FUNDAÇÃO GETULIO VARGAS
ESCOLA DE MATEMÁTICA APLICADA
Data Structures and Algorithms

INVERTED INDEX
&
COMPARATIVE ANALYSIS OF DATA STRUCTURES

ERIC MANOEL RIBEIRO DE SOUSA
EVERTON COSTA REIS
LUCAS MENEZES DE LIMA
RODRIGO SEVERO ARAÚJO
VICTOR GABRIEL HARUO IWAMOTO

Rio de Janeiro – RJ
June 2025

Contents

1 Introduction

Inverted Index and Comparative Analysis of Data Structures is a project developed as part of the second assessment for the Data Structures course in the Applied Mathematics program at FGV.

The project's goal is to demonstrate proficiency in the C++ programming language by applying concepts covered throughout the classes. To this end, different tree-based data structures are implemented—including **Binary Search Tree (BST)**, **AVL**, and **Red-Black Tree (RBT)**—in order to analyze their behavior and performance in building an inverted index.

The inverted index consists of associating words with lists of documents in which they appear, also allowing for the alphabetical ordering of these words. The project, therefore, focuses on comparing the efficiency of the different trees in terms of insertion, sorting, and searching within this context.

2 Structs

2.1 Binary Search Tree - BST

2.1.1 Concept

A **Binary Search Tree (BST)** is a data structure used in computer science to organize and store data in an ordered manner. Conceptually, a tree must adhere to specific criteria to be considered a binary search tree:

Each node in a Binary Search Tree has, at most, two children: a left child and a right child. The left child contains values smaller than the parent node, while the right child contains values larger than the parent node.

This hierarchical structure allows for efficient search, insertion, and deletion operations on the data stored in the tree. In time complexity analysis, the efficiency of a BST is linear in relation to the height of the tree. Furthermore, the best time for basic operations—insertion, deletion, and search—takes $O(\log n)$ for n nodes. In the worst case, when the tree degenerates (becoming similar to a linked list), the time can reach $O(n)$.

2.1.2 Implementation

- **Insert Function:**

  For inserting a word into the tree, the main idea is to preserve the existing structure. Thus, the insertion process is divided into two primary cases: the word already exists in the tree, or it is a new word. This distinction is made using the previously implemented 'Search' function.

  In the first case, where the word already belongs to a node in the tree, the function searches for the ID of the text to be inserted within the 'documentIds' list of the corresponding node. If the ID is already present, no action is taken. Otherwise, the ID for that text is added to the list. As an optimization, the list of IDs is traversed in reverse. Since texts are inserted in ascending order, this means that if the word was already present in the text, its ID would have been the last one added, thus saving computational time. However, to ensure the code remains functional regardless of the file insertion order, the decision was made to retain a full loop through the list.

  On the other hand, if the word is not already in the tree, it must be added as a child node of a leaf. Since the 'Search' function already stores the parent node, one simply compares the new word with the parent node's word to determine if the new node will be inserted as a left or right child. A special case occurs when the tree is empty (i.e., the parent is null); in this scenario, the new node is set as the root of the tree.

2.2 AVL

2.2.1 Concept

An **AVL Tree** is a **Binary Search Tree (BST)** that distinguishes itself by being self-balancing. The key characteristic of an AVL tree is that for any node in the tree, the absolute difference between the heights of its left and right subtrees must be at most 1. This difference is known as the *balancing factor*.

Whenever an operation (such as insertion or deletion) causes an imbalance—that is, the balancing factor of a node becomes greater than 1—the tree performs a rebalancing. This process is carried out through specific rotations, which reorganize the nodes to restore the balancing property.

This self-balancing mechanism ensures that operations like insertion, search, and deletion are executed with a time complexity of $O(\log n)$, where n is the number of nodes in the tree. This is crucial for maintaining efficiency, even in trees with a large volume of data. Furthermore, it prevents the degeneration that can occur in a standard BST.

2.2.2 Implementation

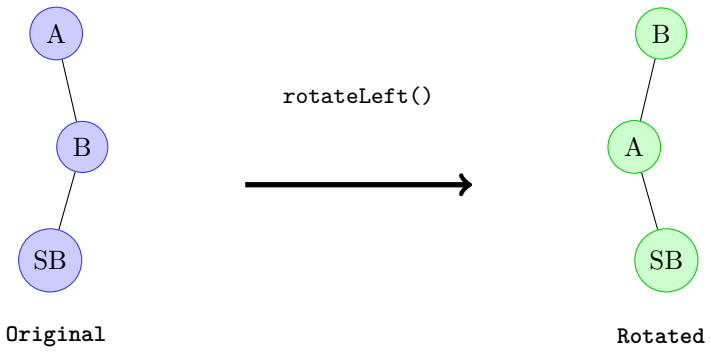
- **Function BalancingFactor:**

  In short, the function calculates the balancing factor of a given node by finding the difference between the heights of the right and left subtrees.

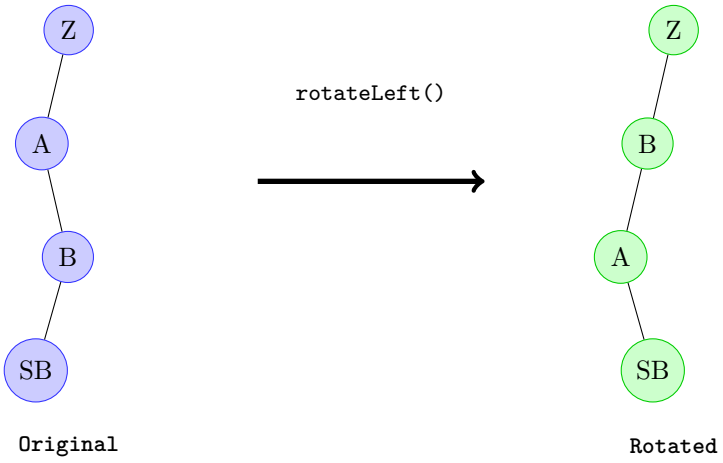
- **Function Rotates:**

  As part of the tree's rebalancing, it is necessary to perform rotations on parts of it. Thus, these functions execute rotations in each direction. The transformations performed by each function will be visually represented in the following graphs:

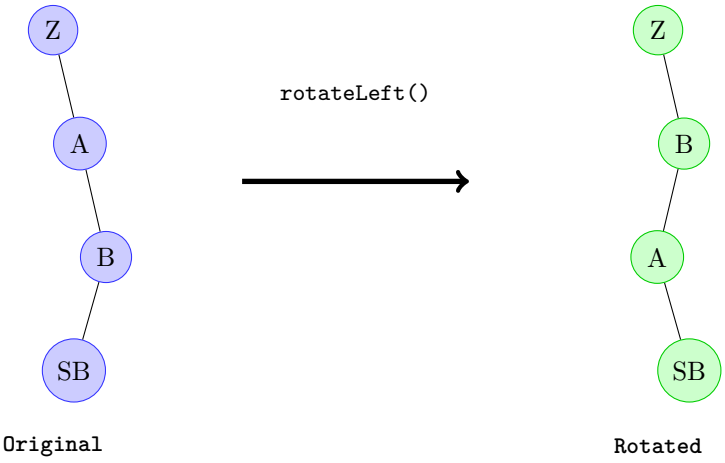
  - ***First Case:*** The node is the root and has no parent.



  - ***Second Case:*** The node is a left child.



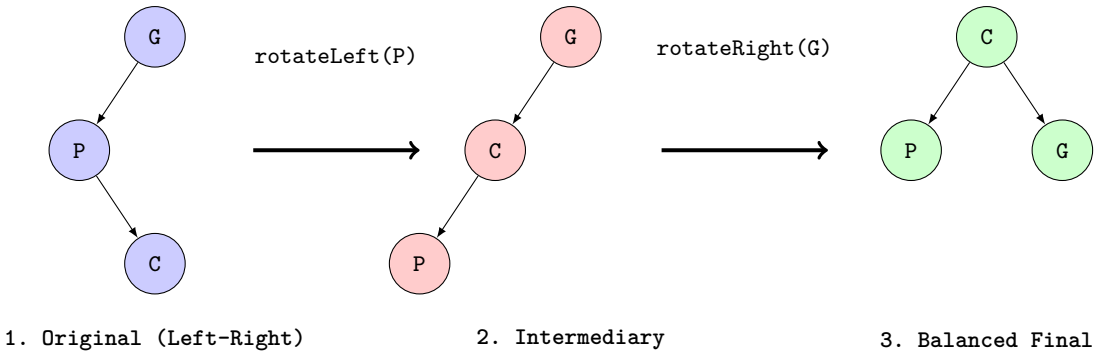
– ***Third Case:*** The node is a right child.



The rotations presented are derived from the left rotation operation, where the node to be rotated is A and SB represents a subtree. For a right rotation, a similar but mirrored logic is applied. It is important to note that the cases shown do not occur in valid binary search trees, but they constitute the base cases for the rotation algorithms.

Furthermore, there are what are known as double rotations, which are used when an imbalance occurs at two levels—for example, when a node is a right child, but its own child is a left child (or vice-versa). In these situations, the `rotateRightLeft` and `rotateLeftRight` functions are used to restore the tree's balance.

– ***Rotate Left Right:*** Double Rotation



- **Balance Function:**

  The `balance` function is responsible for checking if a given node violates the balancing property of an AVL tree. To do this, it first calculates the node's balancing factor. If the absolute value of this factor is greater than 1, it indicates an imbalance and, therefore, the need for a rotation.

  Next, the function determines if the imbalance is in the left or right subtree and checks whether a single or double rotation is needed based on the balancing factor of the corresponding child. After applying the appropriate rotation, the function returns the new, balanced node.

- **Insert Function:**

  In general, insertion into an AVL tree follows the same logic as insertion into a BST, divided into two cases: the word is already in the tree, or it is a new entry. After this check, the balancing process begins.

  If the word does not yet exist in the tree, after its insertion as a child of a leaf node, a recursive check is performed from the new node's parent up to the root. At each step, it verifies if the current node maintains the AVL balancing property. If the tree remains balanced, no action is necessary; otherwise, the `balance` function is called to restore the local balance.

  This process ensures that, by the end of the insertion, all nodes above the new element—all the way to the root—are also balanced. In this way, the entire tree structure continues to adhere to the AVL balancing rules, maintaining its efficiency for search, insertion, and deletion operations.

2.3 Red-Black Tree - RBT

2.3.1 Concept

A **Red-Black Tree** is a variation of the binary search tree that uses a set of properties to stay balanced. This guarantees that insertion, search, and deletion operations have logarithmic time complexity, even in the worst cases.

To ensure this balance, each node in the tree is assigned a color—either red or **black**—and after each modification (insertion or deletion), the tree's balance is restored through rotations and recolorings as needed.

A Red-Black Tree must obey the following properties:

1. Node Color: Every node is either red or black.
2. Root Property: The root of the tree is always black.
3. Red Property: A red node cannot have red children (i.e., there are no two consecutive red nodes on any path).

4. Black Property: Every path from a given node to any of its descendant NIL nodes contains the same number of black nodes.
5. Leaf Property: All leaves (NIL nodes) are black.

2.3.2 Implementation

- **Transplant Function:**

  The `transplant` auxiliary function is responsible for replacing one node, u, with another node, v, in the tree. In other words, it makes the parent of u point to v. Furthermore, if v is not null, the function updates the parent pointer of v to point to the former parent of u.

- **FixInsert Function:**

  Given the properties of a Red-Black Tree, we first need to determine the color of the newly added node, which we will call z. As established, z can only be black or red. Let's see what happens in each case:

  If z is black:

  - It violates Property 4, unless z is the root.

  If z is red:

  - Property 4 is maintained.
  - If its parent is red, Property 3 is violated.
  - If it is the root, Property 2 is violated.

  Therefore, by convention, every node initially inserted into a Red-Black Tree is colored red. When this insertion leads to a violation of the tree's properties, the issue can be resolved by handling a series of specific cases.

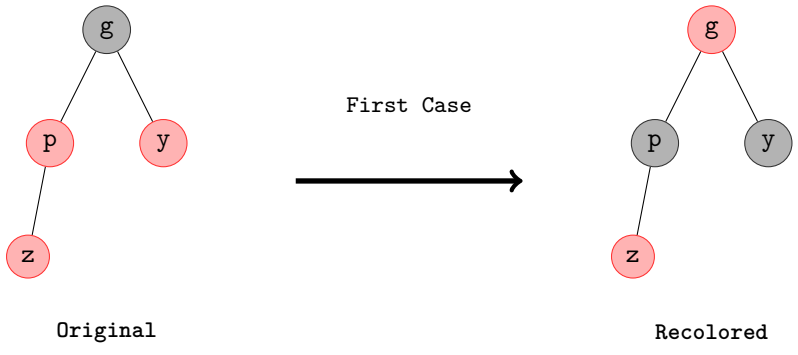
  In addressing these violations, three main cases (and their symmetric counterparts) are distinguished.

  - ***First Case:*** *The uncle of the newly inserted node z is red.*

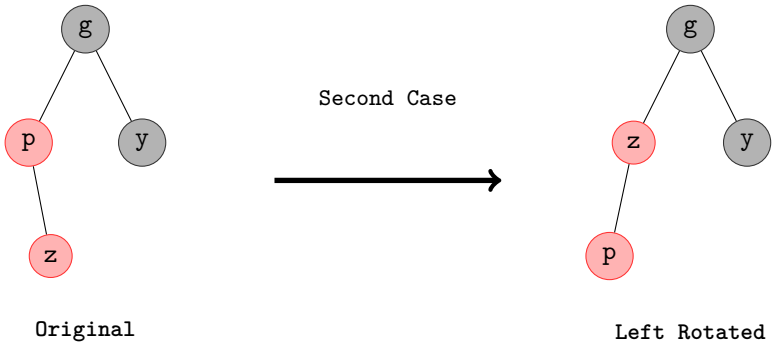
    Since z is inserted as a red node, its parent, p, might also be red. This configuration violates the Red Property of a Red-Black Tree (a red node cannot have a red child). Assuming that the uncle of z, denoted by y, is also red, we perform a recoloring to fix the violation.

    In this situation, we assume that the grandparent of z, denoted by g, was black before the insertion—which guarantees the tree was valid up to that point. The fix consists of recoloring both p and y to black, and g to red. This resolves the local conflict.

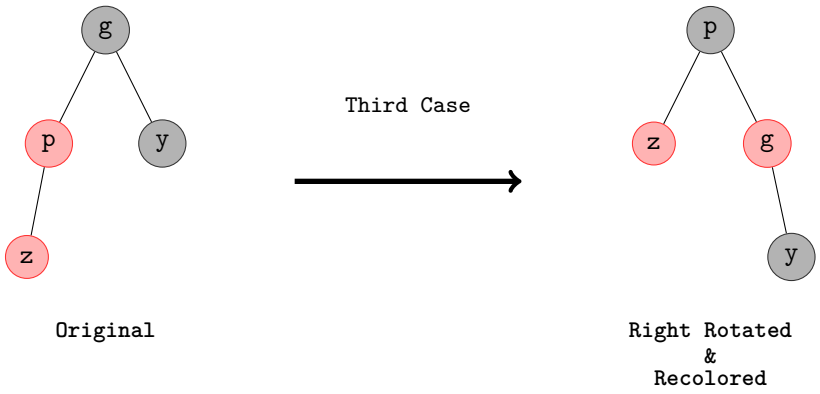
    However, this recoloring might have introduced a new conflict at the level of the grandparent, g, if its parent is also red. Therefore, the verification and correction must be applied recursively up the tree until all properties are restored.



– ***Second Case:*** *The uncle is black and the inserted node is a right child.*
In this scenario, node z is inserted as red, its parent p is also red, while the uncle is black. Furthermore, z is a right child of p, and p is a left child of the grandparent, g.
Since the uncle is black, we cannot resolve the issue with a simple recoloring. To handle the imbalance created by z's position as a right child, we apply a **left rotation** on its parent, p. This rotation transforms the current problem into the configuration of the **Third Case**, which is simpler to resolve.
This rotation repositions the nodes, effectively preparing the structure for the final adjustments described in the next step.



– ***Third Case:*** *The uncle is Black and z is a left child.*
Since the uncle is black, it is considered stable. Therefore, the imbalance caused by z can be resolved by applying a **right rotation** on the grandparent, g, and performing a **recoloring**.



- **Function Insert:**
Similar to the other implementations, the `Insert` function for the RBT handles two cases: if the word already exists in the tree, it simply adds the document ID to the list. Otherwise, it adds a new node as a leaf. For the RBT, at the end of this process, the tree is rebalanced using the `FixInsert` function.

2.4 General Functions

Since BST, AVL, and Red-Black trees are all binary trees, they naturally share similar functions due to their fundamentally similar structure. Therefore, the `create`, `search`, and `destroy` functions are the same for all of them.

- **Create Function:**
The `create` function is responsible for creating the tree by initializing the root and the NIL node (used in the Red-Black Tree) as null pointers, thereby preventing invalid memory access.

- **Search Function:**

  By definition, a binary tree is constructed so that for any given node, smaller values are stored in the left subtree and larger values in the right. Based on this principle, the search operation is executed efficiently by leveraging the structure's inherent order.

  The search algorithm traverses the tree by comparing the target word with the keys in each node, using a `compare` function that returns the difference of the first diverging letter based on the ASCII table. Depending on this return value, the algorithm decides whether to traverse to the left or right child. This process repeats until the word is found or a null pointer is reached (a child of a leaf), indicating the word is not in the tree. If the word is located, the function returns a structure containing the result information. For statistical purposes, a timer is used to measure the search execution time.

  Regarding complexity, the algorithm's efficiency is closely tied to the tree's height, with a complexity of $O(h)$, where h is the height of the binary tree. In the best-case analysis, when the tree is perfectly balanced, the complexity is $O(\log n)$, where n is the number of nodes. On the other hand, in the worst case—when the tree degenerates into the shape of a linked list—the search has a complexity of $O(n)$, this applies to the BST, which does not self-balance.

- **Destroy Function:**

  Finally, the `destroy` function is responsible for freeing all memory allocated by the binary tree structure. To accomplish this, it uses a helper function called `deleteTreeRecursive`, which employs a Post-Order Depth-First Search (DFS). Through recursion, it deletes nodes from the leaves up to the root. Once the node deallocation process is complete, the tree's root pointer is reset to null, ensuring the structure is completely cleared.

3 Analysis and Graphics

The analysis was performed using the documents available from the first database.

3.1 Insertion Performance

Table 1 presents performance metrics for insertion operations, for different quantities of documents (n), where **Time (s)** measures the total duration of the process, **Comparisons** counts the total number of comparison operations between nodes, and **Relative Efficiency** indicates the comparative gain in relation to the BST.

n (docs)	Structure	Time (s)	Comparisons	Relative Efficiency
10	BST	0.021	38,668	1.00
	AVL	0.020	39,187	0.99
	RBT	0.021	34,948	1.11
100	BST	0.230	548,119	1.00
	AVL	0.233	507,573	1.08
	RBT	0.229	484,434	1.13
1000	BST	1.965	5,689,724	1.00
	AVL	1.912	5,037,297	1.13
	RBT	1.922	5,028,881	1.13
5000	BST	10.763	29,821,186	1.00
	AVL	10.556	26,207,636	1.14
	RBT	10.503	26,660,495	1.12
10103	BST	24.268	59,612,852	1.00
	AVL	23.721	52,301,192	1.14
	RBT	23.678	53,405,044	1.12

Table 1: Insertion performance with relative efficiency compared to BST

Despite the additional balancing operations, AVL and RBT show comparable or superior efficiency to BST in insertions. For large volumes ($n \geq 1000$), both perform 12-14% fewer comparisons, which is reflected in consistently lower times. The RBT stands out in smaller scenarios ($n = 10$) with an 11% gain in relative efficiency, while the AVL maintains a stable advantage (14%) in larger sets ($n \geq 5000$).

Figure 1 demonstrates the distribution of insertion time, showing that RBT has the lowest temporal variability among the analyzed structures, while BST exhibits the greatest dispersion in insertion times.

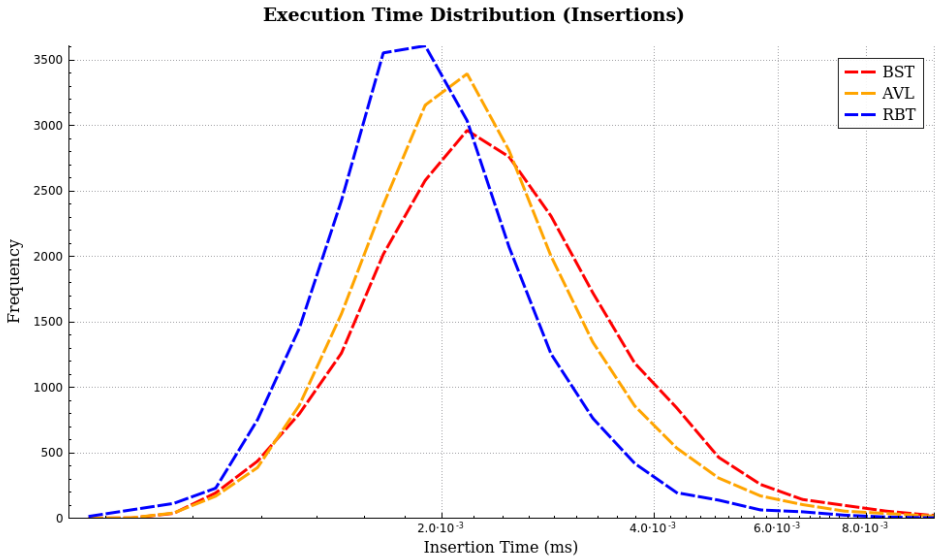


Figure 1: Number of comparisons by number of words

Figure 2 reveals that, although insertions in AVL and RBT involve additional balancing operations, the superior efficiency in search operations compensates for this initial cost, particularly in datasets with many repeated words.

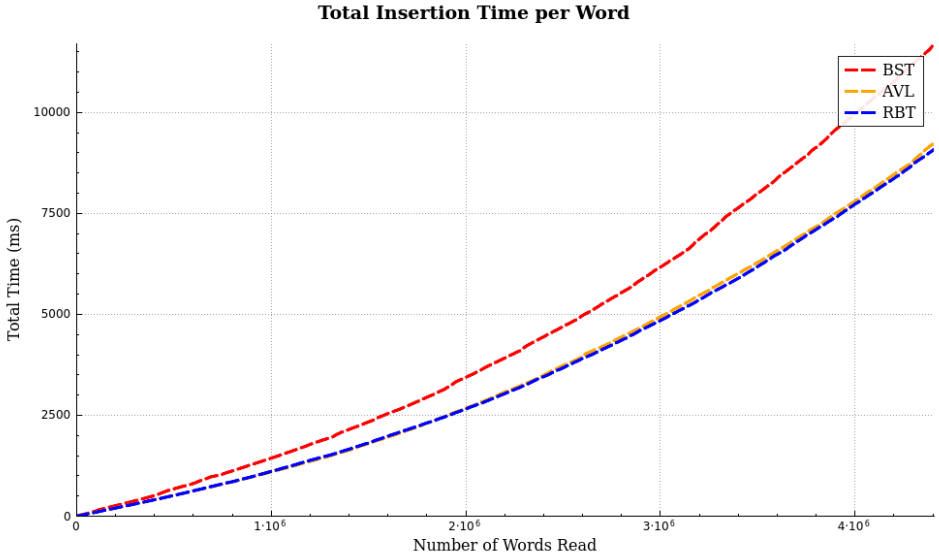


Figure 2: Number of comparisons by number of words

3.2 Search Performance

Table 2 displays results for search operations for different numbers of documents (n), with **Time (ms)** representing the average duration per operation, **Comparisons** showing the average number of comparisons per search, and **Relative Efficiency** indicating the comparative gain in relation to the BST.

n (docs)	Structure	Time (ms)	Comparisons	Relative Efficiency
10	BST	2.477	16,657	1.00
	AVL	2.466	12,252	1.264
	RBT	2.422	12,312	1.261
100	BST	12.755	108,933	1.00
	AVL	12.215	79,420	1.271
	RBT	12.321	79,940	1.266
1000	BST	28.333	313,739	1.00
	AVL	26.595	226,639	1.278
	RBT	27.285	228,744	1.271
5000	BST	34.393	380,500	1.00
	AVL	33.366	274,630	1.278
	RBT	34.111	277,234	1.271
10103	BST	35,627	382,313	1.00
	AVL	34.379	275,883	1.278
	RBT	34.022	278,510	1.271

Table 2: Search performance with relative efficiency compared to BST

The balanced trees demonstrate expressive gains in search operations, consistently reducing comparisons by 26-28% relative to the BST across all scales. This optimization is directly reflected in the time: the AVL and RBT maintain a temporal advantage in all cases.

Figure 3 presents the distribution of the number of comparisons made during searches, as a function of the number of words, for the three analyzed structures: BST, AVL, and RBT. It is observed that the balanced trees (AVL and RBT) have distributions that are significantly more concentrated around a single value, with frequency peaks at approximately 14 comparisons—indicating that most words are located after this number of steps. In the BST, the distribution is more dispersed, with a peak shifted to around 19 comparisons, reflecting greater variability compared to the balanced trees.

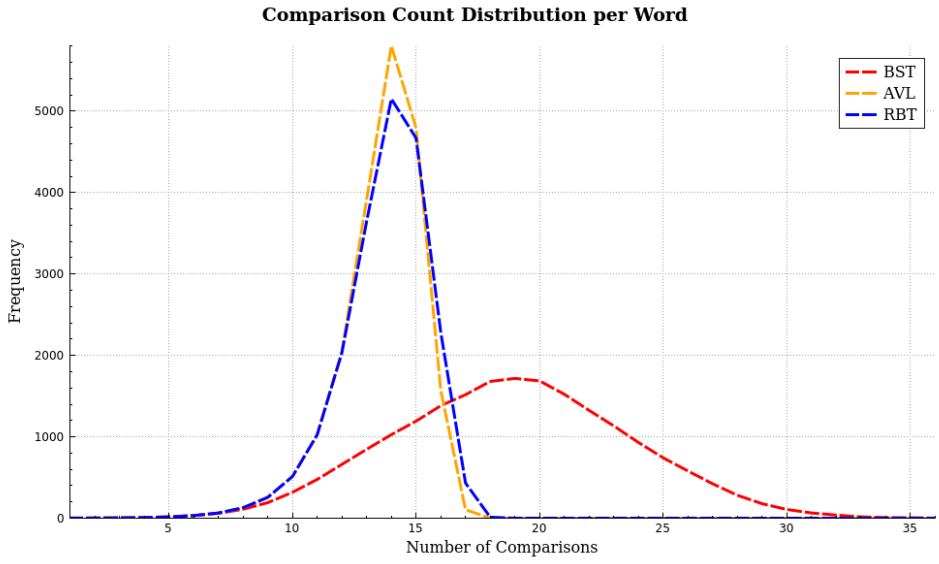


Figure 3: Number of comparisons by number of words

In particular, this is the level of the tree that has the most nodes. This implies that searches in AVL and RBT are not only faster but also more predictable. On the other hand, the BST—by not guaranteeing balancing—can suffer structural degradations in unfavorable scenarios, which leads to an increase in the average number of comparisons.

3.3 Maximum and Minimum Branch

Table 3 shows the tree height statistics, where **Max. Dist** represents the longest distance from the root to any leaf, **Min. Dist** indicates the shortest distance from the root to a leaf, **Difference** calculates the variation between these heights, and **Ratio** expresses the proportion between maximum and minimum height, with values close to 1 being indicative of better balancing.

n (docs)	Structure	Max. Dist	Min. Dist	Difference	Ratio
10	BST	23	4	19	5.75
	AVL	11	7	4	1.57
	RBT	12	7	5	1.71
100	BST	29	5	24	5.80
	AVL	14	9	5	1.56
	RBT	15	9	6	1.67
1000	BST	34	5	29	6.80
	AVL	16	10	6	1.60
	RBT	16	10	6	1.60
5000	BST	36	5	31	7.20
	AVL	16	10	6	1.60
	RBT	17	10	7	1.70
10103	BST	36	5	31	7.20
	AVL	16	10	6	1.60
	RBT	17	10	7	1.70

Table 3: Tree height statistics for different data volumes

It is noted that even for small sets ($n = 10$), the BST already exhibits a significant disparity between minimum and maximum heights (ratio of 5.75). As the data volume increases, this difference widens, reaching ratios greater than 7. In contrast, the balanced structures (AVL and RBT) maintain height differences of less than 7 nodes and maximum ratios of 1.70, demonstrating consistently stable behavior regardless of the data scale. The following graphs are on a logarithmic scale for better visualization.

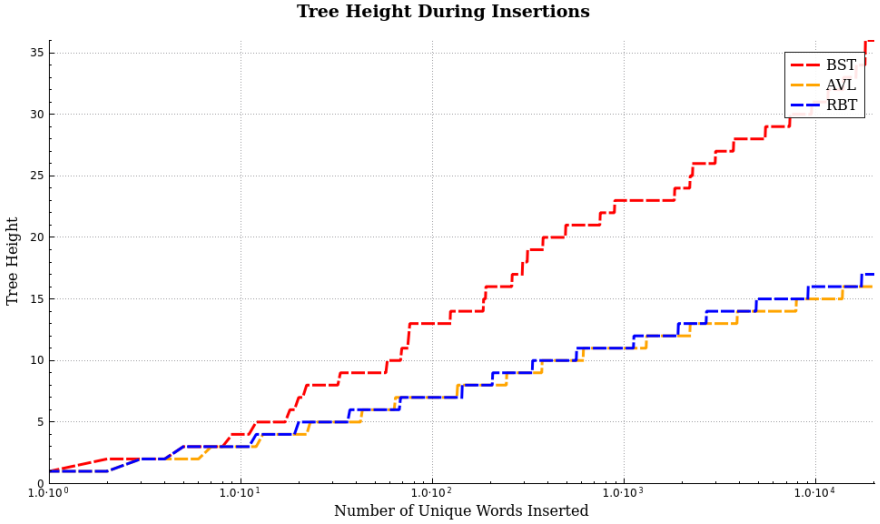


Figure 4: Tree height per insertion

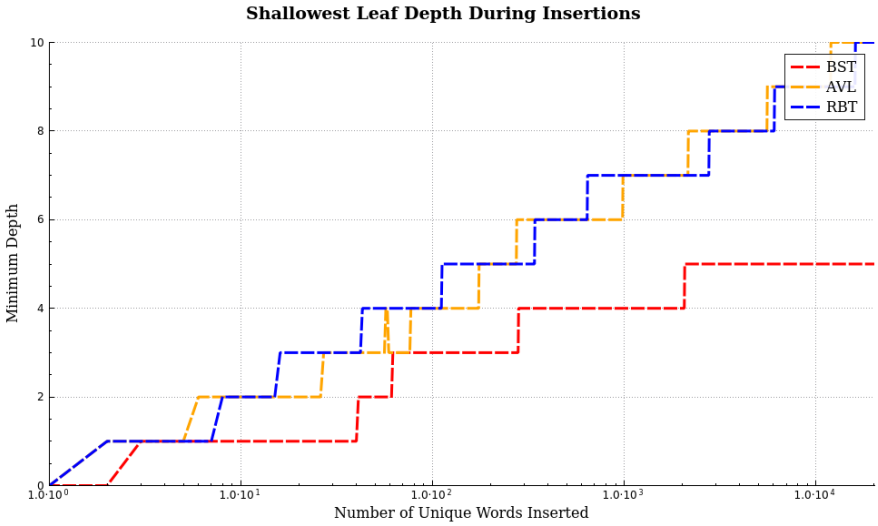


Figure 5: Tree height per insertion

Note that, in general, as evidenced by figures 4-5, in AVL and RBT trees—being balanced—it is necessary to insert a larger quantity of nodes for the total height of the tree (or the size of the smallest branch) to increase. Their balancing restricts the excessive growth of the branches. On the other hand, in the BST, this behavior is not guaranteed: in pathological cases, such as ordered insertions, two new nodes may be sufficient to increase the tree height by two units.

Note that in figure 5, there was a moment when the size of the smallest branch decreased. At first, we might think this is incorrect. But in fact, this is not a problem, as it can really occur, see the example:

In figure 6 we have an AVL tree such that the size of the smallest branch is 4, and in figure 7 it is the same tree after inserting node 650. After this insertion, this node becomes the root and the size of the smallest branch will be 3.

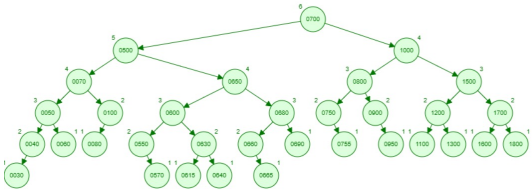


Figure 6: Before inserting node 650

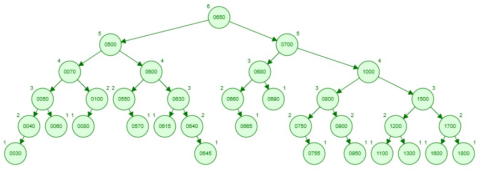


Figure 7: After inserting node 650

3.4 Conclusion

Comparing the three structures, the BST stands out for its simplicity, but can have poor performance when data is inserted in an orderly fashion—becoming practically a list. The AVL tree keeps the tree always very well balanced, which guarantees fast searches, but requires more rotations, making it better for cases with many queries and few insertions. The RBT, on the other hand, finds a middle ground: it performs fewer rotations than AVL and still maintains good overall performance, making it a good choice when there are many insertions and modifications to the tree.

4 Project Organization

4.1 Structure of the Project

Below is an overview of the project's structure. Files named with a [trees] placeholder (e.g., [trees].cpp) have three corresponding versions, one for each data structure: bst, avl, and rbt.

```
shattered/
|-- src/                        # Main source code
|   +-- Makefile
|   +-- trees.cpp
|   +-- trees.h
|   +-- data.cpp
|   +-- data.h
|   +-- main_trees.cpp
|   +-- test_trees.cpp
|   +-- test_tree_utils.cpp
|   +-- tree_utils.cpp
|   +-- tree_utils.h
|-- plots/                      # Plots
|   +-- plot.cpp
|   +-- plot.pro
|   +-- qcustomplot.cpp
|   +-- qcustomplot.h
|-- docs/                       # Documentation
|   +-- files.tex
|   +-- main.pdf
|-- data/                       # First Dataset
|   +-- data.zip
|-- data2/                      # Second Dataset
|   +-- data_v2.zip
|-- README.md                   # General view of the project
```

5 What Each One Has Done

5.1 Individual Contribution

Throughout the project's development, all team members remained consistently active, as evidenced by the commit history. Each member contributed to both discussions and coding. To optimize the workflow, the project was modularized, allowing specific parts to be primarily developed by certain members—albeit with collective support and review.

To ensure the application of knowledge acquired in Data Structures and Algorithms, each member actively participated in implementing at least one tree structure. Development began with the common auxiliary functions for all trees, such as node creation, printing, and property calculation, which were implemented by Rodrigo Severo.

The project is founded on the Binary Search Tree (BST), which serves as the basis for the other trees. Accordingly, the implementation of the BST, along with the initial project structure, was carried out by Victor Iwamoto. He developed the core tree functions, such as search, destroy, and create.

For the self-balancing trees, Everton Reis and Lucas Menezes jointly developed the entire logic for the AVL Tree, implementing the necessary rotations to maintain its balance during insertions.

Given the greater complexity of the Red-Black Tree, the pair programming technique was adopted

to ensure collaborative development and higher quality control. In this process, Rodrigo Severo acted as the driver (coder), while Eric Ribeiro served as the navigator (strategist and reviewer).

Furthermore, Eric Ribeiro was also responsible for the user experience design of the command-line interface (CLI). He mapped out potential user errors and implemented appropriate handling mechanisms. To improve overall usability, he created the project's *Makefile* and developed the data reading system.

Everton Reis was in charge of the internal documentation of the source code. This required him to review and understand the modules developed by the other members, which meant he also acted as a technical reviewer.

Lucas Menezes focused on the project's statistical analysis, developing graphs from the trees' performance metrics using a C++ library specialized in data visualization.

Based on the statistics generated by the CLI and the graphical visualizations, Rodrigo Severo wrote the project's conclusions, comparing the practical performance of the structures and highlighting their respective advantages and limitations.

Finally, Victor Iwamoto was responsible for drafting the final report, in which he detailed the developed structures and functions. This task required a complete review of the code, allowing him to also take the lead on quality assurance, identifying bugs and relaying them to their respective authors for correction. He was also in charge of the report's visual identity and final formatting.

6 How to Run

This guide provides instructions for compiling and running the project on Linux systems or the Windows Subsystem for Linux (WSL).

6.1 Prerequisites

Ensure the following packages are installed on your system:

- C++ Compiler (g++ 11.4.0 or later)
- `make`
- `unzip`
- Qt5 development libraries

6.2 Setup

6.2.1 1. Install Dependencies

On Debian-based systems like Ubuntu, run the following commands to update your package lists and install the required dependencies:

```
# Update package lists
sudo apt update

# Install make and unzip
sudo apt install make unzip

# Install Qt5 development tools
sudo apt install qtbase5-dev qtchooser qt5-qmake qtbase5-dev-tools -y
```

6.2.2 2. Prepare the Datasets

The project datasets are provided as compressed `.zip` files. Due to how they are archived, unzipping them creates a nested directory structure (e.g., `data/data/`). The commands below will unzip the archives and move the files into their correct locations.

From the project's root `InvertedIndex/` directory, run:

```
# Unzip the primary dataset into the data/ directory
unzip data/data.zip -d data/
# Unzip the secondary dataset into the data2/ directory
unzip data2/data_v2.zip -d data2/

# Move the files out of the nested 'data' directories
mv data/data/* data/
mv data2/data/* data2/
```

6.3 Compilation & Execution

This project uses a `Makefile` located in the `src/` directory to simplify compilation and execution. You can run the program with default settings or provide custom parameters.

6.3.1 Using the Makefile

The following commands should be run from the project's root directory.

```
#To run with default settings:
(cd scr/ ; make)

#To specify the number of documents (uses default BST tree):
(cd scr/ ; make num=n)

#To run with fully custom parameters:
(cd scr/ ; make tree=tree path="../data_folder/" num=num cmd=stats)

#Delete object files and executables:
(cd scr/ ; make clean)
```

The following parameters can be passed to the `make` command to customize execution.

Parameter	Description	Default	Arguments	Example
`tree`	Data structure to use	`bst`	bst, avl, rbt	`tree=avl`
`num`	Number of documents to process	`10103`	1, 2, 3, ...	`num=1000`
`cmd`	Command to execute	`search`	search, stats	`cmd=stats`
`path`	Path to the documents folder	`../data/`	`../data/`, `../data2/`	`path=../data2/`

Table 4: Makefile Parameters

Note: If parameters are not specified, the program will default to using the **BST** structure with all **10103** documents from the `data/` folder and will execute the `search` command.

6.4 Plot Graphs

The graphs will be created on the plots folder.

```
#Compiling files to plot graph
(cd plots && qmake && make && ./plot)
```

7 Challenges

During the project's development, several challenges were faced, requiring organization, adaptability, and collaborative work from the team.

One of the main obstacles was finding a balanced way to divide tasks, ensuring that all members could actively participate without overburdening anyone. The modularization of the code and the clear definition of responsibilities required several discussions and adjustments throughout the process.

Another point of difficulty was the interpretation of the project statement. The text provided by the professor contained ambiguities in some requirements, which led to uncertainty during implementation. A collective effort was required to correctly interpret the guidelines and align the team's expectations with the project's objectives.

The second dataset used also posed a significant technical challenge. Due to its large size, processing the data completely demanded more computational resources than the team's personal computers could offer. As a consequence, we experienced frequent VSCode freezes and unexpected terminal crashes.

Creating the illustrations for the report also presented difficulties. We opted to use LaTeX as our scientific writing tool, which provided advantages in formatting and standardizing the document. However, creating graphs and visual representations, such as the trees, required the use of complex libraries like TikZ, which demanded a significant time investment to learn and apply correctly.

Initially, the tree printing function only displayed data in the terminal. As the project progressed, the need to save this information to files arose, which required modifying the code to allow for automatic output redirection.

Another challenge was visualizing the performance statistics. Although the team opted to use C++ for this task, the language lacks robust native support for generating graphs. It was therefore necessary to find specialized libraries and learn how to use them effectively.

Finally, as is common in projects of this nature, we encountered logic errors during implementation, especially during the tree construction and balancing phases. Identifying and correcting these flaws required careful code review, constant testing, and collaboration among the members to ensure the stability and efficiency of the data structures.

8 Conclusion

Insert the conclusion HERE

References

[1] Wikipedia. *Binary Search Tree*. Available at: `https://en.wikipedia.org/wiki/Binary_search_tree`

[2] GeeksforGeeks. *Binary Search Tree (BST) - Data Structure*. Available at: `https://www.geeksforgeeks.org/binary-search-tree-data-structure/`

[3] Wikipedia. *AVL Tree*. Available at: https://en.wikipedia.org/wiki/AVL_tree

[4] GeeksforGeeks. *Introduction to AVL Tree*. Available at: https://www.geeksforgeeks.org/introduction-to-avl-tree/

[5] GeeksforGeeks. *Introduction to Red-Black Tree*. Available at: https://www.geeksforgeeks.org/introduction-to-red-black-tree/

[6] Wikipedia. *Red–black tree (PT translation)*. Available at: https://en-m-wikipedia-org.translate.goog/wiki/Red%E2%80%93black_tree?_x_tr_sl=en&_x_tr_tl=pt&_x_tr_hl=pt&_x_tr_pto=tc