A2 - Estrutura de Dados

Victor Gabriel Iwamoto

June 2025

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

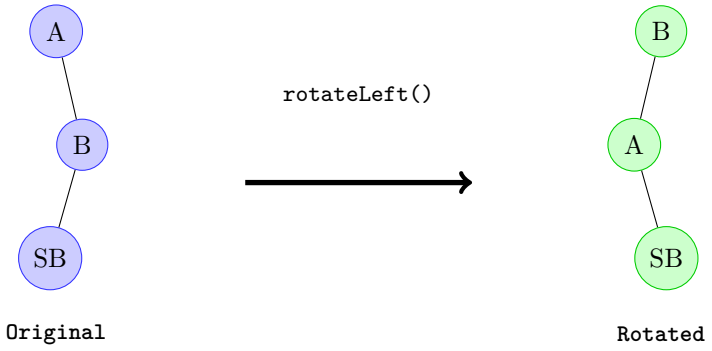
- **Function BalancingFactor:**

  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.

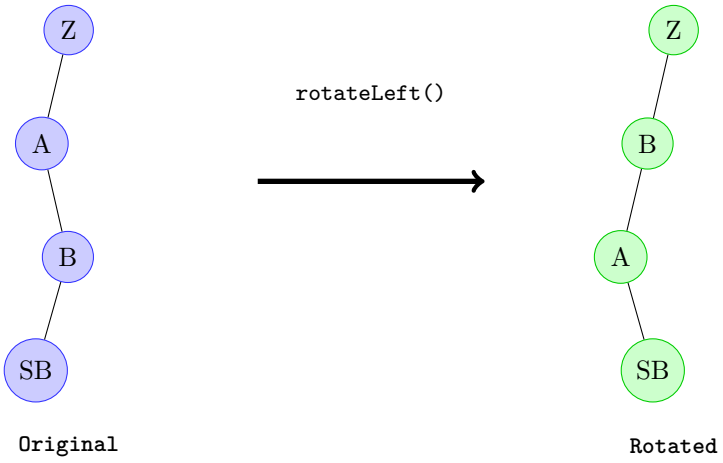
- **Function Rotates:**

  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:

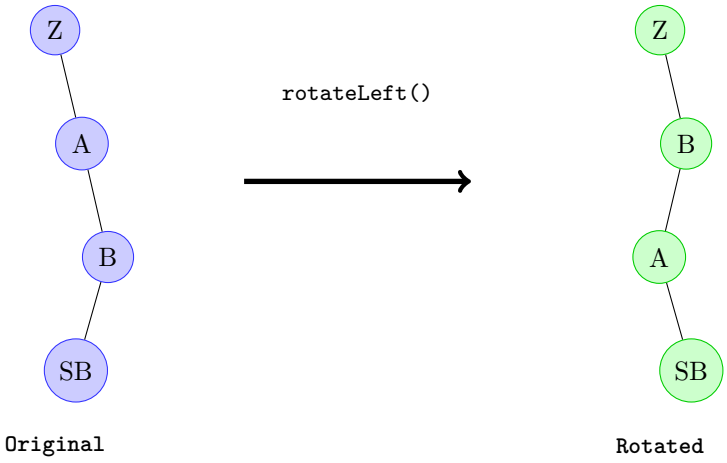
  - ***First Case:*** The node is the root and has no parent.



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



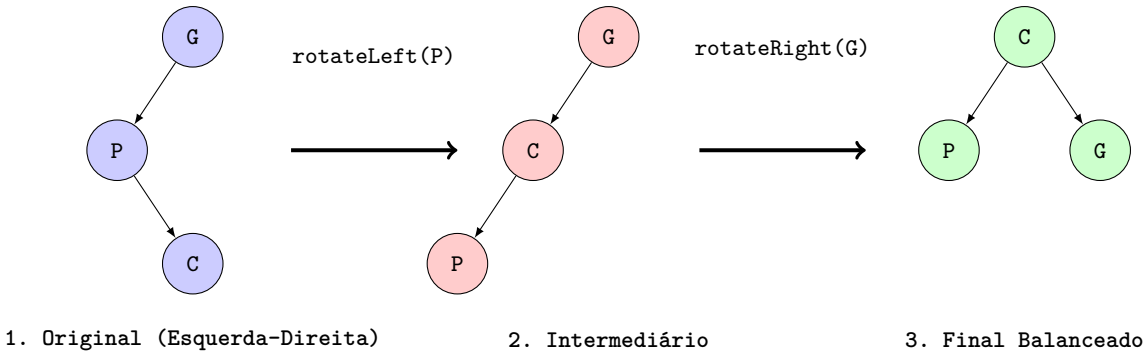
– ***Third Case:*** The node is a right child.



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.

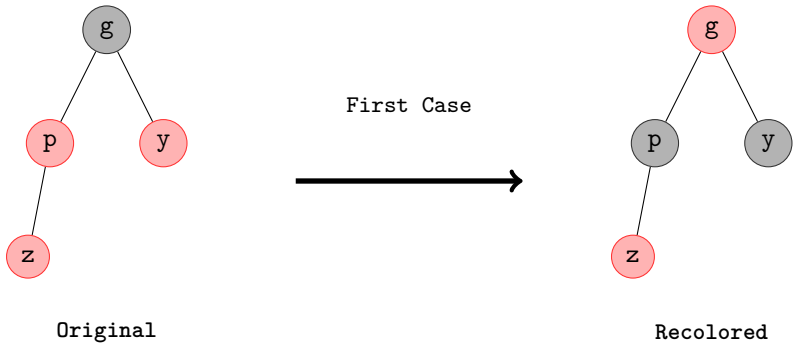
  In addressing these violations, three main cases (and their symmetric counterparts) are distinguished.

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

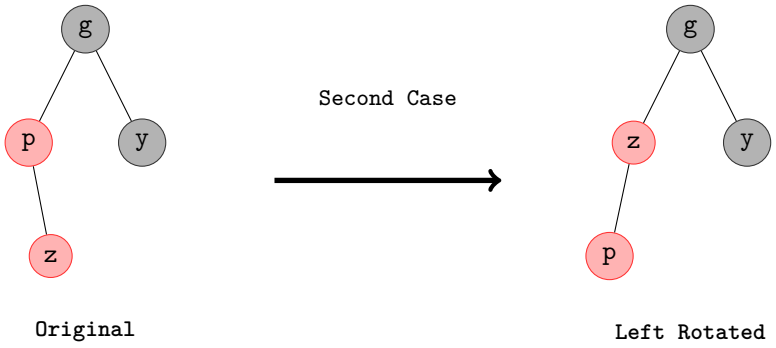
    Since z is inserted as a red node, its parent, p, might also be red. This configuration violates the Red Property of a Red-Black Tree (a red node cannot have a red child). Assuming that the uncle of z, denoted by y, is also red, we perform a recoloring to fix the violation.

    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.

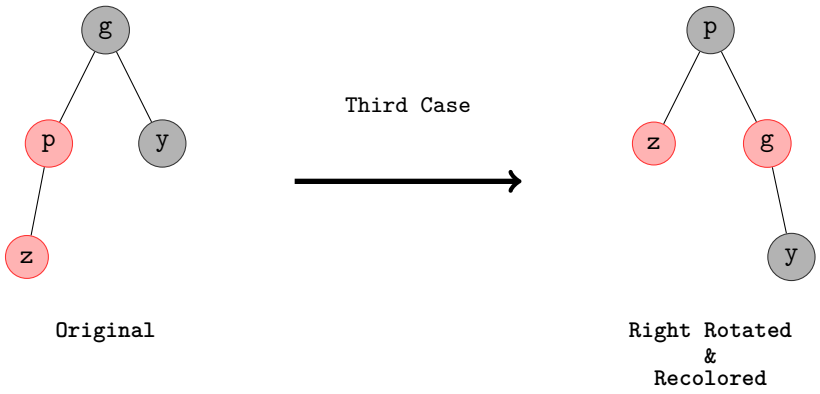
    However, this recoloring might have introduced a new conflict at the level of the grandparent, g, if its parent is also red. Therefore, the verification and correction must be applied recursively up the tree until all properties are restored.



– ***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.