

Design Document

Both the red-black tree and AVL tree are efficient data structures as they provide means to insert, delete, and search the structure in $O(\log n)$ time. This program aims to implement the two efficient data structures in Rust and adds new features for them.

This program is written by Ahmed Al Dallal, Mohammad Amin Vafadar and Pengfei Gao.

If you want more information for the program, click [here](#).

Contents

Design Document

- Contents
- Part 1: Major innovations
 - Additional features
 - Red-black tree
 - AVL tree
 - Detailed Rationale
- Part 2: Current limitations
- Part 3: User manual
 - Operating environment
 - Quick start
 - Command-line Interface
 - Red-black Tree
 - AVL Tree
 - Code Interface
 - Red-black Tree
 - AVL Tree
- Part 4: Performance
 - Binary search tree
 - The benchmark cases
 - Results
 - Red-black tree and AVL tree
 - The benchmark cases
 - Results
 - Line Chart
 - Violin Plot

Part 1: Major innovations

Additional features

Red-black tree

1. Print Pre-order traversal of the tree.
2. Print Post-order traversal of the tree.
3. Check whether the node is in the tree.
4. Get the minimum value in a tree.
5. Get the maximum value in a tree.

6. Count the number of nodes in a tree.
7. Clear the tree, removing all elements.

AVL tree

1. Print Pre-order traversal of the tree.
2. Print Post-order traversal of the tree.
3. Check whether the node is in the tree.
4. Get the minimum value in a tree.
5. Get the maximum value in a tree.
6. Count the number of nodes in a tree.
7. Clear the tree, removing all elements.

Detailed Rationale

Pre-order and Post-order traversal

Red-black tree and AVL tree are data structures based on binary tree. Pre-order traversal and post-order traversal are the two most common types of binary tree traversal. **Pre-order** traversal is mainly used when a tree needs to be duplicated. The feature of **Post-order** traversal is that the left and right child nodes of the node must have been traversed during operation, so it is suitable for destructive operations, such as deleting all nodes.

Check the **existence of the node**

Both the insertion and deletion operations of the tree need to judge whether the node to be operated exists in the tree structure. We also need this function when benchmarking code.

Maximum and Minimum value

There are also many practical applications to obtain the maximum and minimum values of a set of data. For example, find employees with the longest working hours in the company, check the lowest and highest scores of this exam.

The number of nodes

Red black tree and AVL tree are often used to store large-scale data. When analyzing these data, it is very important to obtain the total data. For example, calculate the average number and standard deviation.

Part 2: Current limitations

###

1. Red black and AVL trees currently do not support inserting duplicate numbers.

Part 3: User manual

Operating environment

- Rust 1.50.0 or newer

if you need help to install the Rust on your computer, please click the link below

<https://www.rust-lang.org/tools/install>

Quick start

Command-line Interface

First, please enter cargo run on the terminal.

```
cargo run
```

You will see the welcome page.

There are three data structures you can use.

```
----- Trees Command Line Interface -----  
-----  
  
Available trees:  
- Red-Black Tree  
- AVL tree  
- Binary Search Tree  
  
Available operations:  
1- Insert  
2- Delete  
3- Count Leaves  
4- Count Nodes  
5- Height  
6- Maximum  
7- Minimum  
8- Empty  
9- Search  
10- Traverse  
11- Print  
12- Clear  
  
How to use the Command Line Interface:  
-----  
You can select a tree number to start or type 'exit' to leave!  
Select a tree!  
1- Red-Black Tree  
2- AVL Tree  
3- Binary Search Tree  
input >
```

Red-black Tree

In the welcome interface, please enter 1 to select the red black tree.

When you input 1, the output is as follows:

----- Red-Black Tree branch -----

Availabe Operations:

Enter the number corresponding to the operation you want to **perform!**

- 1) Insert - insert a node into the tree.
- 2) Delete - delete a node from the tree.
- 3) Count Leaves - count the number of leaves **in** the tree.
- 4) Count Nodes - count the number of nodes **in** the tree
- 5) Height - **return** the height of the tree
- 6) Maximum - find the maximum value **in** the tree
- 7) Minimum - find the minimum value **in** the tree
- 8) Empty - check **if** the tree is empty
- 9) Search - check **if** the tree contains a certain value
- 10) Traverse - traverse the tree (Inorder, Preorder, or Postorder)
- 11) Print - print the tree
- 12) Clear - clear the tree, removing all elements.
- Back - Go back to previous menu and erase current tree

Operation >

1. Select 1 to insert a node into the tree.

Operation > 1

When you input 1, the output is as follows:

insert value >

Please enter the value you want to insert.

We input 1 as an example. The output is as follows:

insert value > 1

The insert operation **for '1' in** the tree is **complete!**

If the entered value already exists, the output is as follows:

insert value > 1

The insert operation **for '1' in** the tree is **complete!**

The node already exists.

2. Select 2 to delete a node from the tree.

Operation > 2

When you input 2, the output is as follows:

delete value >

Please enter the value you want to delete.

We input 1 as an example. The output is as follows:

delete value > 1
The delete operation for '1' in the tree is complete!

If the value to delete does not exist, the output is as follows:

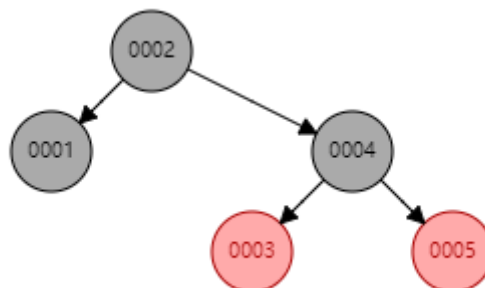
delete value > 1
The delete operation for '1' in the tree is complete!
The node of value: 1 doesn't exist.

3. Select 3 to count the number of leaves in the tree.

Operation > 3

When you input 3, the output is as follows:

Suppose the nodes in the tree are 1, 2, 3, 4, 5.



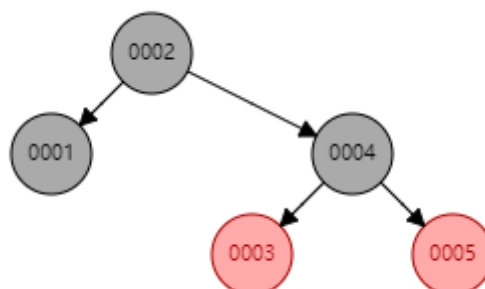
Number of leaves: 3

4. Select 4 to count the number of nodes in the tree.

Operation > 4

When you input 4, the output is as follows:

Suppose the nodes in the tree are 1, 2, 3, 4, 5.



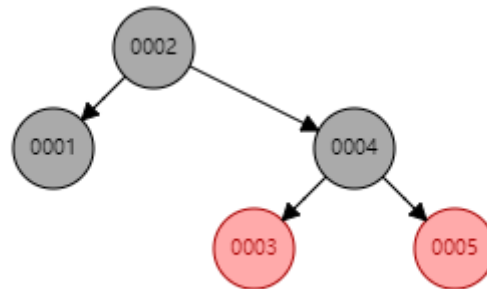
Number of nodes: 5

5. Select 5 to get the height of the tree.

Operation > 5

When you input 5, the output is as follows:

Suppose the nodes in the tree are 1, 2, 3, 4, 5.



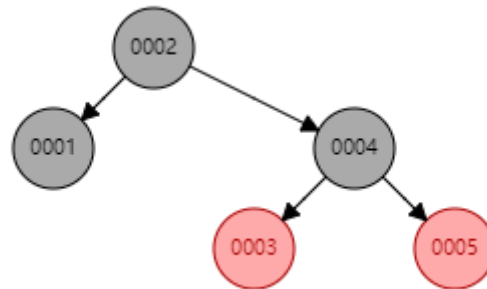
Height of the tree: 3

6. Select 6 to find the maximum value in the tree.

Operation > 6

When you input 6, the output is as follows:

Suppose the nodes in the tree are 1, 2, 3, 4, 5.



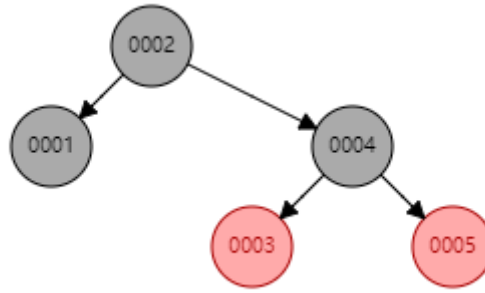
Maximum Value: 5

7. Select 7 to find the minimum value in the tree.

Operation > 7

When you input 7, the output is as follows:

Suppose the nodes in the tree are 1, 2, 3, 4, 5.



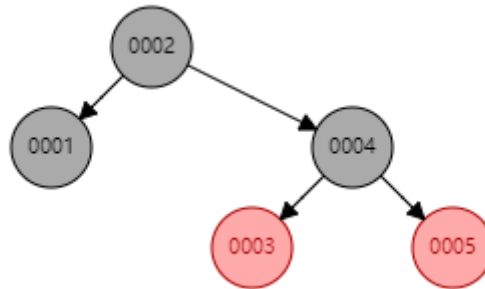
Minimum Value: 1

8. Select 8 to check if the tree is empty.

Operation > 8

When you input 8, the output is as follows:

Suppose the nodes in the tree are 1, 2, 3, 4, 5.



Is the tree empty?: false

If the tree is empty, the output is as follows:

Is the tree empty?: true

9. Select 9 to check if the tree contains a certain value.

Operation > 9

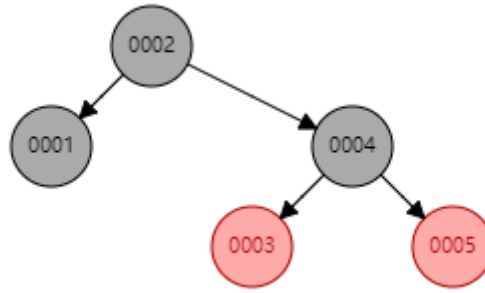
When you input 9, the output is as follows:

search value >

Please enter the value you want to delete.

We input 3 as an example. The output is as follows:

Suppose the nodes in the tree are 1, 2, 3, 4, 5.



search value > 3
The search operation for '3' in the tree is complete!
Value found? true

When the value you entered does not exist, the output is as follows:

search value > 6
The search operation for '6' in the tree is complete!
Value found? false

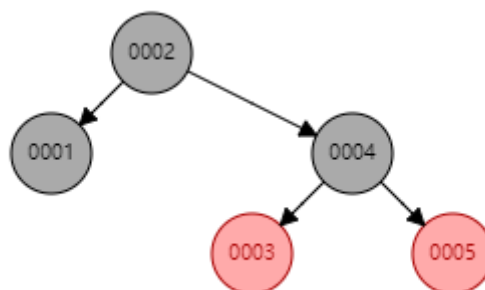
10. Select 10 to traverse the tree (Inorder, Preorder, or Postorder)

Operation > 10

When you input 10, the output is as follows:

Enter the number corresponding to the tree traversal type you want or type 'back' to select a different operation!
1-Inorder
2-Preorder
3-Postorder
input >

Suppose the nodes in the tree are 1, 2, 3, 4, 5.



When you enter 1, the output will be In-order traversal.

input > 1
Your tree:
1
2
3
4
5

When you enter 2, the output will be Pre-order traversal.

```
input > 2
Your tree:
2
1
4
3
5
```

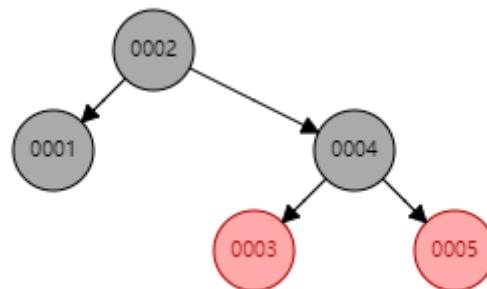
When you enter 3, the output will be Post-order traversal.

```
input > 3
Your tree:
1
3
5
4
2
```

11. Select 11 to print the tree.

```
Operation > 11
```

Suppose the nodes in the tree are 1, 2, 3, 4, 5.



When you input 11, the output is as follows:

```
Root 2 Black
|___ L 1 Black
|___ R 4 Black
    |___ L 3 Red
    |___ R 5 Red
```

12. Select 12 to clear the tree and remove all elements.

```
Operation > 12
```

When you input 12, the output is as follows:

```
Operation > 12
Clear operation is complete!
```

Enter 8 to check if the tree is empty.

```
Operation > 8
Is the tree empty?: true
```

13. Enter back to return to the previous menu.

```
Operation > back

You can select a tree number to start or type 'exit' to leave!
Select a tree!
1- Red-Black Tree
2- AVL Tree
3- Binary Search Tree
input >
```

AVL Tree

In the welcome interface, please enter 2 to select the AVL tree.

When you input 2, the output is as follows:

```
----- AVL Tree branch -----

Availabe Operations:
-----
Enter the number corresponding to the operation you want to perform!

1) Insert      - insert a node into the tree.
2) Delete      - delete a node from the tree.
3) Count Leaves - count the number of leaves in the tree.
4) Count Nodes - count the number of nodes in the tree
5) Height      - return the height of the tree
6) Maximum     - find the maximum value in the tree
7) Minimum     - find the minimum value in the tree
8) Empty       - check if the tree is empty
9) Search      - check if the tree contains a certain value
10) Traverse   - traverse the tree (Inorder, Preorder, or Postorder)
11) Print      - print the tree
12) Clear      - clear the tree, removing all elements.
Back          - Go back to previous menu and erase current tree

Operation >
```

1. Select 1 to insert a node into the tree.

```
Operation > 1
```

When you input 1, the output is as follows:

```
insert value >
```

Please enter the value you want to insert.

We input 1 as an example. The output is as follows:

insert value > 1
The insert operation for '1' in the tree is complete!

If the entered value already exists, the output is as follows:

insert value > 1
The insert operation for '1' in the tree is complete!
The node already exists.

2. Select 2 to delete a node from the tree.

Operation > 2

When you input 2, the output is as follows:

delete value >

Please enter the value you want to delete.

We input 1 as an example. The output is as follows:

delete value > 1
The delete operation for '1' in the tree is complete!

If the value to delete does not exist, the output is as follows:

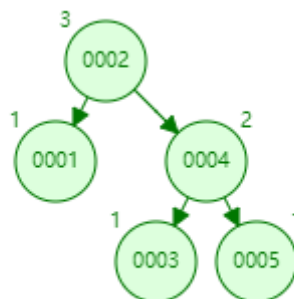
delete value > 1
The delete operation for '1' in the tree is complete!
The node of value: 1 doesn't exist.

3. Select 3 to count the number of leaves in the tree.

Operation > 3

When you input 3, the output is as follows:

Suppose the nodes in the tree are 1, 2, 3, 4, 5.



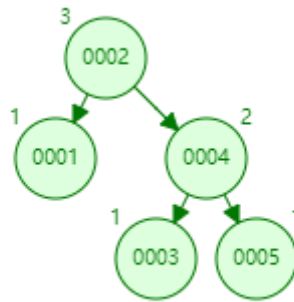
Number of leaves: 3

4. Select 4 to count the number of nodes in the tree.

Operation > 4

When you input 4, the output is as follows:

Suppose the nodes in the tree are 1, 2, 3, 4, 5.



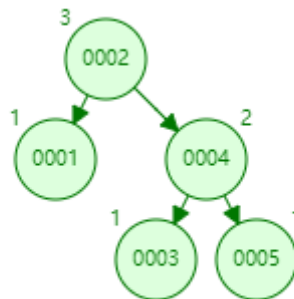
Number of nodes: 5

5. Select 5 to get the height of the tree.

Operation > 5

When you input 5, the output is as follows:

Suppose the nodes in the tree are 1, 2, 3, 4, 5.



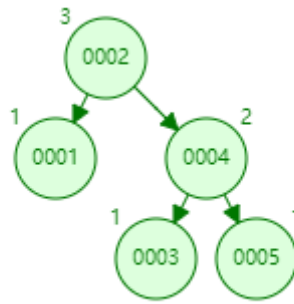
Height of the tree: 3

6. Select 6 to find the maximum value in the tree.

Operation > 6

When you input 6, the output is as follows:

Suppose the nodes in the tree are 1, 2, 3, 4, 5.



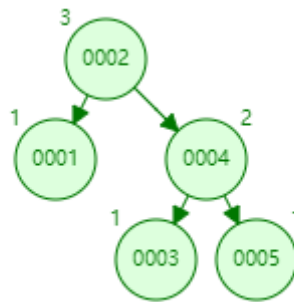
Maximum Value: 5

7. Select 7 to find the minimum value in the tree.

Operation > 7

When you input 7, the output is as follows:

Suppose the nodes in the tree are 1, 2, 3, 4, 5.



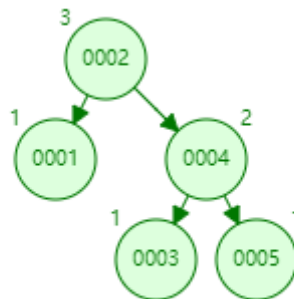
Minimum Value: 1

8. Select 8 to check if the tree is empty.

Operation > 8

When you input 8, the output is as follows:

Suppose the nodes in the tree are 1, 2, 3, 4, 5.



Is the tree empty?: false

If the tree is empty, the output is as follows:

Is the tree empty?: true

9. Select 9 to check if the tree contains a certain value.

Operation > 9

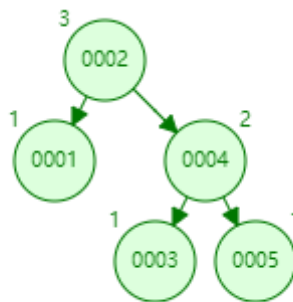
When you input 9, the output is as follows:

search value >

Please enter the value you want to delete.

We input 3 as an example. The output is as follows:

Suppose the nodes in the tree are 1, 2, 3, 4, 5.



search value > 3

The search operation for '3' in the tree is complete!

Value found? true

When the value you entered does not exist, the output is as follows:

search value > 6

The search operation for '6' in the tree is complete!

Value found? false

10. Select 10 to traverse the tree (Inorder, Preorder, or Postorder)

Operation > 10

When you input 10, the output is as follows:

Enter the number corresponding to the tree traversal type you want or type 'back' to select a different operation!

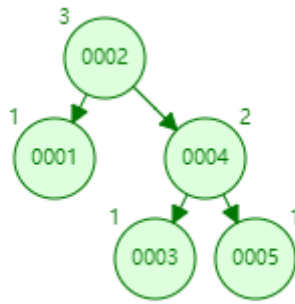
1-Inorder

2-Preorder

3-Postorder

input >

Suppose the nodes in the tree are 1, 2, 3, 4, 5.



When you enter 1, the output will be In-order traversal.

```
input > 1
Your tree:
1
2
3
4
5
```

When you enter 2, the output will be Pre-order traversal.

```
input > 2
Your tree:
2
1
4
3
5
```

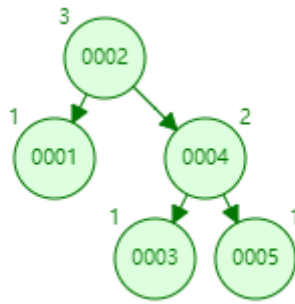
When you enter 3, the output will be Post-order traversal.

```
input > 3
Your tree:
1
3
5
4
2
```

11. Select 11 to print the tree.

```
Operation > 11
```

Suppose the nodes in the tree are 1, 2, 3, 4, 5.



When you input 11, the output is as follows:

```

Root 2
|__ L 1
|__ R 4
    |__ L 3
    |__ R 5
  
```

12. Select 12 to clear the tree and remove all elements.

```
Operation > 12
```

When you input 12, the output is as follows:

```

Operation > 12
Clear operation is complete!
  
```

Enter 8 to check if the tree is empty.

```

Operation > 8
Is the tree empty?: true
  
```

13. Enter back to return to the previous menu.

```

Operation > back

You can select a tree number to start or type 'exit' to leave!
Select a tree!
1- Red-Black Tree
2- AVL Tree
3- Binary Search Tree
input >
  
```

Code Interface

Red-black Tree

The structure for red-black tree is

```

type RcRefCellIRBTNode<T> = Rc<RefCell<RBTreeNode<T>>>;
type OptionNode<T> = Option<RcRefCellIRBTNode<T>>;
  
```


1. Create a new empty red-black tree

```
let mut tree=rbtree::RBTree::new();
```

2. Insert nodes to the red-black tree

```
//Assume that the value of the new node is 1
let new_node_val=1;
tree.insert(new_node_val);
```

3. Delete nodes from the red-black tree

```
//Assume that the value of the node to be deleted is 8
let delete_node_val=8;
tree.delete(delete_node_val);
```

4. Count the number of leaves in the red-black tree

```
tree.count_leaves();
//Print the result
println!("The number of leaves: {}",tree.count_leaves());
```

5. Get the height of the red-black tree

```
tree.get_height();
//Print the result
println!("The height of tree: {}",tree.get_height());
```

6. Print In-order traversal of the red-black tree

Print Pre-order traversal of the red-black tree

Print Post-order traversal of the red-black tree

```
tree.traverse_inorder();
//for example we insert 3,2,1,4,5 into the tree
//the result will be 1,2,3,4,5
tree.traverse_preorder();
//for example we insert 3,2,1,4,5 into the tree
//the result will be 2,1,3,4,5
tree.traverse_postorder();
//for example we insert 3,2,1,4,5 into the tree
//the result will be 1,3,4,5,2
```

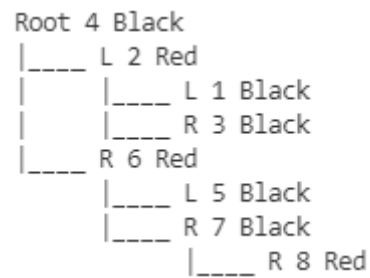
7. Check if the tree is empty

```
tree.is_empty()
//if you want to see the result
println!("The tree is empty? {}",tree.is_empty());
//when the tree is empty, it will print true. Otherwise it will print false
```

8. Print the tree

```
tree.print_tree();
```

For example, if we insert 1,2,3,4,5,6,7,8. You should see output similar to the following:



AVL Tree

The structure for AVL tree is

```
type RcRefcellAVLNode<T> = Rc<RefCell<AVLTreeNode<T>>>;
type OptionNode<T> = Option<RcRefcellAVLNode<T>>;
```

1. Create a new empty AVL tree

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

2. Insert nodes to the AVL tree

```
//Assume that the value of the new node is 1
let new_node_val=1;
tree.insert(new_node_val);
```

3. Delete nodes from the AVL tree

```
//Assume that the value of the node to be deleted is 8
let delete_node_val=8;
tree.delete(delete_node_val);
```

4. Count the number of leaves in the AVL tree

```
tree.count_leaves();
//Print the result
println!("The number of leaves: {}",tree.count_leaves());
```

5. Get the height of the AVL tree

```
tree.get_height();
//Print the result
println!("The height of tree: {}",tree.get_height());
```

6. Print In-order traversal of the AVL tree

Print Pre-order traversal of the AVL tree

Print Post-order traversal of the AVL tree

```

tree.traverse_inorder();
//for example we insert 3,2,1,4,5 into the tree
//the result will be 1,2,3,4,5
tree.traverse_preorder();
//for example we insert 3,2,1,4,5 into the tree
//the result will be 2,1,3,4,5
tree.traverse_postorder();
//for example we insert 3,2,1,4,5 into the tree
//the result will be 1,3,4,5,2

```

7. Check if the AVL tree is empty

```

tree.is_empty()
//Print the result
println!("The tree is empty? {}",tree.is_empty());
//when the tree is empty, it will print true. Otherwise it will print false

```

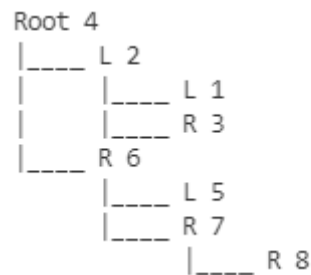
8. Print the AVL tree

```

tree.print_tree();

```

For example, if we insert 1,2,3,4,5,6,7,8. You should see output similar to the following:



Part 4: Performance

Binary search tree

The benchmark cases

```

for tree_size in (10, 100, 1,000, 5,000) 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

```

Results

Size	Time
10	699.02 ns
100	32.018 us
1000	3.4898 ms
5000	99.200 ms

The result shows that when size equals 1000, the binary search tree takes longer time than the red black tree and AVL tree with 10000 data.

Red-black tree and AVL tree

The benchmark cases

```

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

```

Results

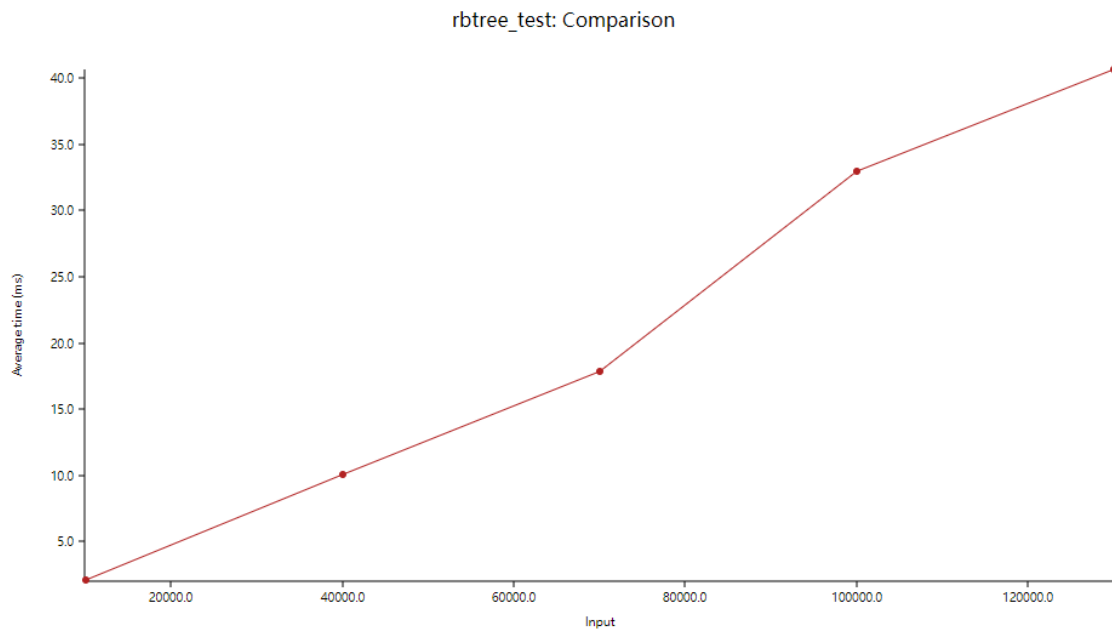
For the results, we run the benchmark code ten times and take the average value

Size	Red-black Tree	AVL Tree
10,000	2.1634 ms	3.8656 ms
40,000	10.108 ms	15.781 ms
70,000	17.897 ms	28.721 ms
100,000	32.948 ms	42.683 ms
130,000	40.769 ms	62.057 ms

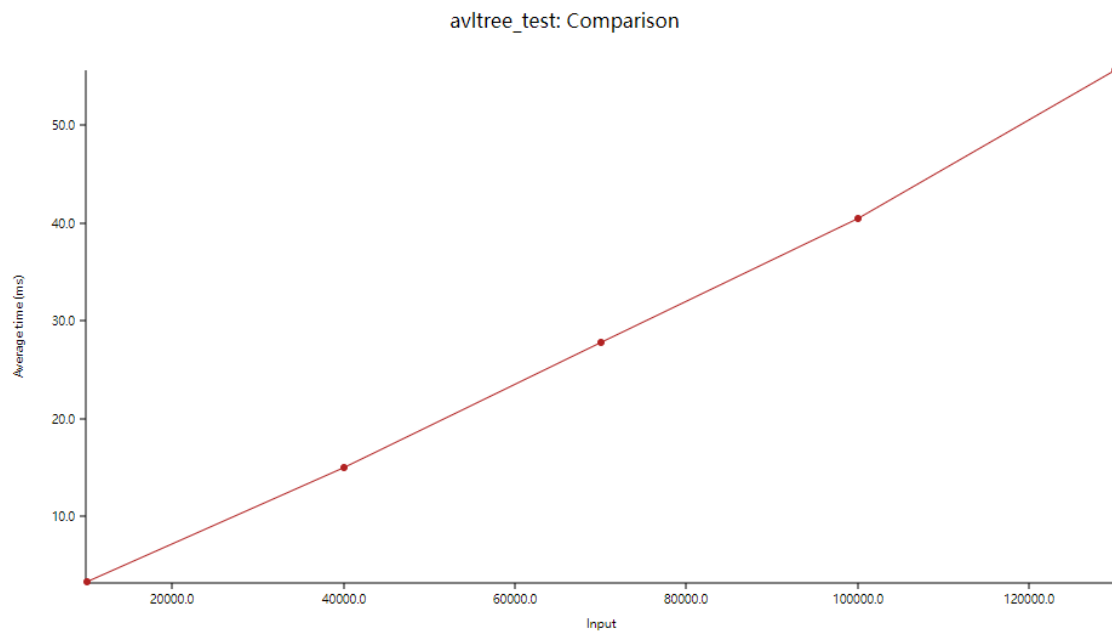
Line Chart

This chart shows the mean measured time for each function as the input (or the size of the input) increases.

Line Chart



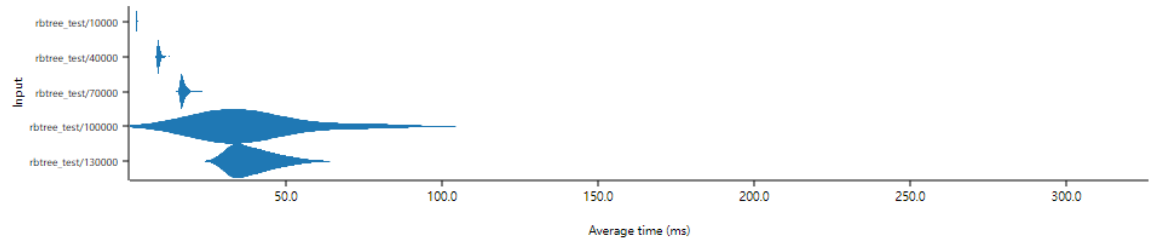
Line Chart



Violin Plot

This chart 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.

rbtree_test: Violin plot



avltree_test: Violin plot

