EC504 Final Project:

Efficiency Comparison between AVL and Red-Black Tree

Ziyan Chen, Zhiqing(Joey) Wang, Jìa Wilkins, Hin Lui Shum

Boston University, College of Engineering

Table of Contents

0 Abstract **3**
1 Introduction **3**
2 AVL Trees and properties **4**
2.1, BST properties 4
2.2, Rotation 4
3 Red-Black Tree and properties **4**
3.1, Establishment and properties 5
3.2 Inserting a node 5
3.3 Deleting a node 7
4 B+ Tree (Excluded from Main Comparison) **9**
4.1 Tree Construction, Insertion, & Deletion 10
4.2 Reason for Exclusion from the Main Comparison 10
5 Similarities and Differences between AVL Trees and RB Trees **10**
6 Efficiency Comparison **12**
6.1 Testbench and Test Method 12
6.2 Data Results and Visualization 12
6.2.1 RBT with AVL Tree 12
6.3 Data Analysis 14
6.3.1 Efficiency Analysis Between AVL and RB Trees 14
6.3.2 Pros and Cons Between AVL and RB Trees 15
7 Conclusion **16**
8 Appendix **16**

0 Abstract

Our final project presents a time efficiency analysis of two self-balancing binary search tree (BST) structures: the AVL Tree and the Red-Black Tree. The primary objective is to compare their efficiency in performing various operations, including building a tree, inserting and deleting nodes, finding maximum & minimum nodes, and searching for random elements in differently sized trees. Our findings offer practical guidance for software developers seeking to implement an appropriate tree structure tailored to fit specific computational scenarios.

1 Introduction

The efficiency of data structures in computer algorithms is a critical aspect of software engineering as it directly impacts the time performance and space scalability of a given application. Among the myriad of existing data structures, we chose self-balancing binary search trees, specifically AVL Trees and Red-Black Trees, as our project's main focus due to their ability to maintain balanced heights for ensuring optimal operation times.

Initially, our exploration also encompassed B+ trees, a non-binary tree structure known for its efficiency in database and file system indexing like mySQL. However, due to its fundamental differences from binary search trees, we decided to narrow our focus to provide a more direct and meaningful comparison between these two closely related structures.

This report explores AVL and Red-Black trees in detail, examining their structural properties, balancing mechanisms, and operational algorithms. The aim is to conduct an efficiency comparison of these tree types across various operations at different sizes—specifically focusing on their time efficiency in tree construction, searching, insertion, and deletion of nodes. By evaluating their performance in a controlled test environment, the report seeks to uncover the practical implications of their theoretical differences. The resulting analysis will provide valuable insights into the suitability of each tree type for different computational scenarios.

2 AVL Trees and properties

Published by Georgy Adelson-Velsky and Evgenii Landis in the 1962 edition of the *Proceedings of the USSR Academy of Sciences*, the eponymous AVL tree offers a self-balancing binary search tree that minimizes search time through the application of various rotation algorithms during insert and delete operations such that it maintains as short a tree as possible.

2.1, BST properties

As a subtype of binary search tree, it inherits several properties from the generalized BST, namely that each node has at most two children, with each resultant subtree recursively defined as another BST such that every node in the left BST is smaller than or equal to the parent node and every node in the right BST is greater than the parent node. When building and searching through an AVL tree, one takes the same approach as through creating a plain BST: comparing the value of interest and traversing down the tree left or right until a relevant end node.

However, a major weak-point of the regular BST is that there is no guarantee that the height of the tree will always be bounded by log(n); in the worst case, if the nodes are inserted in sorted order, the tree will have a height of n. AVL trees address this flaw by performing a self-balancing operation whenever the tree begins to get too far out of balance.

2.2, Rotation

As defined above, the children of any given node in an AVL tree are themselves the roots of a recursively defined AVL tree. When the difference in height between two subtrees of any node in an AVL tree becomes greater than one, a specific rotation can be performed to bring the tree back into balance. Depending on a given node's relationship to its parents and its unbalanced children, one of four rotations are performed: Left, Right, Left-Right, and Right-Left. In general, a lineage of three problem nodes are first identified and then rearranged such that they are flat, after which the four relevant subtrees (which may be null) are reattached in valid BST order.

3 Red-Black Tree and properties

Invented in 1978 by Leonidas J. Guibas and Robert Sedgewick, the Red-Black Tree (RBT) is a tree-shaped data structure with several helpful properties. Achieved by coloring every node in either red or black and following a certain ruleset, RBTs can maintain a relatively balanced shape, which enhances performance in overall tree operations.

3.1, Establishment and properties

To build a RBT, we should follow the following rules:

1. RBT should be a BST first
2. Every node is red or black
3. All NIL nodes are black
4. Red nodes don't have red children (so red nodes' children must be black)
5. Every number from a given node to any of its descendant NIL nodes goes through the same number of black nodes.

For rule 5, mark the black node number on a certain path from a node x to leaf nodes as black height, note as bh(x).

3.2 Inserting a node

For inserting nodes into a RBT, we need one more step: rearranging color to fit the rules in 3.1.
Here is the pseudo code for inserting a node into a RBT:

```
RB-INSERT($T, z$)
$y = T.nil$
$x = T.root$
while $x \neq T.nil$
    $y = x$
    if $z.key < x.key$
        $x = x.left$
    else $x = x.right$
$z.p = y$
if $y == T.nil$
    $T.root = z$
elseif $z.key < y.key$
    $y.left = z$
else $y.right = z$
$z.left = T.nil$
$z.right = T.nil$
$z.color$ = RED
RB-INSERT-FIXUP($T, z$)
```

fig 3.2.1: Pseudo code for RBT node insertion

Notice that coloring z as red may violate the rule before, to solve this problem, we use insert-fixup function shown in pseudo code below to rearrange the colors of each node.

```
RB-INSERT-FIXUP(T, z)
while z.p.color == RED
    if z.p == z.p.p.left
        y = z.p.p.right
        if y.color == RED
            z.p.color = BLACK
            y.color = BLACK
            z.p.p.color = RED
            z = z.p.p
        else if z == z.p.right
                z = z.p
                LEFT-ROTATE(T, z)
            z.p.color = BLACK
            z.p.p.color = RED
            RIGHT-ROTATE(T, z.p.p)
    else (same as then clause
            with "right" and "left" exchanged)
T.root.color = BLACK
```

fig 3.2.2 Pseudo code for RBT self-balancing after insertion

Here are 3 different situations:

1) **z's uncle y is red**

This situation shows in the picture (fig 3.2.4) below. Since z.p.p is black, we color z.p and y black to resolve conflict between z and z.p, and color z.p.p red to maintain rule 5.

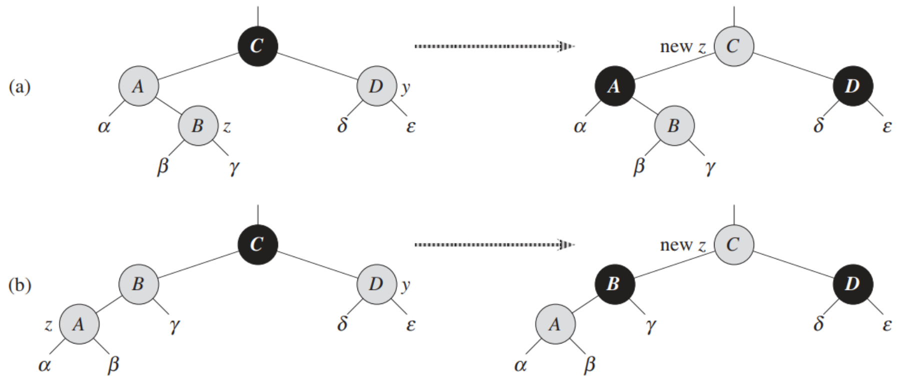


fig 3.2.4: Visualization for situation 1

2) **z's uncle y is black and z is a right child**
3) **z's uncle y is black and z is a left child**

These situations are shown in the picture [1] below. These situations are similar. If we are in case 2 where z is a right child, we perform a left rotation to make it case 3. This rotation won't affect the existing color. For case 3, we perform a right rotation, which doesn't violate rule 5. Then the while loop terminates, since there's no longer 2 red nodes in a row.

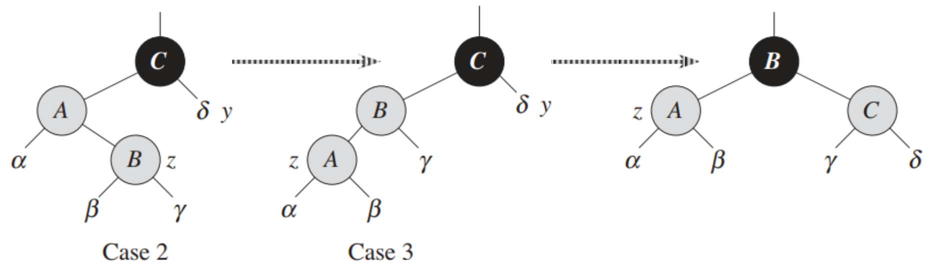


fig 3.2.5: Visualization for situation 3

3.3 Deleting a node

Deleting a node is similar to insertion before, the problem comes when color rules are violated. Again we need a fixup function to make colors in place.

Here is the pseudo code in fig 3.3.2 for deleting a code in RB Tree:

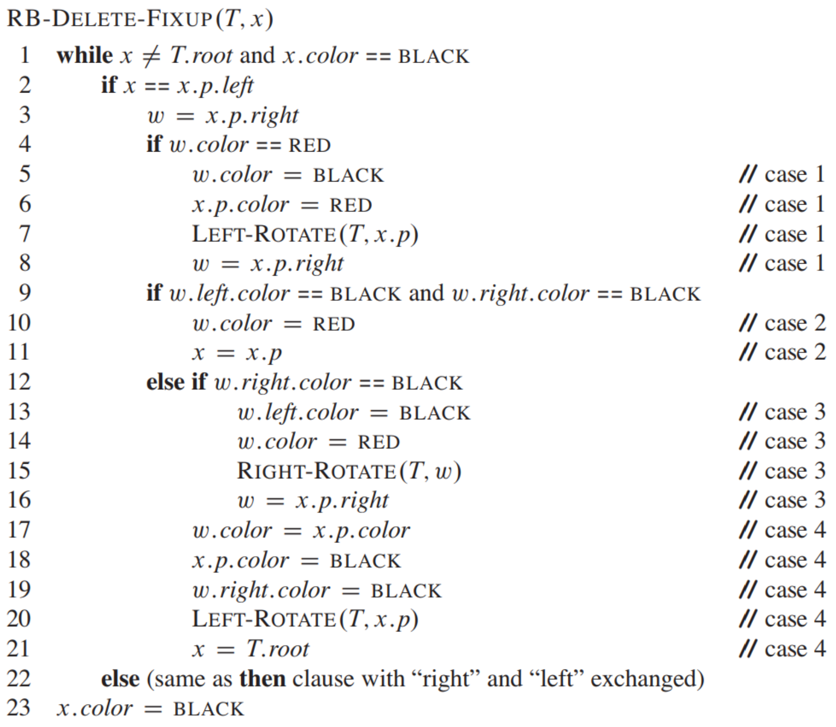
```
RB-TRANSPLANT(T, u, v)
if u.p == T.nil
    T.root = v
elseif u == u.p.left
    u.p.left = v
else u.p.right = v
v.p = u.p
```

fig 3.3.1: Pseudo code for Red-Black Tree transplant function

```
RB-DELETE(T, z)
y = z
y-original-color = y.color
if z.left == T.nil
    x = z.right
    RB-TRANSPLANT(T, z, z.right)
elseif z.right == T.nil
    x = z.left
    RB-TRANSPLANT(T, z, z.left)
else y = TREE-MINIMUM(z.right)
    y-original-color = y.color
    x = y.right
    if y.p == z
        x.p = y
    else RB-TRANSPLANT(T, y, y.right)
        y.right = z.right
        y.right.p = y
    RB-TRANSPLANT(T, z, y)
    y.left = z.left
    y.left.p = y
    y.color = z.color
if y-original-color == BLACK
    RB-DELETE-FIXUP(T, x)
```

fig 3.3.2: Pseudo code for Red-Black Tree delete function

For a RBT, deleting a node may cause 4 types of situation as the following pseudo code in fig 3.3.3:



```
RB-DELETE-FIXUP(T, x)
while x ≠ T.root and x.color == BLACK
    if x == x.p.left
        w = x.p.right
        if w.color == RED
            w.color = BLACK                                  // case 1
            x.p.color = RED                                  // case 1
            LEFT-ROTATE(T, x.p)                              // case 1
            w = x.p.right                                    // case 1
        if w.left.color == BLACK and w.right.color == BLACK
            w.color = RED                                    // case 2
            x = x.p                                          // case 2
        else if w.right.color == BLACK
                w.left.color = BLACK                         // case 3
                w.color = RED                                // case 3
                RIGHT-ROTATE(T, w)                           // case 3
                w = x.p.right                                // case 3
            w.color = x.p.color                              // case 4
            x.p.color = BLACK                                // case 4
            w.right.color = BLACK                            // case 4
            LEFT-ROTATE(T, x.p)                              // case 4
            x = T.root                                       // case 4
    else (same as then clause with "right" and "left" exchanged)
x.color = BLACK
```

fig 3.3.3: Pseudo code for Red-Black Tree self-balancing after deletion

Below [1] are the 4 situations when deleting a node from RBTree. Notice that only case 2 causes the loop, while other cases can change into case 2 by changing colors and rotations. In case 2, the extra black represented by the pointer x moves up the tree by coloring node D red and setting x to point to node B. If we enter case 2 through case 1, the while loop terminates because the new node x is red-and-black, and therefore the value c of its color attribute is red.

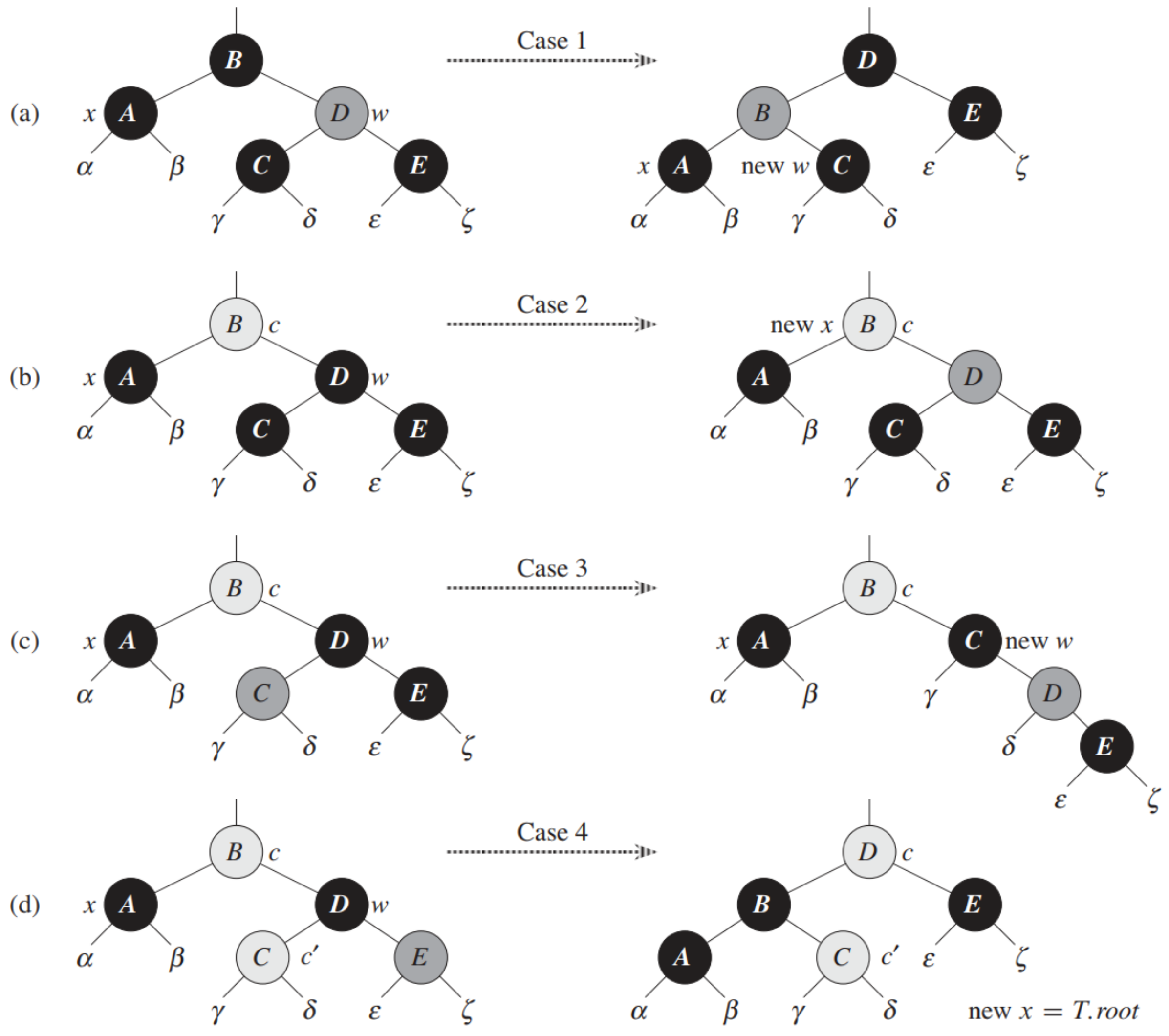


fig 3.3.4: Visualization of Red-Black Tree Deletion

4 B+ Tree (Excluded from Main Comparison)

B+ Trees are an essential data structure in database and file system indexing. Unlike AVL and Red-Black Trees, which are binary trees, B+ Trees are multi-way trees. This means that each node in a B+ Tree can have more than two children, making them well-suited for systems that handle large blocks of data. B+ Trees are designed for efficient range searching and sequential data access, with all values stored in the leaf nodes and internal nodes serving as pointers to these leaves.

4.1 Tree Construction, Insertion, & Deletion

Building a B+ Tree involves creating a balanced multiway tree structure. Initially starting as an empty tree, it grows by splitting nodes as more elements are added. Each non-leaf node in the tree contains keys that act as separators for the ranges of values stored in the nodes beneath it.

Insertion in a B+ Tree is a process that maintains the tree's balanced nature. When a new element is added, it is inserted into the appropriate leaf node. If this causes the leaf node to overflow (exceed its maximum number of elements), it is split into two nodes, and the median value is promoted to the parent node. This process may cascade up the tree if necessary, ensuring that the tree remains balanced.

Deletion, like insertion, requires maintaining the tree's balance. When an element is removed, if it causes a node to have fewer elements than the minimum required, the tree undergoes a rebalancing process. This might involve merging nodes or redistributing elements between neighboring nodes, followed by appropriate adjustments up the tree.

4.2 Reason for Exclusion from the Main Comparison

Despite its relevance and unique properties, the B+ Tree was ultimately excluded from our main comparison with AVL and Red-Black Trees. The fundamental structural differences between multi-way and binary trees make a direct efficiency comparison challenging. B+ Trees excel in scenarios distinct from those where binary search trees are typically employed, such as disk-based storage systems.

Our report, therefore, concentrates on AVL and Red-Black Trees — both binary search trees — allowing for a more relevant and in-depth comparison of their properties, efficiency, and applicability in various computational scenarios.

5 Similarities and Differences between AVL Trees and RB Trees

AVL trees and Red-Black Trees are both self-balancing binary search trees, but they have some key differences and similarities in their algorithms and properties:

5.1 Similarities

Self-Balancing:

Both AVL and Red-Black Trees are self-balancing. Each node in both trees maintains extra information to help keep the tree balanced during insertions and deletions.

Binary Search Trees:

They are both binary search trees (BSTs). This means that for every node, the nodes in the left subtree are all less than or equal to the node, and the nodes in the right subtree are all greater.

Complexity:

Both offer O(log n) time complexity for insertion, deletion, and search operations, where n is the number of nodes in the tree.

Rotations:

They use tree rotations as a fundamental operation to maintain balance. Both perform single and double rotations.

5.2 Differences:

Balancing Criteria:

AVL trees use the balance factor of nodes (the height difference between the left and right subtrees) to determine balance. An AVL tree is balanced if, for every node, the difference in the height of the left and right subtrees is at most 1.

Red-Black trees follow a different set of rules involving the color of nodes (red or black). These rules include that every path from a node to its descendant NULL nodes must have the same number of black nodes, and red nodes cannot have red children (no two red nodes can be adjacent in lineage).

***Balancing Operations*:**

AVL trees may require more frequent rebalancing. They perform balancing on almost every insertion and deletion, which can be more frequent than Red-Black Trees.

Red-Black trees tend to have fewer balancing operations. They are more forgiving than AVL trees regarding the balance factor but still maintain a roughly balanced tree.

***Height*:**

AVL trees are more rigidly balanced and thus, have a smaller height compared to Red-Black Trees. This can result in faster lookups.

Red-Black trees may have a larger height in the worst case, but they often perform better in insertion and deletion cases due to fewer rotations.

***Usage Scenarios*:**

AVL trees are preferred when search operations are more frequent than insertions and deletions because they are more strictly balanced and thus have shorter search paths.

Red-Black trees are often preferred in scenarios where insertions and deletions are more frequent, as they balance more quickly than AVL trees.

In summary, the choice between an AVL tree and a Red-Black Tree depends on the specific needs of the application, particularly the frequency of different types of operations (search, insert, delete) and the necessity for maintaining minimal tree height.

6 Efficiency Comparison

6.1 Testbench and Test Method

In our experiment, we tested our data structures on the school's SCC cluster. The Hardware information is shown as follows:

Linux Core: Linux scc1 4.18.0-513.9.1.el8_9.x86_64 #1
SMP Sat Dec 2 05:23:44 EST 2023 x86_64 x86_64 x86_64 GNU/Linux

CPU: 32 Intel(R) Xeon(R) Gold 6242 CPU @ 2.80GHz,16 core

Memory: 16GB

6.2 Data Results and Visualization

6.2.1 RBT with AVL Tree

The table of results is the average time of 20 experiments obtained with different datasets. The datasets were generated using *numpy.random.randint*, the dataset contains only integers and has a uniform distribution.

AVL tree Dataset size	Build (seconds)	Search (seconds)	Minimum (seconds)	Maximum (seconds)
10	1.28E-06	1.69E-07	8.04E-08	8.01E-08
100	1.31E-05	8.24E-08	7.54E-08	7.84E-08
1000	0.000139799525	8.64E-08	7.93E-08	8.30E-08
10000	0.0019633785	1.08E-07	1.09E-07	9.90E-08
100000	0.03369510833	2.00E-07	3.23E-07	1.53E-07
1000000	0.033027805	2.38E-07	3.44E-07	1.66E-07
10000000	0.0339503025	2.68E-07	3.22E-07	3.22E-07
Red-Black tree Dataset size	Build (seconds)	Search (seconds)	Minimum (seconds)	Maximum (seconds)
10	2.34E-06	1.02E-07	7.15E-08	7.77E-08
100	1.45E-05	8.93E-08	7.91E-08	8.09E-08
1000	0.00016454445	1.11E-07	8.59E-08	9.66E-08
10000	0.001994779	1.39E-07	1.23E-07	1.37E-07
100000	0.02931233125	3.09E-07	2.16E-07	3.13E-07
1000000	0.029569399	2.76E-07	2.07E-07	2.93E-07
10000000	0.02971608	2.64E-07	2.23E-07	2.92E-07

fig 6.2.1: Time efficiency comparison of RBT vs AVL at different dataset size

We can graph the data in the table above as a line graph:

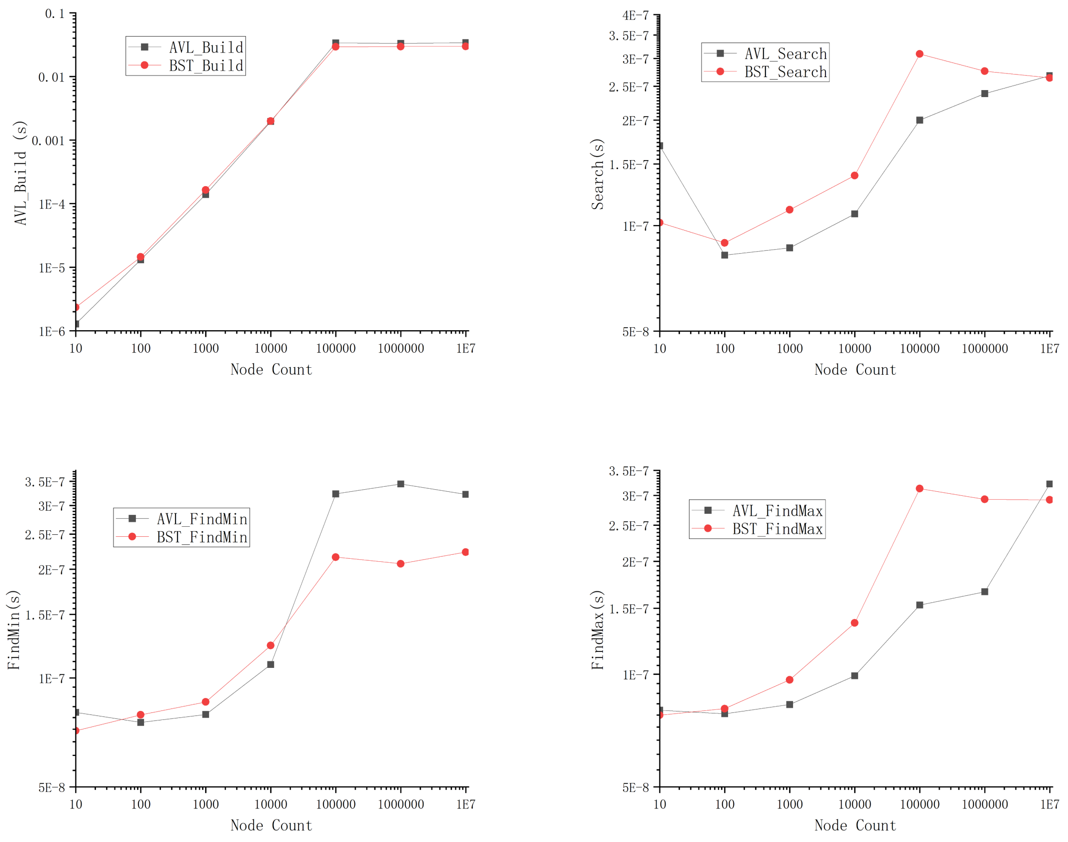


fig 6.2.2: Visualization of RBT vs AVL at different dataset size

6.3 Data Analysis

6.3.1 Efficiency Analysis Between AVL and RB Trees

➢ Build Operation

In comparing the build times of AVL and RB trees across various sizes, AVL trees demonstrate a slightly faster build time for smaller sets of nodes. Specifically, this advantage is observed in AVL trees for datasets containing 10, 100, and 1,000 nodes. However, as the dataset size increases, particularly at 10,000 nodes and beyond, RB trees begin to exhibit a faster build time compared to AVL trees. The less stringent balancing rules of Red-Black trees result in fewer rebalancing operations during insertions. This feature will become obvious in larger dataset sizes.

- ➢ Search Operation

When performing the search operation, there is no significant difference between the timing of the AVL tree and the Red-Black tree. From the table, the search operation time in **AVL trees** appears to increase with the size of the dataset. This increase is gradual and remains within the same order of magnitude (10^-7 seconds) even as the dataset size grows significantly. This suggests that AVL trees maintain their efficiency in search operations even with large datasets, aligning with their logarithmic time complexity characteristics. However, the search operation time in Red-Black trees also tends to increase with the size of the dataset, though the increase is not strictly linear. The time remains within the order of magnitude of (10^-7 seconds) for most dataset sizes, also indicating good efficiency in search operations. Interestingly, there exists a noticeable jump in the search time for a dataset size of 100,000, after which it stabilizes for larger datasets. This suggests two possible factors:

- ○ While Red-Black trees efficiently handle large datasets, there might be specific dataset sizes where the search operation time can vary more significantly.
- ○ It might be caused by the cache strategies of the experiment environment.

- ➢ Find Minimum Operation

For smaller datasets (10, 100, and 1,000 nodes), the minimum operation times of AVL and Red-Black trees are quite similar, with AVL trees showing marginally longer times in the smallest dataset. As the dataset size increases, the AVL tree's minimum operation time increases more significantly compared to the Red-Black tree, particularly at the 100,000 and 1,000,000 nodes level. In even larger datasets (10,000,000 nodes), the minimum operation times converge again, with AVL and Red-Black trees showing similar performance. In summary, both AVL and Red-Black trees maintain efficient minimum operation times across all dataset sizes, with Red-Black trees having a slight advantage in larger datasets.

- ➢ Find Maximum Operation

For smaller datasets (10 and 100 nodes), the maximum operation times are quite similar for both AVL and Red-Black trees, with negligible differences. As the dataset size increases, the operation times for both tree types increase, but Red-Black trees show a slightly higher increase in time compared to AVL trees, especially on the order of 10,000,000 nodes. Overall, both AVL and Red-Black trees maintain efficient maximum operation times across all dataset sizes, with times consistently within the order of (10^-7 seconds). The increase in operation time with larger datasets is expected due to the larger tree size and complexity.

6.3.2 Pros and Cons Between AVL and RB Trees

Based on the results of our experiments, we have concluded the following Pros and Cons of AVL trees and Red-Black trees.

AVL Tree Pros:

- ➔ *Better Search Performance*: AVL trees generally have slightly better search performance, especially in smaller datasets. This is due to their more stringent balancing, which keeps the trees shorter and more uniform.
- ➔ *Efficient Minimum and Maximum Operations*: For smaller datasets, AVL trees perform minimum and maximum operations very efficiently, with times comparable to Red-Black trees.

AVL Tree Cons:

- ➔ *Performance in Larger Datasets:* As the dataset size increases, AVL trees show a more significant increase in operation times for minimum and maximum operations compared to Red-Black trees. This could be a downside for applications with very large datasets.
- ➔ *Potentially Higher Rebalancing Cost*: Due to stricter balancing rules, AVL trees might require more rotations during insertion and deletion, which can lead to higher computational costs in dynamic scenarios.

Red-Black Tree Pros:

- ➔ *Consistent Performance Across Datasets*: Red-Black trees maintain more consistent performance across different dataset sizes, particularly for minimum and maximum operations. They handle large datasets slightly more efficiently than AVL trees.
- ➔ *Lower Rebalancing Cost*: With looser balancing rules, Red-Black trees generally require fewer rotations during insertions and deletions, which can be advantageous in scenarios with frequent data updates.

Red-Black Tree Cons:

- ➔ *Slightly Slower Search in Smaller Datasets*: Red-Black trees have marginally slower search times in smaller datasets compared to AVL trees. However, this difference is quite minimal and may not be significant in most practical applications.
- ➔ *Variable Performance in Specific Operations*: While generally consistent, the performance of Red-Black trees in specific operations (like minimum and maximum) can vary more noticeably with changes in dataset size compared to AVL trees.

7 Conclusion

In this paper we mainly introduce 3 types of tree-size algorithms, AVL Tree, Red-Black Tree and B+ Tree, introduce how to build and modify them, and compare the efficiency in different data scales between the first two. From data we can know that, for our testbench, Red Black Tree is more efficient on building the trees and searching minimum node when facing large scales of data (>10000) due to its special structure.

8 Appendix

[1] Cormen, T.H., Leiserson, C.E., Rivest, R.L. and Stein, C., 2022. *Introduction to algorithms*. MIT press.