Trabalho de Estrutura de Dados - A2

Carlos Daniel, Gabriela Barbosa, Gabrielly Chácara,
Jaime Willian e Walléria Simões

**Relatório do Trabalho de Estrutura de Dados -
Índice Invertido e Análise Comparativa de Estruturas de Dados**

18-06-2025

Sumário

Introdução..... 3
Binary Search Tree - BST..... 4
Divisão de tarefas..... 4
Implementação..... 4
Características Principais da Implementação:..... 4
Integração com Estatísticas:..... 5
Vantagens:..... 5
Desvantagens:..... 6
Observações..... 6
AVL..... 7
Divisão de tarefas..... 7
Implementação..... 7
Características Principais da Implementação:..... 7
Vantagens:..... 8
Desvantagens:..... 9
Observações..... 9
Árvore Rubro-Negra - RBT..... 10
Divisão de tarefas..... 10
Implementação..... 10
Principais Funções da Implementação:..... 10
Otimizaçoes Implementadas:..... 11
Vantagens:..... 11
Desvantagens:..... 12
Observações..... 12
Estatísticas..... 13
Criação e análise das estatísticas criadas..... 13
Implementação de profundidade da árvore..... 13
Otimizações de Desempenho na Geração do Relatório Evolutivo (CSV)..... 15
Comparação de estruturas utilizando gráficos..... 16
Altura..... 16
Densidade:..... 17
Profundidade média..... 19
Profundidade mínima..... 21
Estatísticas de busca..... 25
Conclusões sobre eficiência de cada estrutura..... 29
Últimas considerações e conclusão..... 30

Introdução

Este relatório mostra o processo de implementação e análise comparativa de três estruturas de dados fundamentais: a **Árvore Binária de Busca** (**BST**), a **Árvore AVL** e a **Árvore Rubro-Negra** (**RBT**). Este trabalho foi desenvolvido com o objetivo de codificar as três árvores em C++ e analisar suas diferenças, tanto em suas implementações quanto em desempenho.

As árvores binárias de busca (BST) foram projetadas para tornar a busca de dados mais rápida. No entanto, elas têm um problema: se os dados forem inseridos em ordem alfabética (ou numérica), a árvore pode ficar desbalanceada e se comportar como uma lista encadeada, perdendo as propriedades em $\log_2(n)$ e tornando as operações, como buscas, lentas. Para corrigir isso, foram criadas as árvores AVL e Rubro-Negra (RBT). Elas se ajustam automaticamente para manter o balanceamento, garantindo que as buscas sejam sempre eficientes e, por consequência, as remoções e inserções também.

Neste documento, vamos explicar como cada árvore foi implementada e os desafios que encontramos. Também faremos uma análise de desempenho usando gráficos para comparar as estruturas. Ao final, apresentaremos nossas conclusões sobre as vantagens e desvantagens de cada uma, com base nos resultados dos testes.

Binary Search Tree - BST

A Árvore Binária de Busca é uma estrutura baseada em nós, onde cada nó armazena um índice. No contexto do nosso projeto de índice invertido, cada palavra do índice é uma chave. A principal característica de uma BST é que, para qualquer nó, todas as chaves em sua subárvore esquerda são menores que a chave do nó, e todas as chaves na subárvore direita são maiores.

Divisão de tarefas

A implementação da BST foi organizada da seguinte forma entre os membros do grupo:

- **Implementação da estrutura (bst.cpp/h):** Carlos e Jaime ficaram responsáveis por codificar a lógica central da árvore, incluindo as funções de inserção, e busca de nós.
- **Leitura de dados (data.cpp/h):** Walléria ficou responsável pelo módulo capaz de ler os dados de entrada a partir dos arquivos de texto, permitindo popular a árvore para os testes.
- **Interface de linha de comando (main_bst.cpp):** Gabriela implementou a interface com o usuário, focando no comando **search** e na integração com a estrutura de dados criada.
- **Coordenação e testes (test_bst.cpp):** Gabrielly realizou os testes unitários para validar a implementação e elaborar o *README* inicial do projeto.

Implementação

Características Principais da Implementação:

1. **Nós (`Node`):** Os nós da BST armazenam uma palavra (`word`) (string) e um `vector` de `documentIds`, que registra em quais documentos essa palavra aparece. Cada nó possui ponteiros para seus filhos esquerdo e direito, e para seu pai, facilitando a navegação na árvore.
2. **Criação da árvore (`BST::create()`):** Inicia uma estrutura de árvore binária com a raiz nula, representando uma árvore vazia.
3. **Inserção (`BST::insert()`):**
    - A inserção de uma nova palavra segue a lógica padrão da BST: compara-se a palavra a ser inserida com o nó atual, percorrendo a subárvore esquerda se for menor ou a direita se for maior, até encontrar uma posição vazia.

- Para palavras que já existem na árvore (duplicatas), o novo `documentId` é adicionado ao `vector documentIds` do nó existente, mantendo-o ordenado através de uma busca binária (`TREE_UTILS::binarySearch`).
- A função mede e retorna o número de comparações e o tempo de execução para cada operação de inserção.
- **Não Balanceada:** A implementação da BST não inclui mecanismos de auto-balanceamento (como rotações).

4. **Busca (`BST::search()`):**
   - A busca por uma palavra segue o algoritmo padrão da BST, percorrendo a árvore de forma eficiente (em média, O(log n) para árvores balanceadas; O(n) para degeneradas).
   - A função retorna uma estrutura `SearchResult` contendo se a palavra foi encontrada, os IDs dos documentos associados e as métricas de desempenho (comparações e tempo).
5. **Destruição (`BST::destroy()`):** Garante a liberação de toda a memória alocada para os nós da árvore de forma recursiva, prevenindo vazamentos de memória.

Integração com Estatísticas:

A BST se integra perfeitamente com o módulo de utilitários de árvore (`TREE_UTILS`). Embora a BST não utilize o campo `node->height` para balanceamento interno, o sistema de coleta de estatísticas (`TREE_UTILS::collectAllStats`) foi projetado para calcular a altura real da árvore (maior galho) de forma independente desse campo, garantindo que as métricas estruturais (altura, profundidade média, densidade) sejam precisas e comparáveis com as demais implementações de árvores no projeto. As comparações e tempos de execução são rastreados em todas as operações de inserção e busca.

Vantagens:

- **Simplicidade de implementação:** As operações básicas (inserção, busca, remoção) são relativamente fáceis de entender e codificar.
- **Ordem mantida:** A travessia em ordem (in-order *traversal*) da BST retorna os elementos em ordem crescente, o que é útil para listar dados ordenados.
- **Estrutura dinâmica:** Permite a adição e remoção eficiente de nós, adaptando-se a conjuntos de dados que mudam com o tempo.

Desvantagens:

- **Degeneração (pior caso):** A principal desvantagem. Se os dados forem inseridos em uma ordem já classificada (crescente ou decrescente), a BST pode se degenerar em uma "lista encadeada" (uma árvore completamente desequilibrada). Nesse cenário, as operações (busca, inserção, remoção) têm complexidade linear de tempo, ou seja, O(n), tornando-a ineficiente.

- **Falta de balanceamento automático:** Diferente de árvores balanceadas (como AVL ou RBT), a BST não garante automaticamente um bom desempenho. Para evitar a degeneração, mecanismos de balanceamento adicionais são necessários, o que aumenta a complexidade da estrutura.

AVL

A **Árvore AVL**, é a primeira árvore binária de busca auto balanceada. Ela foi projetada para resolver o principal problema da BST: o risco de desbalanceamento. A AVL garante que a diferença de altura entre as subárvores de qualquer nó (conhecida como **fator de balanceamento**) nunca seja maior que 1.

Para manter essa propriedade, a árvore executa operações de rotação (simples ou duplas) sempre que uma inserção ou remoção viola o fator de balanceamento.

Divisão de tarefas

- **Implementação da estrutura (avl.cpp/h):** Carlos e Jaime ficaram responsáveis por codificar a lógica central da árvore AVL, incluindo o cálculo do fator de balanceamento e a implementação das rotações.
- **Coleta de estatísticas (main_avl.cpp):** Jaime implementou a funcionalidade de coleta de métricas, como tempo de execução e número de comparações, e adicionou o comando **stats** à interface.
- **Melhorias na interface (CLI):** Gabriela e Walléria refinaram a interface de linha de comando, melhorando a experiência do usuário. Walléria também focou no tratamento de erros da aplicação.
- **Testes e gestão do projeto (test_avl.cpp):** Gabrielly desenvolveu os testes unitários para garantir a corretude da implementação, além de decidir como os gráficos seriam criados, a estrutura do relatório e o *README* .

Implementação

A Árvore AVL é uma evolução da BST, sendo uma Árvore Binária de Busca **auto-balanceada**. Sua implementação procura garantir que as operações de busca, inserção e remoção sempre ocorram em tempo logarítmico, evitando os problemas de degeneração da BST simples.

Características Principais da Implementação:

1. **Nós (`Node`) com Altura:** Cada nó da AVL, além da `palavra` e dos `documentIds`, armazena explicitamente sua `altura`. Este campo é crucial para determinar o fator de balanceamento e identificar desequilíbrios.
2. **Criação da árvore (`AVL::create()`):** Inicia uma árvore vazia com a raiz nula.
3. **Inserção (`AVL::insert()`):**

a. A inserção começa como uma inserção comum de BST, encontrando a posição correta para o novo nó.

b. **Balanceamento automático:** Após a inserção, a árvore verifica o caminho da inserção da folha até a raiz (ou até encontrar o primeiro nó que ficou desbalanceado ou que não teve a altura alterada após a inserção). Para cada nó nesse caminho:

c. A altura do nó é atualizada (`node->height`).

d. O fator de balanceamento (diferença entre as alturas das subárvores esquerda e direita) é calculado.

e. Se um nó estiver desbalanceado (fator de balanceamento maior que 1 ou menor que -1), uma ou duas operações de rotação (simples ou duplas) são executadas (`sideRotate`). As rotações rearranjam os nós para restaurar a propriedade de balanceamento da AVL.

f. As operações de `sideRotate` também garantem a atualização correta das alturas dos nós envolvidos, pois, para os nós acima do primeiro desbalanceado, as alturas seguiram iguais ao momento antes da inserção, ou seja, não precisam ser incrementadas, enquanto as abaixo delas terão a altura incrementada. Vale ressaltar que as alturas dos nós rotacionados são atualizadas na função `sideRotate` após a rotação.

4. **Busca (`AVL::search()`):**

   a. A busca por uma palavra é idêntica à da BST, aproveitando a propriedade de ordenação.

   b. Sua eficiência é garantida pelo balanceamento da árvore.

   c. As métricas de desempenho (comparações e tempo) são rastreadas.

5. **Destruição (`AVL::destroy()`):** Libera recursivamente toda a memória alocada para a árvore, semelhante à BST.

Vantagens:

- **Desempenho logarítmico garantido:** A principal vantagem. Todas as operações (inserção, busca e remoção) são garantidamente O(log n) no pior caso, pois a altura da árvore é sempre mantida em um limite logarítmico em relação ao número de nós. Isso a torna altamente previsível e eficiente, mesmo com dados de entrada ordenados.

- **Balanceamento Estrito:** Mantém um balanceamento muito rigoroso (fator de balanceamento de -1, 0 ou +1), o que resulta em caminhos de busca muito próximos do ótimo.

Desvantagens:

- **Maior Complexidade de Implementação:** A lógica para rastrear alturas e realizar rotações e rebalanceamento adiciona uma complexidade significativa ao código, tornando-a mais difícil de implementar e depurar do que uma BST simples.
- **Overhead nas Inserções:** Embora o desempenho seja O(log n), cada operação de inserção pode envolver múltiplas atualizações de altura e rotações. Isso introduz um pequeno custo constante extra por operação em comparação com uma BST que, por acaso, permaneça balanceada. Esse overhead pode ser perceptível em cenários com muitas operações de modificação e poucas buscas.
- **Maior Uso de Memória por Nó:** Cada nó requer um campo adicional para armazenar sua altura.

Observações

Na Entrega 2, focamos na implementação das estatísticas de inserção (tempo, altura, densidade da árvore e tamanho dos galhos) no CLI de **stats**. Contudo, a integração completa das estatísticas de busca nesse mesmo CLI não foi finalizada, ficando restrita ao CLI de **search** com métricas básicas (comparações e tempo de busca por palavra individual). Essa lacuna deve-se à incerteza sobre onde e em que nível de detalhe estas estatísticas deveriam ser exibidas (no próprio CLI ou apenas detalhadas no relatório final), além da sobrecarga de trabalho com projetos paralelos das disciplinas de Álgebra Linear Numérica e Técnicas e Algoritmos em Ciência de Dados. As estatísticas de busca acabaram sendo implementadas apenas na Entrega Final.

Árvore Rubro-Negra - RBT

A **Árvore Rubro-Negra** é outra variação de árvore binária de busca auto balanceada, que, assim como a AVL, garante que as operações principais tenham uma complexidade de tempo no pior caso de **O(log n)**. No entanto, ela atinge o balanceamento de uma forma diferente.

Em vez de usar um fator de balanceamento, a RBT utiliza um sistema de cores. Cada nó é pintado de vermelho ou preto, e o balanceamento é mantido através de um conjunto de regras estritas (como "um nó vermelho não pode ter um filho vermelho" e "todo caminho de um nó até suas folhas descendentes deve conter o mesmo número de nós pretos"). Embora suas regras de balanceamento sejam menos rígidas que as da AVL, permitindo que a árvore seja ligeiramente mais desbalanceada, isso muitas vezes resulta em menos rotações durante as inserções, tornando-a uma escolha eficiente em cenários de escrita intensiva.

Divisão de tarefas

- **Implementação da Estrutura (rbt.cpp/h):** Walléria e Carlos ficaram responsáveis por codificar a lógica complexa da árvore Rubro-Negra, incluindo as operações de inserção, as regras de coloração e as rotações necessárias para manter as propriedades da árvore. Eles também focaram na documentação completa do código.

- **Testes e Integração Final:** Gabriela assumiu a tarefa de unificar as três estruturas de dados em um sistema coeso e desenvolveu os testes unitários (**test_rbt.cpp**) para validar a nova implementação.

- **Análise Comparativa:** Gabrielly e Jaime trabalharam juntos na fase de análise. As responsabilidades incluíam a coleta dos dados de desempenho de todas as árvores, a geração dos gráficos e tabelas para a comparação visual e a redação do relatório final.

Implementação

Principais Funções da Implementação:

- **Inicialização:** A função `create()` inicia a árvore com uma raiz nula. Ao adicionar nós, `initializeNode()` é chamada para alocar memória e definir valores padrão, sendo o mais importante a cor do nó, que é inicializada como `vermelha`.

- **Inserção (insert):** A inserção é um processo de duas etapas. Primeiro, a palavra é inserida seguindo a lógica padrão de uma BST. Caso a palavra já exista, a função apenas atualiza a lista de documentos associada a ela. Em seguida, a função `fixInsert()` é chamada para corrigir quaisquer violações das propriedades da RBT que a inserção possa ter causado.

- **Balanceamento (`fixInsert`):** Esta é a função central da RBT. A partir do nó recém-inserido, ela sobe recursivamente pela árvore até a raiz, verificando e corrigindo violações através de rotações e recolorações. A lógica de correção foi dividida em três casos principais, determinados pela cor do "tio" do nó atual:
    a. **Tio Vermelho:** O caso mais simples, resolvido apenas com a recoloração do pai, do tio e do avô do nó.
    b. **Tio Preto (Caso Triângulo):** Requer uma rotação simples para se transformar no "caso linha".
    c. **Tio Preto (Caso Linha):** Requer uma rotação e uma recoloração para resolver o desbalanceamento.

- **Busca (`search`):** O algoritmo de busca é idêntico ao de uma BST padrão. Durante sua execução, o tempo de operação e o número de comparações são medidos para posterior análise estatística.

- **Destruição (`destroy`):** Libera toda a memória alocada pela árvore de forma recursiva, garantindo que não haja vazamentos de memória.

Otimizações Implementadas:

- **Controle de Altura:** Embora a RBT não utilize a altura de um nó para seu balanceamento, o campo `height` foi mantido em cada nó. Isso foi feito para garantir a compatibilidade com as funções de estatística genéricas do projeto, que calculam a altura e outras métricas para todas as três árvores.

- **Inserção Eficiente de Documentos:** Para otimizar a atualização da lista de documentos em palavras duplicadas, foi implementada uma busca binária (`binarySearch`) auxiliar. Isso garante que a inserção de novos IDs de documentos no vetor `documentIds` seja feita de forma eficiente, mantendo o vetor sempre ordenado.

- **Verificação de Propriedades:** Durante a fase de desenvolvimento e testes, foi criada uma função de validação (`checkBlackHeight`) para verificar programaticamente se a propriedade da "altura negra" (todo caminho da raiz a uma folha tem o mesmo número de nós pretos) era mantida após as operações, agilizando a depuração. Além disso, a função `validBlackHeight` garante que todos os nós da árvore (não apenas a raiz) satisfaçam a propriedade de altura preta.

Vantagens:

- **Desempenho Logarítmico Garantido:** Semelhante à AVL, todas as operações fundamentais (inserção, busca e remoção) têm complexidade de tempo de O(log N) no pior caso. Isso assegura que a RBT é altamente eficiente e previsível, mesmo com grandes volumes de dados ou com padrões de inserção que degeneram em uma BST simples.

- **Balanceamento Flexível e Eficiente para Modificações:** A RBT mantém o balanceamento através de regras de coloração menos rígidas que a AVL, permitindo que a árvore seja ligeiramente mais "profunda" (altura máxima de $2 * \log_2 (n+1)$). No entanto, essa flexibilidade geralmente resulta em menos operações de rotação e rebalanceamento por inserção ou remoção em comparação com a AVL. Isso pode tornar as operações de modificação (inserção e remoção) marginalmente mais rápidas em termos de fatores constantes, em cenários com muitas atualizações.

Desvantagens:

- **Maior complexidade de implementação:** A RBT é consideravelmente mais complexa de implementar que uma BST simples, e sua lógica de balanceamento baseada em cores e casos específicos pode ser até mais complicada que a da AVL em alguns aspectos.
- **Altura Ligeiramente Maior que a AVL:** Por suas regras de balanceamento serem menos estritas, uma RBT pode ter uma altura marginalmente maior do que uma AVL para o mesmo número de nós. Isso se traduz em um número ligeiramente maior de comparações por operação de busca em média, comparado à AVL.
- **Mais memória por nó:** Cada nó da RBT requer um bit extra para armazenar sua cor (vermelho ou preto), o que representa um pequeno aumento no uso de memória por nó em comparação com uma BST que não armazena altura

---

Observações

Foi identificado um erro nas estatísticas das árvores RBT e BST, onde a altura calculada era menor que a profundidade mínima (o que é uma impossibilidade lógica). A causa era que, ao contrário da árvore AVL, a propriedade de altura dos nós não era atualizada corretamente nessas estruturas.

Para resolver o problema, a função de coleta de estatísticas `collectTreeStats` foi alterada para calcular a altura dinamicamente durante a travessia. Ela agora rastreia a maior profundidade encontrada em qualquer ramo da árvore e, ao final, atribui esse valor como a altura correta. Com essa mudança, a estatística de altura passou a ser precisa para todas as árvores, garantindo uma medição correta e uma comparação justa entre elas.

Estatísticas

Implementação de profundidade da árvore

No nosso projeto o cálculo da profundidade é feito de maneira abrangente, cobrindo a profundidade de cada nó, a profundidade média da árvore, a profundidade mínima (menor caminho até uma folha) e a profundidade máxima (altura da árvore).

O cálculo principal ocorre dentro da função recursiva `collectTreeStats`, que é chamada por `collectAllStats`.

Cálculo da Profundidade e Métricas Relacionadas na Árvore

A medição da profundidade e de suas métricas associadas na estrutura da árvore é realizada principalmente através de um percurso recursivo, garantindo a precisão para todos os tipos de árvores (BST, AVL, RBT).

1. Profundidade do Nó Atual (`currentDepth`):

- **Como é Calculado:** Durante a travessia recursiva da árvore pela função `collectTreeStats(Node* node, int currentDepth, ...)`, o parâmetro `currentDepth` rastreia a profundidade do nó sendo visitado.
- Para a raiz da árvore, `currentDepth` é iniciado em `0` (primeiro nível).
- A cada chamada recursiva para um filho (`node->left` ou `node->right`), `currentDepth` é incrementado em `1` (`currentDepth + 1`), refletindo a descida para o próximo nível da árvore.

2. Profundidade Média (`stats.averageDepth`):

- **Como é calculado:** A profundidade média é uma estatística geral que indica a "profundidade típica" de um nó na árvore.
- Dentro de `collectTreeStats`, uma variável de referência `totalDepth` acumula a profundidade de *cada nó visitado* (`totalDepth += currentDepth;`).
- Uma variável `nodeCount` acumula o número total de nós (`nodeCount++`).

- Na função `collectAllStats`, `stats.averageDepth` é calculado dividindo a `totalDepth` acumulada pelo `nodeCount` total: `(double)totalDepth / nodeCount.`

3. Profundidade Mínima (`stats.minDepth`):

- **Como é calculada:** A profundidade mínima representa o caminho mais curto da raiz até qualquer nó folha na árvore.
- Na função `collectAllStats`, `minDepth` é inicializado com `INT_MAX` (o maior valor possível para um `int`, importado de `<climits>`).
- Dentro de `collectTreeStats`, quando um **nó folha** é encontrado (ou seja, `node->left == nullptr && node->right == nullptr`), a profundidade atual desse nó folha (`currentDepth`) é comparada com a `minDepth` registrada até então. `minDepth = std::min(minDepth, currentDepth);` garante que `minDepth` sempre armazene a menor profundidade encontrada entre todas as folhas.

4. Altura da Árvore / Profundidade Máxima (`stats.height` como `maxDepthFound`):

- **Como é Calculado**: A altura da árvore é definida como a profundidade do caminho mais longo da raiz até qualquer nó folha.
- Na função `collectAllStats`, uma variável de referência `maxDepthFound` é inicializada com -1 (para garantir que a primeira profundidade válida seja maior).
- Dentro do `collectTreeStats`, para *cada nó visitado*, `maxDepthFound` é atualizado para ser o maior entre seu valor atual e a `currentDepth` do nó. `maxDepthFound = std::max(maxDepthFound, currentDepth);.`
- Após a conclusão da travessia recursiva, o valor final de `maxDepthFound` é atribuído à `stats.height` em `collectAllStats`. Essa abordagem garante a altura real da árvore, independentemente de como o campo `height` individual de nós é mantido internamente por cada tipo de árvore (BST, AVL, RBT).

Em resumo, a profundidade é rastreada durante uma travessia completa, e essa informação é então usada para derivar as métricas de profundidade média, mínima e máxima (altura da árvore), fornecendo uma visão abrangente da forma e do balanceamento da estrutura.

Estatísticas de Busca

A avaliação completa do desempenho de estruturas de dados como Árvores AVL, BST e RBTS não se limitou apenas ao tempo e comparações de inserção ou às características estruturais. Na entrega final, foram adicionadas novas métricas de desempenho relacionadas à operação de busca no relatório evolutivo da árvore, ou seja, analisando a evolução da árvore ao longo dos `n_docs` que são passados como parâmetro

Estatísticas de Busca Adicionadas ao Relatório CSV

Para cada número de documentos indexados, um conjunto amostral de buscas é realizado e as seguintes estatísticas são coletadas e registradas:

- `Tempo_Maximo_Busca_Amostra`**:** Tempo (em milissegundos) da busca mais longa ocorrida dentro da amostra de buscas realizadas para o estado atual da árvore. É uma métrica importante para identificar cenários de pior caso de desempenho.

- `Comparacoes_Total_Busca_Amostra`**:** Indica o número total acumulado de comparações entre chaves realizadas durante todas as buscas na amostra.

- `Tempo_Medio_Busca_Amostra_Por_Palavra`: É o tempo médio (em milissegundos) que cada busca leva para ser concluída, calculado dividindo o tempo total da amostra pelo número de buscas efetivamente realizadas.

- `Comparacoes_Medias_Busca_Amostra_Por_Palavra`**:** Representa o número médio de comparações realizadas por busca na amostra.

Num_Docs	Tipo_Arvore	Altura	Profundidade_Média	Profundidade_Minima	Max_Desbalanceamento
100	AVL	14	10.9717	9	1
100	BST	29	15.4204	5	0
100	RBT	15	11.05	9	6
1000	AVL	16	12.3435	10	1
1000	BST	34	17.4718	5	0
1000	RBT	16	12.4675	10	7

5000	**AVL**	**16**	**12.6024**	**11**	**1**
5000	**BST**	**36**	**17.8466**	**5**	**0**
5000	**RBT**	**17**	**12.7318**	**10**	**7**
10000	**AVL**	**16**	**12.6085**	**11**	**1**
10000	**BST**	**36**	**17.8588**	**5**	**0**
10000	**RBT**	**17**	**12.7386**	**10**	**7**

Desafios e Soluções na Implementação das Estatísticas de Busca

A implementação das estatísticas de busca no relatório evolutivo enfrentou quatro desafios principais.

- **Custo Computacional:** Realizar milhares de buscas a cada passo da análise era muito lento. A solução foi limitar o número de buscas por amostra a um valor fixo (ex: 2000), balanceando o custo e a precisão estatística.

- **Amostra de Busca Relevante:** As palavras buscadas precisavam existir na árvore em cada estágio. Para resolver isso, a amostra de palavras passou a ser gerada dinamicamente, utilizando apenas palavras dos documentos já processados naquela etapa.

- **Generalização para Diferentes Árvores:** A função de estatísticas, sendo genérica, não sabia qual função de busca específica (`BST::search`, `AVL::search`, etc.) chamar. Isso foi solucionado passando a função de busca correta como um **ponteiro de função**, tornando o código flexível.

- **Gerenciamento de Métricas:** Era preciso calcular tanto a média quanto o pior caso do desempenho da busca. A solução foi usar variáveis separadas para rastrear o tempo/comparações totais (para a média) e o tempo/comparações máximas (pior caso).

Otimizações de Desempenho na Geração do Relatório Evolutivo (CSV)

A geração do relatório evolutivo (CSV) era extremamente lenta, especialmente com muitos documentos, devido a dois grandes gargalos de desempenho.

A **primeira otimização** corrigiu o problema principal: em vez de destruir e reconstruir uma árvore de análise do zero a cada passo, a árvore passou a ser construída de forma **incremental**. Agora, ela é criada uma única vez e, a cada etapa, apenas as palavras do novo documento são adicionadas. Isso reduziu drasticamente o tempo de processamento e evitou erros de memória.

A **segunda otimização** (versão atual) melhorou a coleta de palavras para os testes de busca. Em vez de criar, copiar e embaralhar grandes listas de palavras a cada passo, as amostras agora são selecionadas diretamente do conjunto de dados principal usando índices aleatórios, eliminando operações pesadas e o uso excessivo de memória.

Em conjunto, essas mudanças tornaram a geração de relatórios eficiente e viável, mesmo para grandes volumes de dados.

Comparação de estruturas utilizando gráficos

Altura

1. Árvore Binária de Busca (BST)

Observamos um *crescimento de altura acentuado* para a BST à medida que o número de documentos aumenta. A altura da árvore dispara rapidamente, atingindo valores muito elevados.

Implicação: Este comportamento é característico da **degeneração da BST**. Isso acontece porque a BST não se equilibra sozinha, fazendo com que a árvore se assemelhe com uma lista encadeada (ou como apelidamos durante a implementação, um ‘cipó’), chegando a altura 36 já nos 2000 primeiros documentos.

2. Árvore AVL

Em comparação a BST, a altura da árvore AVL se comporta de forma mais estável, isso é reflexo das rotações automáticas feitas para manter a árvore sempre balanceada.

Implicação: A altura da árvore cresce aos poucos, com “degraus” visíveis no gráfico.

Para 10.000 documentos, a altura da AVL se estabiliza em 16, menos da metade da altura alcançada pela BST.

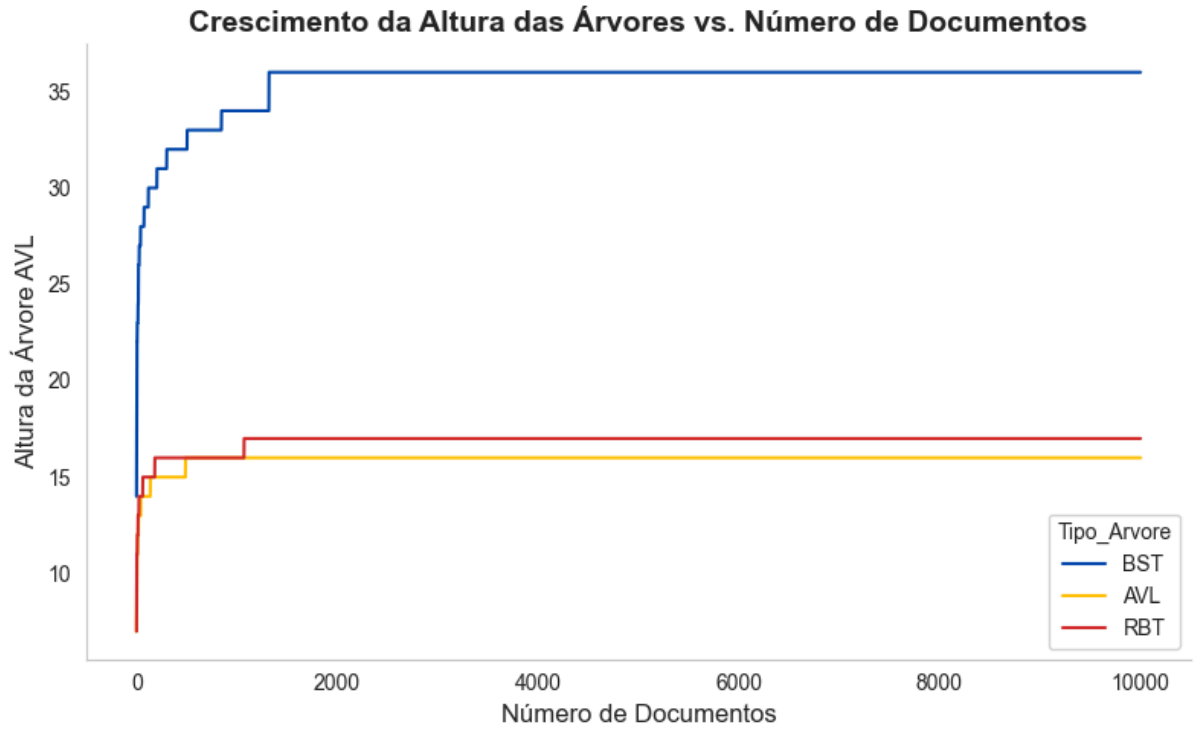
3. Árvore Rubro-Negra (RBT) - (Linha Vermelha):

A RBT também mostra um crescimento muito controlado, parecido com o da AVL. Para 10.000 documentos, sua altura fica um pouco mais alta que a da AVL, mas se estabiliza em 17.

Isso acontece porque a RBT é um pouco mais “flexível” nas regras de balanceamento - ela tolera um pequeno desbalanceamento em troca de realizar menos rotações.

Conclusão:

Os resultados mostram que estruturas de árvores auto-balanceadas, como AVL e Rubro-Negra, mantêm a altura sob controle mesmo com grandes volumes de dados, garantindo desempenho eficiente e estável. Em contraste, a BST simples sofre degeneração rápida, comprometendo gravemente a eficiência das operações.



Densidade:

Tipo_Arvore	Min_Dens	Num_Docs_Min	Max_Densidade	Num_Docs_Max
BST	**1.318550e-07**	**1320**	**0.003937**	**1**
AVL	**1.051190e-01**	**489**	**0.505882**	**1**
RBT	**6.610510e-02**	**1072**	**0.505882**	**1**

Usamos a seguinte definição de Densidade:

- A densidade de uma árvore binária é calculada dividindo o número de nós pela quantidade de nós possíveis na árvore, sendo o número de nós possíveis igual a 2^(h+1) - 1.

1. Árvore Binária de Busca (BST):

A densidade da BST permanece próxima de zero durante toda a inserção dos documentos. Como visto no gráfico das alturas, a BST cresce de forma desbalanceada, ela atinge grandes alturas sem aproveitar bem os níveis da árvore. A fórmula 2^{h+1} - 1 explode para alturas grandes, e a quantidade real de nós é pequena comparada ao total possível.

A densidade alcança seu máximo já com `n_docs` igual a 1, iniciando com densidade igual a 0.003937. Já seu mínimo é alcançado com **1320** arquivos, chegando a 1.318550e-07.

Conclusão: A árvore BST é bem mais “esticada”, sendo ineficiente estruturalmente.

2. Árvore AVL (Adelson-Velsky e Landis):

A densidade da AVL começa mais irregular, mas se estabiliza em torno de 0.15, o que indica uma boa ocupação dos níveis da árvore.

Isso acontece porque a AVL mantém o balanceamento rígido, evitando que a altura cresça demais para um número dado de nós.

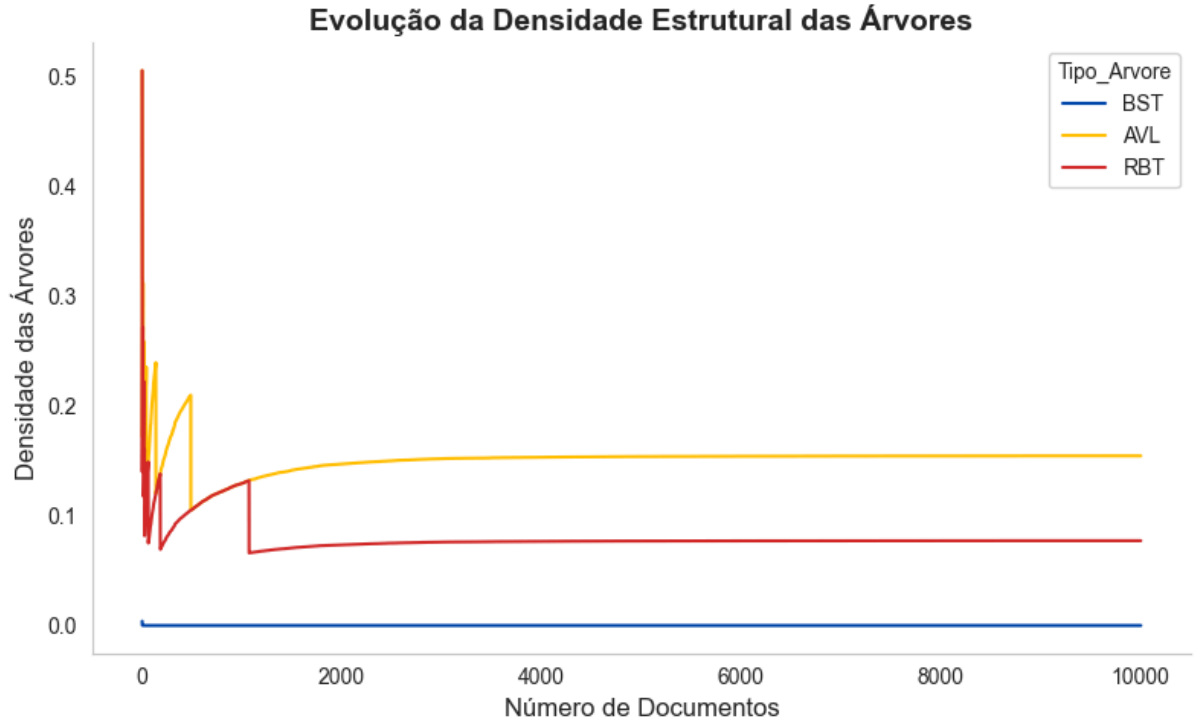
Conclusão: A AVL é **estruturalmente eficiente** — não ocupa o espaço ideal como uma árvore perfeitamente cheia, mas faz bom uso da estrutura.

3. Árvore Rubro-Negra (RBT) - (Linha Vermelha):

A RBT mostra comportamento parecido com a AVL, mas com densidade um pouco menor, finalizando a análise com 0.077336.

Isso é esperado: a RBT permite mais desbalanceamento que a AVL, resultando em árvores ligeiramente mais altas e, portanto, com menor densidade.

Conclusão: A RBT e*quilibra eficiência estrutural com custo menor de rebalanceamento*, mantendo boa densidade para aplicações práticas.



Profundidade média

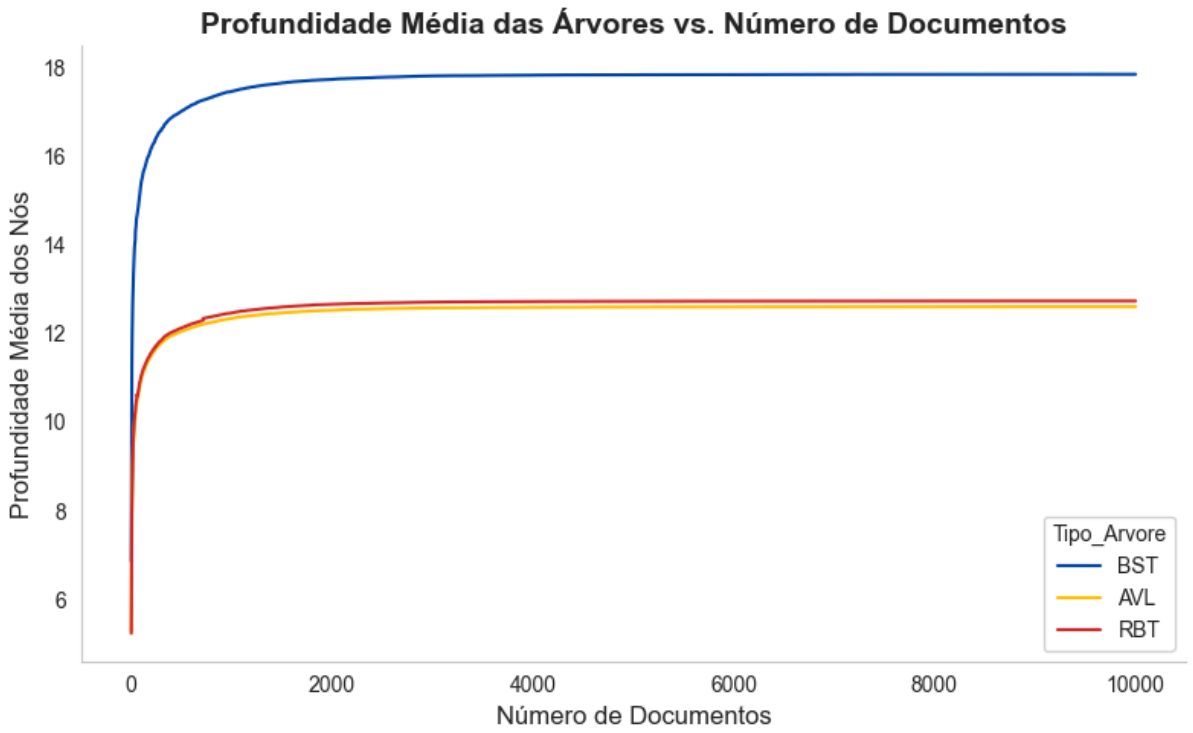
No gráfico de *Profundidade Média das Árvores vs Números de Documentos*, é exibido a profundidade média das árvores implementadas conforme o número de

documentos aumenta. Em suma, a profundidade média de uma árvore indica o quão fundo, em média, os nós estão em relação à raiz e, como podemos observar, possui um formato muito semelhante ao gráfico que compara o crescimento das alturas das árvores. A seguir, realizaremos uma análise individual de cada tipo de árvore:

- **BST:** O comportamento exibido acima evidencia uma limitação importante da BST, a profundidade média de seus nós tende a crescer rapidamente, principalmente quando os dados que estão sendo inseridos estão em ordem crescente ou decrescente. Esse comportamento acaba resultando em árvores desbalanceadas, comprometendo o desempenho das operações de busca e inserção. O motivo da aparente estabilização após 1000 documentos pode decorrer do fato de que, neste ponto, grande parte das palavras já foram inseridas na árvore. Como resultado, o algoritmo passa a realizar atualizações em nós existentes, o que não interfere significativamente na profundidade média.

- **AVL:** Como podemos observar no gráfico, a árvore AVL apresenta a menor profundidade média entre as estruturas analisadas. Isso ocorre graças à principal proposta da AVL: solucionar o problema de desequilíbrio presente nas BSTs tradicionais. Para isso, a AVL adota um mecanismo automático de balanceamento, que é aplicado a cada inserção ou remoção de elementos. Especificamente, a AVL impõe uma condição rigorosa: para todo nó n, a diferença entre as alturas de suas subárvores esquerda e direita não pode exceder 1. Essa regra de balanceamento estrita é o que garante o comportamento mais estável e a baixa profundidade média observada no gráfico.

- **RBT:** No caso da árvore Rubro-Negra (RBT), observamos um comportamento intermediário entre a AVL e a BST, porém consideravelmente mais próximo da AVL. Isso se deve ao fato de que, embora a RBT também implemente um mecanismo de balanceamento, suas regras são menos rigorosas que as da AVL. O balanceamento da RBT permite uma maior variação nas alturas das subárvores, o que resulta em uma profundidade média superior à da AVL, mas ainda significativamente melhor do que a observada na BST.



Conclusão:

A análise da profundidade média das três estruturas deixa evidente o impacto das diferentes estratégias de balanceamento. A BST, por não possuir qualquer mecanismo para corrigir o desbalanceamento, apresenta o pior desempenho, com crescimento rápido e descontrolado da profundidade média, especialmente quando os dados são inseridos de forma ordenada. A AVL, por sua vez, mantém a menor profundidade média, graças à sua política de balanceamento mais rigorosa, que limita a diferença de altura entre as subárvores. Já a RBT apresenta um comportamento intermediário: seu balanceamento é menos estrito que o da AVL, mas ainda assim garante uma profundidade média muito inferior à da BST.

No entanto, a partir de um determinado ponto da inserção de documentos, aproximadamente após os 1000, observa-se uma semelhança de comportamento entre as três árvores: a profundidade média tende a se estabilizar. Isso provavelmente ocorre porque, nesse estágio, a maioria das palavras já foi inserida nas árvores e o algoritmo passa a realizar majoritariamente atualizações em nós existentes em vez de novas inserções. Como tais atualizações não alteram a estrutura da árvore, o impacto na profundidade média torna-se mínimo, fazendo com que todas as curvas apresentem uma tendência de estabilização.

Profundidade mínima

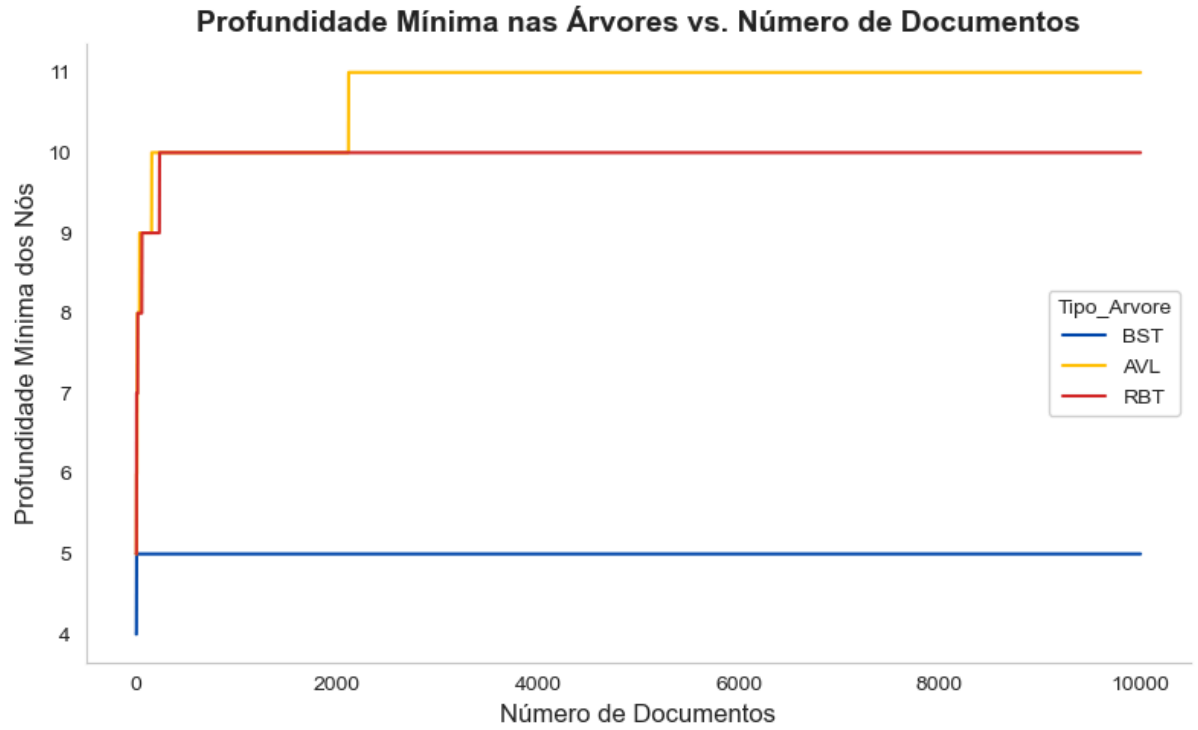
No gráfico *Profundidade Mínima nas Arvores vs Número de Documentos*, é apresentada a evolução da profundidade mínima das árvores implementadas conforme o número de documentos aumenta. A profundidade mínima representa a menor distância entre a raiz e um nó folha da árvore, ou seja, o caminho mais curto necessário para alcançar uma folha. Assim como observado na análise da profundidade média, o formato geral do gráfico acompanha o comportamento estrutural de crescimento das árvores à medida que novos documentos são inseridos. A seguir, detalharemos o desempenho de cada tipo de árvore individualmente.

- **BST:** Na BST, observamos que a profundidade mínima se mantém relativamente baixa ao longo de toda a inserção de documentos. Isso ocorre porque, por não haver qualquer mecanismo de balanceamento, a árvore pode acabar formando ramos muito curtos em uma das extremidades, principalmente dependendo da ordem de inserção dos dados. Mesmo que a árvore, como um todo, fique extremamente desbalanceada, basta a existência de um ramo curto para manter a profundidade mínima reduzida.

- **AVL:** Já a árvore AVL apresenta, de forma consistente, a maior profundidade mínima entre as estruturas. Isso é uma consequência direta de sua política de balanceamento altamente rigorosa, que impõe que, para qualquer nó n, a diferença entre as alturas de suas subárvores esquerda e direita não ultrapasse 1. Como resultado, a AVL força uma distribuição bastante uniforme das folhas, limitando a diferença entre a profundidade mínima e a máxima da árvore.

- **RBT:** A árvore Rubro-Negra (RBT) apresenta um comportamento intermediário. Seu balanceamento é menos estrito do que o da AVL, permitindo maior variação nas alturas das subárvores. Por conta disso, a RBT consegue manter algumas folhas relativamente próximas da raiz, mas sem os extremos desbalanceamentos típicos das BSTs. Assim, sua profundidade mínima permanece maior que a da BST, porém inferior à da AVL.

A análise da profundidade mínima evidencia o impacto do balanceamento de cada árvore. A BST, sem qualquer controle estrutural, apresenta a menor profundidade mínima devido à presença de ramos curtos isolados. A RBT, com um balanceamento mais flexível, mantém uma profundidade mínima intermediária. Já a AVL, com seu balanceamento rigoroso, exibe a maior profundidade mínima, garantindo uma distribuição mais uniforme das folhas. Além disso, a partir de aproximadamente 2000 documentos, todas as árvores tendem à estabilização da profundidade mínima, devido à redução de novas inserções e aumento das atualizações em nós já existentes.



Profundidade Média

O gráfico apresentado mostra a relação entre o "Número de Documentos" (em escala logarítmica) e a "Profundidade Média dos Nós" nas árvores.

Análise:

1. Dados Observados (Pontos nos gráficos): Os pontos usados para construir os gráficos acima representam os dados coletados, mostrando a profundidade média dos nós para diferentes números de documentos e diferentes árvores. Percebe-se que, à medida que o número de documentos aumenta, a profundidade média dos nós também aumenta. Entretanto, o padrão de crescimento nessa escala parece semelhante para todas as árvores, a não ser por uma constante que multiplica cada gráfico e que é maior para a BST.

2. Ajuste Logarítmico (Linha Laranja Tracejada):** Uma linha de ajuste logarítmico foi sobreposta aos dados. Essa linha representa um modelo do tipo $y = a \log_2(x) + b$, onde y é a profundidade média e x é o número de documentos.

3. **Qualidade do Ajuste Logarítmico**:

Para um número maior de documentos (a partir de aproximadamente 10^2 ou 10^3 documentos), o ajuste logarítmico parece se encaixar muito bem aos dados observados. A linha preta tracejada acompanha de perto os pontos dos gráficos, indicando que a profundidade média dos nós em todas as árvores crescem de forma logarítmica com o número de documentos.

Para um número muito pequeno de documentos (abaixo de 10), o ajuste logarítmico não é tão preciso. Há uma diferença notável entre os dados observados e a linha de ajuste, especialmente na porção inicial do gráfico. Isso é esperado, pois o comportamento de uma estrutura de dados para um volume muito pequeno de elementos pode não seguir perfeitamente a complexidade assintótica que se aplica a volumes maiores.

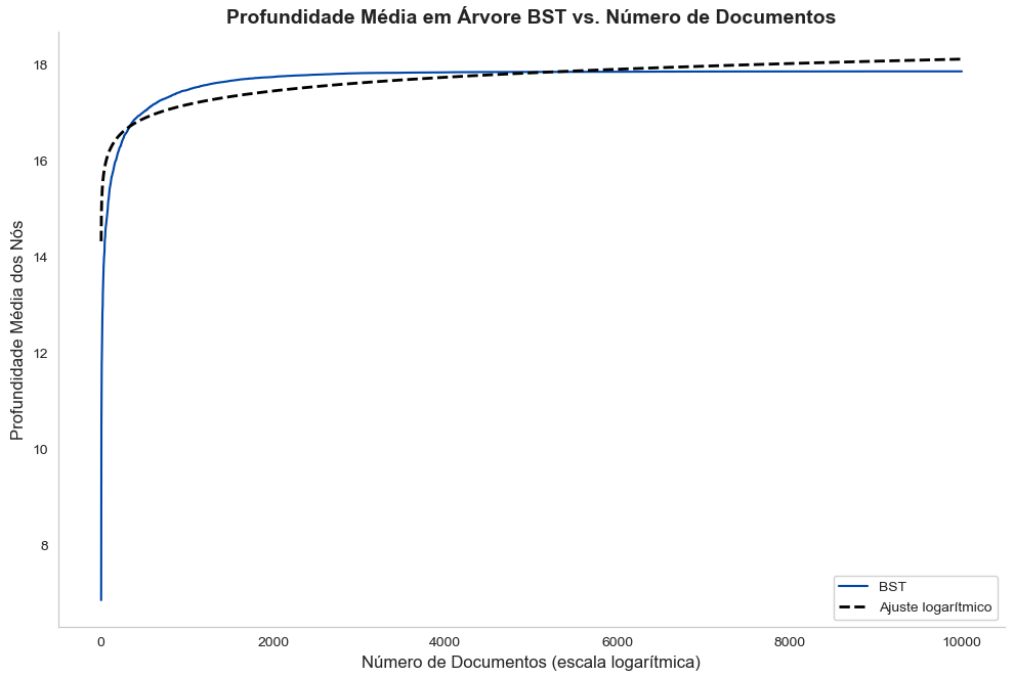
Conclusão:

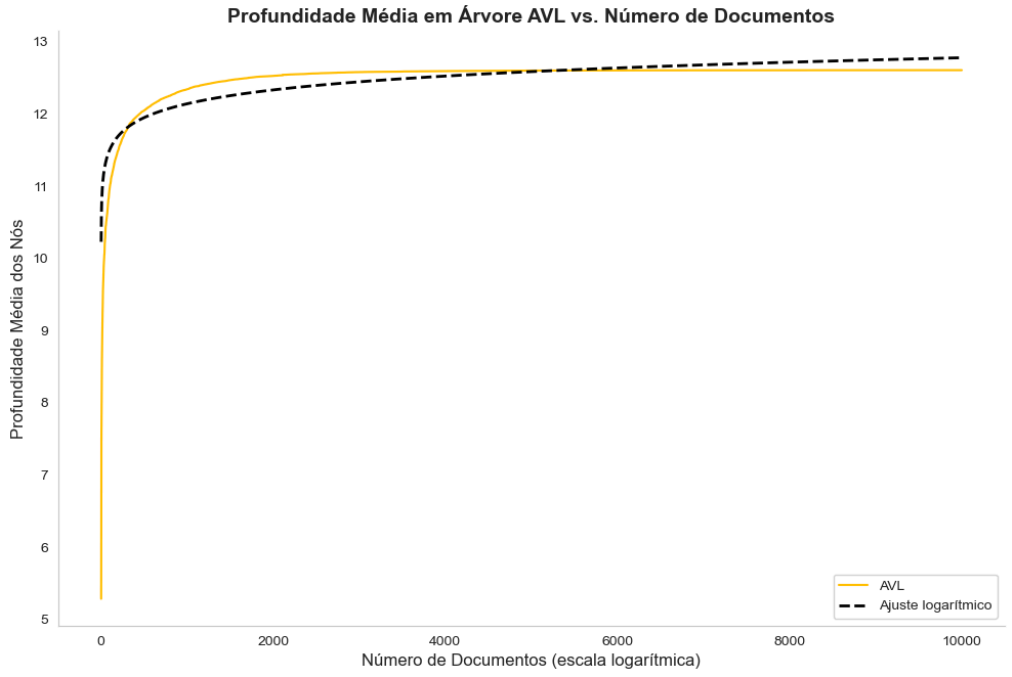
O texto analisa por que as árvores AVL e RBT são necessárias, mesmo quando gráficos mostram que a BST (Árvore Binária de Busca) também cresce de forma logarítmica com dados de texto aleatórios.

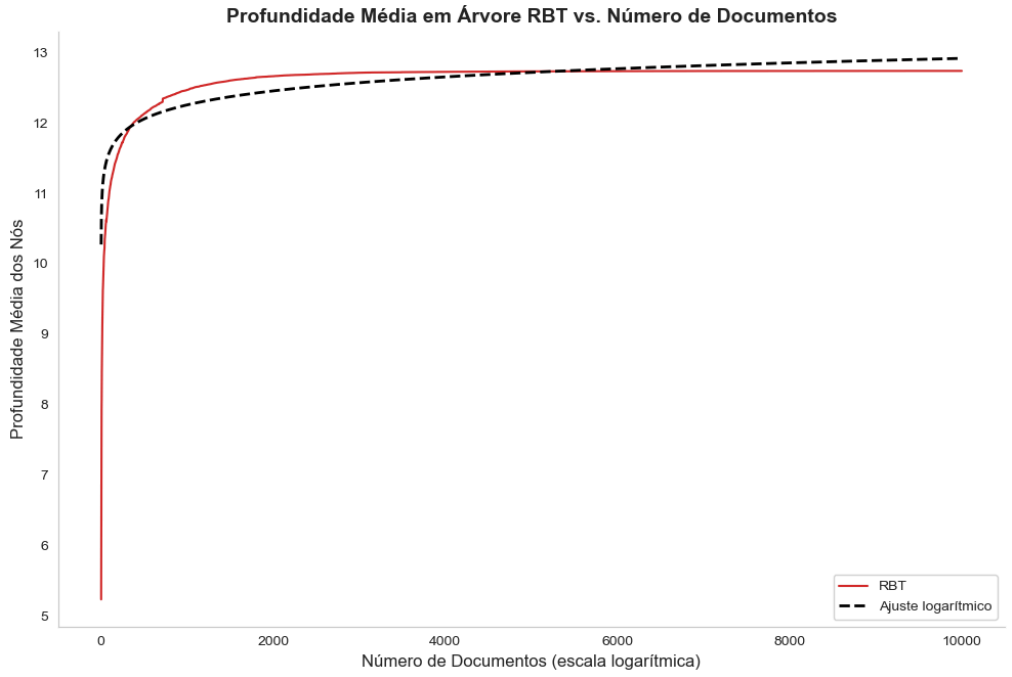
A explicação é que o bom desempenho da BST ocorre apenas no **caso médio**. O verdadeiro valor da AVL e da RBT está em sua garantia de desempenho no **pior caso**.

Para provar isso, o texto descreve o pior cenário para uma BST: a inserção de dados já ordenados (como palavras de um dicionário). Nesse caso, a profundidade da BST cresce de forma quase **linear**, o que a torna ineficiente. Em contraste, a AVL e a RBT, com suas rotações e recolorações, garantem que a altura sempre se mantenha **logarítmica**, independentemente da ordem dos dados de entrada.

Em resumo, o trabalho extra de balanceamento da AVL e RBT é a garantia de que a árvore nunca terá um desempenho ruim, algo que a BST não pode prometer.


Profundidade Média em Árvore BST vs. Número de Documentos
Profundidade Média dos Nós
Número de Documentos (escala logarítmica)
BST
Ajuste logarítmico


Profundidade Média em Árvore AVL vs. Número de Documentos
Profundidade Média dos Nós
Número de Documentos (escala logarítmica)
AVL
Ajuste logarítmico


Profundidade Média em Árvore RBT vs. Número de Documentos
Profundidade Média dos Nós
Número de Documentos (escala logarítmica)
RBT
Ajuste logarítmico

Número de documentos vs Máximo desbalanceamento

Das análises que fizemos, conseguimos observar a clara vantagem que as árvores autoanteriormente -balanceadas têm ante a BST, mas um ponto mencionado é o fato de que, embora a AVL e a RBT aplicam métodos para balanceamento, a preferida para aplicações práticas ainda é a RBT.

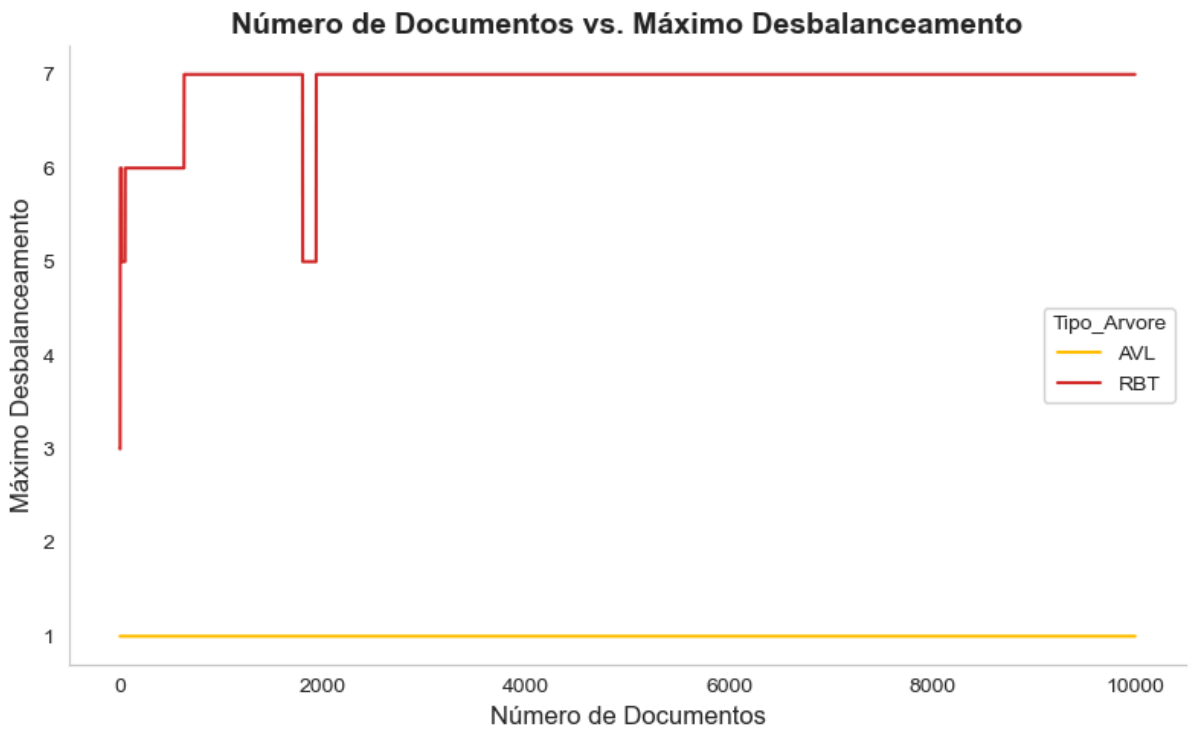
Isso se deve ao fato de que a AVL é muito mais rígida quanto ao critério de balanceamento, de forma que, para qualquer nó, a diferença entre as alturas da subárvore à esquerda e à direita é de, no máximo, 1. Como é esperado, a AVL garante buscas ainda mais rápidas do que as demais.

Entretanto, esse mesmo fato também acarreta num ponto negativo: Um critério mais rígido implica em mais rotações para manter o balanceamento, o que resulta em mais tempo de processamento para inserção de novos nós, especialmente os que deixam a árvore desbalanceada. Então cabe ao desenvolvedor escolher qual dessas estruturas usar para o tipo de problema que está enfrentando.

É de se esperar que para problemas em que não há muitas inserções, onde o foco seja a busca rápida e eficiente de dados, seja comum optar pela AVL, enquanto para modelos mais dinâmicos, onde estamos frequentemente inserindo e removendo dados, a RBT seja a escolha ideal.

Bem, deixando a escolha um pouco mais fácil, podemos comparar essas árvores nos critérios mencionados, “quantificando” o quão melhor em cada um a árvore mais vantajosa se saí. Começamos pelo primeiro cenário, onde as buscas devem ser eficientes, para isso, a árvore deve estar bem balanceada para que, independente do valor procurado, ele seja encontrado de forma eficiente e regular, ou seja, queremos que a árvore seja o mais balanceada possível.

Obviamente a AVL têm vantagem, mas precisamos ver quão atrás está a RBT:



Podemos observar que a RBT está de fato mais desbalanceada que a AVL, que têm maior desbalanceamento igual a 1 para todos os casos (como é de se esperar), mas a RBT alcançou máximos 7 de desbalanceamento, mesmo para muitos dados. Portanto, é normal esperar que, mesmo para os casos em que precisamos de buscas eficientes, a RBT não é uma má escolha, pois une um auto-balanceamento efetivo mas que, ao mesmo tempo, não é tão rígido à ponto de demandar tanto tempo de processamento com a AVL.

Estatísticas de busca

O "Total de Comparações na Busca" é calculado na função `exportEvolutionStatsToCSV`, localizada em `tree_utils.cpp`, para cada estágio de crescimento da árvore (correspondente a uma linha do CSV). Em cada etapa, realiza-se um número fixo de buscas amostrais (até 1000 nós), e, para cada busca, a função específica de busca da árvore (`searchFunc`) retorna o número de comparações realizadas. A soma dessas comparações individuais resulta no "Total de Comparações na Busca", que representa o custo acumulado de comparações ao executar todas as buscas da amostra naquele ponto da evolução da árvore.

BST

- **Comportamento:** A BST permanece em um nível de comparações *muito alto e com bastante flutuação* (ruído) ao longo de todo o espectro de documentos. Ela se mantém na faixa de 12.000 a 13.000 comparações totais.

- **Interpretação:** Este é o comportamento esperado para a BST não balanceada em um cenário onde a árvore degenera (como visto no gráfico de altura). Um número elevado de comparações totais para uma amostra de buscas indica que *cada busca individual é muito custosa*, percorrendo caminhos longos na árvore. As flutuações podem ser resultado da amostragem ou de pequenas variações na degeneração da árvore. A BST claramente exibe o pior desempenho de busca nessa amostragem.

2. **AVL**

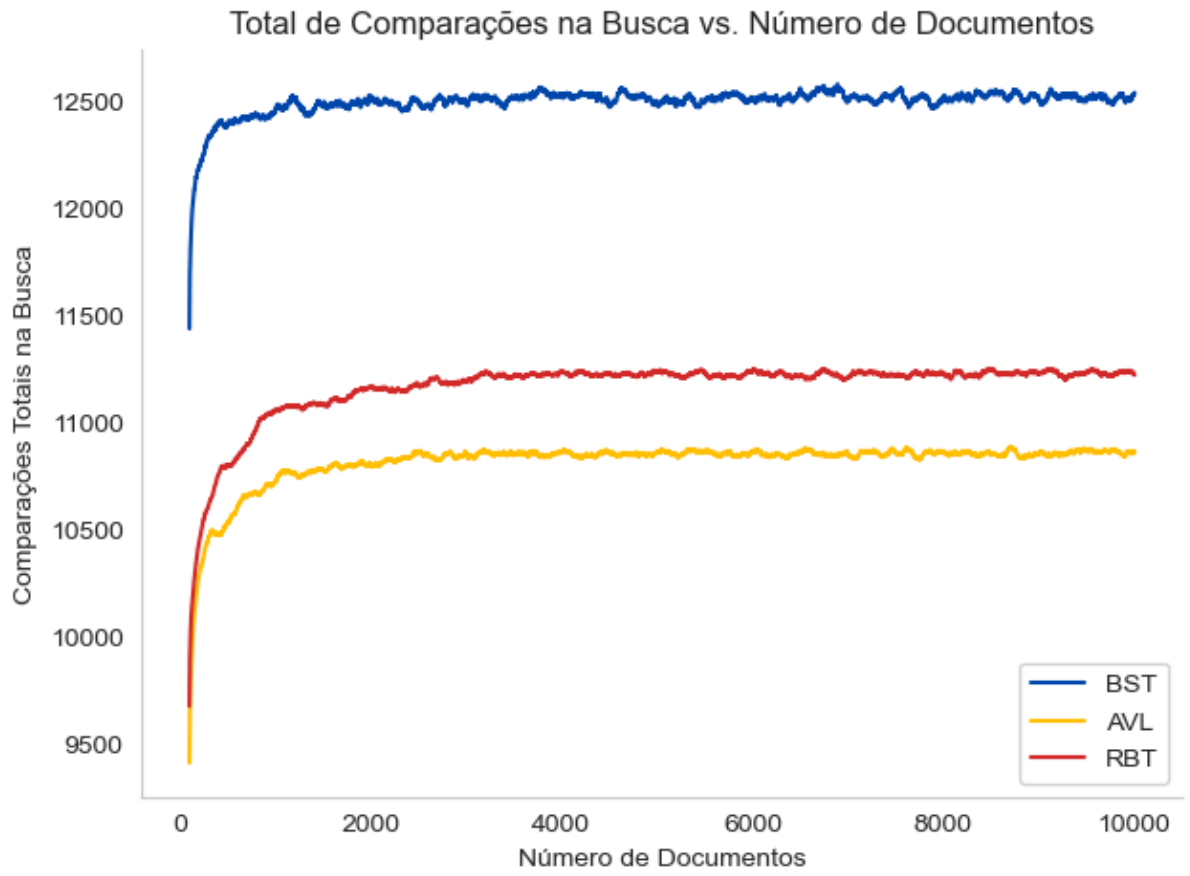
Comportamento: A linha amarela da AVL começa em um nível de comparações *significativamente menor e se mantém notavelmente estável e suave*, com pouquíssima flutuação, ao longo de todo o crescimento do número de documentos. Ela se mantém na faixa de 10.000 a 11.000 comparações totais.

Interpretação: Este comportamento demonstra a alta eficiência e previsibilidade da AVL nas operações de busca. Graças ao seu balanceamento estrito, a altura da árvore cresce de forma logarítmica (O(log n)), o que se traduz em um número consistentemente baixo de comparações por busca. O total de comparações para a amostra reflete diretamente essa eficiência logarítmica.

3**. RBT**

Comportamento: A linha vermelha da RBT também se inicia em um nível de comparação *baixo e se mantém muito estável*, com flutuações mínimas. Ela está ligeiramente acima da linha da AVL, na faixa de 11.000 a 12.000 comparações totais.

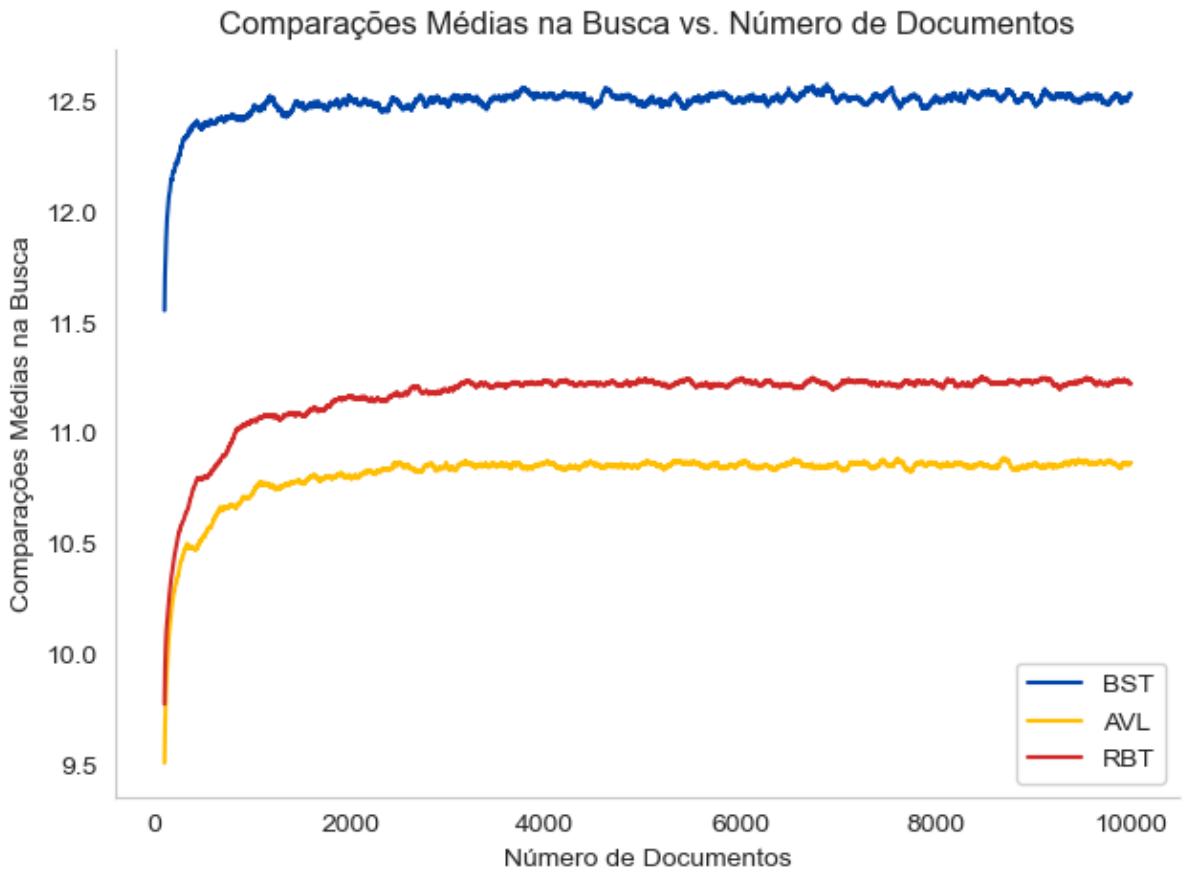
Interpretação: O fato de estar ligeiramente acima da AVL em número de comparações totais é coerente com a observação no gráfico de altura de que a RBT tende a ser marginalmente mais "profunda" que a AVL (devido às suas regras de balanceamento menos rigorosas), o que leva a um número ligeiramente maior de comparações por busca.



Conclusão:

A **BST** demonstra um desempenho de busca ineficiente e inconsistente, tornando-a inadequada para cenários onde a velocidade de busca é crítica em grandes volumes de dados.

Ambas as árvores **AVL** e **RBT** exibem um excelente e previsível desempenho de busca, com um número de comparações totais que escala de forma eficiente (logarítmica) com o volume de dados. A AVL é marginalmente mais eficiente em termos de comparações diretas, mas ambas são vastamente superiores à BST não balanceada.



Comparações médias por palavra

1. BST

Comportamento: A linha azul da BST mostra consistentemente o maior número de comparações médias por palavra, flutuando geralmente entre 12 e 13.

Interpretação: Este alto valor médio confirma que as operações de busca individuais na BST são significativamente mais custosas. Isso reflete diretamente a degeneração da árvore (já observada no gráfico de altura), onde os caminhos da raiz até os nós se tornam muito longos. Para uma árvore degenerada, a busca pode exigir a travessia de muitos nós, tornando-a ineficiente.

2. AVL

Comportamento: A linha da AVL demonstra o *menor número de comparações médias por palavra*, mantendo-se de forma muito estável em torno de 10.5 a 11.

Interpretação: Este comportamento ressalta a boa eficiência de busca da árvore AVL. Suas regras de balanceamento rigorosas garantem uma altura mínima

logarítmica (O(log n)), o que se traduz em um número previsível e consistentemente baixo de comparações necessárias para encontrar qualquer palavra, aproximando-se do desempenho teórico ideal para árvores binárias de busca.

3. RBT

Comportamento: A linha vermelha da RBT apresenta um número médio de comparações por palavra que é *ligeiramente maior que a AVL, mas consideravelmente menor que a BST*, variando tipicamente entre 11 e 11.5.

Interpretação: A RBT também oferece uma eficiência de busca muito alta, com uma altura logarítmica (O(log N)). O pequeno aumento nas comparações médias em relação à AVL é consistente com a RBT permitir uma altura marginalmente maior (devido às suas regras de balanceamento menos estritas), o que pode levar a um caminho de busca ligeiramente mais longo em média, mas ainda assim extremamente eficiente e previsível.

Conclusão:

Este gráfico é uma forte validação da eficácia das árvores auto-balanceadas (AVL e RBT) em manter a eficiência das operações de busca à medida que o volume de dados aumenta. Ele ilustra claramente que:

- A BST sofre de uma degradação de desempenho perceptível em buscas individuais, devido à sua incapacidade de manter o balanceamento.
- Tanto a AVL quanto a RBT oferecem um desempenho de busca consistentemente rápido e previsível, com um número médio de comparações que reflete sua complexidade logarítmica. A AVL demonstra uma ligeira vantagem em termos de menor número médio de comparações, enquanto a RBT oferece um desempenho muito próximo com um balanço talvez mais favorável para inserções/remoções em outros cenários.

Conclusões sobre eficiência de cada estrutura

A análise comparativa entre as estruturas de dados BST, AVL e RBT revelou distinções claras em sua eficiência para a aplicação de índice invertido. A **BST** demonstrou um desempenho de busca e inserção ineficiente e inconsistente, devido à sua incapacidade de manter o balanceamento, resultando em degeneração e comprometimento grave da eficiência das operações.

Em contraste, tanto a **AVL** quanto a **RBT** exibiram um excelente e previsível desempenho de busca e inserção. Ambas mantêm a altura da árvore sob controle logarítmico, garantindo operações rápidas mesmo com grandes volumes de dados. A AVL se destacou por manter a menor profundidade média, e por apresentar um

número marginalmente menor de comparações em buscas , enquanto a RBT oferece um balanço entre a eficiência estrutural e um custo de rebalanceamento potencialmente menor, mantendo um desempenho muito próximo ao da AVL.

Últimas considerações e conclusão

Em suma, este trabalho demonstrou que, embora a BST seja mais simples de implementar, suas limitações de balanceamento a tornam inadequada para lidar com grandes volumes de dados dinâmicos. As árvores **AVL e Rubro-Negra (RBT)**, com seus mecanismos de autoajuste, são soluções robustas que garantem a eficiência e a escalabilidade das operações. A escolha entre AVL e RBT dependerá do cenário específico, considerando o **balanço** entre a altura ligeiramente menor da AVL (mais rápida em buscas) e a menor frequência de rebalanceamentos da RBT (potencialmente mais rápida em inserções/remoções intensas).