FGV/EMAp

Final Project - Inverted Index and Comparative Analysis

Arthur Rabello Oliveira[1], Gabrielle Mascarelo, Eliane Moreira, Nícolas Spaniol, Gabriel Carneiro

Abstract

We present the implementation and comparative analysis of three fundamental data structures for indexing and searching textual documents: the Classic Binary Search Tree (BST), the AVL Tree, and the Red-Black Tree (RBT). Each structure was implemented with its core operations, including insertion and search. Unit tests were developed to validate the correctness and performance of these implementations. We also provide a further comprehensive comparative study of the three trees based on their time complexity, balancing efficiency, and suitability for document indexing. The results demonstrate the trade-offs between implementation complexity and query performance, offering insights into the practical considerations for choosing appropriate search tree structures in information retrieval systems.

Contents

[1]Escola de Matemática Aplicada, Fundação Getúlio Vargas (FGV/EMAp), email: arthur.oliveira.1@fgv.edu.br

1. Introduction

1.1. Context

Humanity now produces more text in a single day than it did in the first two millennia of writing combined. Search engines must index billions of web pages, e-commerce sites hundreds of millions of product descriptions, and DevOps teams terabytes of log lines every hour. Scanning those datasets sequentially would be orders of magnitude too slow; instead, virtually all large-scale retrieval systems rely on an **inverted index**, a data structure that stores, for each distinct term, the identifiers of documents in which it appears.

1.2. Problem Statement

While the logical view of an inverted index is a simple dictionary, its physical realisation must support two conflicting workloads:

- **Bulk ingestion** of millions of (term, docID) pairs per second.
- **Sub-millisecond** queries for ad-hoc keyword combinations.

Choosing the proper data structure is therefore a trade-off between build-time cost and implementation complexity.

1.3. Objectives

1. **Implement** BST, AVL and Red-Black Tree insertion, deletion and search in C++.
2. **Build** an inverted index over a 10 k-document corpus with each tree.
3. **Measure** build time, query latency, and memory usage under identical workloads.
4. **Discuss** which structure best balances simplicity and performance for mid-scale information-retrieval tasks.

2. Motivation

2.1. Why Inverted Index?

Domain	Real-world system	Role of the inverted index
Web search	Google, Bing, DuckDuckGo	Core term → page mapping
Enterprise search	Apache Lucene & Elasticsearch	Underlying index files and query engine
Database systems	Postgres GIN & CockroachDB	Full-text and JSONB indexing
Observability / Logs	Splunk, OpenObserve	Fast filtering / aggregation of terabyte-scale logs
Bioinformatics	VariantStore, PAC	Searching billions of DNA k-mers
Operating systems	Linux schedulers & timers	Kernel subsystems use RBTs—conceptually an inverted index over time or PID keys

These examples shows the ubiquity of inverted indexes in modern era. From web search engines to bioinformatics, inverted indexes are the backbone of efficient information retrieval systems.

3. Definitions

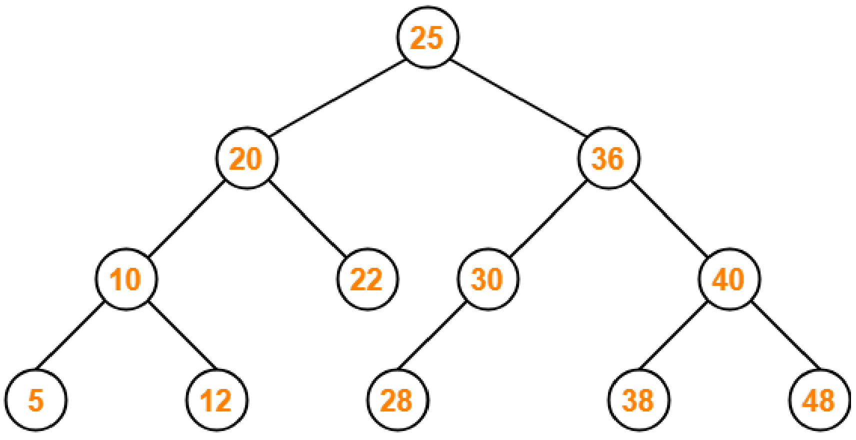
3.1. BST

In graph theory, a tree is a connected acyclic graph. A Binary Search Tree (BST) is a tree with the following properties:

1. Each node has at most two children, referred to as the left and right child.
2. The left child of a node contains only nodes with keys less than the node's key.
3. The right child of a node contains only nodes with keys greater than the node's key.

These properties are true $\forall$ node $\in T = (V, E)$.

Here are some examples of BSTs:



Binary Search Tree

Figure 1: Example of a Binary Search Tree

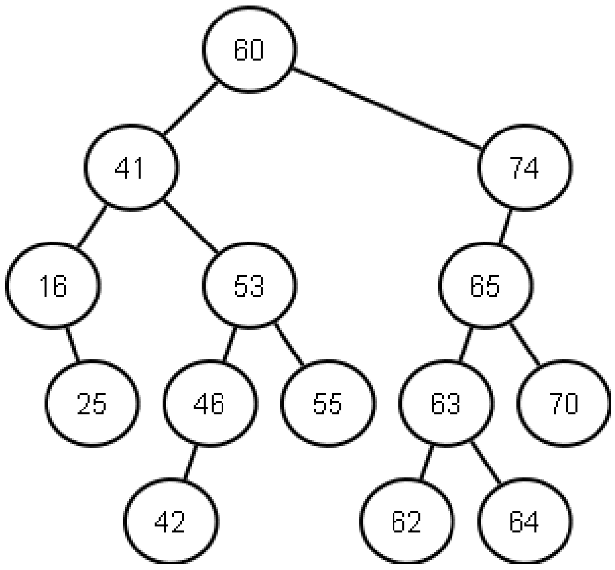


Figure 2: Another example of a Binary Search Tree

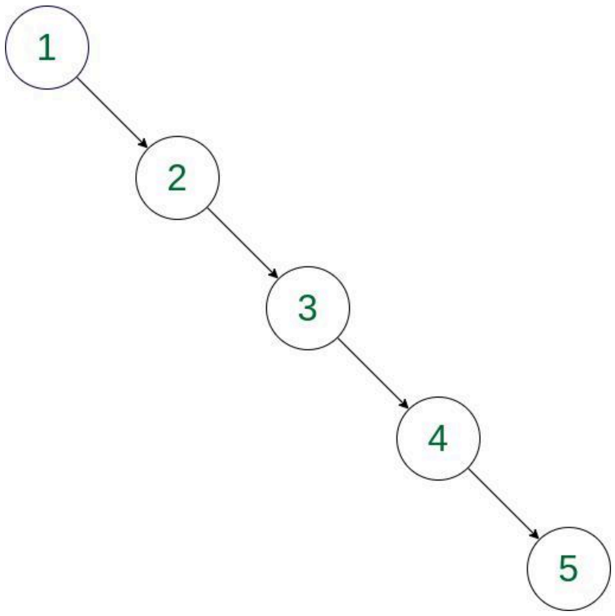


Figure 3: Degenerated Binary Search Tree

Figure 1 and Figure 2 are ordinary BST's, while Figure 3 is a *degenerated* BST. Which is a tree in which every node has at most one child, making it a linear chain of nodes. This is the worst case for a BST, due to computational complexity of some operations.

3.2. AVL Tree

The AVL tree can be seen as a solution to the degenerated BST problem. It is a self-balancing binary search tree, built so the heights of the two child subtrees of any node differ by at most one. This balance condition ensures that the tree remains approximately balanced, preventing the worst-case linear chain structure.

In order to maintain this balance rule, we define the balance factor of a node $n \in V$ as:

$$B_F(n) = h(\text{RightSubtree}(n)) - h(\text{LeftSubtree}(n)) \tag{1}$$

If $B_F(n) < 0$, the tree is categorized as "left-heavy", if $B_F(n) > 0$, the tree is "right-heavy", and if $B_F(n) = 0$, the tree is "balanced". For the event $B_F(n) \neq 0$, *rotations* are applied to restore balance.

Rotations are local tree restructuring operations that change the structure of the tree without violating the binary search tree property. There are 3 types of rotations:

1. Left Rotation: Applied when a right-heavy subtree needs balancing.
2. Right Rotation: Applied when a left-heavy subtree needs balancing.
3. Permutations of Left and Right Rotations: Used to balance more complex imbalances.

This is a very good website to better visualize the creation of a tree. We recommend the user to play with it, inserting and deleting nodes, to see how the tree is balanced.

The AVL Tree was created by Georgy Adelson-Velsky and Evgenii Landis in 1962, and it was the first self-balancing binary search tree. The name "AVL" comes from the initials of their last names. More information can be found here. Below is an example of the creation of an AVL tree:

Figure 4: Creation of an AVL Tree

3.3. RBT Tree

The RBT is another self-balancing BST, which uses a different balancing strategy than the AVL. RBT stands for Red-Black Tree. The Red-Black stands for the additional propery that each node is colored either red or black, and the tree satisfies the following properties:

1. The root node is always black.
2. Every leaf node is black.
3. If a node is red, then both of its children must be black (no two red nodes can be adjacent).
4. Every path from a node to its descendant leaf nodes must have the same number of black nodes (black-height).

Properties 3 and 4 forces long paths to pick up extra black nodes, capping the tree's height. Here is an example:

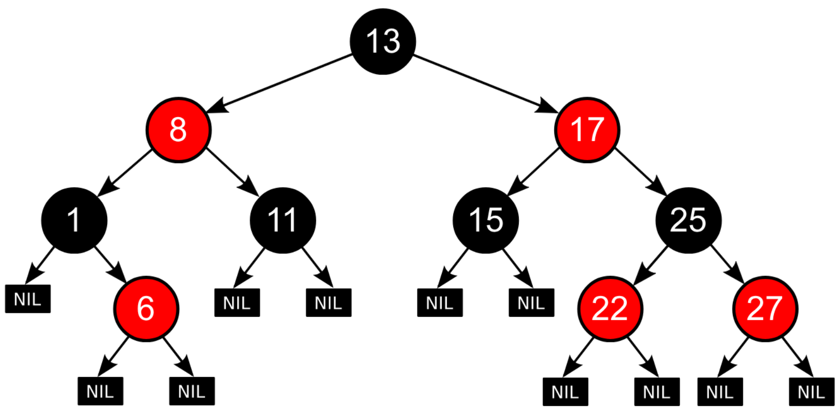


Figure 5: Example of a Red-Black Tree

4. Implementations

4.1. Binary Search Tree (BST)

Since the AVL and RBT trees are both subsystems of the classic BST, we have implemented the classic BST in the `tree_utils` module, later used as based for the AVL and BST

4.1.1. Algorithms

Here are all the functions written for the classic BST, and here is the header file with the corresponding documentation. The list of functions is:

- `createNode`
- `createTree`
- `search`
- `deletionPostOrder`
- `destroy`
- `calculateHeight`
- `updateHeightUp`

4.1.2. Complexity Analysis

Below is a full complexity analysis:

`createNode`:

Clearly $O(1)$, the function allocates memory for a new node and initializes it, independent of the size of the tree.

`createTree`:

Also $O(1)$, an empty BST is allocated and initialized with a `nullptr` root.

`search`:

The search operation, unavoidably, has a time complexity of $O(h)$, with h being the height of the tree. The function follows a single root-to-leaf path in the BST, making at most h comparisons. No recursion, only a few local variables.

`deletionPostOrder`:

This function is $O(\varphi)$, where φ is the size of the subtree rooted at the node to be deleted. It is a classic post-order traversal: each node is visited & deleted precisely once.

`destroy`:

This functon simply calls `deletionPostOrder` on the root node, so it is $O(h)$.

`calculateHeight`:

This is $O(k)$, with $k =$ subtree size. It recursively explores both children of every node once to compute `max(left,right)+1`.

`updateHeightUp`:

This function is $O(h)$. It iterates upward, recomputing height until it stops changing or reaches the root; at most h ancestor steps, no recursion on children.

4.2. AVL Tree

4.2.1. Algorithms

The functions implemented *stritly* for the AVL can be found here, and the header file with the corresponding documentation here. We have used many of the BST functions, as previously stated. The list of AVL-functions is:

- `getHeight(Node*)`
- `bf(Node*)` [balance factor]
- `leftRotation(BinaryTree&, Node*)`
- `rightRotation(BinaryTree&, Node*)`
- `insert(BinaryTree& binary_tree, const std::string& word, int documentId)`
- `remove(Node*&, key)`
- `remove(BinaryTree&, key)`
- `updateHeightUp(Node*)`
- `clear(BinaryTree*)`
- `printInOrder`
- `printLevelOrder`

4.2.2. Complexity Analysis

Below is a full complexity analysis of the AVL functions:

`getHeight`:

$O(1)$, it simply returns the stored `height` of a node.

`bf(Node*)`:

Also $O(1)$, this is nothing more than a subtraction of two integers.

`leftRotation(BinaryTree&, Node*)`:

This is a constant quantity of pointer rewires and some height updates, so $O(1)$.

`rightRotation(BinaryTree&, Node*)`:

Same as above, $O(1)$.

`insert(BinaryTree& binary_tree, const std::string& word, int documentId)`:

This is $O(h)$, where h is the height of the tree. It does one BST descent $O(h)$ at most one rebalance per level. The rotations are constant.

`remove(Node*&, key)`:

This is $O(h)$. It is a classic BST deletion + at most one structural deletion + up-to-root rebalancing.

`remove(BinaryTree&, key)`:

Same as above, $O(h)$.

`updateHeightUp(Node*)`:

This function iterates upward recomputing height until the value stabilises or the root is reached. Therefore it is $O(h)$.

`clear(BinaryTree*)`:

Clearly $O(n)$, where n is the size of the tree. It removes each node once.

`printInOrder`:

This is a classic traversal visiting every node once, so it is $O(n)$.

`printLevelOrder`:

Same as above, $O(n)$.

4.3. Red-Black Tree (RBT)

4.3.1. Algorithms

The functions implemented *strictly* for the RBT can be found here, and the header file with the corresponding documentation here. We have used many of the BST functions, as previously stated. The list of RBT-functions is:

- `insert(BinaryTree& binary_tree, const std::string& word, int documentId)`
- `int fixInsert(Node** root, Node* z)`
- `Node* getUncle(Node* node)`
- `Node* getSibling(Node* node)`

4.3.2. Complexity Analysis

Below is a full complexity analysis of the RBT functions:

`int fixInsert(Node** root, Node* z)`:

The while loop climbs the tree one level per iteration, recolouring and performig ≤ 2 rotations per level. Since the maximum height of an RBT is $O(\log n)$[1], this is $O(\log n)$.

`insert(BinaryTree& t, const std::string& word, int documentId)`:

This is a standard descent to find the leaf position, and height $\leq O(\log n)$, then it calls `fixInsert`, so the total complexity is also $O(\log n)$.

`Node* getUncle(Node* node)`:

There is no loop or recursion, just a few pointer dereferences, so this is $O(1)$.

`getSibling(Node* node)`:

Same reasoning as above, $O(1)$.

4.4. Inverted Index

4.4.1. Algorithms

4.4.2. Complexity Analysis

5. Testing and Validation

5.1. Unit Testing Method

We have used the **Unity** submodule for unit testing (espanio faz teu nome)

All trees were equivalently tested under the same principles blablabla

The testing module can be found here blablabla

6. Visualization with JavaScript

espanio faz teu nome

7. Comparative Analysis

Here we develop a full comparative analysis of the three implemented data structures. We will use the following metrics:

1. Memory usage
2. Insertion time
3. Search time

7.1. The Experiment

For the graphs seen in Section 7.2, we have used the following:

```shell
./[tree_name] stats 1000 data
```

All trees were testes with the same first 1000 .txt file, which can be found here.

7.2. Graphs

For a better visualization of the statistics we will cover here, we recommend, through the repository, to run:

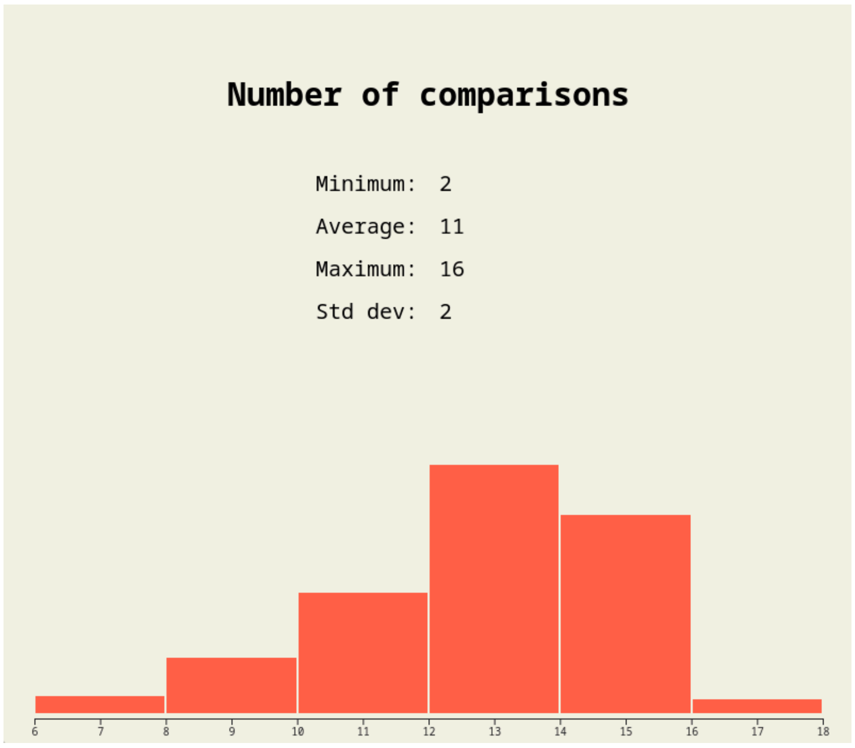
```shell
make bst && make avl && make rbt
./(tree_name) stats 1000 data
```

Opening the generated JavaScript visualization in a browser, you will be able to see the graphs and the tree visualizations in a more interactive way.

All statistics are also available in the `(tree_name).csv` file, generated by the `stats` command.

7.2.1. AVL

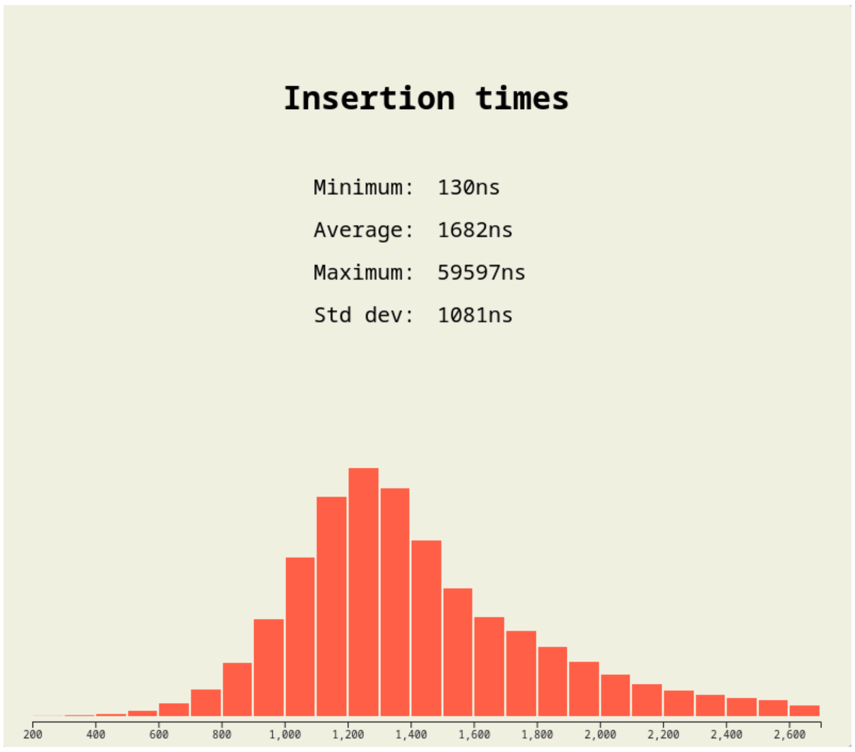
These graphs were generated by thye JavaScript visualizer.



Number of comparisons

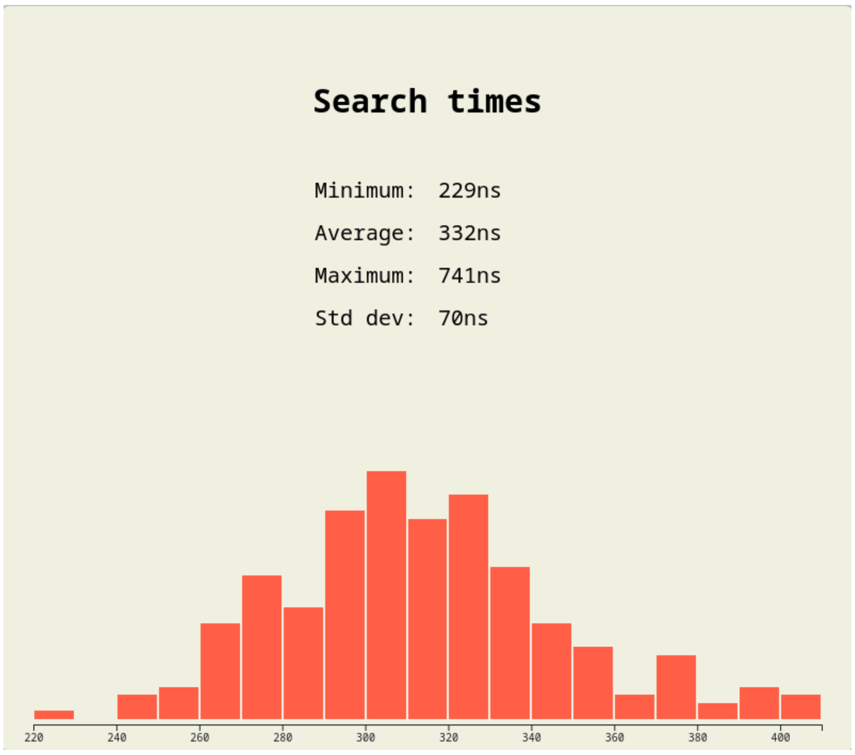
```
Minimum:  2
Average:  11
Maximum:  16
Std dev:  2
```

Figure 6: Number of Comparisons done by the AVL Tree



```
Minimum:  130ns
Average:  1682ns
Maximum:  59597ns
Std dev:  1081ns
```

Figure 7: Insertion time of the AVL Tree



```
Minimum:  229ns
Average:  332ns
Maximum:  741ns
Std dev:  70ns
```

Figure 8: Search time of the AVL Tree

7.2.2. BST

These graphs were generated by thye JavaScript visualizer.

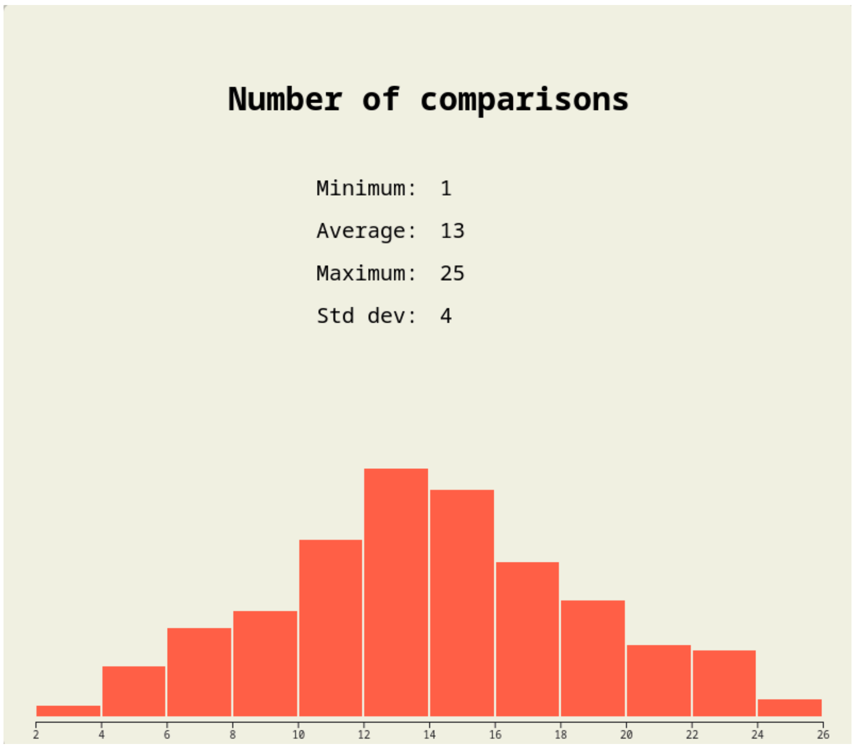


Figure 9: Number of Comparisons done by the BST Tree

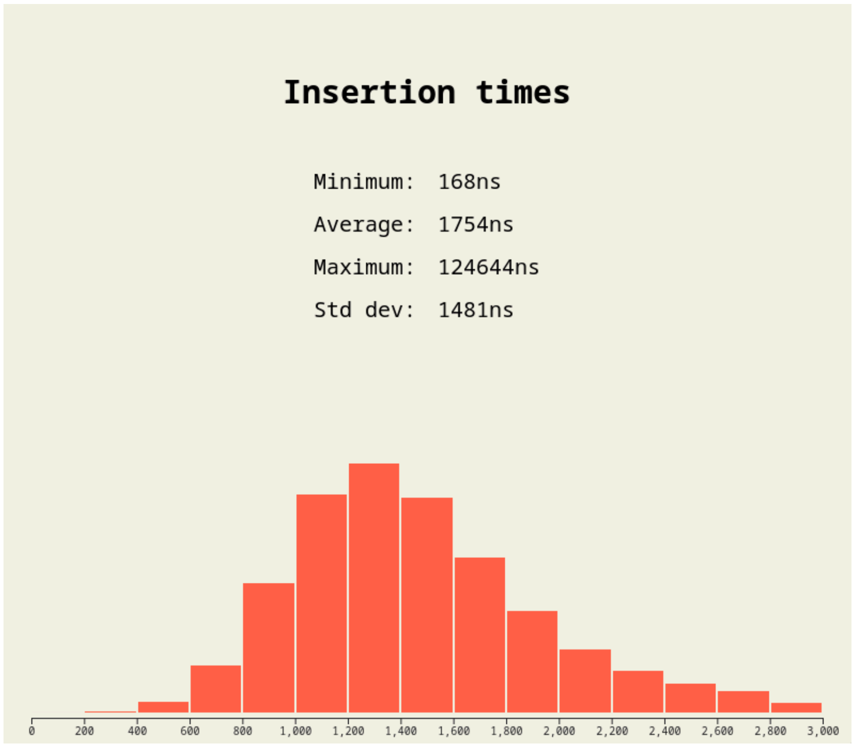


Figure 10: Insertion time of the BST Tree

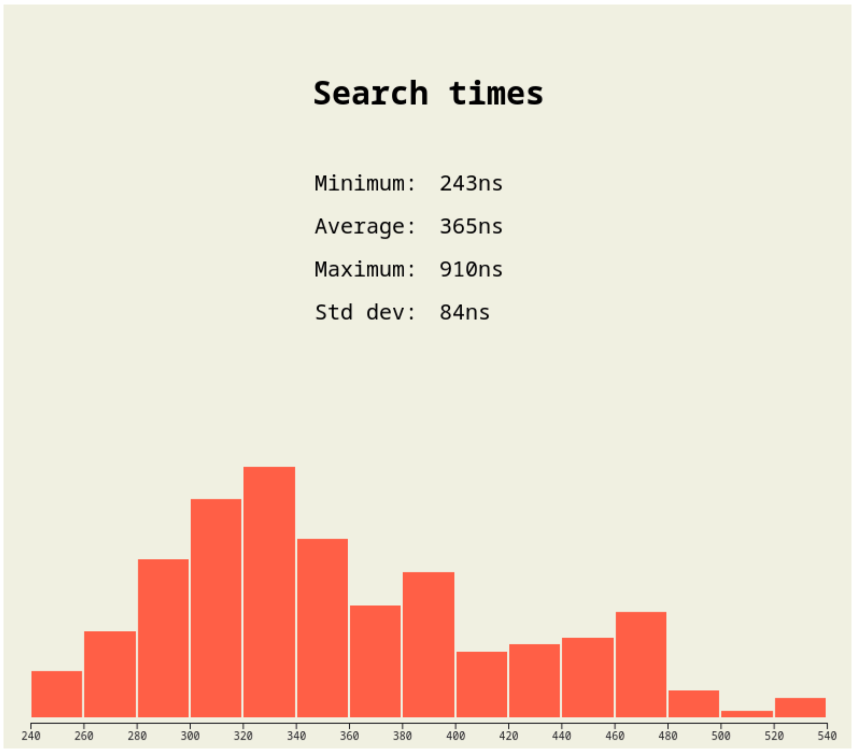


Figure 11: Search time of the BST Tree

7.2.3. RBT

These graphs were generated by thye JavaScript visualizer.

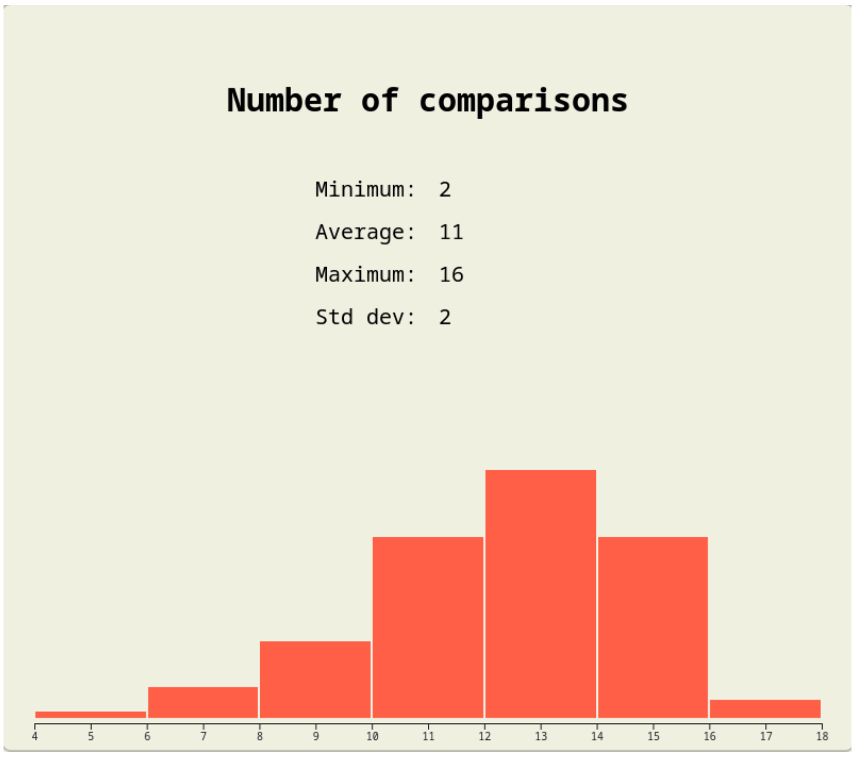


Figure 12: Number of Comparisons done by the RBT Tree

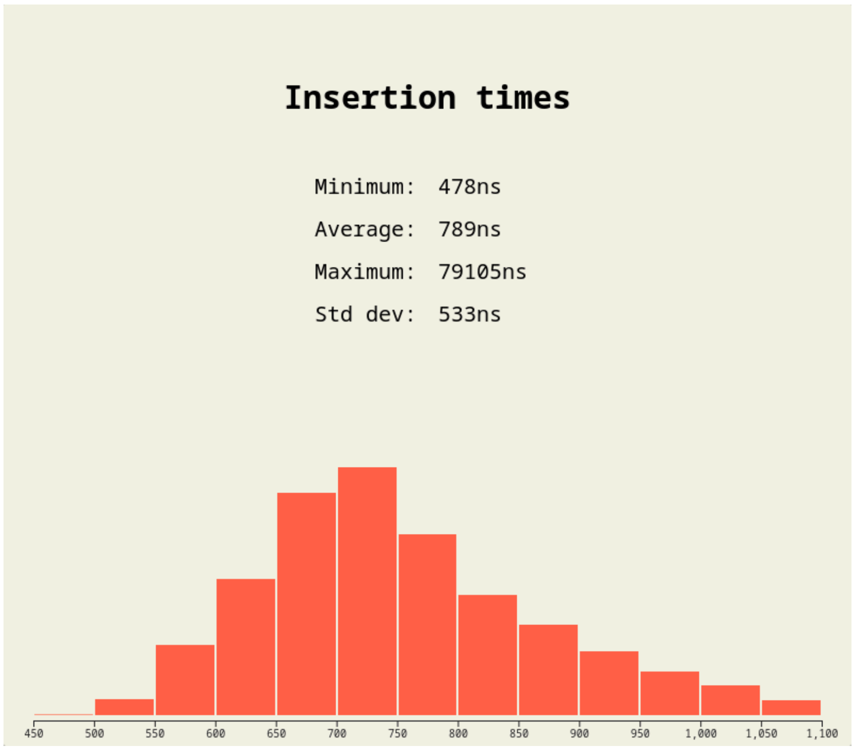


Figure 13: Insertion time of the RBT Tree

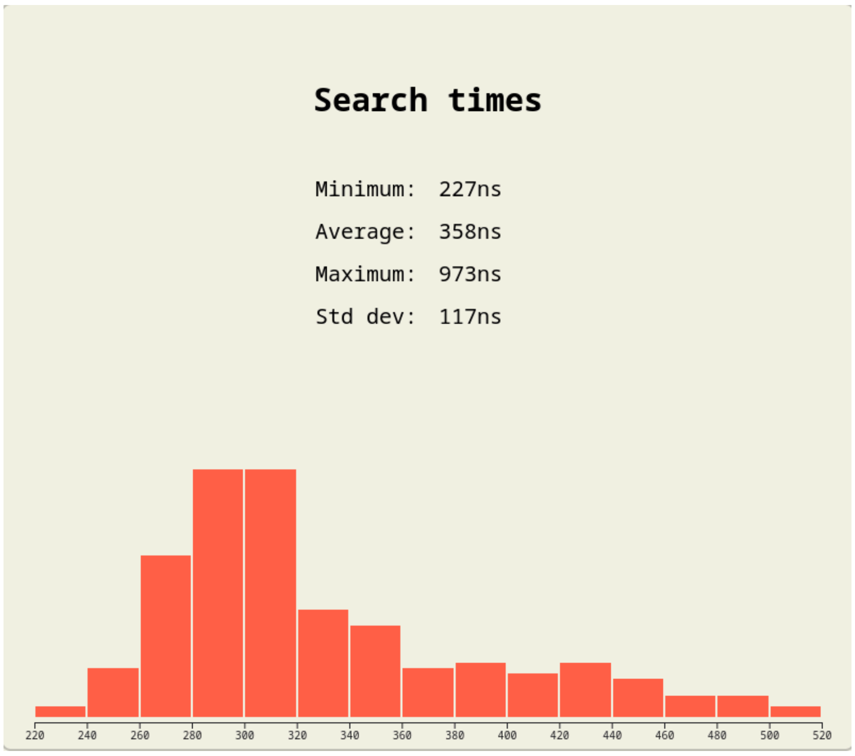


Figure 14: Search time of the RBT Tree

8. Conclusion

8.1. Raw stats

The raw statistics for the trees are:

Metric	BST	AVL	RBT
Docs indexed	1000	1000	1000
Words indexed	213099	213099	213099
Total insertion time (ns)	373 834 912	393 149 216	168 135 111
Avg insertion time (ns)	1754	1844	789
Max insertion time (ns)	124 644	189 938	79 105
Min insertion time (ns)	168	147	478
Total search time (ns)	90 034	81 607	88 426
Avg search time (ns)	365	331	358
Max search time (ns)	910	773	973
Min search time (ns)	243	232	227
Total comparisons (search)	3356	2942	2947
Avg comparisons (search)	13	11	11
Max comparisons (search)	25	16	16
Min comparisons (search)	1	2	2
Node count	16 986	16 986	16 986
Tree height	32	16	24
Min depth	4	10	8
Balance diff	28	6	16
Relative balance	8	1.6	4

8.2. Actual Analysis

Let $h(n)$ be the height of node n. Because every AVL is a BST with an additional constraint, every theorem about BST search order also applies to AVL unless it contradicts the balance rule.

For the BST, If the input keys arrive in sorted order, each new key is inserted as the right child of the previous one, yielding a linear chain of **n** nodes, so:

$$h_{\max}(n) = n - 1 \tag{2}$$

For the AVL, the maximum height is logarithmical [2], as it is a balanced tree. The height of an AVL tree with **n** nodes is at most:

$$h_{\max}(n) = O(\log n) \qquad (3)$$

Which might make the AVL interesting.

8.2.1. Time Complexity

When analyzing time complexity for these trees, we look at the cost of each operation:

Operation	BST	AVL	RBT
Insertion	$O(h(n))$	$O(h) + O(1)$ rotations	$O(\log n) + O(1)$ recolor/rotations
Search	$O(h(n))$	$O(h(n))$	$O(\log n)$
Deletion	$O(h(n))$	$O(h) + O(h)$ rotations	$O(\log n) + O(1)$ rotations

8.2.2. Memory Usage

All implementations use:

```C++
struct Node {
    std::string word;
    std::vector<int> docIds;
    Node *left, *right, *parent;
    int height;       // both codes already keep this
    bool isRed;       // reserved for RBT
};
```

Field counts are identical, therefore heap consumption is $O(n)$ for the AVL and BST. Recursive algorithms allocate one activation record per visited level.

- BST worst-case stack depth: $h = n - 1 \rightarrow O(n)$ extra bytes; may overflow for large n.
- AVL stack depth: $h = O(\log n) \rightarrow$ asymptotically optimal.

No additional global buffers are required; rotations operate with $O(1)$ local variables.

9. Challenges and Difficulties (Required by the professor)

9.1. Arthur Rabello Oliveira

The hardest challenge i faced was to keep the GitHub repository organized, reviewing pull requests from my fellow collaborators. Writing this report and the `README.md` have not been trivial tasks too.

Another very big difficulty was to write the `Makefile` for compiling the project, as the syntax is rather counterintuitive. One could scroll through the commit history and get a glance at how ugly the first version of this file was.

Reviewing the work of my fellow collaborators was also a challenge, as I had to understand their code and suggest improvements. This was a very good learning experience, though.

I must say that with other 4 courses, time management was a **very big challenge** too. Not only 4 other courses, but 4 other **hard** courses, with many assignments and deadly exams and tests, but unfortunately my days have not more than 24 hours, so I had to manage my time very carefully to be able to finish this project on time. And i am very happy with the result.

9.2. Eliane Moreira

9.3. Gabrielle Mascarelo

9.4. Nícolas Spaniol

9.5. Gabriel Carneiro

10. Source code

All implementations and tests are available in the public repository at https://github.com/arthurabello/dsa-final-project

11. Task Division (Required by the professor)

11.1. Arthur Rabello Oliveira

Contributed with:

- Keeping the repository civilized (main-protecting rules, enforcing code reviews)
- Writing and documenting the `Makefile` for building the project
- Writing the `README.md` and `report.typ`
- Writing and documenting functions for the classic BST in `bst.cpp`

11.2. Gabrielle Mascarelo

Contributed with:

- Writing and documenting functions to read files in `data.cpp`
- Structuring statistics in the CLI

11.3. Eliane Moreira

Contributed with:

- Testing all function related to the BST
- Writing and documenting functions for `tree_utils.cpp`
- Fixing bugs in `data.h`

11.4. Nícolas Spaniol

Contributed with:

- Code reviews and suggestions for improvements in the codebase.
- Built from scratch the JavaScript visualization of all trees

11.5. Gabriel Carneiro

Contributed with:

- Writing and documenting functions for the classic BST
- Writing and documenting functions for the AVL Tree
- Testing functions for `tree_utils.cpp`

Bibliography

[1] U. of Wisconsin-Madison, “Red Black TRees.” [Online]. Available: https://pages.cs.wisc.edu/~naleah/

[2] MIT, “AVL Trees Height Proof.” [Online]. Available: https://people.csail.mit.edu/alinush/6.006-spring-2014/avl-height-proof.pdf