FUNDAÇÃO GETULIO VARGAS

GRADUAÇÃO EM CIÊNCIA DE DADOS E INTELIGÊNCIA ARTIFICIAL
GRADUAÇÃO EM MATEMÁTICA APLICADA

Alex Júnio Maia de Oliveira, João Pedro Jerônimo de Oliveira, Lucas Batista Pereira, Matheus Vilarino de Souza Pinto, Thalis Ambrosim Falqueto

PROFESSOR: Matheus Werner

ESTRUTURA DE DADOS:
RELATÓRIO PROJETO FINAL

Rio de Janeiro - Rio de Janeiro
2025

SUMÁRIO

1. INTRODUÇÃO

No contexto tangente a estruturas de dados, buscamos estudar como se comportam, principalmente, as operações de inserção, remoção e busca em determinadas estruturas de dados.

Nesse cenário, índices invertidos são muito utilizados para sistemas de busca em pequena e grande escala, em que associamos elementos a uma lista de documentos dos quais eles aparecem, ou seja, cada elemento é um nó na estrutura de dados utilizada.

Para os fins do projeto, implementamos o índice invertido utilizando três estruturas de dados: Árvore Binária de Busca (BST), Árvore Binária de Busca Balanceada (AVL) e Árvore Rubro Negra (RBT).

Foi oferecido um conjunto de dados contendo mais de 20.000 arquivos para fazermos as implementações, testes e estatísticas. Além disso, seria implementada uma interface por linha de comando (CLI), pois assim seria possível executar comandos no terminal de forma clara e bem definida.

2. METODOLOGIA

Para realizar o objetivo do projeto, que é a comparação performática dos índices invertidos gerados entre as três estruturas de dados (BST, AVL e RBT), utilizamos a linguagem de programação *C++* e uma organização modular e organizada no GitHub.

O código fonte está contido na pasta src e está dividido em módulos de implementação das três árvores, criação dos objetos para auxiliar na compilação dos arquivos, testes unitários para validar as estruturas das árvores, estruturas comuns para as árvores, busca e inserção de resultados. Além disso, há *mains* para testar o CLI de cada estrutura e um arquivo makefile para facilitar na compilação dos módulos.

Somado ao descrito acima, fizemos uma limpeza e validação da base de dados, tirando símbolos, acentos e letras maiúsculas e uma validação do CLI. Além disso, para fazer o uso correto dos CLIs, deve-se adicionar uma pasta chamada data na raiz do repositório (ela é ocultada pelo GitHub).

3. COMO FOI DIVIDIDO

Para facilitar o andamento e não permitir uma sobrecarga de funções aos membros, as tarefas foram divididas da seguinte forma: Thalis ficou responsável pela implementação da BST adaptada para o problema e pela formulação do relatório, Alex ficou responsável pela implementação da AVL adaptada para o problema e pela formulação do relatório, Matheus ficou responsável pela implementação da RBT adaptada para o problema e pela formulação do relatório, e, por fim, Lucas e João Pedro ficaram dando o suporte necessário em todo o restante do projeto: modularização, estatísticas, CLI e funções extras importantes (como a função que imprime a árvore, imprime o index, etc.).

Além disso, dividimos o projeto em branchs para maior organização e entendimento das funções de cada integrante do grupo.

4. ESTATÍSTICAS

Ao iniciar o programa referente a alguma das 3 árvores no modo "*stats*", são gerados arquivos CSV com as estatísticas de tempo de inserção, quantidade de comparações, altura das árvore no momento da inserção, quantidade de nós inseridos até o momento, quantidade de rotações (para AVL e RBT). Com os dados gerados, foi criado um programa em *Python*, com objetivo de gerar gráficos para analisar e elaborar hipóteses.

4.1 Consistência e previsibilidade das árvores

Realizando testes com as implementações das três árvores, observamos que árvores balanceadas (AVL e RBT) tendem a ter um desempenho de busca mais consistente e previsível (menor variância) em termos de tempo e número de comparações do que uma BST, especialmente à medida que o número de elementos aumenta.

Para realizar essa comparação e validar tal hipótese, fizemos dois gráficos: o boxplot das comparações na busca e o boxplot dos tempos de execução na busca.

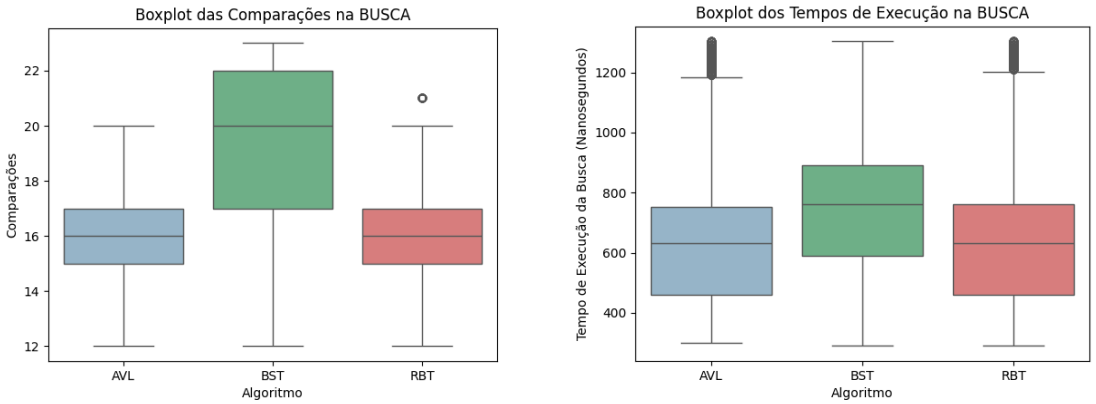

Figura 1 - Boxplot quantidade de comparações (esquerda) e boxplot tempo de execução (direita).

Analisando algoritmo por algoritmo, conseguimos ver que, no gráfico de comparações, a AVL e a RBT apresentam números de comparações mais concisos do que a BST, estando bem localizados na região de mediana 16, enquanto as comparações da BST são esparsas e maiores, justamente pela falta de balanceamento que acontece nos outros dois algoritmos. Analogamente, temos um resultado parecido no segundo gráfico quando analisamos a performance dos algoritmos. Perceba que as medianas da AVL e da RBT são simétricas em relação ao intervalo interquartil.

Por fim, comparando com os resultados obtidos no artigo “Performance analysis of BSTs in system software”(página 6, tabela 2), especificamente na parte parents (os nomes “parents”, “plain”, “threads”, etc são todas formas diferentes de representar a mesma árvore, usando nós, threads, entre outros. considere que a representação usada por nós no projeto foi a parents) apresentam resultados que validam a nossa hipótese. Perceba:

Conjunto de dados	Análise	BST	AVL	RBT
Mozilla	*tempo de execução (s)*	15,67	3,65	3.78
	comparações	635.957	54.842	68.942
VMware	*tempo de execução (s)*	447,40*	6,31	7,32
	comparações	14.865.389	122.087	175.861
Squid	*tempo de execução (s)*	12,52	3,69	3,80
	comparações	487.818	62.467	72.867

Random	*tempo de execução (s)*	1,63	1,67	1,64
	comparações	28.123	25.958	25.983

Tabela 1 - tempo em segundos e número de comparações de 1000 execuções de cada conjunto de dados.

Analisando tempo de execução e comparações das 4 bases de dados usadas por Ben, fica nítido que o tempo de execução e contagem de comparações são extremamente altos para o algoritmo BST quando comparado com AVL e RBT, o que confirma a hipótese.

Vale lembrar que retiramos alguns outliers na hora da plotagem. Além disso, o * na tabela significa que a BST se transformou em uma “lista encadeada”.

4.2 Rotações entre árvores AVL e RBT

Embora a AVL e a RBT sejam ambas balanceadas, a AVL pode apresentar um custo de balanceamento ligeiramente maior (mais rotações) por inserção do que a RBT. Para isso, vamos analisar a média de rotações em função da quantidade de palavras únicas dado os gráficos da *“Figura 2”*:

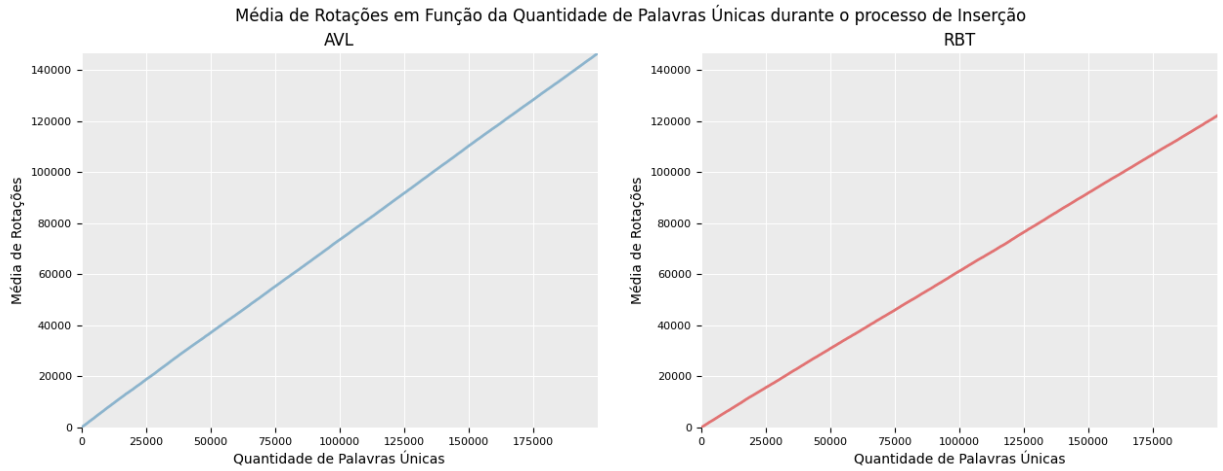

Figura 2 - Rotações em relação a quantidade de palavras únicas.

Essa hipótese parece ser validada ao olharmos a inclinação de ambas as retas, onde, nitidamente, a reta do algoritmo AVL possui um ângulo maior quando comparado com o algoritmo da RBT. Por exemplo, no ponto de 150000 palavras únicas, observamos um valor perto de 110000 rotações para a AVL, enquanto, no mesmo ponto, temos um valor próximo de 95000 rotações para a RBT.

Isso pode ser observado nas tabelas 2, 3, 4 e 5 da página 18 do artigo "***Comparative Performance Evaluation of the AVL and Red-Black Trees***" criado pelos pesquisadores Milo Tomašević e Svetlana Štrbac-Savic:

	TABELA 2		TABELA 3		TABELA 4		TABELA 5	
Arquivo	**AVL**	**RBT**	**AVL**	**RBT**	**AVL**	**RBT**	**AVL**	**RBT**
R10	0,052	0,040	0,023	0,019	0,021	0,015	0,058	0,047
R100	0,056	0,046	0,018	0,010	0,023	0,018	0,042	0,035
R1000	0,058	0,048	0,009	0,006	0,023	0,020	0,018	0,014
R10000	0,066	0,059	0,014	0,013	0,029	0,028	0,031	0,028
R100000	0,083	0,073	0,023	0,021	0,033	0,030	0,042	0,037
RMilion	0,165	0,136	0,060	0,050	0,066	0,055	0,130	0,109

Tabela 2 - Média de rotações por operação em diferentes distribuições de inserção, pesquisa e deleção.

Analisando a tabela, vemos que nenhuma média de rotação do algoritmo AVL calculada fica menor ou sequer igual quando comparado com o algoritmo RBT, o que sugere a validação da hipótese. Ainda, segundo os autores do artigo, "[...] *it is clear that, as expected, it takes more rotations to make the initial AVL tree then the RB tree.*", ou seja, a AVL realmente usa mais rotações do que a RBT.

4.3 Relação entre comparações e altura

Durante a experimentação das propriedades das árvores, percebemos que a BST tende a ter um desempenho de inserção altamente variável e pode degenerar em uma lista encadeada, apresentando uma altura maior quando comparada com outras árvores, em cenários de dados ordenados ou quase ordenados, enquanto AVL e RBT manterão um desempenho próximo de $O(log(n))$.

Para validar a afirmativa, vamos analisar a média de comparações em função da altura da árvore dado os gráficos da *"Figura 3"*:

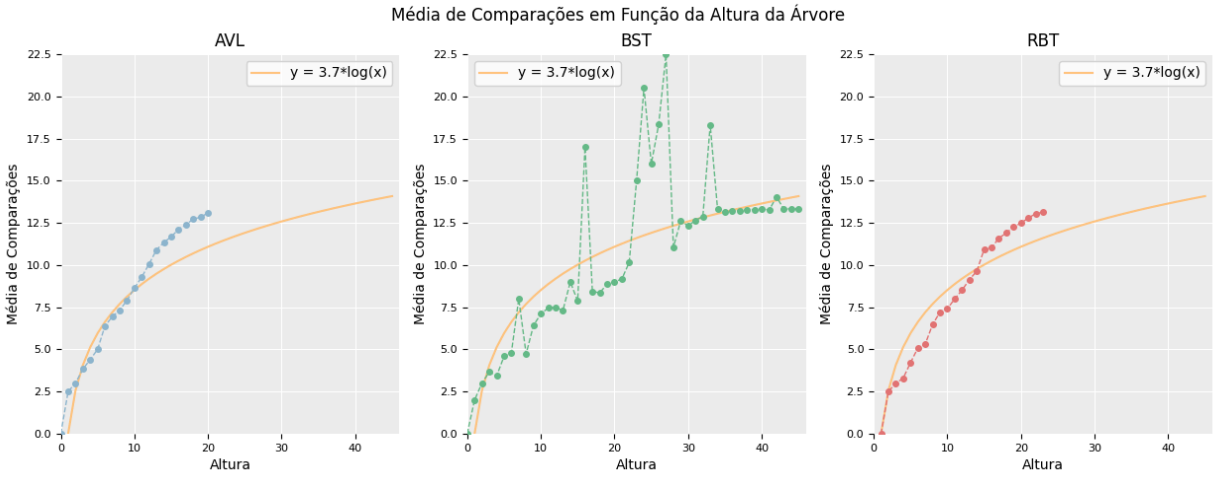

Figura 3 - Média de comparações em função da altura.

Essa hipótese seria melhor observada num contexto de palavras ordenadas, onde, obrigatoriamente o algoritmo BST deveria tender a uma “lista encadeada”, já que, como não há rotação de acordo com que a árvore vai ficando mais e mais desequilibrada e por estar em ordem, a árvore nunca teria nós a esquerda e viraria uma “lista”.

Porém, o que é possível observar é que o algoritmo BST apresenta grande variação na média em algumas alturas, em teoria porque, como não há balanceamento no algoritmo, uma mesma altura é mantida por muito tempo, aumentando então a quantidade de comparações na mesma altura, e, portanto, aumentando a média respectiva. Além disso, como o algoritmo BST é o que mais tem pontos, olhando o gráfico, parece plausível que, se não houvessem essas variações, o gráfico pareceria com uma função $y = c\,log(x)$, o que é esperado dado que as palavras não estão ordenadas.

Por fim, ao analisarmos os algoritmos AVL e RBT, apesar de termos uma quantidade menor de pontos por causa do rebalanceamento (quase metade quando comparada com o algoritmo da BST), e, plotando a função $y = c\,log(x)$, vemos que ambas se parecem bastante, constatando a hipótese.

4.4 Comparações para busca e inserções nas árvores

Sabemos que as três árvores lidam de forma distintas com a inserção de um nó. Assim, estudando esse comportamento, observamos que a média de comparações para inserção tende a ser menor na AVL e na RBT do que na BST,

refletindo a altura mais controlada das árvores balanceadas. O mesmo vale para a média de comparações durante a busca por palavras dentro da árvore.

Para analisar a hipótese, vamos representar o total de comparações de inserção e de busca feitas em cada árvore dados nos gráficos das *“Figura 4” e “Figura 5”*:

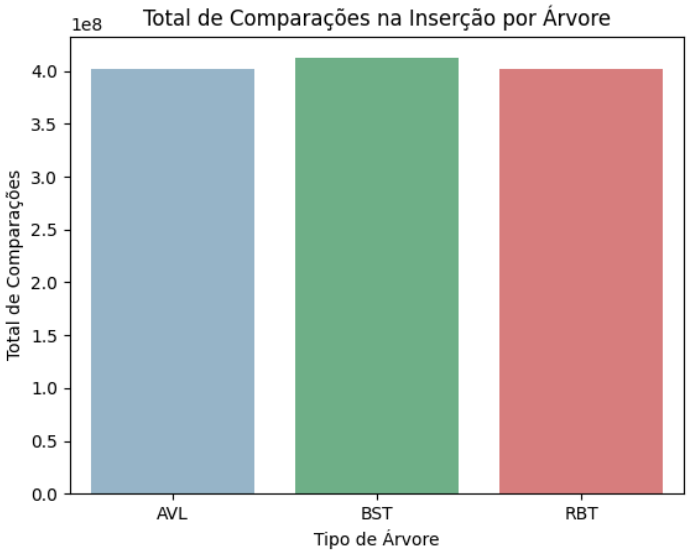

Figura 4 - Total de comparações por inserção.

Representando o total de comparações na inserção por árvore, vemos que existe uma sutil diferença na quantidade dos algoritmos, onde BST é levemente maior que os dois outros algoritmos. Embora cada árvore não tenha uma diferença muito acentuada, isso acaba por validar parte da hipótese.

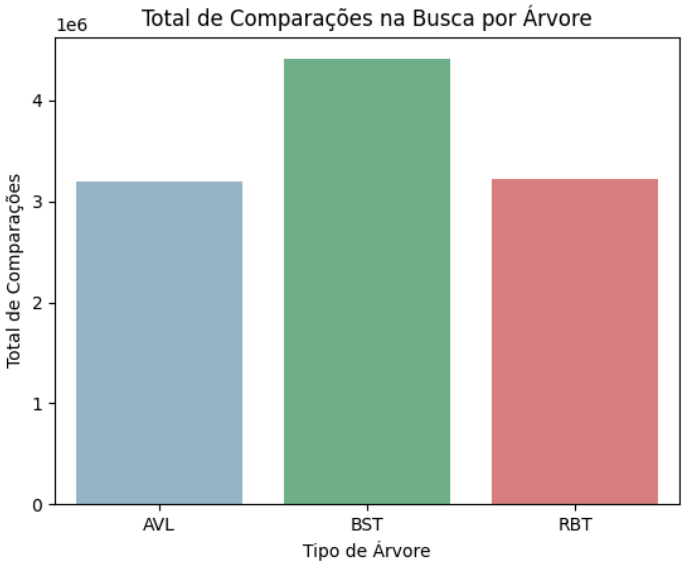

Figura 5 - Total de comparações por busca.

Esse gráfico apresenta o total de comparações por árvore durante a busca por palavras-chave e, como é possível visualizar, a árvore BST tem uma diferença significativa para as outras duas árvores, contendo mais comparações para encontrar a mesma palavra entre seus nós, enquanto a AVL e a RBT mantém uma média equivalente devido ao seu balanceamento automático, ou seja, a segunda parte da hipótese é válida.

4.5 Avaliação do tempo de Inserção

No contexto de estruturas de dados, um fator crucial é o tempo de execução de determinado algoritmo usando uma certa estrutura de dados. Observando tal comportamento nas três árvores, constatamos que o tempo médio de inserção em nanosegundos tende a seguir um padrão diferente das comparações, com AVL e RBT sendo menos eficientes que BST.

Para analisar a hipótese, traçamos a média do tempo de inserção para cada tipo de árvore analisada dado o gráfico da *"Figura 6"*.

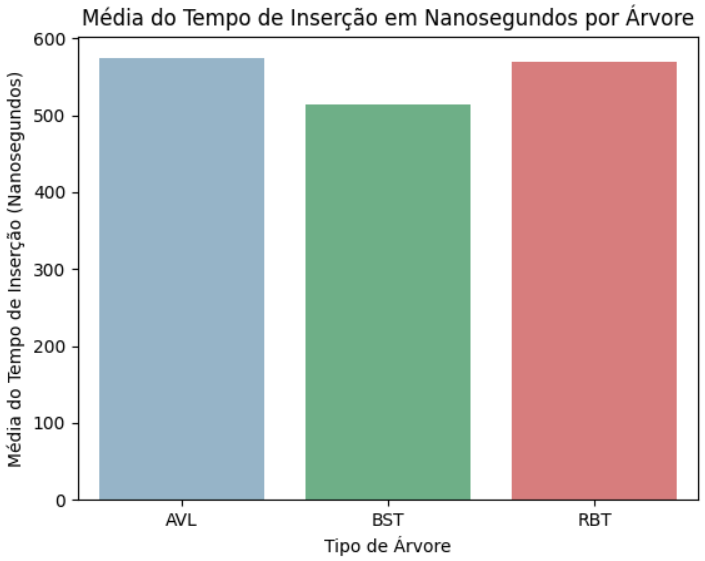

Figura 6 - Tempo médio de inserção.

Segundo Niklaus Wirth, na seção 4.4.7 do seu livro "***Algorithms + Data Structures = Programs***", as árvores balanceadas são mais preferíveis quando, em nosso problema, a frequência de inserção é menor do que a de busca. Isso se deve, segundo o autor, ao fato de que teremos muitos rebalanceamentos durante a

construção destas árvores. Sendo assim, em relação ao tempo de inserção, é esperado que não tenhamos um melhor desempenho para as árvores AVL e RBT quando comparadas com a BST. Como visto no gráfico acima, os tempos de inserção tendem a ser maiores para as árvores RBT e AVL, estando de acordo com a teoria.

4.6 Problema do Heap

Durante a análise dos gráficos que comparam as performances das estruturas da AVL, da RBT e da BST em relação às suas alturas, observamos a presença recorrente de *outliers* em certas implementações, como podemos observar na representação da média de performance em função das alturas das árvores dado a *"Figura 7":*

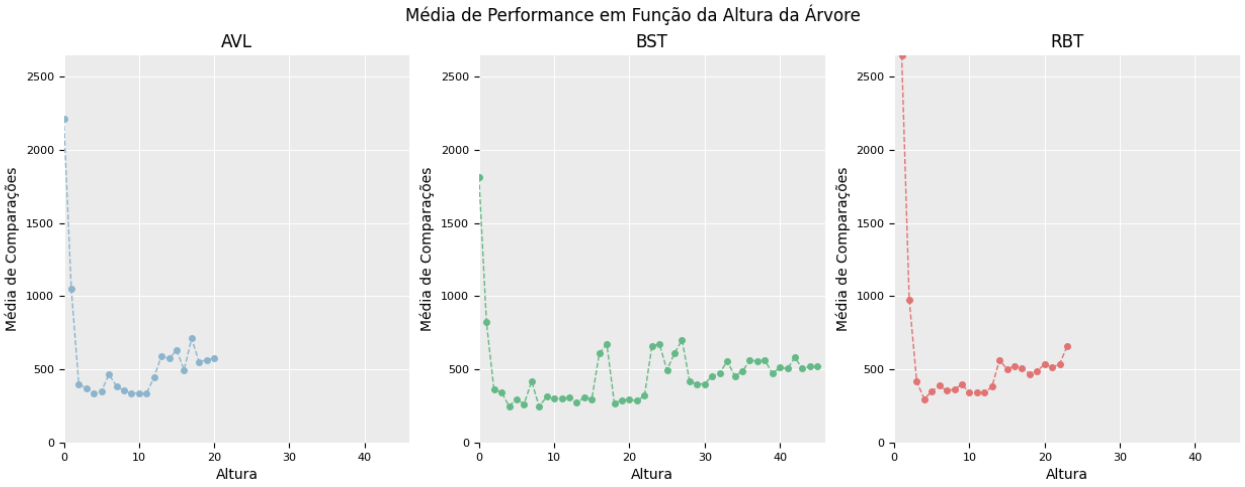

Figura 7 - Performance média em função da altura da árvore.

Inicialmente, suspeitou-se que fatores externos, como o *cache* do processador, a carga computacional momentânea do sistema ou o tempo do computador, pudessem influenciar essas anomalias.

No entanto, após testes controlados e revisão da alocação de memória, identificamos que o comportamento inconsistente está relacionado ao gerenciamento dinâmico de memória (*Heap*), especificamente à alocação de nós durante as operações nas árvores (principalmente na AVL e na RBT). Como o *Heap* armazena dados com tempo de vida indeterminado em tempo de compilação, variações no seu estado podem atrasar operações e acessar indevidamente

espaços da memória. Tal fato é confirmado pois os *outliers* foram mais frequentes em estruturas com mais inserções dinâmicas (AVL e RBT).

Para testar a veracidade da hipótese, fizemos o seguinte experimento: zeramos o cache do computador reinicializando ele. Após isso, rodamos as estatísticas de busca e inserção de cada árvore, obtendo assim, vários *outliers* (principalmente na primeira inserção da árvore). Prosseguindo, fizemos um código para o computador realizar várias operações para encher o cache do computador e fazer com que ele acesse alguns pontos de memória indevida. Por fim, rodamos as estatísticas de cada árvore novamente e constatamos que havia uma quantidade menor de *outliers* e os valores extremos ficaram mais normalizados (apesar de ainda serem extremos).

Pesquisando na literatura sobre a temática, achamos os seguintes livros: "***The Art of Computer Programming, Volume 1: Fundamental Algorithms, Seção 2.5 - D. E Knuth***" e "***Introduction to Algorithms, Capítulo 13 - T. H. Cormen***" que nos dizem, respectivamente, que algoritmos de alocação de memória podem impactar nas operações dinâmica (desempenho irregular na AVL e RBT) e que estruturas que utilizam muitas rotações podem sofrer com custos de localidade de memória, comparando estruturas balanceadas e não balanceadas.

5. DIFICULDADES

Em geral, o algoritmo de Binary Search Tree (BST) para inserção e busca não é muito complexo, as principais dificuldades foram em separar todos os casos possíveis, como a verificação do caso em que a árvore é vazia e o caso onde a palavra é nula serem resolvidos do mesmo jeito, as verificações de quando a árvore não é nula mas não tem raiz, quando é nula mas tem raiz, e, por fim, depois de verificar tudo da árvore, ao ligarmos a árvore com o novo nó, devemos fazer outra verificação para descobrirmos se o filho será adicionado a direita ou à esquerda do pai, após descobrir o pai na verificação anterior. Após a inserção feita, a função de pesquisa foi mais fácil pois já tínhamos uma boa noção das verificações necessárias.

Para a parte da implementação da Árvore de Busca Binária Balanceada (AVL), encontramos dificuldade na construção das funções de rotação (rotação simples e dupla) e na verificação do fator de balanceamento para cada nó quando

inserimos um valor na árvore, pois devemos fazer muitas rotações para balancear toda a árvore. Além disso, outro ponto de dificuldade foi fazer os testes unitários para as rotações, pois criamos uma outra função que diz se duas árvores são iguais e tivemos que dissecar os casos base de rotação e testá-los.

Outra dificuldade foi a organização da equipe e a navegação na nossa própria modularização do código, prejudicando a navegação, a construção do *makefile* e a compilação dos arquivos e dos testes, envolvendo também os problemas em acessar os diretórios corretos para armazenar os dados e entender em qual parte estamos, considerando que o código pode ser rodado pela raiz do repositório tanto quanto pela pasta build, buscando além de tentar deixar o arquivo em ordem para esses casos, permitir que haja pleno funcionamento independente do sistema operacional ser Windows ou Linux.

No quesito de estatísticas, houve uma grande dificuldade na criação dos arquivos CSV contendo as informações que seriam tratadas para geração dos gráficos. O principal problema foi na otimização, tendo em vista que são muitos arquivos com muitas palavras para serem analisadas. Por exemplo, executando o código de geração com o segundo dataset, são realizadas 30 milhões de operações de inserções, cada uma que seria registrada no arquivo CSV. Anteriormente, a cada inserção, as informações já eram inseridas diretamente no arquivo, o que gerava um atraso tendo em vista que, a cada inserção, o canal de inserção do arquivo era utilizado, gerando um custo operacional grande. Posteriormente, utilizamos uma concatenação de strings simples e fazendo uma única inserção enorme no fim de todas as inserções. Outro problema foi monitorar a altura das árvores conforme as palavras eram inseridas. As árvores BST e AVL não foram um problema, tendo em vista que o próprio código de inserção fazia esse gerenciamento da altura, ao contrário da RBT, a qual não conseguimos fazer um monitoramento fácil. Para superar essa dificuldade, criamos uma função recursiva que pega a altura de uma árvore e, especificamente nas estatísticas da RBT, em vez de registrar a altura da árvore a cada inserção, a altura é medida a cada novo documento que é analisado, diminuindo significativamente o tempo de execução do código. Porém, vale ressaltar que, posteriormente, uma melhoria de código foi imposta para a RBT, permitindo um monitoramento da altura da árvore a cada inserção.

Outra grande dificuldade foi na geração dos gráficos. Para contextualizar, utilizamos a linguagem *Python* e as bibliotecas *Pandas* e *Seaborn* para gerar esses

gráficos. O problema surgiu em vista do tamanho dos arquivos CSV gerados nas estatísticas (Por volta de 30 milhões de linhas), então ler eles e fazer as contas normalmente não era uma opção, inclusive, isso foi feito e o código nem conseguia rodar. Isso foi superado utilizando um esquema do próprio Pandas que é a leitura de chunks. Porém, outro problema surgiu, se o código seguisse um paradigma funcional, mesmo com o esquema de chunks, a leitura dos arquivos era demorada, e fazer as operações em cada *dataset* era custoso. Esse problema foi superado ao utilizar de um esquema multi processos, onde usufruímos da biblioteca *Multiprocessing* do python, de forma que cada gŕafico era gerado por um processo separado que rodava simultaneamente e independente aos outros processos, dividindo o código e diminuindo significativamente seu tempo de execução.

Por fim, também houve dificuldade na limpeza do código, já que foram várias ramificações em paralelo, algumas melhorias ficavam perdidas e tinham que ser adicionadas novamente, módulos feitos para versões antigas, etc, apesar de ser confuso e complexo a organização, esses erros ajudaram a moldar o grupo para que haja melhor contribuição em equipe (que foi aprimorada ao longo do tempo), mas que imediatamente provoca um envolvimento total do grupo no projeto.

6. CONCLUSÃO

Este projeto realizou uma análise comparativa aprofundada entre as árvores BST, AVL e RBT aplicadas à construção de um índice invertido, validando experimentalmente as trocas fundamentais entre simplicidade de implementação e eficiência operacional. As estatísticas coletadas demonstraram de forma inequívoca a superioridade das árvores balanceadas (AVL e RBT) em cenários de busca intensiva, que são característicos da aplicação estudada.

Os resultados confirmaram que a Árvore Binária de Busca (BST), apesar de sua simplicidade, apresenta um desempenho de busca inconsistente e imprevisível, com uma tendência a um maior número de comparações e, em casos desfavoráveis, à degeneração de sua estrutura. Em contrapartida, as árvores AVL e RBT garantem uma altura logarítmica ($O(log(n))$), resultando em operações de busca significativamente mais rápidas e com menor variância, um fator crucial para a confiabilidade de um sistema de recuperação de informação.

No entanto, essa eficiência tem um custo. A análise das inserções revelou que o tempo de construção e a sobrecarga computacional, mensurada pelo número de rotações, são notavelmente maiores nas árvores AVL e RBT. O estudo também confirmou a nuance teórica de que a árvore AVL, por manter um critério de balanceamento mais rigoroso, tende a realizar mais rotações que a RBT. Além disso, a investigação de anomalias de desempenho (outliers) apontou para a complexa interação entre os algoritmos e o gerenciamento dinâmico de memória (Heap), uma lição prática valiosa sobre como fatores de baixo nível podem impactar a performance de estruturas de dados dinâmicas.

Nesse contexto, surge a importância do uso de linguagens de baixo nível, como C ou C++, pois permitem um estudo mais detalhado do funcionamento interno da máquina, proporcionando maior controle sobre a alocação de funções, variáveis e o acesso à memória. No entanto, vale destacar também que linguagens de alto nível, como Python, facilitam significativamente a análise e visualização de dados, sendo mais indicadas para prototipagem e exploração inicial.

Portanto, a escolha da estrutura de dados ideal não é absoluta, mas uma decisão de engenharia que depende dos requisitos da aplicação. Para o problema do índice invertido, onde a frequência e a velocidade das buscas são primordiais, o custo adicional de manutenção das árvores AVL ou RBT é um investimento justificável. Ele assegura a estabilidade e a previsibilidade do sistema, amenizando os riscos de degradação de desempenho inerentes a uma estrutura não balanceada como a BST.

7. REFERÊNCIAS

CORMEN, Thomas H. ***Introduction to algorithms***. 3. ed. Cambridge: MIT Press, 2009. Cap. 13. (Autores adicionais: Charles E. Leiserson, Ronald L. Rivest, Clifford Stein.)

GEEKSFORGEEKS. **Introduction to Red-Black Tree**. Disponível em: https://www.geeksforgeeks.org/dsa/introduction-to-red-black-tree/. Acesso em: 18 jun. 2025.

KNUTH, Donald E. ***The art of computer programming. Volume 1: Fundamental algorithms***. 3. ed. Boston: Addison-Wesley, 1997. Seção 2.5.

OLIVEIRA, A. J. M, PEREIRA, L. B, FALQUETO, T. A, PINTO, M. V., OLIVEIRA, J. P. J., *et all*. ***Projeto final ED***. GitHub, 2025. Disponível em: https://github.com/Os-Estudiosos/projeto-final-ed. Acesso em: 26 maio 2025.

PFAF, Ben. **Performance analysis of BSTs in system software**. *ACM SIGPLAN Notices*, v. 38, n. 2, p. 30–39, 2003. Disponível em: https://benpfaff.org/papers/libavl.pdf#:~:text=height%20to%20O%28lg%20n%29,7%2C%208%2C%209%2C%2010. Acesso em: 18 jun. 2025.

TOMAŠEVIĆ, Milo; ŠTRBAC-SAVIĆ, Svetlana. **Comparative performance evaluation of the AVL and Red-Black Trees**. In: *The 6th International Scientific Conference on Information Technology and Data Related Research – Sinteza 2019*. Belgrado: Singidunum University, 2019. p. 17–23. Disponível em: https://www.researchgate.net/publication/262401026_Comparative_performance_evaluation_of_the_AVL_and_red-black_trees#:~:text=are%20lower%20in%20the%20AVL,the%20number%20of%20comparisons%20is. Acesso em: 18 jun. 2025.

WIRTH, Niklaus. ***Algorithms + data structures = programs***. Englewood Cliffs: Prentice Hall, 1976. Seção 4.4.7. Disponível em: https://www.cl72.org/110dataAlgo/Algorithms%20%20%20Data%20Structures%20=%20Programs%20%5BWirth%201976-02%5D.pdf.