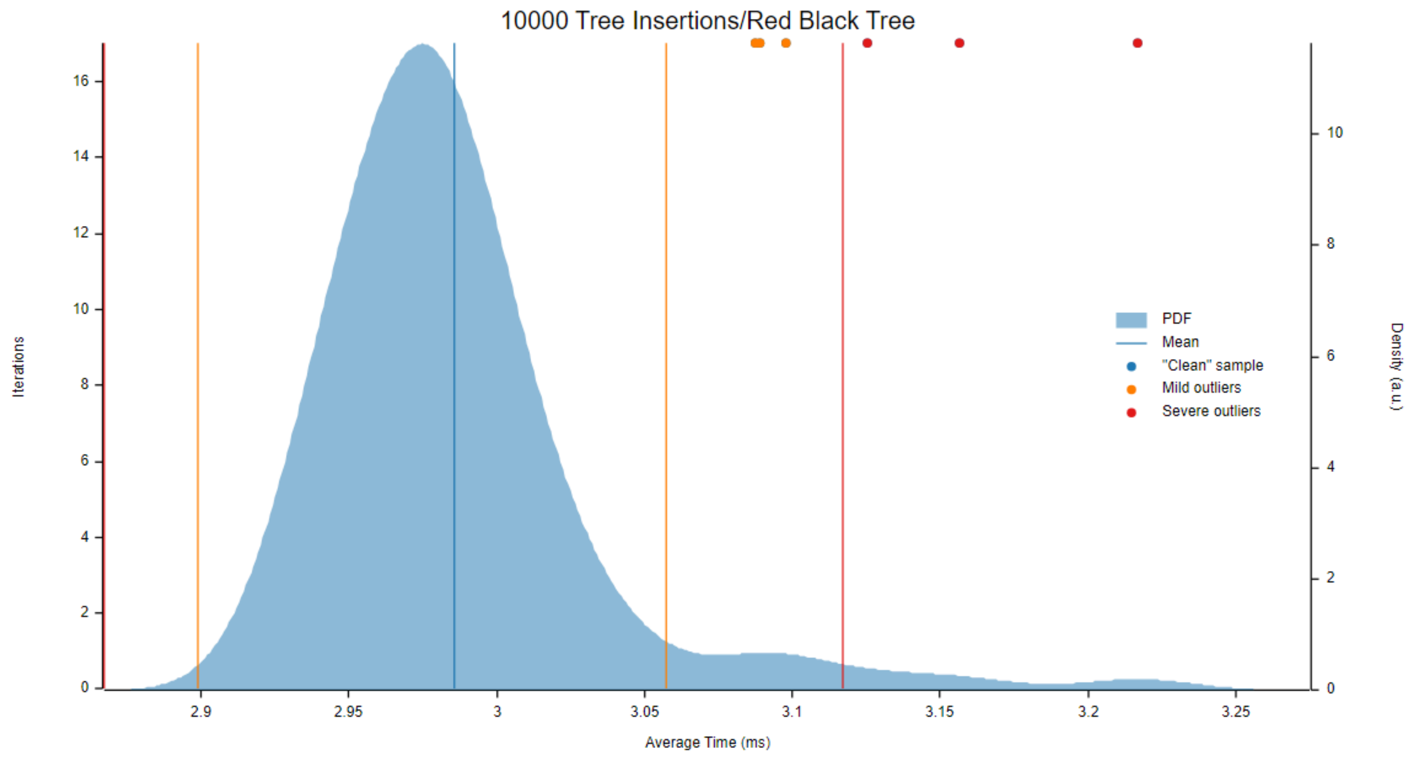

40000 Tree Insertions/AVL Tree


70000 Tree Insertions/AVL Tree
Iterations
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
50
100
150
200
250
Average Time (s)
Density (a.u.)
0.01
0.008
0.006
0.004
0.002
0
PDF
Mean
"Clean" sample
Mild outliers
Severe outliers

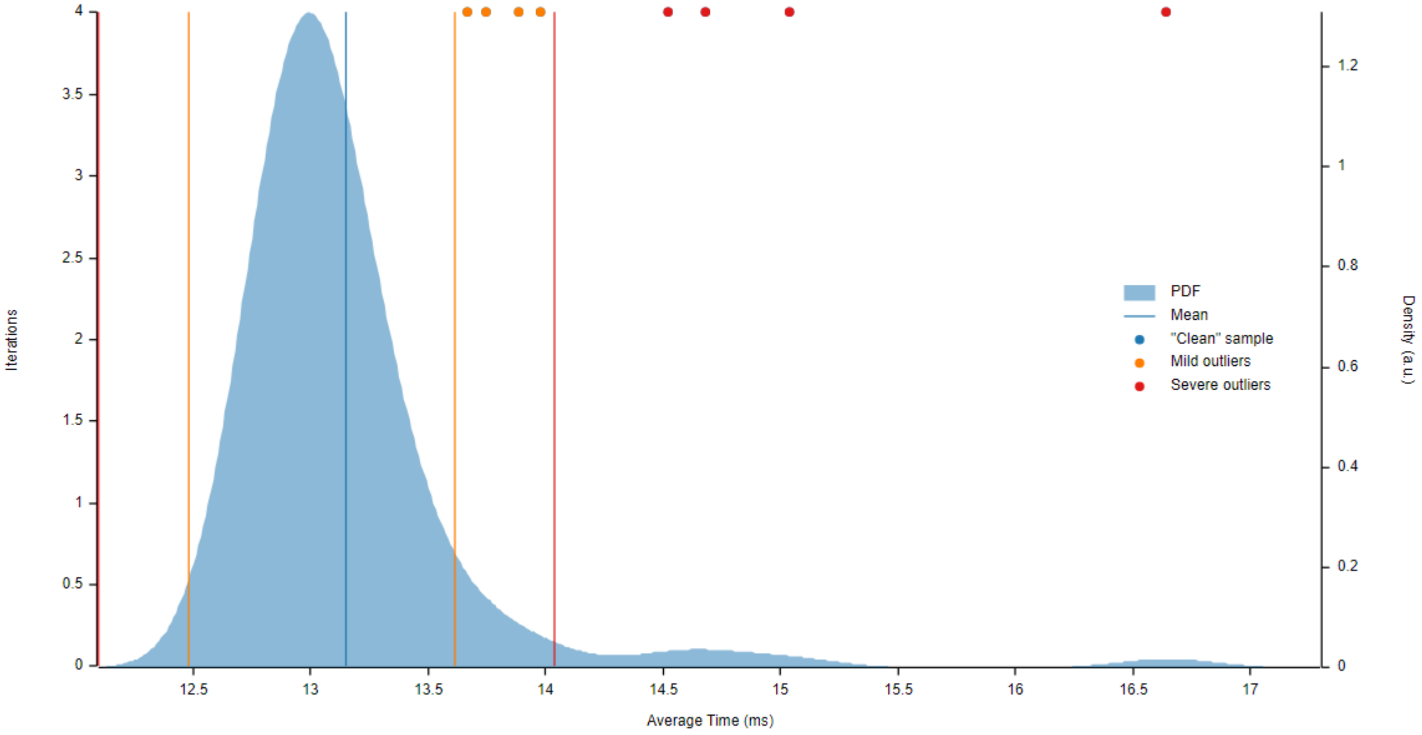

100000 Tree Insertions/AVL Tree
Iterations
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
120
140
160
180
200
220
240
Average Time (s)
Density (a.u.)
0.03
0.025
0.02
0.015
0.01
0.005
0
PDF
Mean
"Clean" sample
Mild outliers
Severe outliers



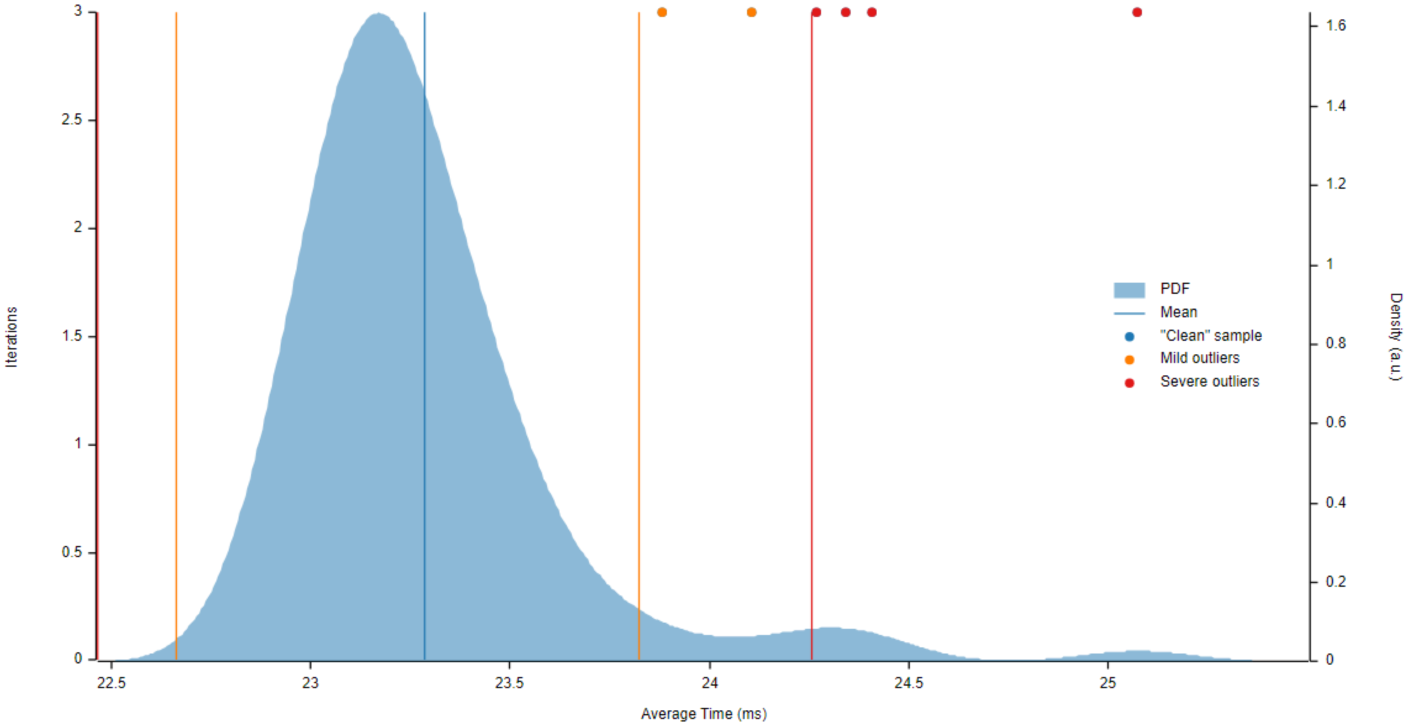
Red-Black Tree

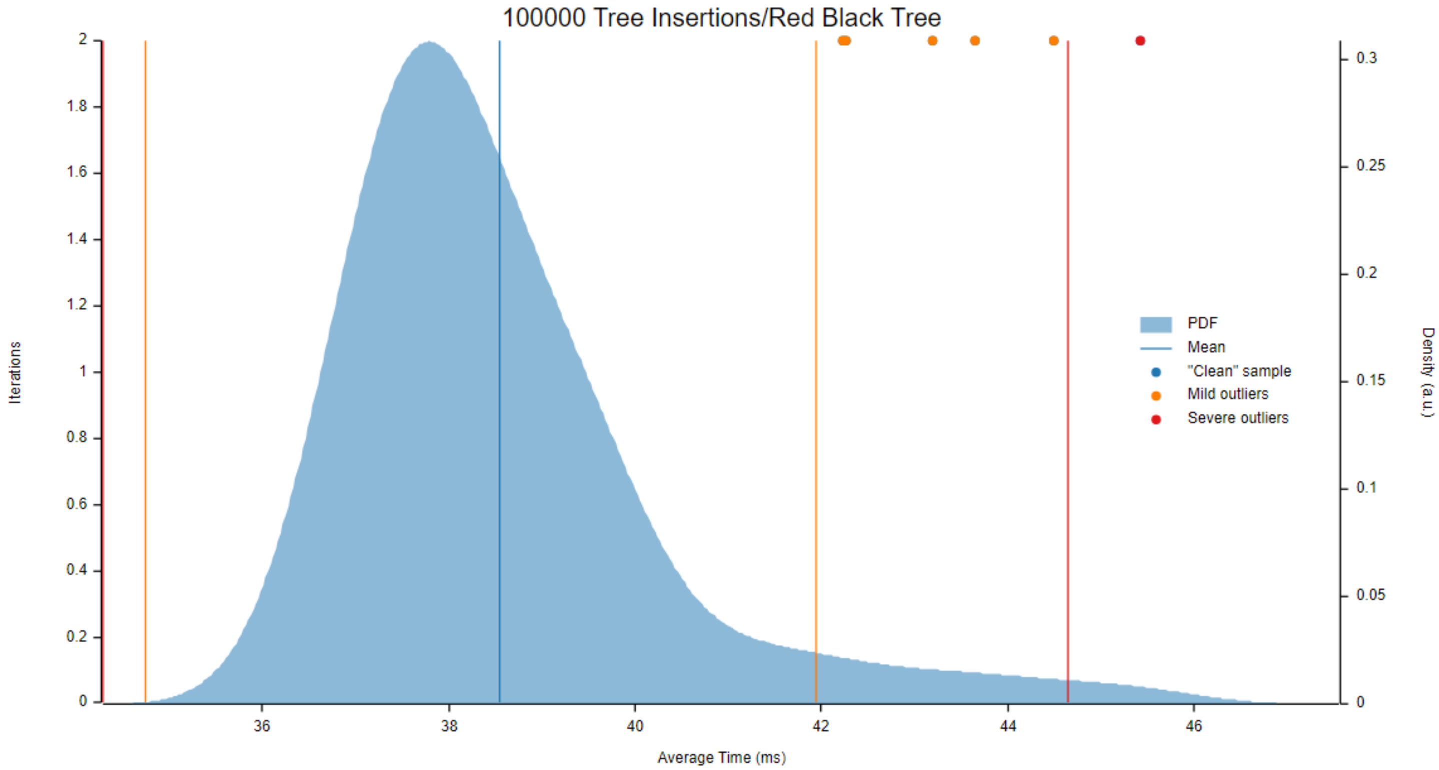


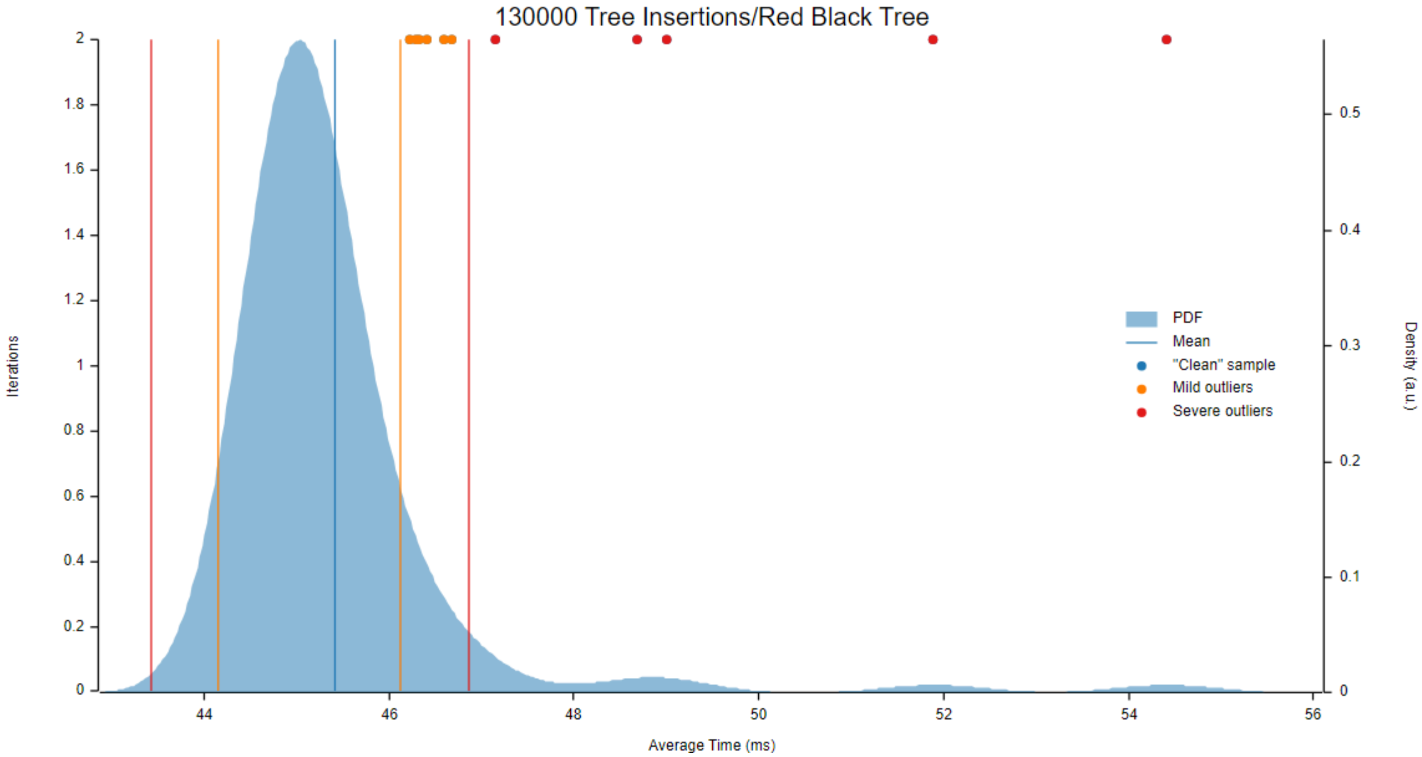
40000 Tree Insertions/Red Black Tree



70000 Tree Insertions/Red Black Tree




100000 Tree Insertions/Red Black Tree
Iterations
2
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
36
38
40
42
44
46
Average Time (ms)
Density (a.u.)
0.3
0.25
0.2
0.15
0.1
0.05
0
PDF
Mean
"Clean" sample
Mild outliers
Severe outliers

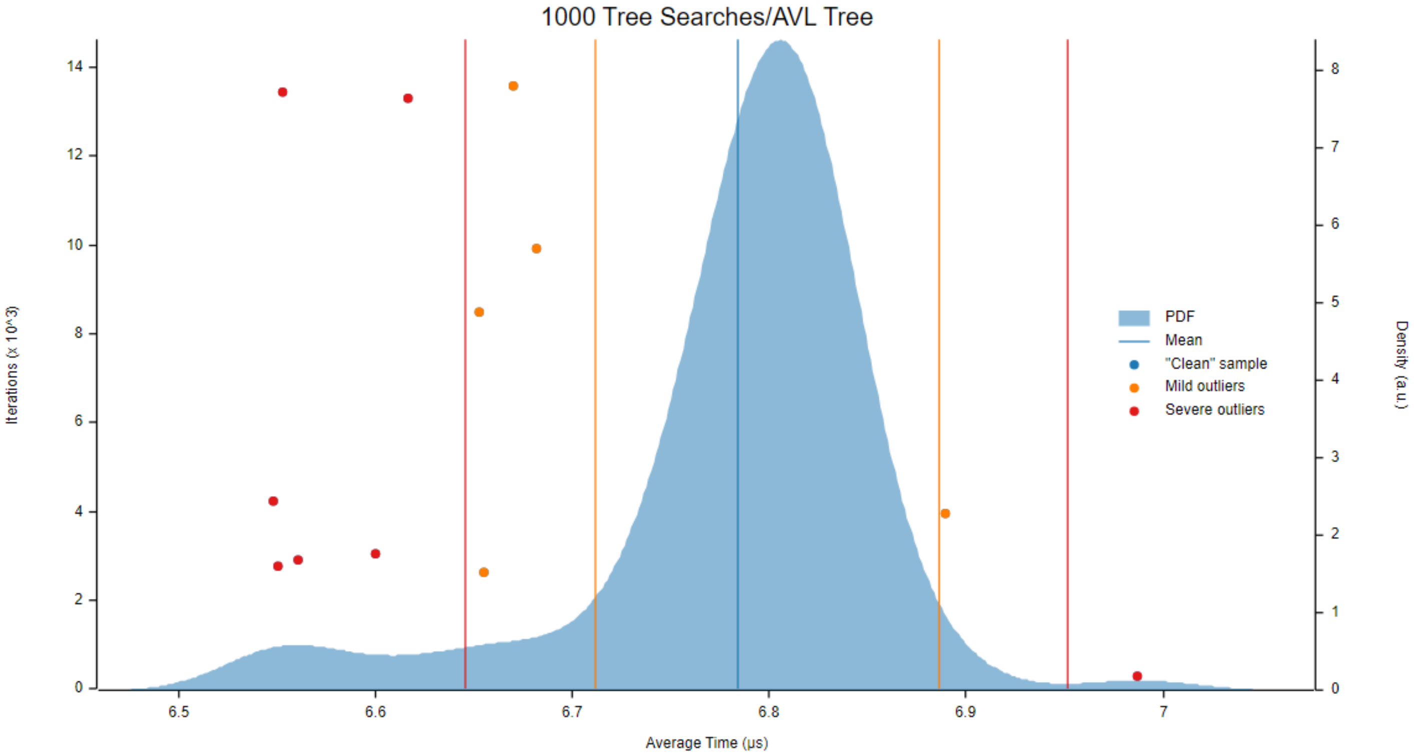

130000 Tree Insertions/Red Black Tree
Iterations
2
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
44
46
48
50
52
54
56
Average Time (ms)
Density (a.u.)
0.5
0.4
0.3
0.2
0.1
0
PDF
Mean
"Clean" sample
Mild outliers
Severe outliers

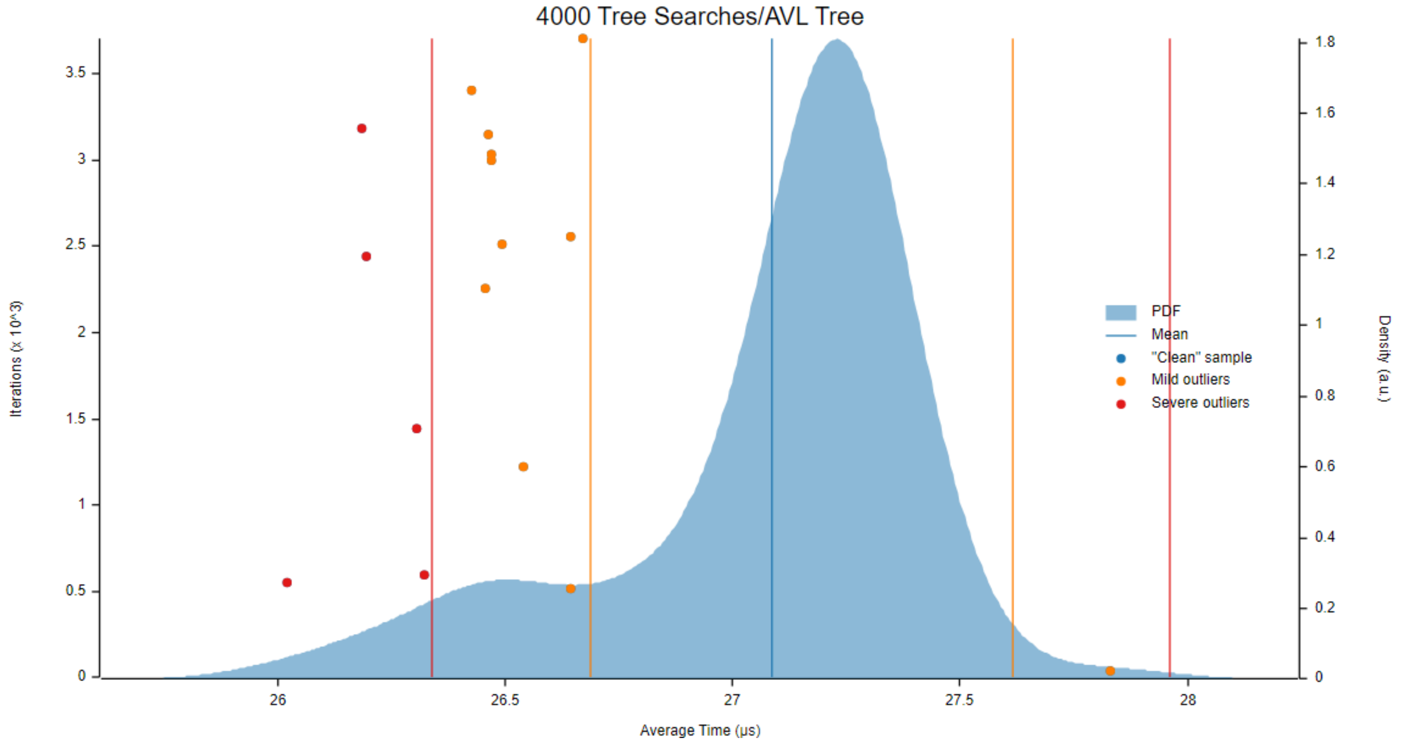
Search Charts

The charts below shows the relationship between function/parameter and iteration time. The thickness of the shaded region indicates the probability that a measurement of the given function/parameter would take a particular length of time.

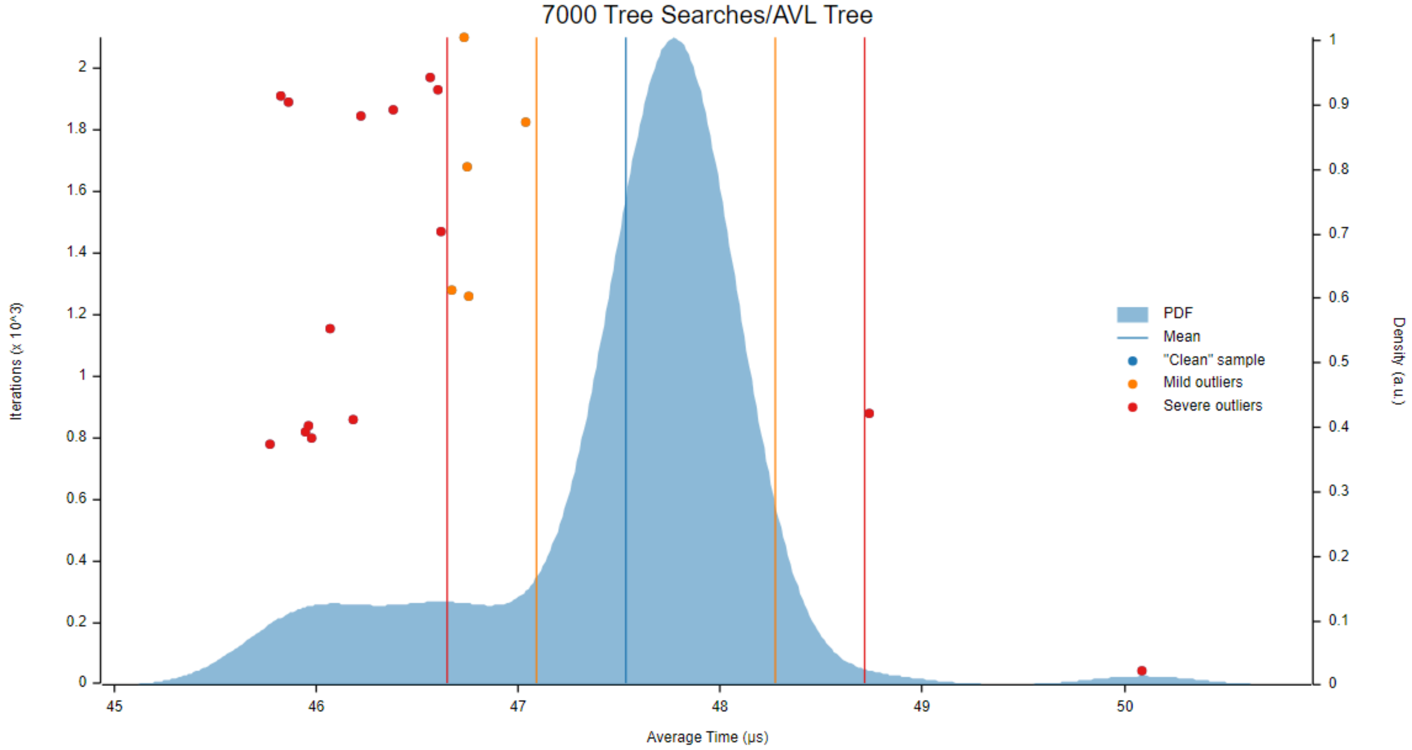
When searching we search for an x/10 amount based on the amount of insertions (x). If we insert 10,000 elements into the tree as done above in the insertion section we search for the 10,000/10 elements (x/10) which would just be 1000 searches. It's important to note that when searching we search for the lowest values in order to achieve worse case scenario results for each tree. So if we did 100 insertions for a tree, insert 100,99,98,....,1 we would search 10 times for the values 10,9,...,1 as it would be the worst case scenario for this specific example.

AVL Tree

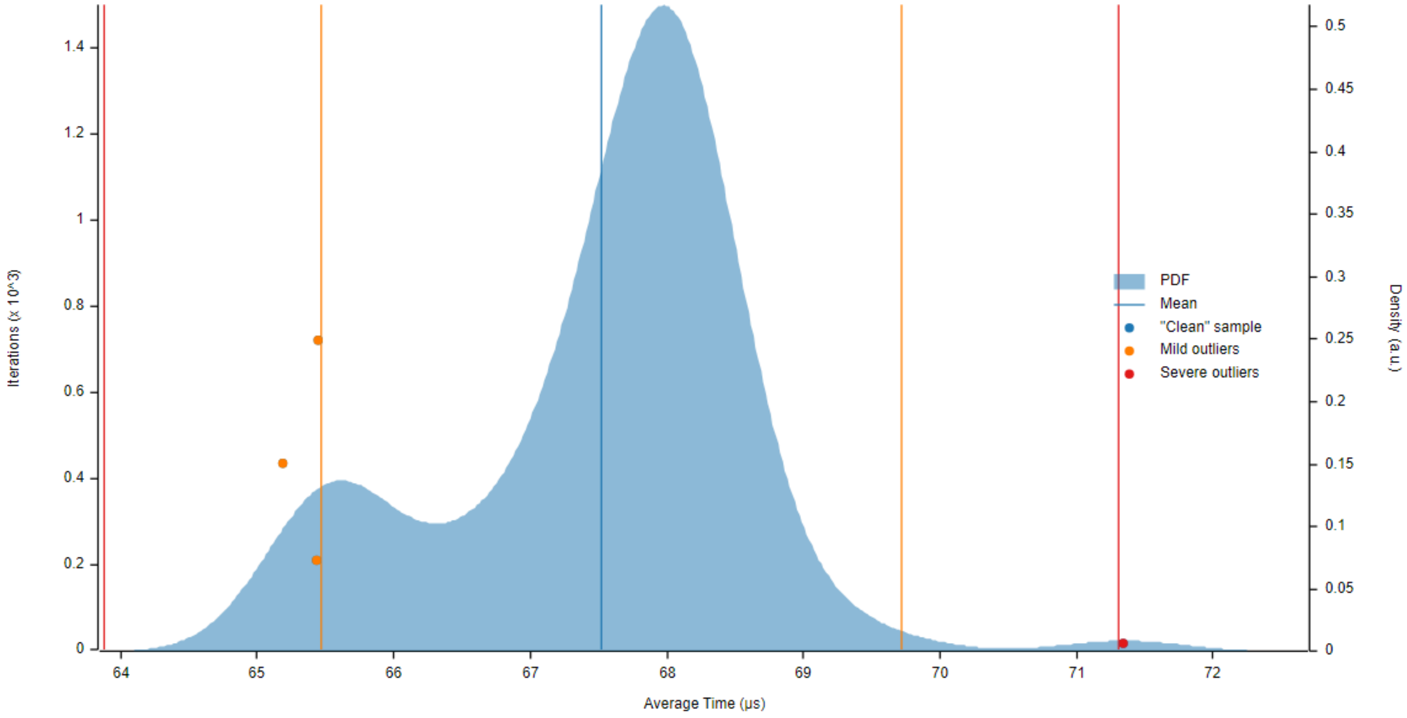


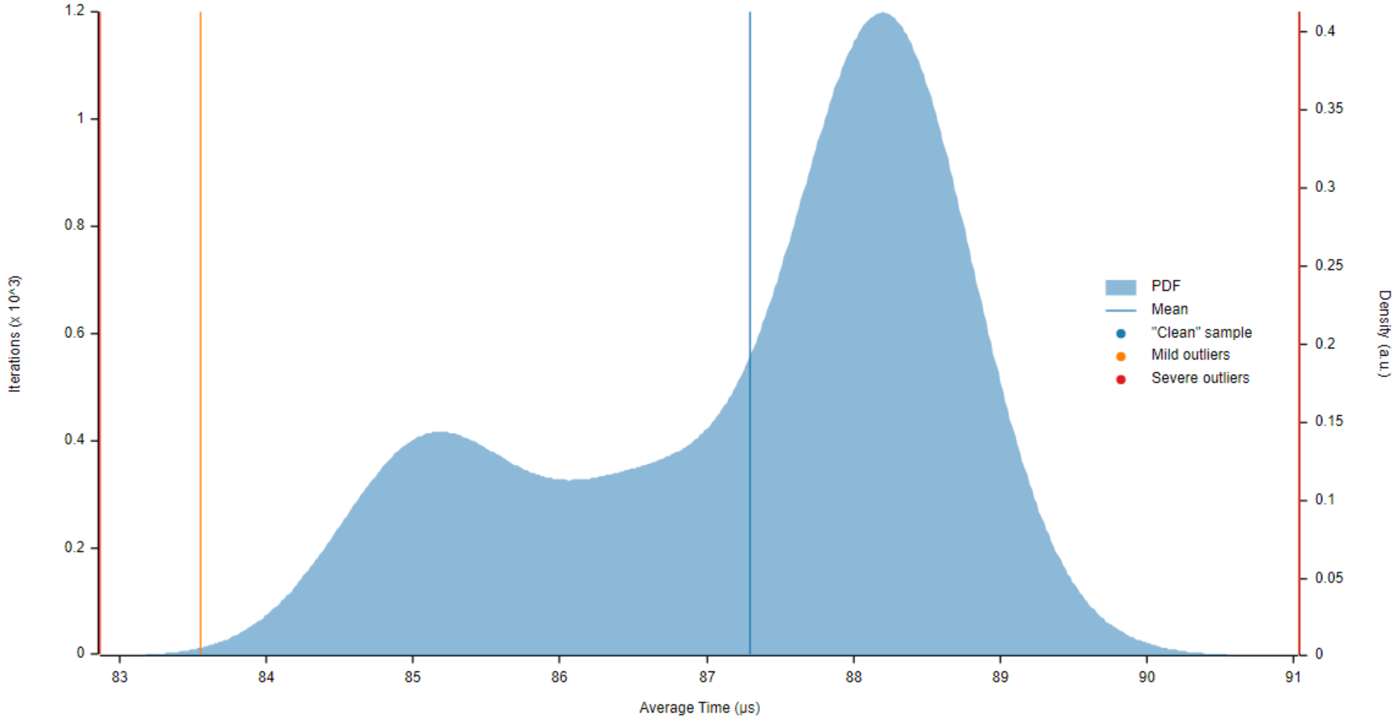


4000 Tree Searches/AVL Tree

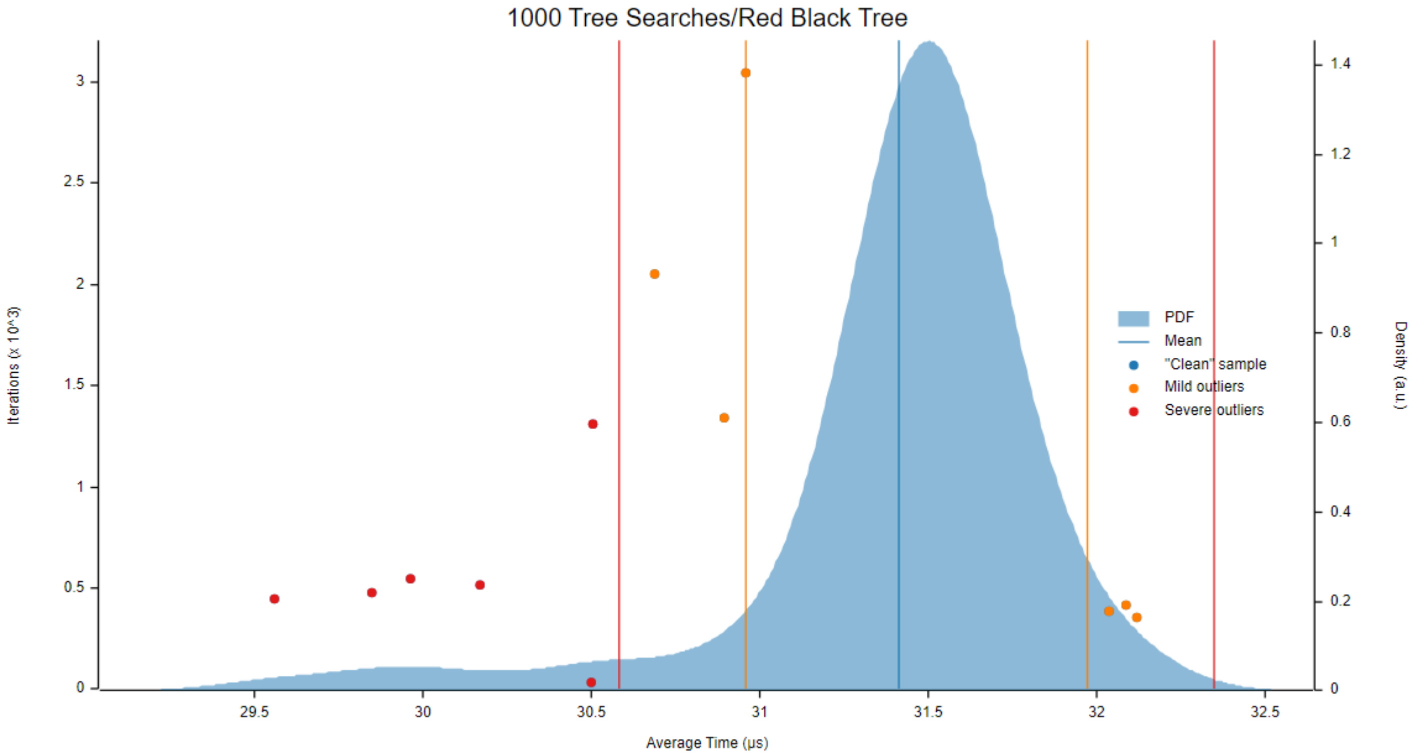


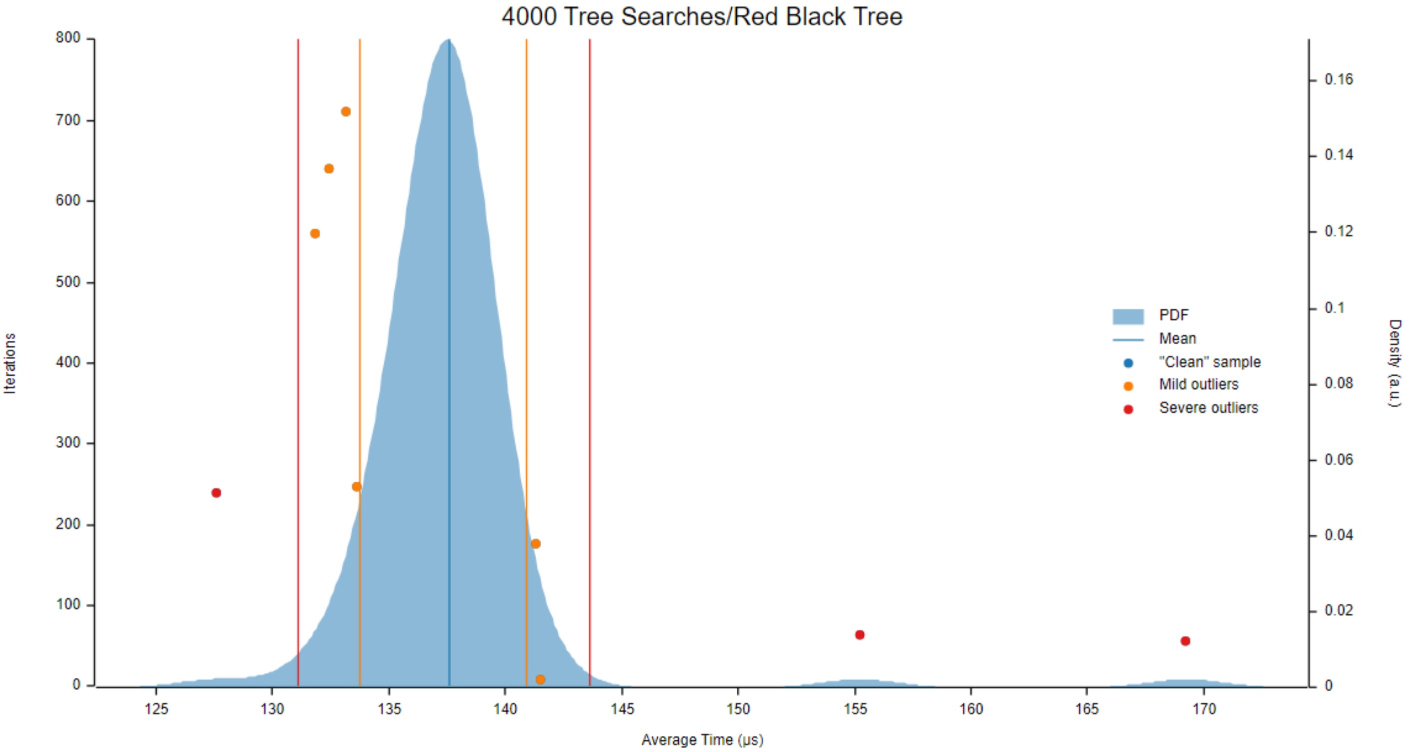
7000 Tree Searches/AVL Tree

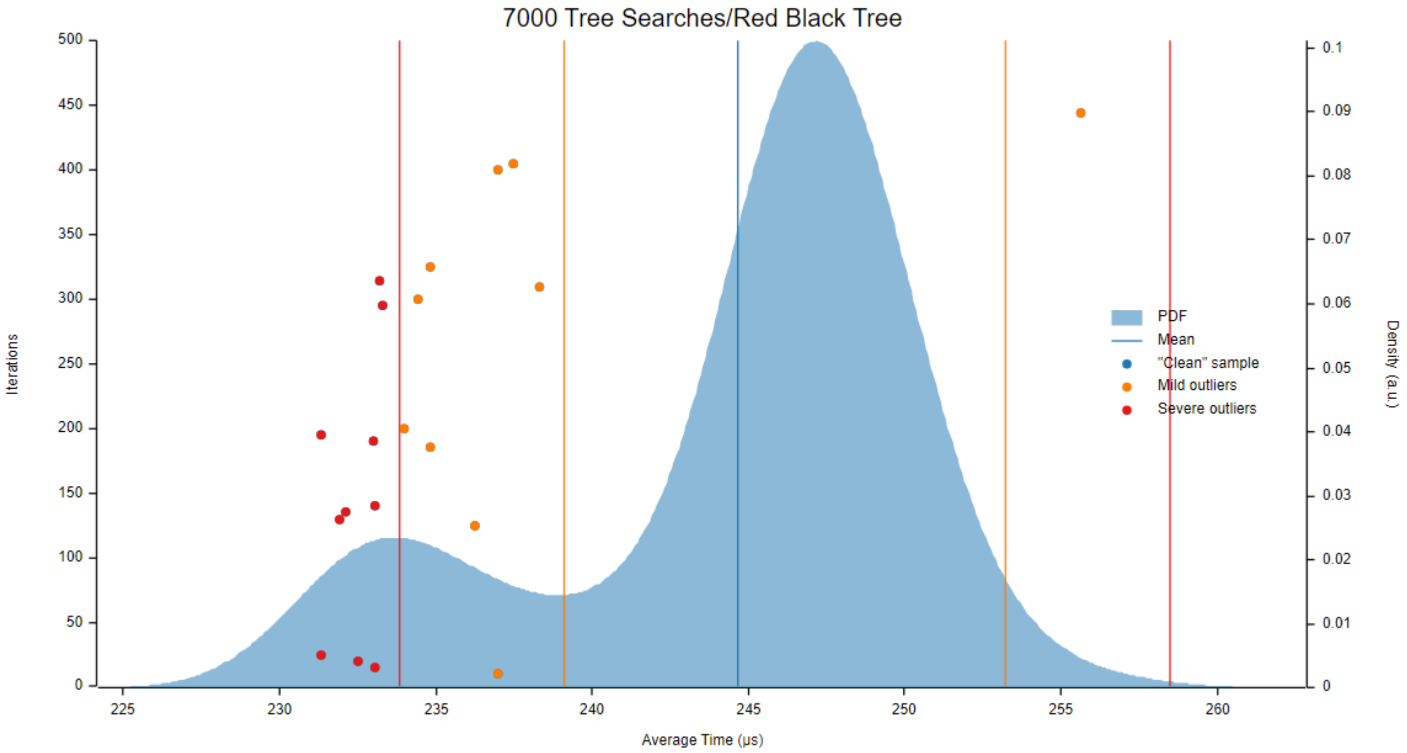

10000 Tree Searches/AVL Tree

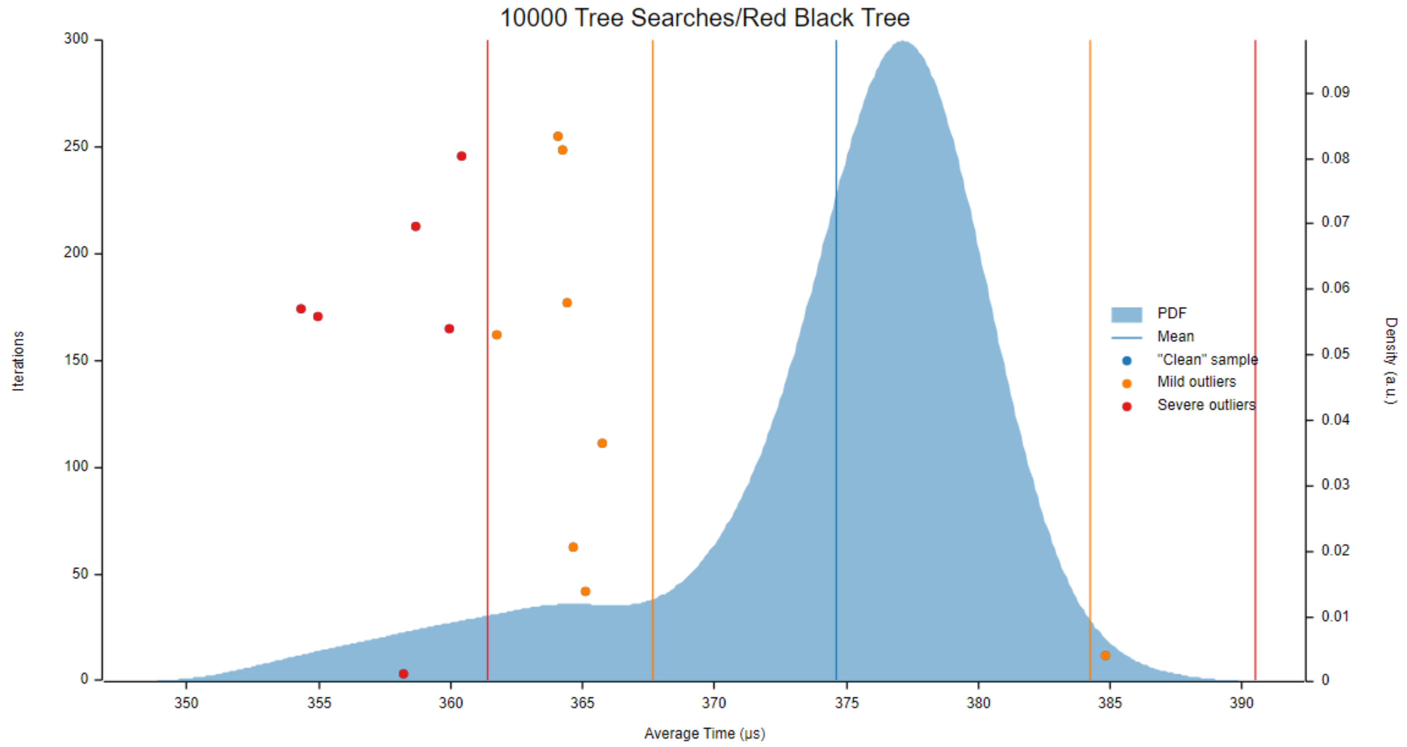

13000 Tree Searches/AVL Tree

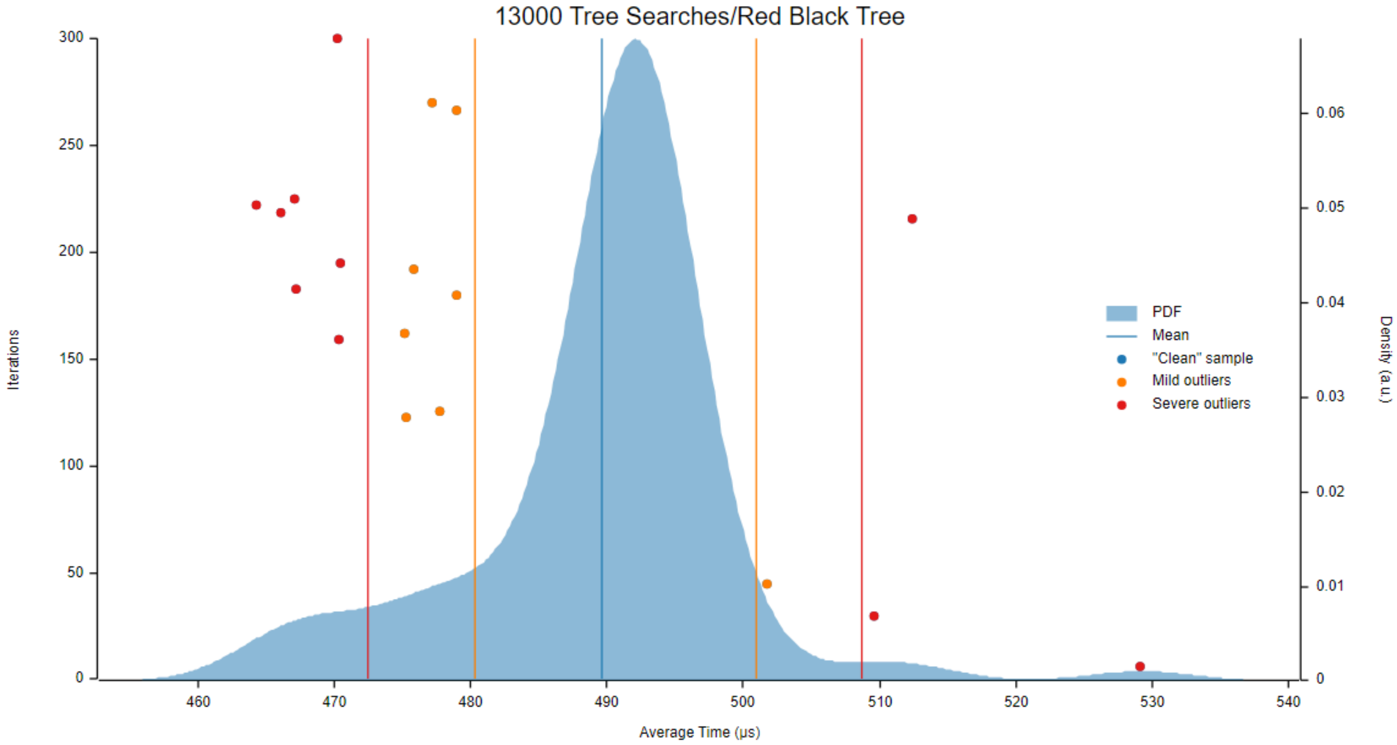
Red-Black Tree






7000 Tree Searches/Red Black Tree
Iterations
0
50
100
150
200
250
300
350
400
450
500
225
230
235
240
245
250
255
260
Average Time (µs)
Density (a.u.)
0
0.01
0.02
0.03
0.04
0.05
0.06
0.07
0.08
0.09
0.1
PDF
Mean
"Clean" sample
Mild outliers
Severe outliers


10000 Tree Searches/Red Black Tree
Iterations
0
50
100
150
200
250
300
350
355
360
365
370
375
380
385
390
Average Time (µs)
Density (a.u.)
0
0.01
0.02
0.03
0.04
0.05
0.06
0.07
0.08
0.09
PDF
Mean
"Clean" sample
Mild outliers
Severe outliers



Results

From the insertion times, it's evident that Red-Black trees heavily out perform AVL trees in terms of insertion speed. AVL trees consistently show lower insertion times across all tested data sizes. However, in the case of search operations, AVL trees maintain an advantage, albeit a narrower one compared to Red-Black. AVL trees show faster search times across different search volumes compared to Red-Black trees.

While the provided test cases offer insights into the performance of AVL and Red-Black trees, it might be beneficial to include more diverse scenarios. For instance, testing with different distribution patterns of input data (e.g., random, sorted, reverse-sorted) could reveal how each tree type performs under various real-world conditions. Additionally, testing with larger data sizes could provide a clearer picture of scalability and potential performance bottlenecks

AVL and Red-Black trees are two prominent self-balancing binary search trees, including a baseline data structure like a binary search tree (BST) could provide valuable context. Comparing the performance of AVL and Red-Black trees against a non-self-balancing BST could highlight the efficiency gains achieved by using self-balancing techniques. Moreover, including other data structures such as hash tables or skip lists could offer a broader perspective on the trade-offs between different data structure choices.

Average insertion times

Insertions	AVL Tree (s)	Red-Black Tree (ms)
10,000	1.2878	2.9857
40,000	20.659	13.152
70,000	102.27	23.288
100,000	149.69	38.540
130,000	218.84	45.410

Average search times

Searches	AVL Tree (µs)	Red-Black Tree (µs)
1000	6.7842	31.413
4000	27.085	137.62
7000	47.532	244.67
10,000	67.519	374.60
13,000	87.298	489.68