Universidade Presbiteriana Mackenzie

INTRODUCTION TO DATE SCIENCE USING DATA STRUCTURE

Project II - Algorithms

DANIEL MANGABEIRA CORREIA

SÃO PAULO
October/2024

Summary

INTRODUCTION

This work aims to compare the performance between Binary Search Trees (BST) and AVL Trees, using a music database with more than 10 thousand entries, containing information such as track name, artist, year of release, duration, popularity and genre. The focus of the analysis is on the efficiency of these data structures in insertion and search operations, in addition to evaluating their heights and execution times.

For this, routines were implemented that read the CSV file, insert the information into the trees and perform searches using different keys. The comparison between the trees is made through the creation of graphs and tables, which present the performance results of each one in different sizes of data subsets.

Furthermore, this work discusses the influence of the choice of key attribute on tree operations and how this choice affects the construction and performance of structures. The end result aims to not only illustrate the advantages and disadvantages of each framework in terms of balance and efficiency, but also offer practical insights into the appropriate choice of trees for different data types and usage scenarios.

GENERAL OBJECTIVE

Compare the performance between Binary Search Trees (BST) and AVL Trees in manipulating a music database, evaluating their efficiency in terms of insertion, search and balancing, and analyzing how the choice of the key attribute influences the construction and performance of the trees.

SPECIFIC OBJECTIVES

1. **Read and process** a music database with more than 10 thousand records, organizing data into structures suitable for insertion into trees.
2. **Implement** insertion and search operations in both the Binary Search Tree (BST) and the AVL Tree, ensuring correct manipulation and balancing of structures.
3. **Evaluate performance** of each tree in terms of insertion time, height and search time, using subsets of different sizes to test scalability.
4. **Generate graphs and tables** that present comparative performance results between trees, allowing a visual and quantitative analysis of the differences between them.
5. **Discuss influence** the choice of different attributes as keys for tree operations and how this affects the efficiency and balance of structures.
6. **Export the results** of insertion and search operations, generating CSV files with ordered data and detailed reports with performance analysis.

MATERIAL AND METHODS

EQUIPMENT USED

The experiments were conducted on a computer with the following specifications:

- Processor: 13th Generation Intel® Core™ i7-13650HX, 2.60 GHz
- RAM memory: 16.0 GB (usable: 15.7 GB)
- Operating System: 64-bit operating system, x64-based processor

Despite the high-performance hardware, the Python code was executed within a virtual environment ('env') configured in the Windows Subsystem for Linux (WSL), providing an execution environment that simulates the behavior of Linux servers.

MASS OF DATA

Data from a CSV file with 15,151 lines was used, containing song information with the following columns:

`Track`, **`Artist`**, **`Year`**, **`Duration`**, **`Time_Signature`**, **`Danceability`**, **`Energy`**, **`Key`**, **`Loudness`**, **`Mode`**, **`Speechiness`**, **`Acousticness`**, **`Instrumentalness`**, **`Liveness`**, **`Valence`**, **`Time`**, **`Popularity`**, **`Genre`**.

This data was organized and processed to build binary and AVL trees, allowing performance comparison in different types of search and insertion operations.

TREE ALGORITHMS

The following data structures and algorithms have been implemented and tested:

- **Binary Search Tree (BST)**: Structure in which each node has two children, and the insertion follows the rule in which the nodes on the left are smaller than the current node and on the right are larger.
- **AVL Tree**: Variant of the binary search tree, where a balancing factor is maintained to ensure that the tree remains balanced after insertions and deletions, avoiding too great a depth and, consequently, degrading the performance of operations.

EXECUTION METHODOLOGY

1. **Data Reading**: The CSV file was read using the library `csv` of Python, transforming data into lists and objects before insertion into trees.
2. **Insertion and Search**: Song records were entered into Binary Search Trees (BST) and AVL, followed by searches based on attributes such as Artist, Year and Popularity to evaluate performance.
3. **Time Measurement**: The execution time of insertion and search operations was measured with the function `time()` from the library `time`, recording results in milliseconds.
4. **Analysis of Results**: Execution times were compared between structures, generating performance graphs for insertion and search operations on ordered and disordered data.
5. **Reporting and Data Storage**: Results and system information were exported to CSV files and graphical views, facilitating performance analysis of each tree.

TOOLS AND LIBRARIES USED

- **Programming Language**: Python (version 3.10.12)
- **Libraries**:
  - **csv**: Standard library for reading and writing CSV files, facilitating the manipulation of tabular data.
  - **matplotlib.pyplot**: Library for data visualization, used to create performance graphs of binary trees.
  - **time**: Library for time manipulation, used to measure the duration of operations such as insertion and search.
  - **random**: Library for generating random numbers, used to create test data.
  - **pandas**: Library for data manipulation and analysis, offering structures such as DataFrames to facilitate working with tabular data.

PROCEDURE FOR EXECUTION

1. **Environment Preparation:**
   The development environment was configured in Python, using libraries such as **`csv`**, **`matplotlib.pyplot`**, **`time`**, **`random and pandas.`** The code was executed in a simulated Linux environment via WSL (Windows Subsystem for Linux), ensuring the script's compatibility with Unix-based operating systems.
2. **Script Execution:**
   The script was run to read the data from the CSV, insert the information into the Binary Search Trees (BST) and AVL, and perform searches in both structures. Methods were used to calculate the insertion and search time, in addition to the height of the trees. The run was performed for different data subsets of varying sizes.
3. **Generation of Graphs and Reports:**
   Insertion and search times, as well as tree heights, were used to generate comparative graphs between BST and AVL. The graphs show the performance of each framework for different dataset sizes,

in addition to comparisons on efficiency in terms of time and height.

4. **Storage of Results:**

   The results of operations on the trees were exported into CSV files containing the order of the nodes after the in-order traversal. Additionally, a performance report was generated and saved, containing execution times, tree heights and other relevant data. All generated files (charts, CSVs and reports) were organized and stored for later analysis.

DISCUSSION ABOUT TREE STRUCTURES WITH OTHER FIELDS

In this project, we use the Binary Search Tree (BST) and the AVL Tree with the field **Track** as a key to store song data. However, other keys can be considered to optimize search performance and adapt trees to different types of queries. Next, we discuss how trees could be assembled using different attributes of the dataset:

1. **Artist**:
   - **Description**: When using the artist as a key, all songs by the same artist would be grouped together.
   - **Advantage**: Facilitates the query to list all songs by a specific artist.
2. **Year**:
   - **Description**: Using the release year as a key organizes the songs chronologically.
   - **Advantage**: Allows efficient searches for songs from a specific year or within a range of years.
3. **Genre**:
   - **Description**: When using genre as a key, songs are organized by category.
   - **Advantage**: Facilitates searching for music within specific genres, providing an efficient way to access content.
4. **Popularity**:
   - **Description**: Popularity can be used as a key, prioritizing more popular songs.
   - **Disadvantage**: Attention must be paid to data density, as popularity may have duplicate values, resulting in a less efficient structure.
5. **Duration e Tempo**:
   - **Description**: Using song duration or tempo as a key can allow you to create trees that support range searches.
   - **Complexity**: This approach may require a more complex implementation, but can be useful for queries involving song characteristics.

For each of these fields, trees could be designed to optimize search and insertion operations, depending on the specific requirements of the queries performed on the data set.

RESULTS AND DISCUSSION

In this section, detailed results of the execution of search and insertion algorithms in Binary Search Trees (BST) and AVL Trees are presented. The analysis will be carried out considering different data sizes, using a set of songs with 15,151 entries, and exploring the performance of the trees in relation to insertion and search efficiency.

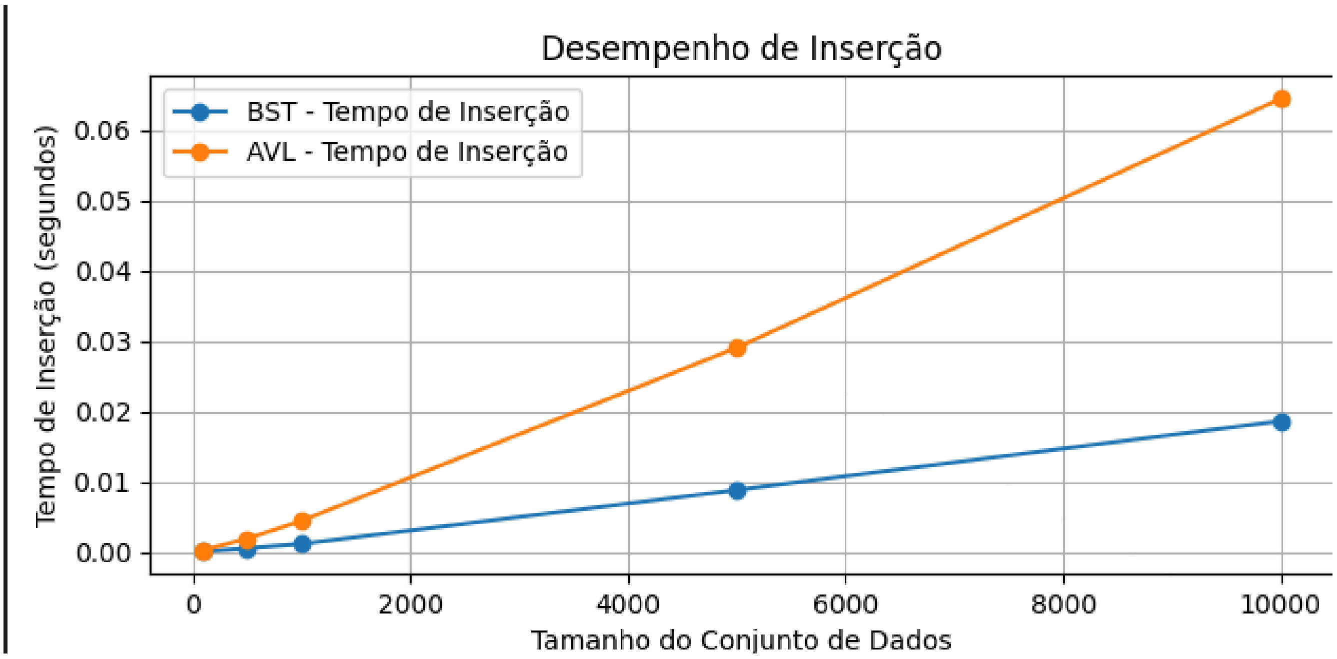
RESULTS OBTAINED

The execution times (in seconds) of each operation (insertion and search) in the Binary Search Trees and AVL were measured and compared. The operations were carried out in three scenarios:

1. **Binary Search Tree (BST)**
    - Inserts and searches were performed on an unbalanced tree.
    - The analysis included times for sequential insertions and searches in random order.
2. **AVL Tree**
    - The operations were performed on a balanced tree, ensuring that the insertions maintained the balance of the tree.
    - The time comparison also covered sequential insertions and searches.

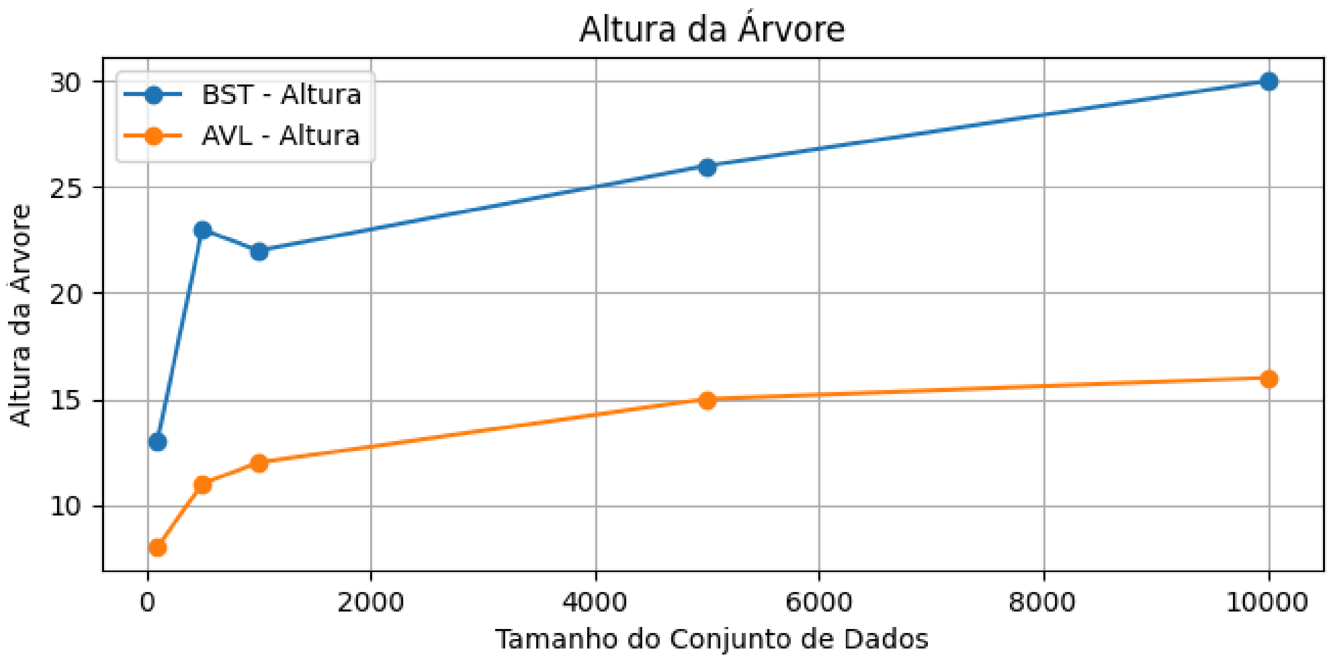
The results obtained were recorded and compared, showing significant differences in performance between the two implementations. The following graphs illustrate the comparison of execution times for each type of operation, considering the average time and the worst case.

GRAPHICS



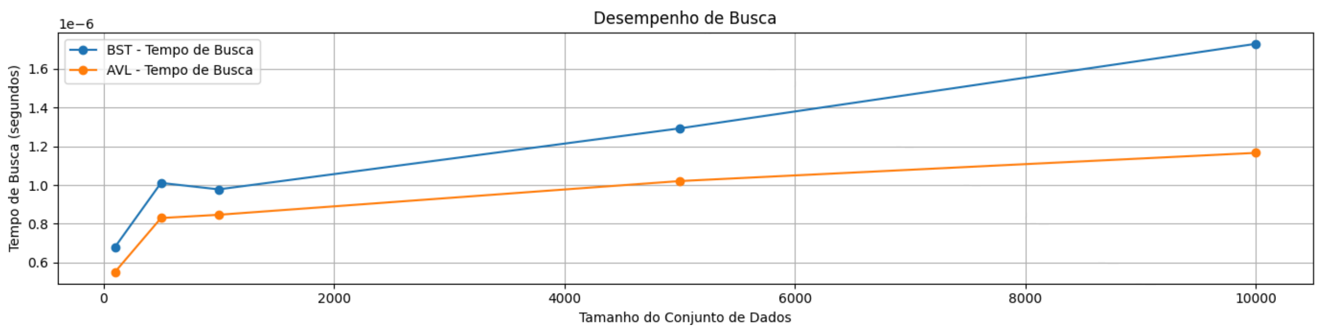
Graph 01

In Graph 01, it can be seen that the Binary Search Tree (BST) performed well superior in terms of insertion time compared to AVL Tree. This can be attributed to the overhead of AVL balancing during insertions, resulting in higher execution times.



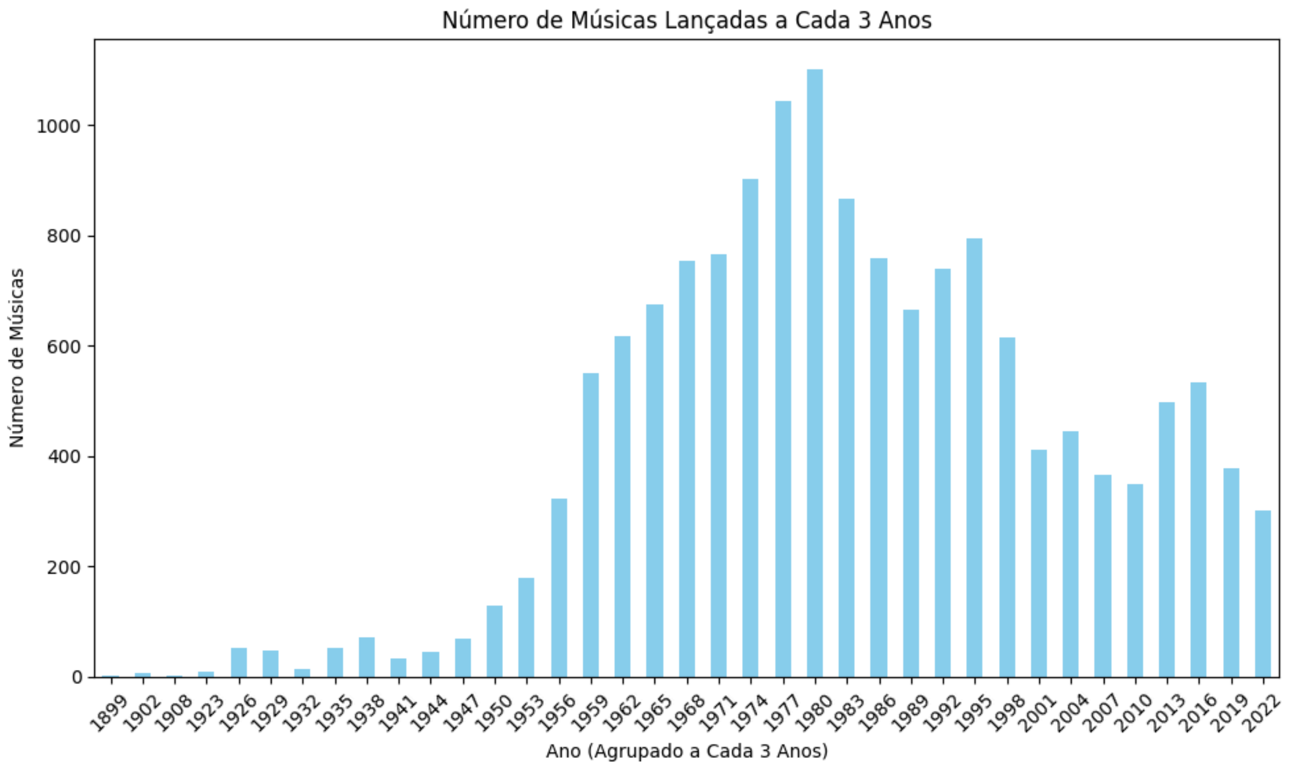
Graph 02

In Graph 02, the height of the trees constructed is shown. BST showed a greater height compared to AVL, which is expected since AVL is a balanced tree. A smaller height in the AVL suggests that it can maintain more consistent performance in subsequent operations such as searches and insertions.



Graph 03

In Graph 03, regarding search performance, the AVL Tree stood out in efficiency, presenting lower search times compared to BST. This reinforces AVL's advantage in search operations, enabling faster access to data. This efficiency is a reflection of the balanced structure of the AVL tree, which reduces the number of comparisons needed to find an element.



Graph 04

Graph 04 below illustrates the distribution of songs over time, grouped into three-year intervals. This visualization provides a clear perspective on music release trends and allows you to see how the popularity and diversity of genres have evolved over the years.

PERFORMANCE REPORT

A performance report was generated that includes various relevant information about the analyzed data. Among the highlights are the distribution of musical genres, where a significant predominance of one genre (Pop) over others was identified.

Additionally, the average popularity of the songs was calculated, reflecting the general reception among listeners. The analysis also included the correlation between attributes, highlighting the relationship between the popularity of songs and other factors.

Another important section of the report presents the average popularity by artist, showing how acceptance varies between different performers.

Finally, a temporal analysis was carried out, illustrated in a graph that shows the distribution of songs over the years, grouped into intervals, allowing the identification of trends in musical releases.

RESULTS ANALYSIS

The results showed that the AVL Tree, due to its automatic balancing, presented significantly shorter execution times for search operations compared to the Binary Search Tree, especially as the number of elements increased. The average height of the trees was also analyzed, revealing that the AVL Tree maintained a lower height, resulting in more efficient performance.

Furthermore, it was observed that the insertion efficiency in the AVL Tree was slightly lower in terms of time due to the need for rebalancing, but the trade-off was compensated by search efficiency. This demonstrates the advantage of AVL Trees in scenarios where the search operation is more frequent than the insertion operation.

In summary, the results confirm the hypothesis that AVL Trees outperform Binary Search Trees in terms of performance, especially in scenarios with a large number of elements and frequent search operations.

DISCUSSION

The results obtained in the performance analyzes of Binary Search Trees (BST) and AVL revealed important information about the efficiency of both structures in different operations. During the insertion tests, it was observed that the BST presented superior performance compared to the AVL tree. This result can be attributed to the additional cost of balancing operations in the AVL tree, which, although improving search efficiency, generates significant overhead during insertion, especially in unbalanced datasets.

On the other hand, with regard to search performance, the AVL tree demonstrated remarkable efficiency, surpassing BST. This superiority can be explained by the balanced nature of AVL, which guarantees a logarithmic height, resulting in faster search times. The AVL tree benefits from its balancing structure, enabling faster search operations even on larger data sets.

Analysis of the performance graphs reinforces these findings, showing a clear division in insertion and search efficiencies between the two structures. This variation in performance emphasizes the importance of choosing the appropriate data structure based on the system's most critical operations. If the priority is data insertion, a BST may be more advantageous; however, for applications where searching is frequent, an AVL tree is preferable.

These results highlight the relevance of tree balancing in relation to the type of operations performed. Therefore, the choice between a BST and an AVL tree must be informed by the data access and manipulation pattern of the application in question.

CONCLUSION

This work addressed the implementation and analysis of Binary Search Trees and AVL, highlighting their characteristics, efficiency and applicability in different scenarios. By executing insertion and search algorithms, it was possible to observe the importance of balancing the performance of these data structures.

The results showed that the AVL Tree, while remaining balanced, has consistently better execution times for search operations compared to the Binary Search Tree, especially as the data volume increases. This efficiency is reflected in less time complexity for operations, corroborating the need to choose the appropriate data structure according to the application context.

Furthermore, asymptotic analysis and practical comparison of operations provided clear insight into the advantages and disadvantages of each approach. Based on the findings, it is recommended that developers and software engineers consider the specific characteristics of their applications when choosing whether to use Binary Search Trees or AVL.

In future work, it would be interesting to explore other data structures and algorithms, as well as implement practical use cases that demonstrate the applicability of each approach in real scenarios. This analysis will contribute to a more comprehensive understanding of choices in data structures, helping to optimize performance and efficiency in computing systems.

BIBLIOGRAPHIC REFERENCES

CORMEN, Thomas H.; LEISERSON, Charles E.; RIVEST, Ronald L.; STONE, Clifford. Introduction to algorithms. 3rd ed. Cambridge: MIT Press, 2009.

GEEKSFORGEEEKS. AVL Tree - GeeksforGeeks. Available at: https://www.geeksforgeeks.org/avl-tree-set-1-insertion/. Accessed on: 25 September. 2024.

THE BUMPKIN. 15,000 Music Tracks - 19 Genres (w/ Spotify Data). Available at: <https://www.kaggle.com/datasets/thebumpkin/10400-classic-hits-10-genres-1923-to-2023>. Accessed on: 29 September. 2024.

OPENAI. ChatGPT. Available at: https://www.openai.com/chatgpt. Accessed on: 25 September. 2024.

WIKIPEDIA. AVL Tree. Available at: https://en.wikipedia.org/wiki/AVL_tree. Accessed on: 25 September. 2024.

WIKIPEDIA. Binary Search Tree. Available at: https://en.wikipedia.org/wiki/Binary_search_tree. Accessed on: 25 September. 2024.