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

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.

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.

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

###

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

Red-black Tree

The structure for red-black tree is

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

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

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:

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

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

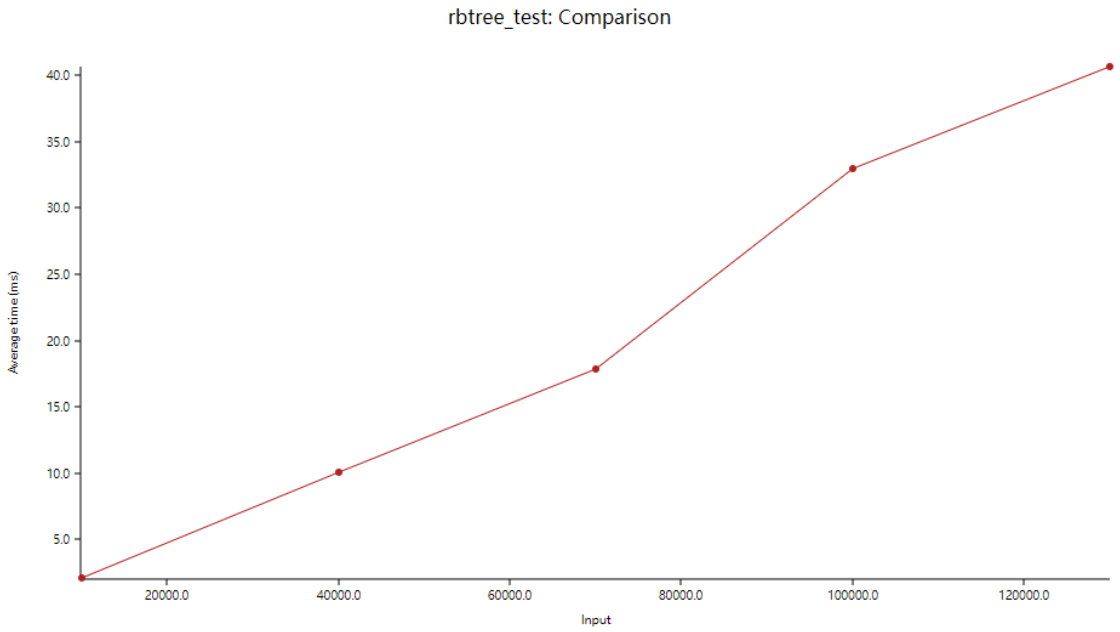
For the results, we run the benchmark code ten times and take the average value

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

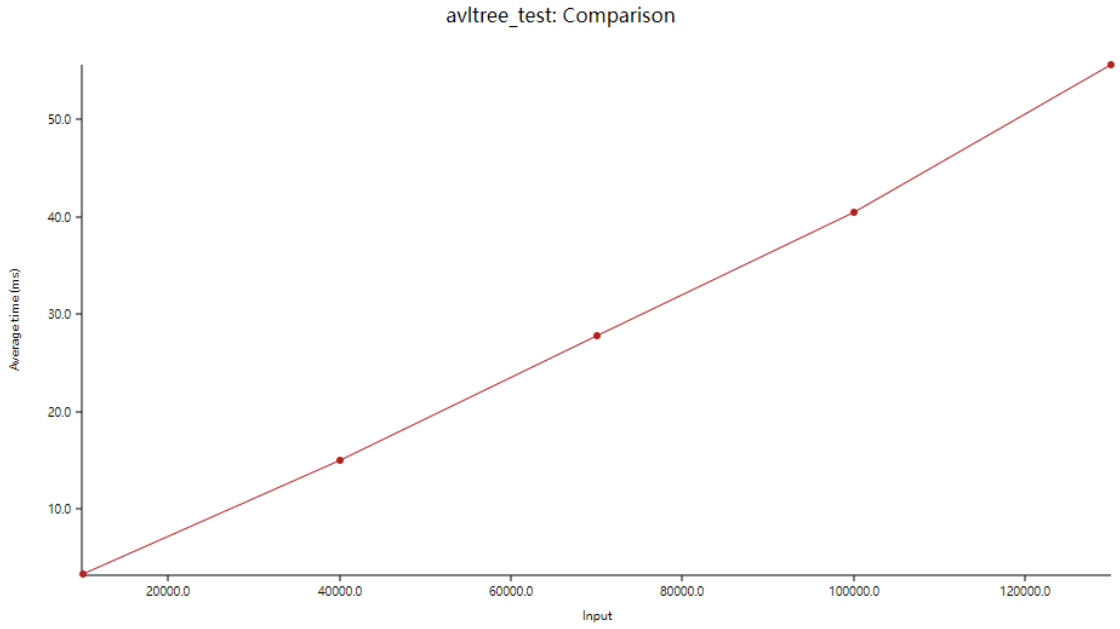
Line Chart

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

Line Chart

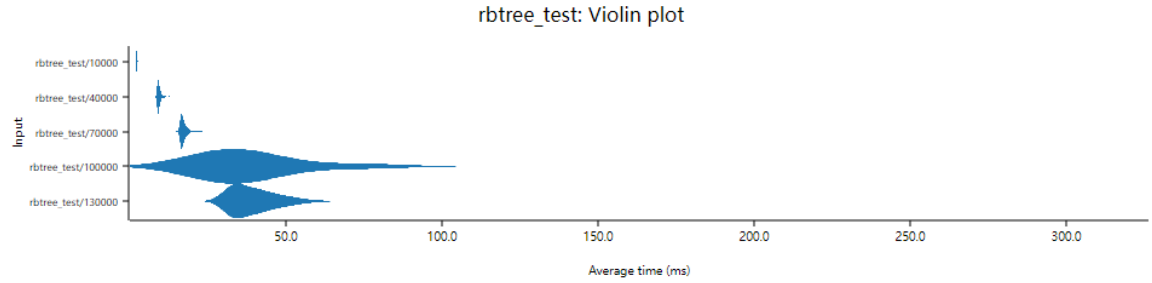


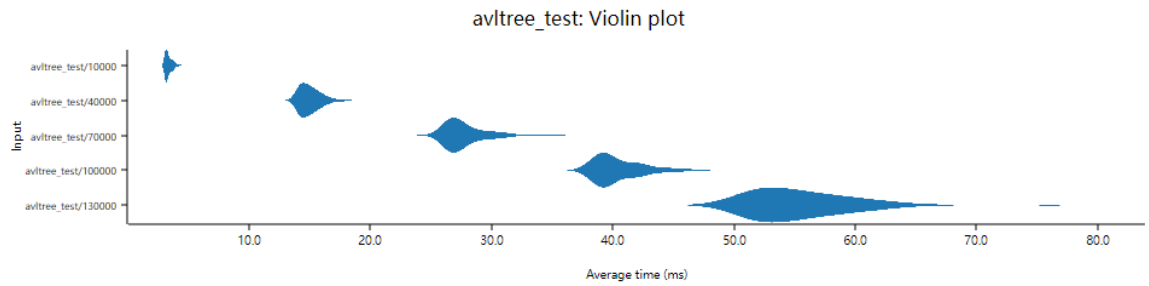
Line Chart



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
rbtree_test/10000
rbtree_test/40000
rbtree_test/70000
rbtree_test/100000
rbtree_test/130000
Input
50.0
100.0
150.0
200.0
250.0
300.0
Average time (ms)


avltree_test: Violin plot
avltree_test/10000
avltree_test/40000
avltree_test/70000
avltree_test/100000
avltree_test/130000
Input
10.0
20.0
30.0
40.0
50.0
60.0
70.0
80.0
Average time (ms)