# Final Project - Inverted Index and Comparative Analysis

Arthur Rabello Oliveira[1], Gabrielle Mascarelo[2] , Eliane Moreira[3], Nícolas Spaniol[4], Gabriel Carneiro[5]

**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. Introduction ..... 3
   1.1. Context ..... 3
   1.2. Problem Statement ..... 3
   1.3. Objectives ..... 3
2. Motivation ..... 3
   2.1. Why Inverted Index? ..... 3
3. Definitions ..... 3
   3.1. BST ..... 3
   3.2. AVL Tree ..... 5
   3.3. RBT Tree ..... 6
4. Implementations ..... 6
   4.1. Binary Search Tree (BST) ..... 6
   4.2. AVL Tree ..... 7
   4.3. Red-Black Tree (RBT) ..... 8
   4.4. Inverted Index ..... 8
5. Testing and Validation ..... 9
   5.1. Unit Testing Method ..... 9
6. Complexity comparative analysis of operations ..... 10
   6.1. Balancing Difference ..... 10
   6.2. Insertion time ..... 10
7. Comparative Analysis ..... 12
   7.1. The Experiment ..... 12
   7.2. Graphs ..... 12
      7.2.1. BST ..... 12
      7.2.2. AVL ..... 15
      7.2.3. RBT ..... 18

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

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

[3] Escola de Matemática Aplicada, Fundação Getúlio Vargas (FGV/EMAp), email: elianemoreira121095@gmail.com

[4] Escola de Matemática Aplicada, Fundação Getúlio Vargas (FGV/EMAp), email: nicolasmspaniol@gmail.com

[5] Escola de Matemática Aplicada, Fundação Getúlio Vargas (FGV/EMAp), email: gabrielcarneiron@gmail.com

8. Conclusion ........ 21
   8.1. Raw stats ........ 21
   8.2. Theoretical Results ........ 22
      8.2.1. Time Complexity ........ 23
      8.2.2. Memory Usage ........ 23
      8.2.3. Data visualization and extras ........ 24
         8.2.3.1. Fun Fact ........ 25
9. Challenges and Difficulties (Required by the professor) ........ 25
   9.1. Arthur Rabello Oliveira ........ 25
   9.2. Eliane Moreira ........ 25
   9.3. Gabrielle Mascarelo ........ 26
   9.4. Nícolas Spaniol ........ 26
   9.5. Gabriel Carneiro ........ 26
10. Source code ........ 27
11. Task Division (Required by the professor) ........ 27
   11.1. Arthur Rabello Oliveira ........ 27
   11.2. Gabrielle Mascarelo ........ 27
   11.3. Eliane Moreira ........ 28
   11.4. Nícolas Spaniol ........ 28
   11.5. Gabriel Carneiro ........ 28
Bibliography ........ 28

# 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 query performance and insertion throughput, besides the edge cases introduced when we choose a more complex tree type.

## 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 ∀ node $\in T = (V, E)$.

Here are some examples of BSTs:

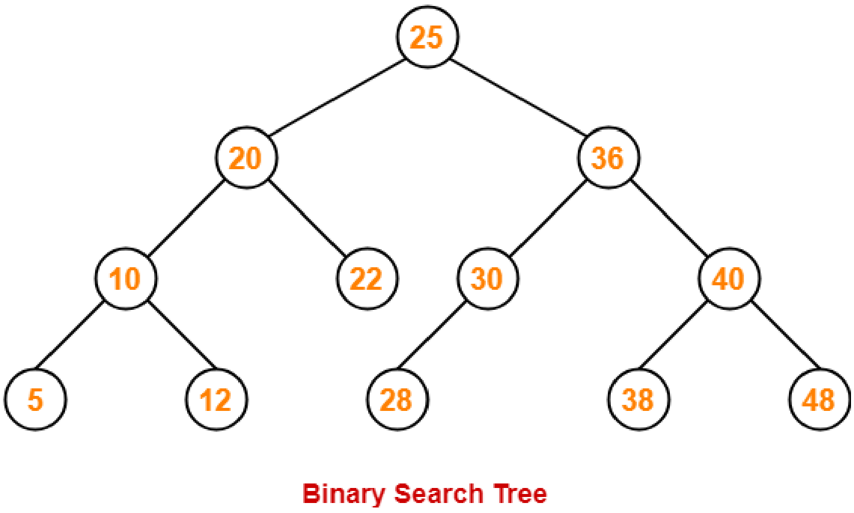


Figure 1: Example of a Binary Search Tree

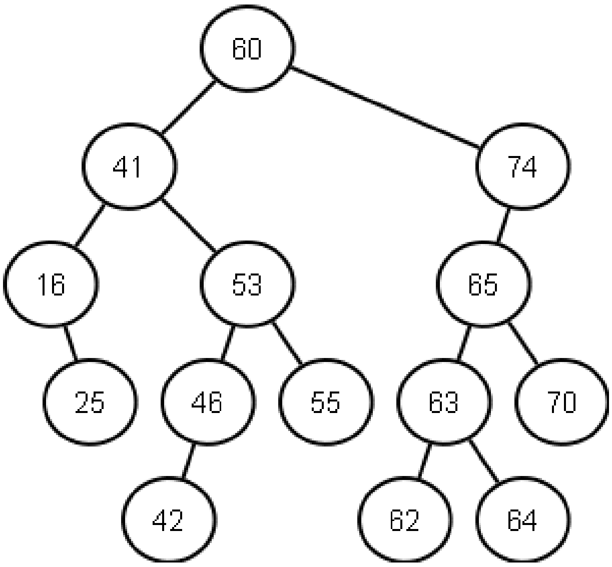


Figure 2: Another example of a Binary Search Tree

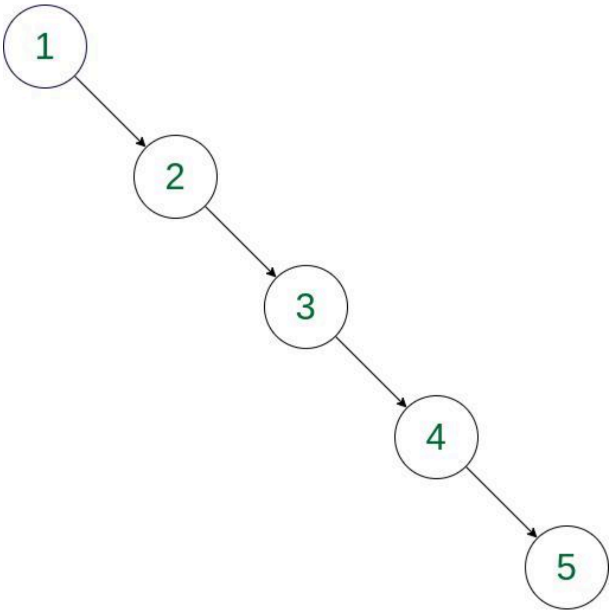


Figure 3: Degenerated Binary Search Tree

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:

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

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.

### 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:

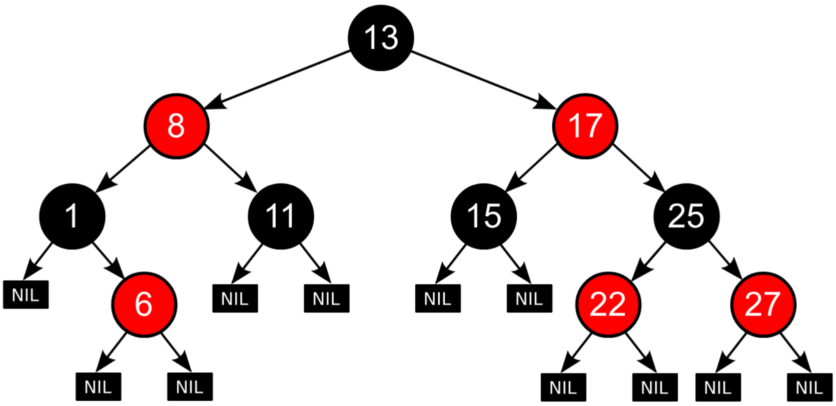


Figure 4: Example of a Red-Black Tree

## 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

Here are all the functions written for the classic BST, and here is the header file with the corresponding documentation. Below is a full complexity analysis alongside with a list of all implemented functions:

- `Node* createNode(std::string, std::vector<int>, int)`:

No loops or recursions, it allocates a node, so it is $O(1)$

- `BinaryTree* createTree()`:

This is a single `new` + pointer initialization. $O(1)$

- `InsertResult BST::insert(BinaryTree*, const string&, int)`:

The while loop descends one edge per iteration $\Rightarrow h$ comparisons. If the word already exists, a `for` scans `current->documentIds` (of size $k$) to avoid duplicate doc-IDs. The Height update (`updateHeightUp`) walks $\leq h$ parents but it does no recursion on kids. So this is $O(h + k)$

- `SearchResult TREE::search(BinaryTree*, const string&)`:

This is an iterative root-to-leaf walk with at most one comparison per leve. Therefore it is $O(h)$.

- **`void updateHeightUp(Node*)`**:

This recomputes height, then climbs parent pointers until the value stabilises or the root is reached that's $\leq h$ iterations, of constant work each. So this function is $O(h)$.

- **`int calculateHeight(Node*)`**:

This is a pure post-order recursion, each node visited exactly once (`1 + max(left,right)`), so it is $O(s)$, where $s$ is the sub-tree size

- **`int calculateMinDepth(Node*)`**:

This recurses into both sub-branches unless one child is `nullptr`, still touches every node in worst case. So it is $O(s)$

- **`int countNodes(Node*)`**:

Classic `1 + count(left)+count(right)`, each node hit once. So it is $O(s)$.

- **`void deletionPostOrder(Node*)`**:

Post-order traversal deletes each node once. Also $O(s)$.

- **`void destroy(BinaryTree*)`**:

Calls `deletionPostOrder` on the whole tree ($n$ nodes) then deletes the wrapper. So it is $O(n)$.

## 4.2. AVL Tree

The functions implemented *strictly* 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. Below is a full time complexity analysis of the AVL functions, alongside with a list of all implemented functions:

- **`int getHeight(Node*)`**:

It's one conditional, so it is $O(1)$.

- **`int bf(Node*)`**:

That's a pair of `getHeight` calls and some arithmetic, so it is $O(1)$.

- **`Node* updateHeightUp(Node*)`**:

Walks at most one edge per iteration from the inserted leaf up to the root, recomputing two child heights per level and stopping early if the height stabilises, no branching into sub-trees. So it is $O(h)$, where $h$ is the current height of the tree.

- **`void leftRotation(BinaryTree&, Node*)`**:

This is a constant number of pointer rewires and 2 height recalculations. So it is $O(1)$.

- **`void rightRotation(BinaryTree&, Node*)`**:

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

- **`void balanceTree(BinaryTree&, Node*)`**:

Given *one* already-detected unbalanced node, it executes at most one of four rotation cases (each case does $\leq 2$ rotations) and then returns. No loops over height, so it is $O(1)$.

- **`InsertResult insert(BinaryTree&, string word, int docId)`**:

Phase 1: classic BST descent to the insertion point touches **h** nodes. Phase 2: `updateHeightUp` climbs back up $h$ levels. Phase 3: `balanceTree` performs at most one constant-time rotation combo. In an AVL we have $h = O(\log n)$ [1], so insert is $O(\log n)$.

## 4.3. Red-Black Tree (RBT)

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. Below is a full time complexity analysis of the RBT functions, alongside with a list of all implemented functions:

- **`InsertResult insert(BinaryTree*, const std::string&, int)`**:

1 BST descent does $\leq h$ key comparisons ($h =$ tree height). In an RBT $h \leq 2\log_2(n+1)$ [2], so descent is $O(\log n)$. After the leaf is linked the function calls fixInsert, whose own cost is $O(\log n)$ (see next row). Constant-time pointer rewires for at most 2 rotations do not change the order. Therefore the total complexity is $O(\log n)$.

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

The while-loop climbs the tree a single level per iteration recolouring or doing $\leq 2$ rotations. The number of iterations is bounded by the black-height of the tree, i.e. $O(\log n)$. All work inside each iteration is constant. So the total complexity is $O(\log n)$.

- **`Node* getUncle(Node*)`**:

This is a constant-time pointer dereference and arithmetic, so it is $O(1)$.

- **`Node* getSibling(Node*)`**:

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

- **`void rotateLeft(Node**, Node*)`**:

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

- **`void rotateRight(Node**, Node*)`**:

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

## 4.4. Inverted Index

The inverted index has been built over the 3 data structures previously presented. Here we provide a list of the tools that actually power this pipeline (file → tokenisation → tree insertion → stats/ visualization):

- **`DATA::normalise`**:

This is a single pass over the input string: one `std::tolower + std::isalnum` per char, no allocations except the output buffer. Therefore it is $O(l)$, where $l$ is the input string length.

- **`DATA::tokenize`**:

Each successful `archive >> word` pulls one token $\Rightarrow W$ iterations. Inside the loop: `uniqWords.find(word)` and `uniqWords.insert(word)` are expected $O(1)$ because `std::unordered_set` is built on open-addressing / chained hash tables. `Normalise()` scans every character of the token once $\Rightarrow \sum \|w_i\|$ overall. So we have a total work of $O(W + \sum \|w_i\|)$

- **`DATA::list_files_txt_in_path`**:

Single pass over `std::filesystem::directory_iterator`, all work per entry is $O(1)$. So the total work is $O(F)$, where $F$ is the amount of entries in the directory.

- **`DATA::buildNodes`**:

This function calls `tokenize`(see above) then does one `createNode` per token (constant-time pointer allocation & initialisation). So the total work is $O(w)$, where $w$ is the number of tokens in the input file.

- **`CLI::formatDouble`**:

Simple `to_string` then substring, linear in string length, negligible in practice. So the total complexity is $O(k)$, where $k$ is the amount of digits in the number.

- **`CLI::colorize`**:

Those are 2 concatenations, so it is $O(L)$, where $L$ is the length of the string to be colorized.

- **`CLI::indexFilesInDir`**:

This is a loop over at most $n$ files, tokenises each, then calls the tree-specific insert once per token. All other body lines are constant time. So it is $O(D + \sum T_i + \sum I_i)$, where $T_i$ is the tokenization cost per file (see above), $I_i$ is the tree-insert cost per token and $D$ is the directory scan (delegated to `list_files_txt_in_path`).

- **`CLI::testSearch`**:

This is an outer loop over distinct sample words, inner fixed 100-iteration loop calling generic TREE::search ($O(h)$). So the total work is $O(Q \cdot h)$, where $Q = \text{words} \cdot \text{numTries}$, $h = \text{TreeHeight}$.

- **`CLI::collectAggStats`**:

Those are linear passes over the stats vectors ($S$ elements) plus calls to `TREE::countNodes, calculateHeight, calculateMinDepth` — each walks the whole tree once ($n$ = nodes). So the total work is $O(S + n)$.

- **`CLI::startViewServer`**:

here `add_node` performs a *single* depth-first traversal; every call touches a node once and does a bounded-cost string append ⇒ $O(n + L)$. The three simple `for` loops concatenate the stats arrays ⇒ $O(V)$ The route registration & server start-up (`server.when(…)`) contains only a fixed number of pointer stores ⇒ $O(1)$. Combining these we have a total work of $O(n + V + L)$.

# 5. Testing and Validation

## 5.1. Unit Testing Method

For this project we used the Unity library for unit testing. *Unity* is a C library that we included as a *git submodule* in our repository, i.e., we cloned the library's folder inside our own and compiled it's source code together with our testing files.

Using a library for this allowed us to better organize our tests, requiring we write each one of them as a function and pass the name of all those functions to Unity afterwards, and making it explicit when we are comparing two values to determine the validity of the test.

The tests are contained in their own file inside each of the tree's subfolders and can be run with *make test-<tree type>*. We wrote similar tests for each of the trees, accounting for rotations and other tree-specific behaviours when needed.

# 6. Complexity comparative analysis of operations

## 6.1. Balancing Difference

As the BST tree is unbalanced this result was expected (it having the most difference in the heights of the highest branch and the shortest branch). On the other hand, AVL and RBT are self-balanced and have this measurement low. AVL is always balancing itself while RBT is sometimes not yet balanced, so AVL is the lowest of them all in this graph.

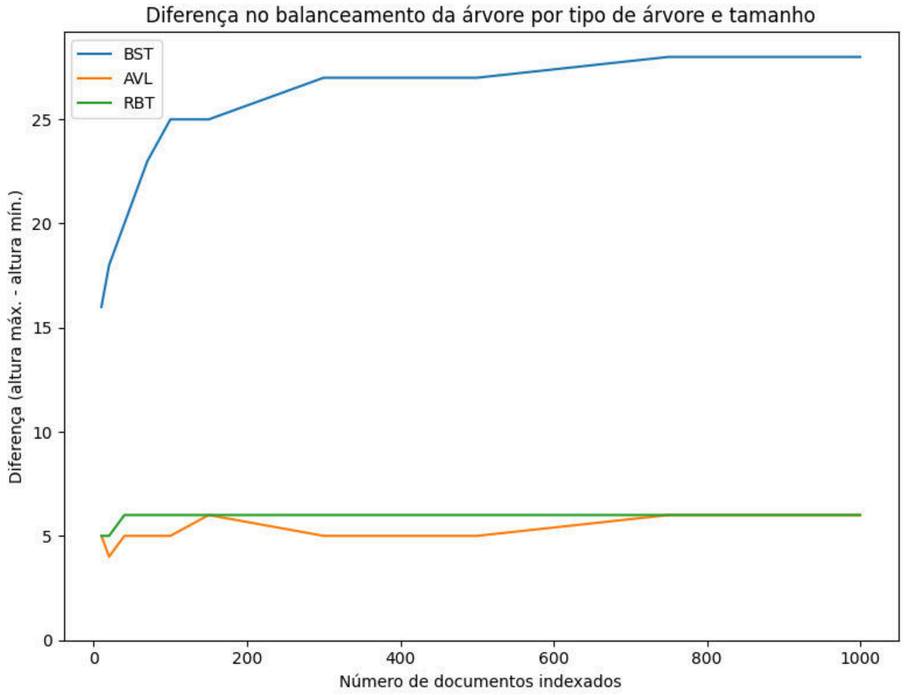


Figure 5: Difference in the trees balancing

## 6.2. Insertion time

In the RBT structure, the insertion is close to being constant because it doesn't recalculate heights like AVL and its balancing is more flexible than AVL, requiring less rotations (even if it requires recoloring operations, they are much more computationally cheaper).

The property that AVL holds by computing rotations in each insertion to balance itself increase significantly its insertion time.

In contrast, the BST structure, the simplest of them all, even not having balancing costs, is high on that measure. It depends on the order the data were inserted, extending the order of time complexity from its height to its number of nodes in the worst case, the degenerated case.

The plateau in BST and AVL reached after 200 documents indexed may occur because, from this point on, the number of new and unique words inserted decreased.

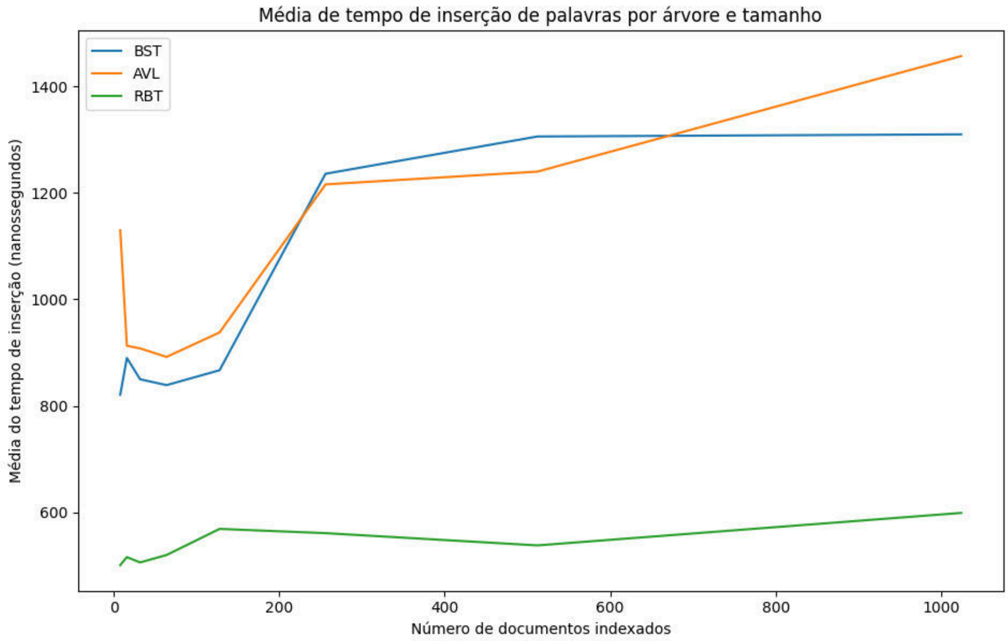


Figure 6: Mean of insertion time in the trees by size

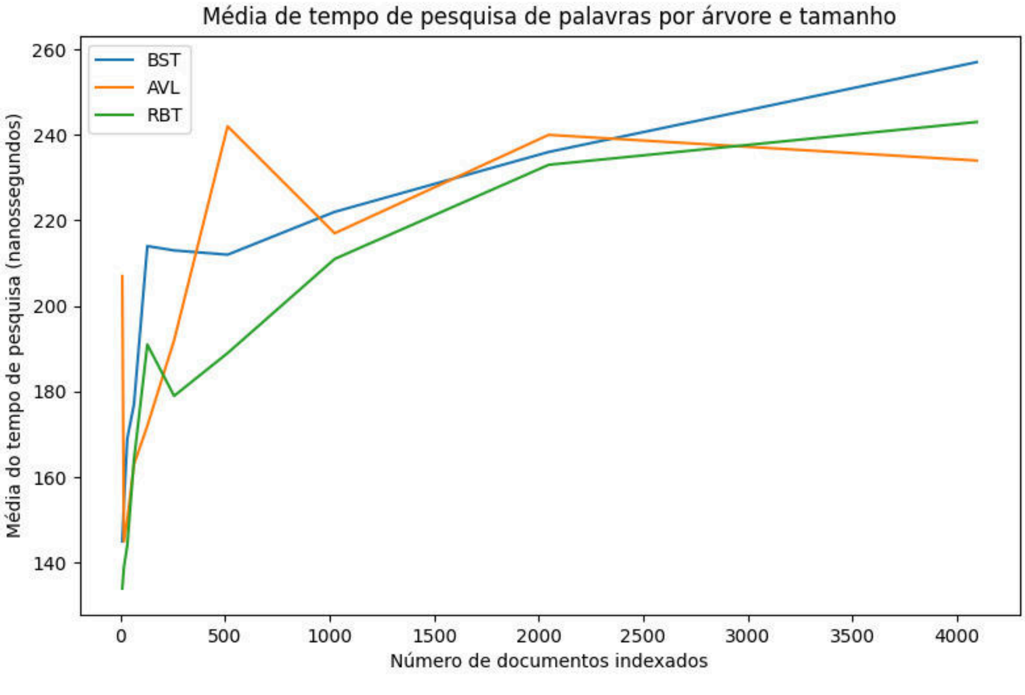


Figure 7: Mean of search time in the trees by size

Here, the mean of all of them appears to have a log(n) shape like, similar to what we expected. In a large number of documents, AVL appears to be the best structure of them in this sense.

It is interesting to look that in both measures of time, the AVL took a bigger while to initialize. In other words, it had the initial iteration way up in the graph comparing to the following iterations.

At last, we could say RBT is way much better in insertion time compared to the others, while in the other measures it is pretty close to being the best, at the side of AVL. We could bear to say that RBT is the overall best, being especially very requested in projects that needs a large number of insertions. AVL perhaps could perform better in projects that need a lot of search operations maintaining few insertions. BST could be put as the worst of all, but nevertheless it may still be used in educational contexts or when the dataset is small and its data is truly random (because it becomes inefficient and degenerated when the data is ordered).

# 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) view 1000 data
./(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. BST

These graphs were generated by thye JavaScript visualizer.

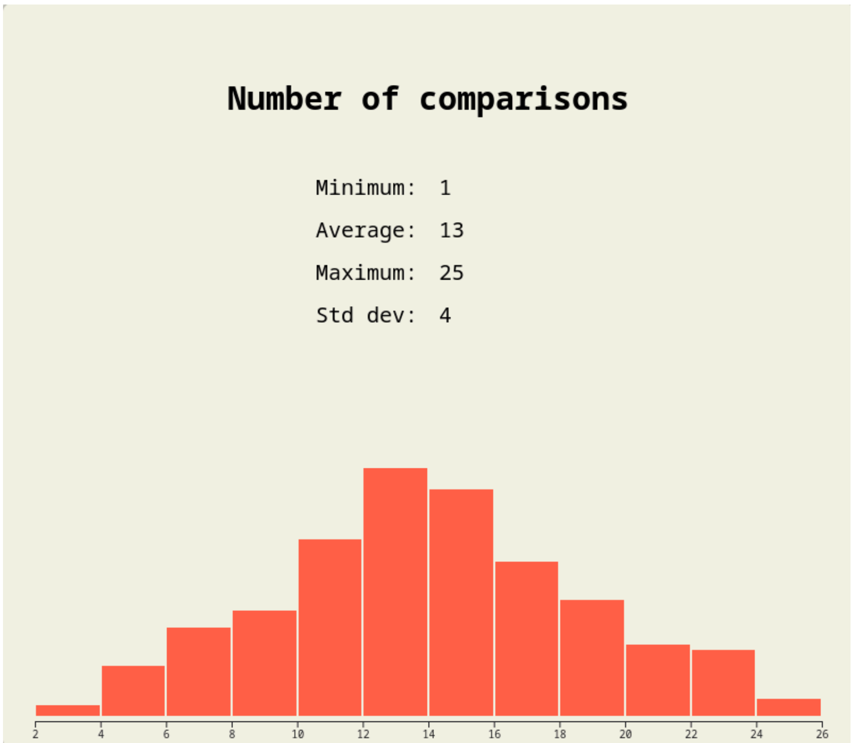


Figure 8: Number of Comparisons done by the BST Tree

- Analysis of the Results

The comparison data in the BST reveal important characteristics of this unbalanced structure. The minimum comparison value represents the best-case scenario, where the element is located exactly at the root of the tree. The average, which is much higher than the minimum, indicates that, in most cases, more operations are required to locate an element, especially when the tree is not perfectly balanced.

The maximum value, which is high relative to the average, demonstrates the main issue of unbalanced BSTs, which is when it resembles a linked list, causing the number of comparisons to increase. The relatively high standard deviation confirms the significant inconsistency in performance.

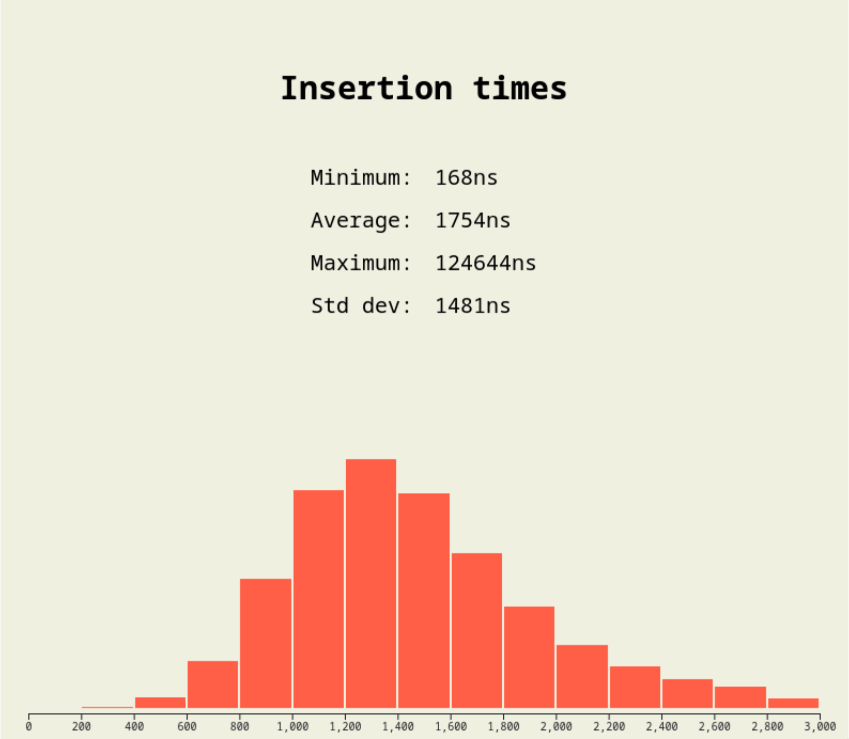


Figure 9: Insertion time of the BST Tree

- Analysis of the Results

The insertion times in the BST show significant variation, revealing the unpredictable behavior of this unbalanced structure. The minimum time represents quick insertions when the new element finds a position close to the root. The high average time reflects the cost of traversing multiple levels of the tree to place a new element.

The maximum time reaches extremely high values, demonstrating when the BST resembles a linked list, requiring traversal of nearly all existing elements before performing the insertion. The high standard deviation confirms this large dispersion in execution times.

The distribution of times in the graph shows a moderate concentration around the average, but with a long tail extending to very high values.

This analysis suggests that the unbalanced nature of BSTs makes them unpredictable, especially when the tree grows uncontrollably.

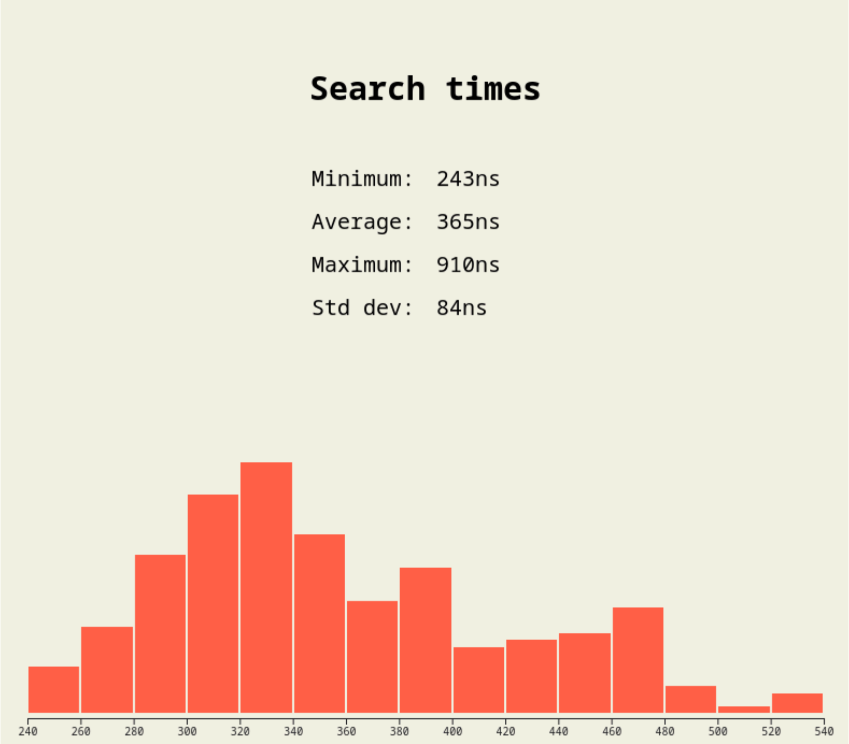


Figure 10: Search time of the BST Tree

- Analysis of the Results

The search times in the BST exhibit behavior characteristic of unbalanced structures. The minimum time occurs when the sought element is at the root or close to it, while the average time reflects the typical cost of traversing multiple levels of the tree. The maximum time, significantly higher, reveals cases where the tree approaches a linked list, requiring traversal of nearly all the nodes.

The relatively high standard deviation indicates considerable variation in search times. The graph shows an asymmetric distribution, with a peak around the average.

These results suggest that, while BSTs can offer fast searches in some cases, their unbalanced nature can lead to inconsistency, especially when compared to AVL trees, which maintain a more stringent balance.

#### 7.2.2. AVL

These graphs were generated by thye JavaScript visualizer.

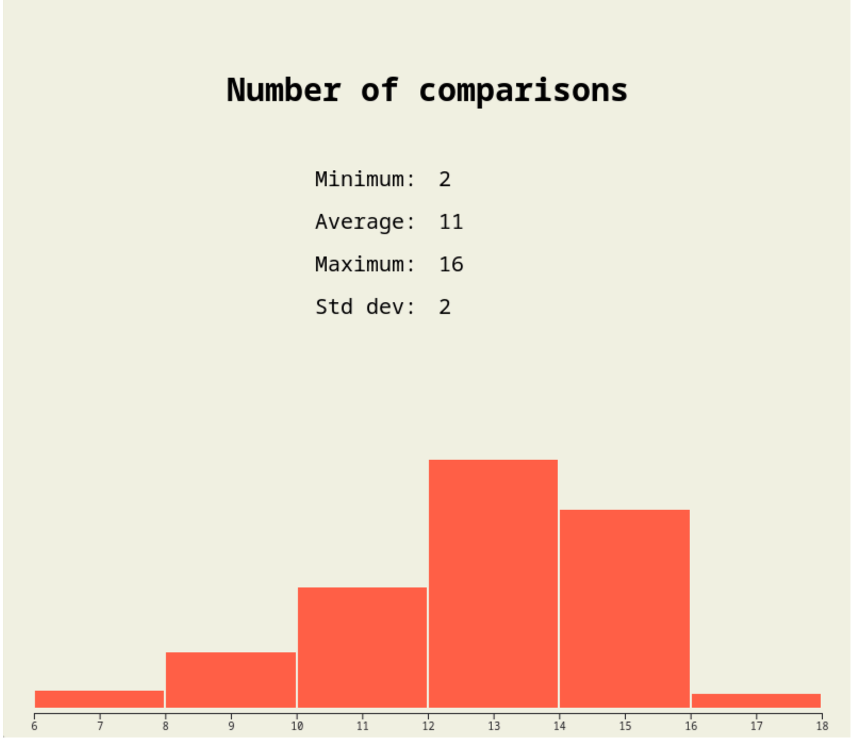


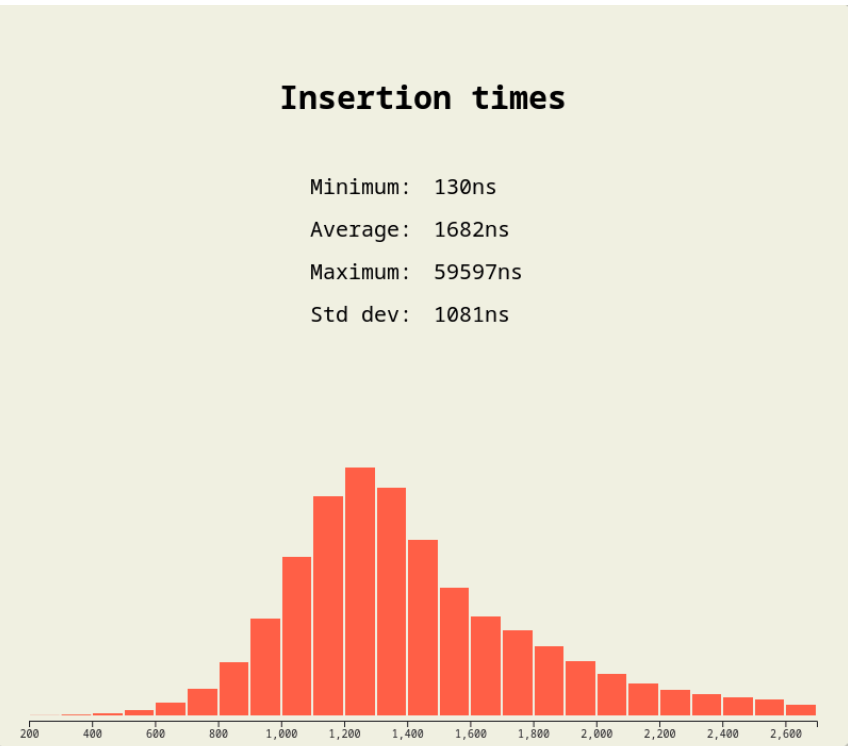
Figure 11: Number of Comparisons done by the AVL Tree

- Analysis of the Results

The table presents statistics on the number of comparisons made in an AVL tree, highlighting its characteristic efficiency. The minimum value represents the smallest number of comparisons required, occurring when the sought element is near the root. The average, on the other hand, reflects the expected performance in most cases, offering a view of the tree's typical behavior. The maximum value indicates the worst-case scenario, where more comparisons are needed, typically for elements located in the deepest nodes. The standard deviation, in turn, shows that most searches perform close to the average, with little variation, confirming the consistency of the structure.

The histogram complements this data by displaying the distribution of comparisons. It is possible to observe that some searches are highly efficient (values on the left), while the majority cluster around the average (central values). Only a few searches require the maximum number of comparisons (values on the right), demonstrating the effectiveness of the AVL's automatic balancing.

These results are possible because the AVL tree maintains its balance, ensuring that its height is always proportional to the logarithm of the number of elements. This ensures that even in the worst case, the number of comparisons remains limited and predictable—an advantage crucial when compared to unbalanced trees, such as degenerate BSTs, which can perform significantly worse in unfavorable situations.



Insertion times

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

Figure 12: Insertion time of the AVL Tree

- Analysis of the Results

The data presented shows the insertion times in an AVL tree, measured in nanoseconds. The minimum time represents the fastest possible scenario, which is, most likely the one where the insertion occurs without the need for rebalancing. The average time reflects the typical performance of the structure, considering that most operations involve some local adjustment of the tree. The maximum time, significantly higher, likely corresponds to cases where a full tree rebalance is necessary, from the inserted leaf all the way up to the root.

The relatively low standard deviation compared to the mean indicates that most insertions are executed in times close to the average value, with few operations requiring extreme times. The distribution graph shows that the majority of insertions occur within time intervals close to the average, with a long tail representing the few insertion cases that demand more processing due to deep rebalancing.

These results demonstrate the efficiency characteristic of AVL trees. Although occasionally more costly operations may occur due to rebalancing, the vast majority of insertions maintain consistent performance. This contrasts with unbalanced structures like BSTs, where insertion times can vary uncontrollably as the tree grows.

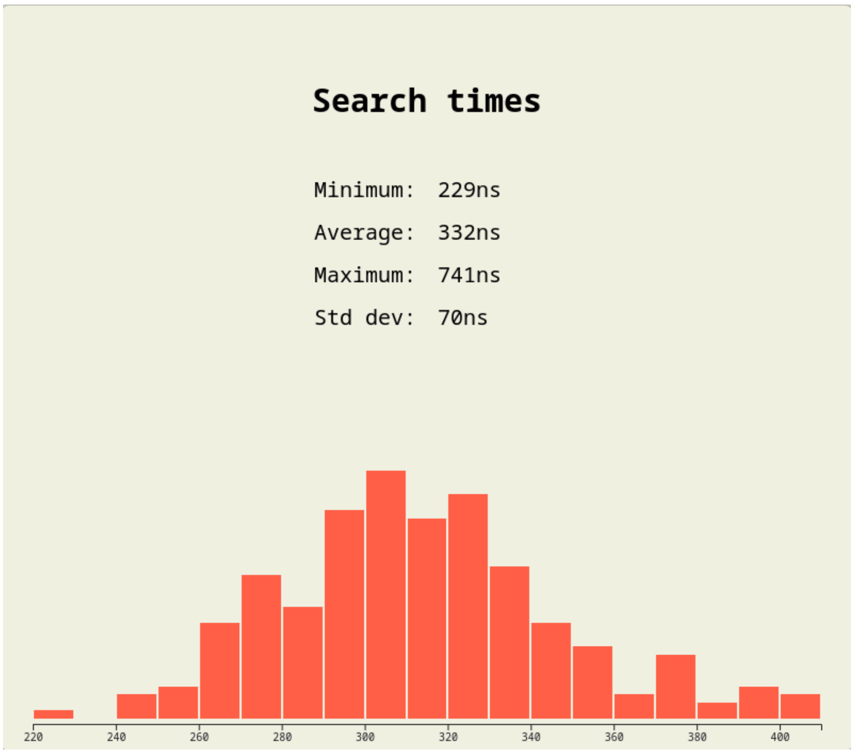


Figure 13: Search time of the AVL Tree

- Analysis of Results

The search times in the AVL tree demonstrate the characteristic efficiency of this balanced structure. The minimum time represents the ideal scenario, when the searched element is close to the root. The average time, only slightly higher than the minimum, shows the consistent performance of the AVL in most cases, reflecting its good balance. The maximum time remains within a reasonable range, indicating that even in the worst case, the search does not become excessively slow.

The low observed standard deviation confirms that most search operations occur in times very close to the average, with little variation. The distribution graph reveals a concentration of searches around the average value, forming a peak with few cases at the extremes. This pattern is typical of well-balanced structures, where the height of the tree is kept under control.

These results suggest that the AVL tree ensures predictable and efficient search times due to its rebalancing mechanism, unlike unbalanced trees, where searches can become slower.

### 7.2.3. RBT

These graphs were generated by thye JavaScript visualizer.

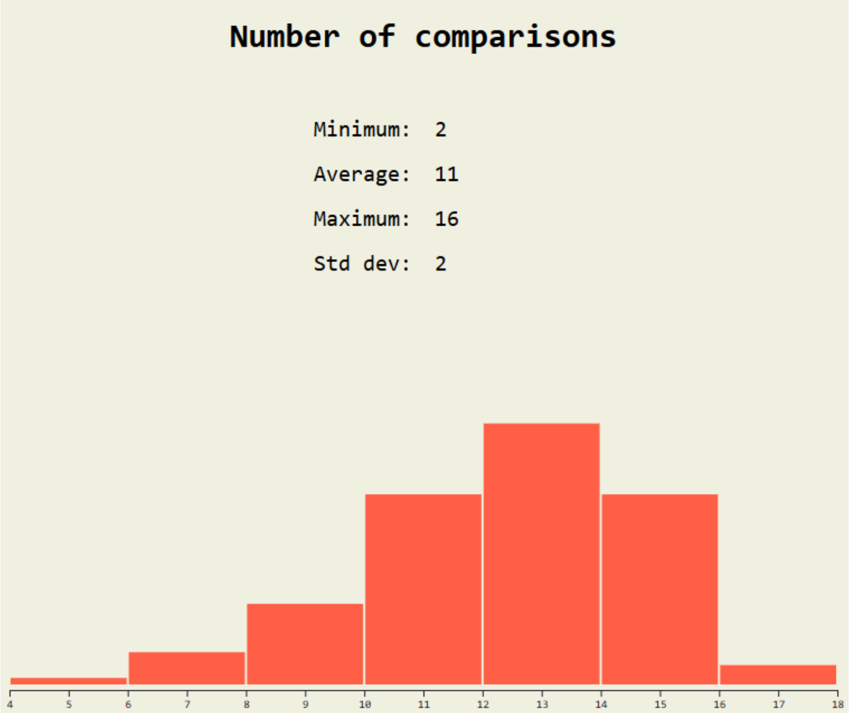


Figure 14: Number of Comparisons done by the RBT Tree

- Analysis of Results

The Red-Black Tree (RBT) demonstrates excellent performance in search operations. The minimum number of comparisons occurs when the element is located near the root, while the average value showcases the efficiency characteristic of this balanced structure. The maximum number of comparisons, though higher than the average, is still controlled, highlighting how the tree's balancing prevents extreme cases.

The small variation in the results, indicated by the low standard deviation, shows that most searches are performed with a number of comparisons very close to the average. This consistency contrasts strongly with the unpredictable behavior of the Binary Search Tree (BST).

The distribution graph reveals a pattern concentrated around the average value, with few operations requiring the maximum number of comparisons, demonstrating how well the tree controls its height, ensuring that even in the worst-case scenario, performance remains strong.

When compared to other structures, the Red-Black Tree shows similar advantages to the AVL Tree, with both maintaining efficient balancing that results in consistent search times. The primary difference lies in the specific balancing algorithms used by each, which lead to small variations in the number of comparisons but still deliver strong overall performance.

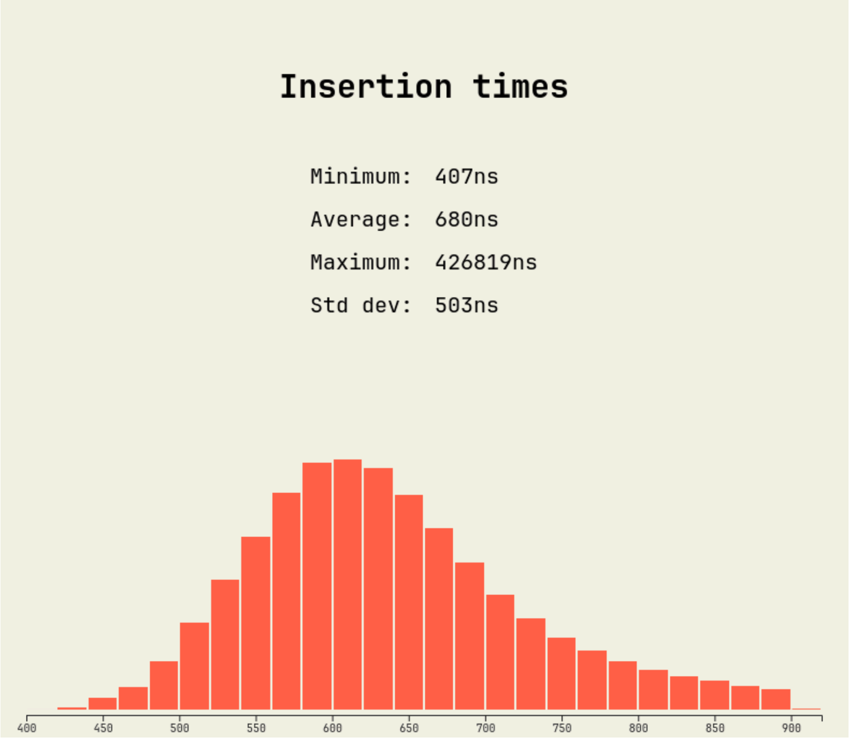


Figure 15: Insertion time of the RBT Tree

- Analysis of Results

***Note:*** *We had to use another computer to regenerate this graph last minute, so the recorded times are approx. half of the ones in past graphs, in virtue of the differences in the speed of both computers.*

As with the AVL tree, the recorded minimum value here probably corresponds to simple insertions where the new node is added as a leaf without requiring significant structural changes. The average time highlights the efficient and stable behavior of the structure in most insertions.

However, the maximum recorded time is significantly higher than the average. This occasional spike likely occurs in specific scenarios where the insertion requires multiple rotations and recolorings that propagate to higher levels of the tree, including cases where the root itself needs adjustment. These additional operations, though rare, are part of the Red-Black Tree's internal mechanisms to maintain its strict balance.

The difference between the standard deviation and the mean, though more pronounced than in search operations, still suggests a predominance of fast insertions, with few outliers. The distribution graph shows that most insertions cluster around the average time but also exhibits isolated spikes resulting from more costly insertions.

Despite occasional spikes in individual operation times, the structure's automatic balancing prevents these costs from persisting. The predictability and self-maintaining equilibrium are the key advantages that distinguish Red-Black Trees from unbalanced structures like traditional BSTs.

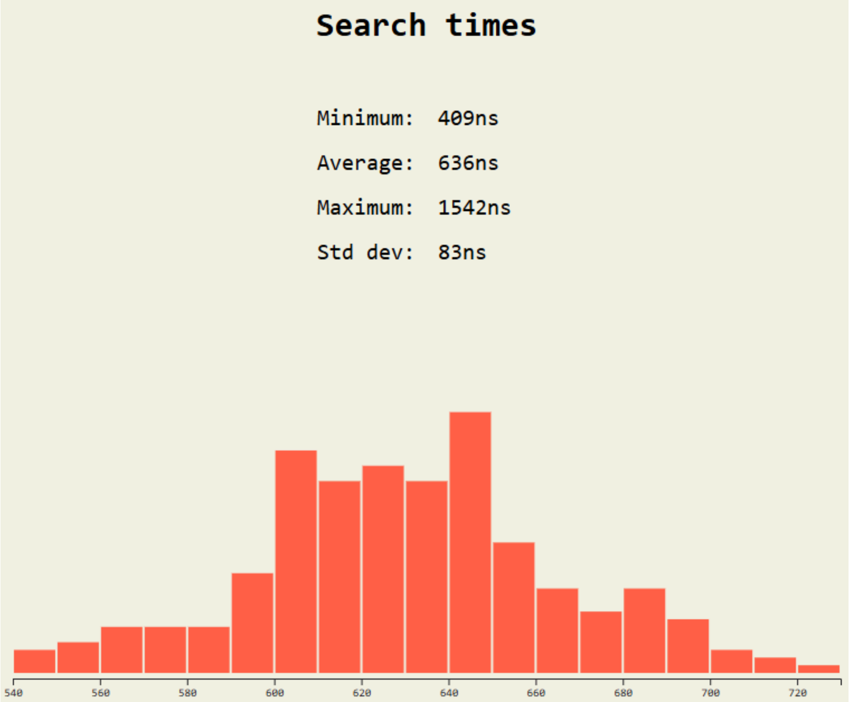


Figure 16: Search time of the RBT Tree

- Analysis of Results

The search times in the Red-Black Tree demonstrate the characteristic efficiency of this balanced structure. The minimum value occurs when the desired element is near the root, while the average time reflects the consistent performance maintained in most cases. The maximum time, although higher, shows how the structure's automatic balancing prevents the extreme cases that can occur in unbalanced trees.

The small difference between the standard deviation and the mean indicates that most search operations are concentrated within a narrow range of values. This consistency is one of the main advantages of the Red-Black Tree, which maintains a strict balance between node depth and operational efficiency.

The distribution chart shows a pattern concentrated around the mean value, with few operations at the extremes. This predictable behavior contrasts with the variable performance of BSTs, where the lack of balancing can lead to significant differences in search times. Like AVL trees, the Red-Black Tree ensures that even in the worst case, search times remain within controlled limits, maintaining the logarithmic complexity characteristic of balanced structures.

# 8. Conclusion

## 8.1. Raw stats

The raw statistics for the complete (with all the 10k documents) trees are:

| **Metric** | **BST** | **AVL** | **RBT** |
|---|---|---|---|
| Docs indexed | 10102 | 10102 | 10102 |

| **Metric** | **BST** | **AVL** | **RBT** |
|---|---|---|---|
| Words indexed | 2201736 | 2201736 | 2201736 |
| Total insertion time | 17.28s | 17.67s | 5.75s |
| Avg insertion time | 7.85 $\mu$s | 8.02$\mu$s | 2.36$\mu$s |
| Max insertion time | 9.39ms | 53.52ms | 41.96ms |
| Min insertion time | 0 | 0 | 0 |
| Total search time | 0.24ms | 0.25ms | 0.53ms |
| Avg search time | 980ns | 1022ns | 2193ns |
| Max search time | 3498ns | 5751ns | 45858ns |
| Min search time | 564ns | 516ns | 582ns |
| Total comparisons (search) | 3357 | 2899 | 2894 |
| Avg comparisons (search) | 13 | 11 | 11 |
| Max comparisons (search) | 25 | 16 | 16 |
| Min comparisons (search) | 1 | 2 | 2 |
| Node count | 20273 | 20273 | 20273 |
| Tree height | 32 | 16 | 16 |
| Min depth | 4 | 10 | 10 |
| Balance diff | 28 | 6 | 6 |
| Relative balance | 8 | 1.6 | 1.6 |

We see that the RBT and AVL tree are quite similar to each other, but the insert operation tends to be faster in the RBT, because the rotations and recoloring are less rigid than in AVL, permitting a better performance. On the other hand, the search is faster in AVL, because the tree is always balanced, meaning that the tree height its always minimized.

The RBT is usually preferred because it has a more general purpose: it offers a good balance between the height(that has implications for search) and stiffness(that has implications for insertion/ deletion).

## 8.2. Theoretical Results

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 [1], as it is a balanced tree. The height of an AVL tree with **n** nodes is at most:

$$h_{\max}(n) = 1,44O(\log(n)) = O(\log(n)) \tag{3}$$

Which might make the AVL interesting.

For the RBT, its known that for a tree with n nodes, the height is:

$$h_{\max}(n) = 2\log(n+1) = O(\log(n+1)) \tag{4}$$

As can be seen in [3].

This explains the behavior seen in Section 8.1 and the previous visualizations.

### 8.2.1. Time Complexity

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

| Operation | BST | AVL | RBT | Proof Idea |
|---|---|---|---|---|
| Insertion | $O(h(n))$ | $2O(h) + O(1)$ rotations(because it has to actualize the height of the nodes, worst case is going up to the root) | $O(h) + O(1)$ rotations and recoloring | Keys are compared on a root-to-leaf path, so the height of the tree determines the number of comparisons. For BST, this is linear in the worst case, while AVL and BRT guarantees logarithmic height due to balancing rules. |
| Search | $O(h(n))$ | $O(h(n))$ | $O(h(n))$ | Shown in Section 4 |

### 8.2.2. Memory Usage

All implementations use:

```cpp
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: $h = n - 1 \to O(n)$ extra bytes; may overflow for large $n$.
- AVL stack depth: $h = O(\log n) \to$ asymptotically optimal.
- BST stack depth: $h = 2O(\log(n + 1)) \to$ worst than AVL, but its better than BST.

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

### 8.2.3. Data visualization and extras

Besides the required "search" and "stats" subcommands, we created a "view" subcommand to visualize some of the data we collected for individual trees in a radial form. Running this subcommand for any type of tree (*./<tree type> view <number of documents> <directory>*) goes through the indexing of the tree and then starts a HTTP server in the running machine. Opening the URL printed to *stdout* will load a visualization of tree, made with *D3.js*. The visualizations give us interesting ways to compare the structure of the trees, as seen by looking at the graphs of BST, AVL and RBT trees built from the same amount of documents (10) :

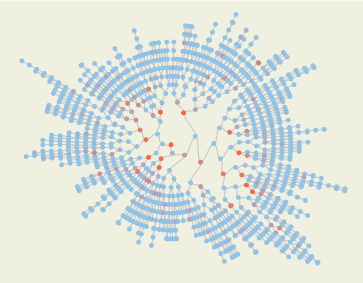


Figure 17: Structure of the BST Tree

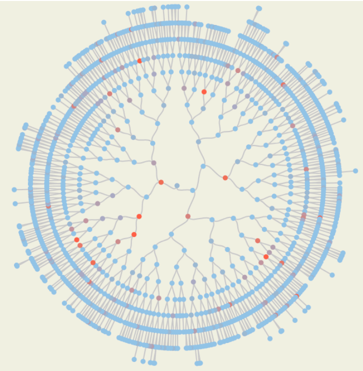


Figure 18: Structure of the AVL Tree

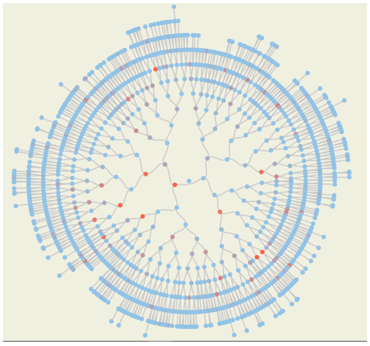


Figure 19: Structure of the RBT Tree

As expected, AVL's and RBT's node heights are less extreme, as a result of the rotations made after each insertion. Notice all trees hold the same amount of nodes, but the BST's and RBT's ones are compressed to a smaller circle to accomodate the greater height of the tree. This facts also makes AVL looks more circular.

These visualizations make clearer how the RBT is a perfect "middle term" between height-efficiency(for being more cirular) and height-distribution (for allowing unbalancing ergo "higher" nodes).

Another interesting analysis we interest ourselves in doing is seeing the words frequency. As we can see below, the most common words are, as expected, articles, prepositions and conjunctions, since these are common for the composition of the words.

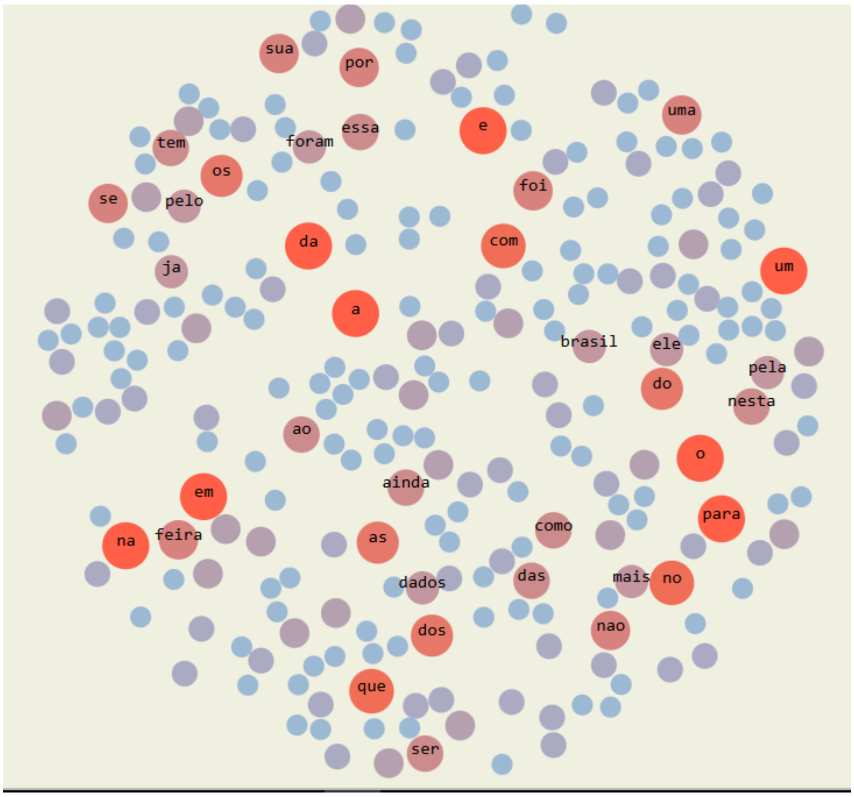


Figure 20: Insertion time of the RBT Tree

These are only extra analysis that give us a greater comprehension on what is happening while constructing the inverted index and give us a interesting way of seeing the elaboration of each tree.

#### 8.2.3.1. Fun Fact

The root of the tree (that is in the middle of the visualization of Section 8.2.3) is the word "meio" ("middle" in portuguese).

## 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

My biggest challenge was dealing with my VSCode, which had many issues, such as not being able to run programs, crashing while I was working on the project, among other things. Additionally, some of my commits were made as "root" instead of my user, due to the various bugs in VSCode. The solution

I found was to write the code on Google Colab (after some time my Chrome start crashing too) and ask my classmates to test it on their computers.

Regarding the project implementation, some of the difficulties were:

- The implementation of the BST insertion had many cases to be checked to ensure they were working as expected.
- There were many errors during the implementation of the RBT, especially in the insertion fix-up required to maintain the tree's logic.
- I faced difficulties dealing with fstream when correcting the use of the library in the data file.

And the worst villain of all was time, or rather, the lack of it… With other assignments to be done alongside this one, test days, to-do lists, English courses (which had to be put on hold for a bit), all of that and more resulted in very stressful submission periods, coinciding with many other tasks.

But, at the end of it all, I enjoyed working with a very diverse group, including people from other courses.

## 9.3. Gabrielle Mascarelo

It may be silly but my main hurdle was to be able to run the MakeFile and dealing with it for the first time. Even though it appeared in our classes, I couldn't get it at the time. It was hard to deal with WSL in my computer.

I also never did something of the kind about the CLI structure. I have to admit I don't quite got the tree structures very easily in my head but that's on me for not going to all the classes. I regret that strongly.

I just think the deadline was short for me to learn and manage that bunch of new things while doing a lot of other stuff.

## 9.4. Nícolas Spaniol

Time management was, by far, my biggest difficulty, given this project was proposed in a very unfortunate moment, when we had lots of other time-consuming assignments from the other courses.

After that, I found some difficulty when dealing with *git submodules* (that we used to include third-party libraries in our project), having to create a fork of *tinyhttp* (our HTTP server library) to deal with a missing library for JSON parsing.

Using *make* also proved to be very difficult, as I had no past experience with this program and had to learn its distinct syntax in order to make compiling our program easier and faster.

## 9.5. Gabriel Carneiro

I've got trough bad times testing and implementing correctly the AVL Tree, since initially I had a wrong interpretation about how rotations worked and how to test correctly, since all examples have to be made by hand. I've also had to be a little bit more careful with pointers and references, so working on the project gave me a good lesson.

For me, the deadlines were quite short. Despite the BST tree being quite simple, the other two are not so trivial (at least for me) and implementing them while having to do other college stuff was challenging, but I guess I made a decent job.

I really think I learned a lot by doing this project, not only about data structures but also on how work together with my group.

# 10. Source code

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

Here is the organization of the code:

```
dsa-final-project

->Unity                    * C++ Library for unit testing
->tinyhttp                 * C++ Library for http servers
->docs                     * Report and images
->data                     * Dataset of approximately 10000 documents (the first one)
->view                     * HTML configuration for visualization
->src
|-->bst                    * BST implementation
|-->avl                    * AVL implementation
|-->rbt                    * RBT implementation
|
|-->cli.cpp                * CLI structuring
|-->cli.h                  * and library
|
|-->data.cpp               * Data scrapping and treating
|-->data.h                 * and library
|
|-->tree_utils.cpp         * General binary tree functions
|-->tree_utils.h           * implementation, library
|-->test_tree_utils.cpp    * and testing
```

# 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 and the initial search in the CLI;

- Testing all functions related to the RBT;
- Fixing bugs and finishing the RBT code;
- Writing `report.typ`.

## 11.3. Eliane Moreira

Contributed with:

- Writing and testing all functions related to the BST;
- Writing all functions related to the RBT;
- Writing and documenting functions for `tree_utils.cpp`;
- Fixing bugs in `data.h`;
- Writing `report.typ`.

## 11.4. Nícolas Spaniol

Contributed with:

- Code reviews, suggestions for improvements in the codebase;
- Built from scratch the JavaScript visualization of all trees;
- CLI structuring and statistics.
- Writing `report.typ`.

## 11.5. Gabriel Carneiro

Contributed with:

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

# Bibliography

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

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

[3] R. A. Brown, "Comparative Performance of the AVL Tree and Three Variants of the Red-Black Tree." [Online]. Available: https://arxiv.org/pdf/2406.05162