Relatório de Estrutura de Dados

Índice Invertido e Análise Comparativa de Estruturas de Dados

Junho de 2025

Adriel Dias 241708013

Gabriel Schuenker 241708060

Matheus Mendes 241708027

Nícholas Farrel 241708006

Vinícius Tavares 241708021

Sumário

1 Introdução

Este relatório tem como objetivo apresentar, descrever e analisar os resultados obtidos no desenvolvimento do projeto da Avaliação 2 (A2) da disciplina de Estrutura de Dados.

O projeto consistiu na implementação de um índice invertido para um conjunto de documentos de texto, utilizando três diferentes estruturas de dados: Árvore Binária de Busca (BST), Árvore AVL e Árvore Rubro-Negra (RBT), todas desenvolvidas na linguagem `C++`.

A partir dessas implementações, foram realizadas análises comparativas de desempenho nas operações de inserção e busca, considerando diferentes volumes de dados. Para a análise de desempenho das estruturas nas operações de inserção e busca, foram utilizados datasets reais formados por mais de dez mil documentos de texto, o que permitiu simular um cenário prático de indexação e acesso de dados.

2 Árvores

Em Estruturas de Dados, uma árvore é um tipo de dado abstrato hierárquico que consiste em nós conectados por arestas, onde um dos nós é designado como a raiz e os demais são organizados em subárvores. Cada nó pode ter zero ou mais filhos, mas apenas um pai, com exceção da raiz, que não possui pai.

Em termos computacionais, uma árvore é uma estrutura recursiva, pois cada subárvore é, por si só, uma árvore. Essa característica torna as árvores extremamente úteis para resolver problemas que envolvem organização hierárquica, como arquivos em um sistema de diretórios, expressões matemáticas e, no caso desse projeto, índices invertidos para busca de palavras

No contexto deste trabalho, utilizamos árvores binárias de busca (BSTs) e duas variações balanceadas: Árvore AVL e Árvore Rubro-Negra. Essas estruturas são ideais para indexar palavras de um acervo de documentos, associando cada termo a uma lista de documentos em que ele ocorre.

3 Binary Search Tree

3.1 Definição

No escopo desse projeto, o tipo mais simples de árvore que apresentaremos é a **Binary Search Tree**, ou, em tradução livre, árvore binária de busca.

Uma árvore binária de busca (Binary Search Tree, ou BST) é uma estrutura de dados hierárquica do tipo árvore binária, ou seja, cada nó possui no máximo dois filhos: um filho à esquerda e um à direita. A principal característica da BST é que ela organiza os elementos inseridos de forma ordenada com base em uma relação de comparação (por exemplo, ordem alfabética ou valor numérico).

Formalmente, para cada nó N da árvore, todos os elementos armazenados na subárvore à esquerda de N são menores que N, e todos os elementos da subárvore à direita são maiores. Essa propriedade deve ser mantida recursivamente para todos os nós da árvore.

Como consequência dessa organização, a Binary Search Tree permite a realização eficiente das operações de busca, inserção e remoção com complexidade média de $O(\log n)$ — desde que a árvore se mantenha aproximadamente balanceada. No entanto, no pior

caso (como ao inserir elementos já ordenados), a estrutura pode se degenerar em uma lista encadeada, elevando a complexidade para $O(n)$.

Do ponto de vista semântico, ao se considerar uma ordem entre os elementos, a subárvore esquerda de um nó armazena elementos que estão atrás do nó nessa ordenação, enquanto a subárvore direita contém os que estão à frente. Essa estrutura é adequada para representar dados ordenáveis e serve como base para árvores balanceadas, como **AVL** e **Red Black Tree**, que introduzem mecanismos de balanceamento para evitar a degeneração da árvore.

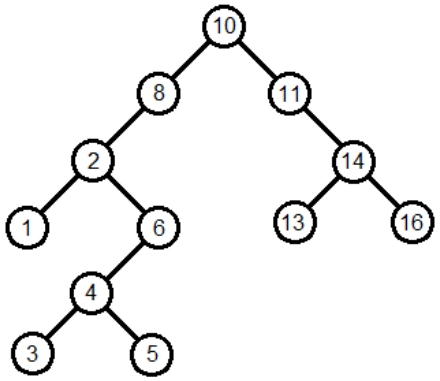

Figura 1: Exemplo de uma árvore binária de busca.
Nesta figura, observa-se que cada nó possui no máximo dois filhos, sendo que todos os elementos à esquerda são menores que o nó e todos à direita são maiores, ilustrando a propriedade fundamental da BST.

3.2 Implementação

A implementação da Binary Search Tree (BST) foi feita em `C++` sem uso de orientação a objetos, seguindo a estrutura funcional proposta pelo projeto. A BST armazena as palavras indexadas e associa a cada palavra uma lista de IDs dos documentos em que ela aparece.

Para a implementação desse Tipo Abstrato de Dado, criamos inicialmente a estrutura `Node`, que representa cada elemento (ou nó) da árvore. Cada `Node` armazena:

- A palavra correspondente a esse nó.
- Os índices dos documentos que essa palavra aparece.
- Ponteiros para o nó pai, o filho à esquerda e o filho à direita.

Como a estrutura `Node` é compartilhada entre as diferentes implementações de árvores, ela também inclui outros atributos específicos, que serão discutidos quando falarmos da implementação das outras árvores.

A estrutura `BinaryTree`, por sua vez, representa a árvore como um todo.Trata-se de um Tipo Abstrato que armazena `root`, um ponteiro que aponta para o nó raiz da árvore, caso ela não esteja vazia. Abaixo explica-se como foi implementado as funções `insert` e `search` na `BinaryTree`.

A operação de **inserção** em uma árvore binária de busca consiste em posicionar um novo elemento na estrutura de acordo com a ordem definida pela relação de comparação. (No caso do nosso projeto, ordem alfabética). O algoritmo deve percorrer a árvore a partir da raiz, comparando o valor a ser inserido com os nós visitados:

- Se o valor for menor, a busca continua pela subárvore esquerda.

- Se for maior, segue-se pela subárvore direita.
- Se o valor já estiver armazenado na árvore, no nosso caso, apenas atualizamos a lista de documentos associada à palavra.

A operação termina ao encontrar um ponteiro nulo, ou seja, uma posição onde o novo nó pode ser inserido como filho de um nó já existente. No pior caso, quando a árvore está completamente desbalanceada, essa operação pode exigir $O(n)$ comparações. Entretanto, em casos favoráveis (como árvores balanceadas), o custo tende a $O(\log n)$.

Na implementação desenvolvida, a função de inserção percorre a árvore partindo da raiz e realiza as comparações até encontrar o local apropriado para inserção. Se a palavra já estiver presente no nó visitado, verifica-se se o identificador do documento já foi armazenado; caso não tenha sido, ele é adicionado à lista daquele nó.

Se a palavra ainda não estiver presente, um novo nó é criado dinamicamente e posicionado na árvore respeitando a ordenação alfabética.

A operação `search()` em uma árvore binária de busca visa localizar um elemento (neste caso, uma palavra) respeitando a ordenação imposta pela estrutura. O algoritmo é conceitualmente semelhante ao da inserção: parte-se da raiz e realiza-se comparações sucessivas para decidir se o percurso deve continuar pela subárvore esquerda ou pela subárvore direita.

- Se a palavra buscada for menor que a palavra no nó atual, a busca continua pela esquerda.
- Se for maior, continua pela direita.
- Se for igual, o nó correspondente é retornado e os dados associados a ele (como os identificadores dos documentos) podem ser consultados.
- Se o ponteiro de nó se tornar nulo durante o percurso, significa que a palavra não está presente na árvore, e o algoritmo retorna 0, indicando que a busca não obteve sucesso.

Essa abordagem, ao explorar a propriedade de ordenação da árvore, permite buscar elementos com complexidade média de $O(\log n)$ em árvores balanceadas, e até $O(n)$ no pior caso. Na implementação realizada, a função de busca recebe a raiz da árvore e a palavra desejada. Em seguida, percorre a estrutura comparando a palavra atual do nó com a palavra alvo. Se encontrar o nó correspondente, a função retorna um objeto contendo um indicador de sucesso da busca (0 em caso de fracasso, 1 em caso de sucesso) e a lista de documentos que a palavra se encontra. Esse objeto possui também outros atributos utilizados para análise, como o número de comparações feitas e o tempo de execução.

Com isso, concluímos a apresentação da implementação da árvore binária de busca, cuja simplicidade estrutural permite uma implementação direta e operações eficientes em muitos cenários. No entanto, a ausência de um mecanismo de balanceamento pode comprometer seu desempenho à medida que a ordem de inserção pode levar à formação de árvores degeneradas em casos extremos, como no exemplo a seguir:

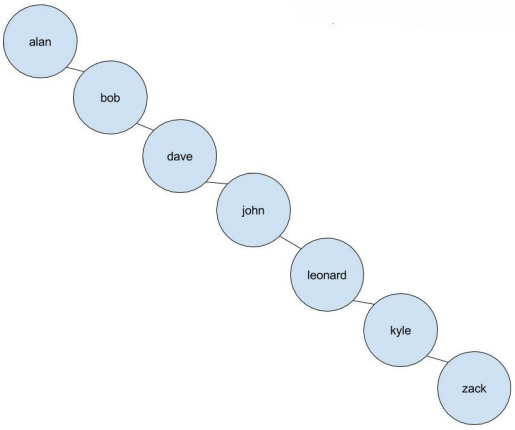

Figura 2: Exemplo de degeneração de uma árvore binária de busca.

A Figura 2 ilustra um caso clássico de degeneração de uma árvore binária de busca: quando elementos são inseridos em ordem crescente (ou decrescente), a árvore perde suas propriedades de balanceamento e passa a se comportar como uma lista encadeada. Isso resulta em profundidade linear e complexidade de tempo $O(n)$ para operações como busca e inserção, anulando os ganhos esperados com a estrutura em árvore.

Essa limitação motiva o estudo e a utilização de estruturas mais robustas, como a árvore AVL, que discutiremos a seguir.

4 AVL

4.1 Definição

A **árvore AVL** é uma variação da árvore binária de busca que introduz um mecanismo de balanceamento automático. Foi proposta por Adelson-Velsky e Landis (daí a sigla AVL) com o objetivo de evitar a degeneração da árvore em uma lista encadeada — problema comum na BST quando os dados são inseridos em ordem.

O princípio básico da AVL é garantir que, para todo nó da árvore, a diferença entre as alturas das subárvores esquerda e direita (o *fator de balanceamento*) seja no máximo 1 em módulo. Caso essa propriedade seja violada após uma inserção (ou remoção), a árvore realiza rotações locais para restaurar o equilíbrio.

Quando o fator de balanceamento sai do intervalo permitido $[-1, 1]$, a árvore precisa ser reequilibrada por meio dessas rotações - operações que reorganizam os nós para manter a propriedade AVL.

A imagem a seguir ilustra um exemplo de rotação utilizada para corrigir um desequilíbrio na árvore.

Na Figura 3, observamos um caso típico de desequilíbrio em uma árvore AVL, onde os nós estão organizados em forma de caminho linear crescente: A é a raiz, B é filho direito de A, e C é filho direito de B. Essa configuração gera um fator de balanceamento fora do intervalo permitido para o nó A, caracterizando um desbalanceamento à direita.

Para corrigir isso, aplicamos uma rotação simples à esquerda em A, que promove B à nova raiz da subárvore, faz de A o filho esquerdo de B, e mantém C como filho direito de B. Essa operação reduz a altura da subárvore e restaura o balanceamento local.

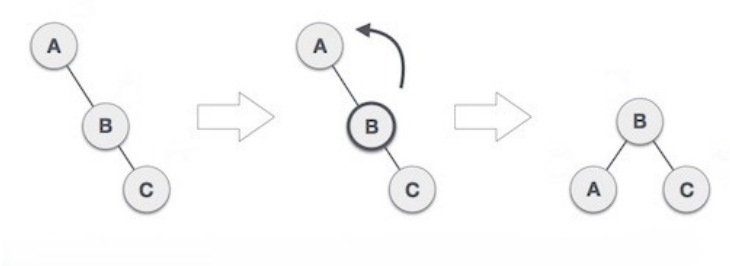

Figura 3: Exemplo de rotação para reequilíbrio em uma árvore AVL.

Assim, a rotação reorganiza os nós para garantir que a propriedade AVL seja satisfeita, evitando a degeneração da árvore em um caminho linear e mantendo as operações eficientes.

4.2 Implementação

Para implementar a árvore AVL, reutilizamos a estrutura básica de nós e da árvore usada na BST, acrescentando o atributo `height` em cada nó para controlar a altura das subárvores e permitir o balanceamento.

A função `height` retorna a altura de um nó, tratando o caso de ponteiros nulos como altura 0. O fator de balanceamento de um nó é calculado pela diferença entre as alturas das subárvores esquerda e direita, usando a função `balanceFactor`. Esse valor deve estar sempre entre -1 e 1 para garantir o balanceamento.

Rotações e inserção

Como comentado anteriormente, para corrigir desequilibrios após inserções, devemos realizar rotações em subárvores da **AVL**. Implementamos no nosso projeto os quatro tipos de rotação usados para restaurar o balanceamento local da árvore:

- **Rotação simples à direita (caso esquerda-esquerda):** ocorre quando um nó tem fator de balanceamento maior que 1 e o nó inserido está na subárvore esquerda do filho esquerdo. Nesse caso, a subárvore esquerda está "pesada", e a rotação à direita reposiciona os nós para redistribuir as alturas.

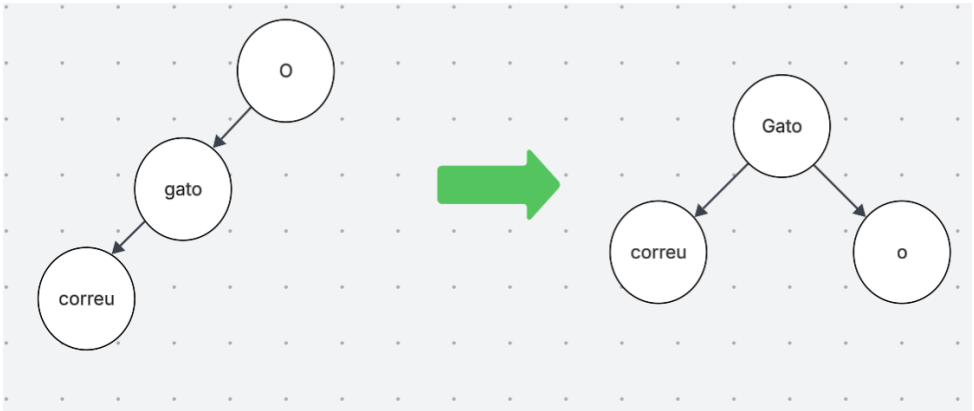

Figura 4: Exemplo de subárvore após aplicar `rotateRight()` no `Node` correspondente a "o".

- **Rotação simples à esquerda :** ocorre quando o fator de balanceamento é menor que -1 e o nó inserido está na subárvore direita do filho direito. A rotação à esquerda compensa o crescimento da subárvore direita.

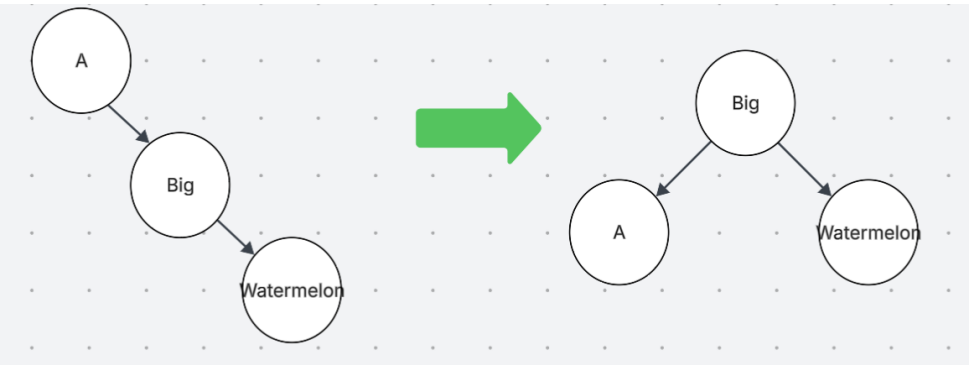

Figura 5: Exemplo de subárvore após aplicar `rotateLeft()` no `Node` correspondente à palavra `A` para corrigir o desequilíbrio direita-direita causado pela inserção em ordem.

- **Rotação esquerda-direita:** ocorre quando o fator de balanceamento é maior que 1 e o nó inserido está na subárvore direita do filho esquerdo. Nesse caso, fazemos primeiro uma rotação à esquerda no filho esquerdo, seguida por uma rotação à direita no próprio nó.

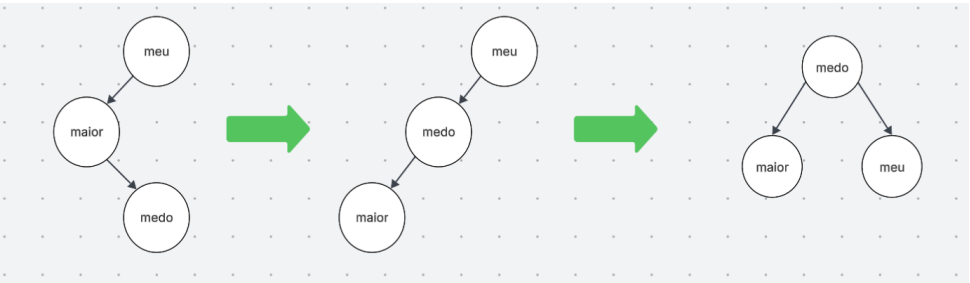

Figura 6: Exemplo de subárvore antes e depois de se corrigir um desequlíbrio esquerda-direita com as funções de rotação `rotateLeft()` e `rotateRight()`

- **Rotação dupla direita-esquerda (caso direita-esquerda):** ocorre quando o fator de balanceamento é menor que -1 e o nó inserido está na subárvore esquerda do filho direito. Primeiro, aplicamos uma rotação simples à direita no filho direito para transformar o caso em direita-direita, e depois uma rotação simples à esquerda no nó atual para restaurar o equilíbrio.

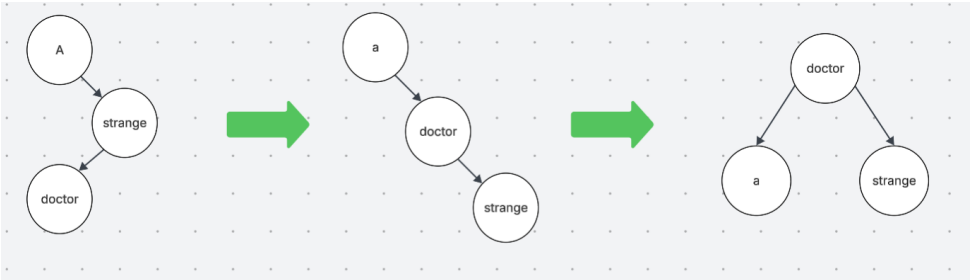

Figura 7: Exemplo de subárvore após aplicar `rotateRight` no nó `strange`, seguido de `rotateLeft` no nó `A`. Esse caso ocorre quando a inserção foi feita na subárvore esquerda do filho direito, caracterizando um desequilíbrio do tipo direita-esquerda.

No código, a função `insert()` detecta automaticamente desequilibrios dos tipos citados acima e aplica as rotações correspondentes, mantendo a árvore balanceada e as operações eficientes.

Repare que a função `search()` segue a mesma lógica para todas as estruturas de árvore binária utilizadas — **BST**, **AVL** e **RBT**. O processo de busca percorre a árvore de forma determinística, comparando a palavra procurada com as palavras armazenadas nos nós e seguindo para a subárvore esquerda ou direita, conforme o caso. A principal diferença entre as estruturas está no balanceamento, o que pode afetar a profundidade da árvore e, consequentemente, o desempenho da busca, mas não altera a lógica do algoritmo em si.

Uma faca de dois gumes

O balanceamento automático da árvore AVL é uma vantagem importante para garantir operações rápidas de busca, com complexidade logarítmica. No entanto, essa mesma característica pode se tornar uma desvantagem em cenários com muitas inserções e remoções, pois as rotações frequentes para manter o equilíbrio podem gerar um custo computacional considerável.

Assim, embora eficiente para buscas, a AVL pode não ser ideal quando a estrutura precisa ser atualizada constantemente. Essa limitação abre espaço para outras abordagens de árvores balanceadas, como as Red-Black trees, que discutiremos a seguir.

5 Red Black Tree

5.1 Definição

A Red-Black tree é uma árvore binária de busca balanceada que utiliza propriedades de cor para garantir o equilíbrio aproximado da árvore. Diferentemente da AVL, a RBT permite um balanceamento menos rígido, o que reduz o número de rotações necessárias durante inserções e exclusões, tornando-a mais eficiente em cenários de muitas atualizações.

A RBT obedece às seguintes propriedades essenciais:

- Cada nó é colorido de vermelho ou preto;
- A raiz da árvore é sempre preta;
- Todas as folhas externas* são pretas; (*chamadas também de NIL)

- Um nó vermelho não pode ter um filho vermelho (não há dois vermelhos consecutivos);
- Para cada nó, todos os caminhos simples da raiz até as folhas descendentes contêm o mesmo número de nós pretos (conhecido como *black-height*);

Essas propriedades garantem que nenhum caminho da raiz até uma folha seja mais do que o dobro do comprimento de qualquer outro, mantendo a árvore aproximadamente balanceada.

Além disso, temos também nós especiais chamados de ***NIL***, que representam os ponteiros nulos da árvore. Diferentemente das outras estruturas vistas anteriormente, na Red-Black Tree esses nós são tratados como nós reais (embora não armazenem dados) e são sempre considerados de cor preta. A presença explícita dos nós *NIL* é fundamental para a definição formal da árvore, pois eles são levados em conta na contagem de nós pretos ao longo dos caminhos da raiz até as folhas, conforme exigido pelas propriedades da RBT.

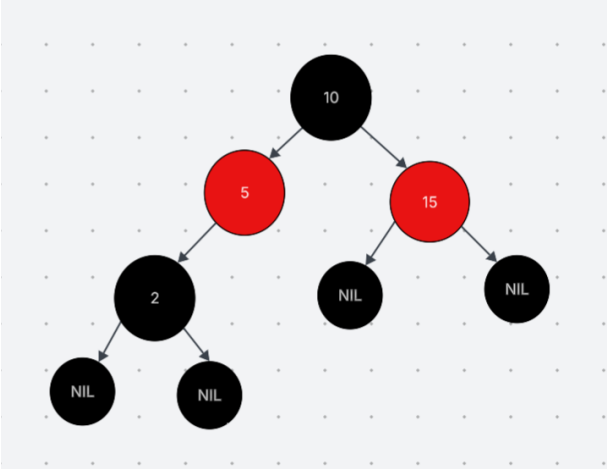

Figura 8: Exemplo de uma árvore Rubro-Negra. A coloração dos nós segue as propriedades da estrutura, e os nós *NIL* (representados como círculos pretos menores) completam as folhas da árvore.

5.2 Implementação

A Red-Black Tree reutiliza a estrutura de árvore binária já empregada nas versões anteriores, com a adição de um campo `isRed` em cada nó para representar sua cor. Além disso, utiliza um nó especial `NIL`, compartilhado por todas as folhas, que representa os terminais da árvore.

A inserção segue a lógica da BST, mas cada novo nó é inicialmente vermelho. Após a inserção, a função `fixInsertion()` é chamada para restaurar as propriedades da RBT, aplicando rotações (`leftRotate()` ou `rightRotate()`) e ajustes de cor, conforme necessário. É imediato saber que para precisar de uma recoloração, o pai do novo `Node` inserido é vermelho, assim, a função atua em três casos específicos, listados abaixo:

- **Caso 1:** Tio vermelho → Recoloração do pai, do tio e do avô. Nenhuma rotação é necessária.

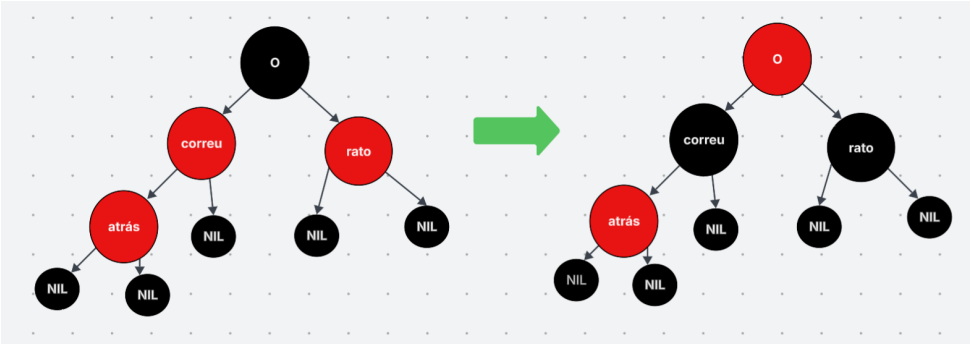

Figura 9: Recoloração após inserir a palavra “atrás”. Note que essa reestruturação só é válida se o avô não for a raiz da árvore.

- **Caso 2:** Tio preto e o nó atual é filho à esquerda → Rotação no avô e troca de cores entre pai e avô.

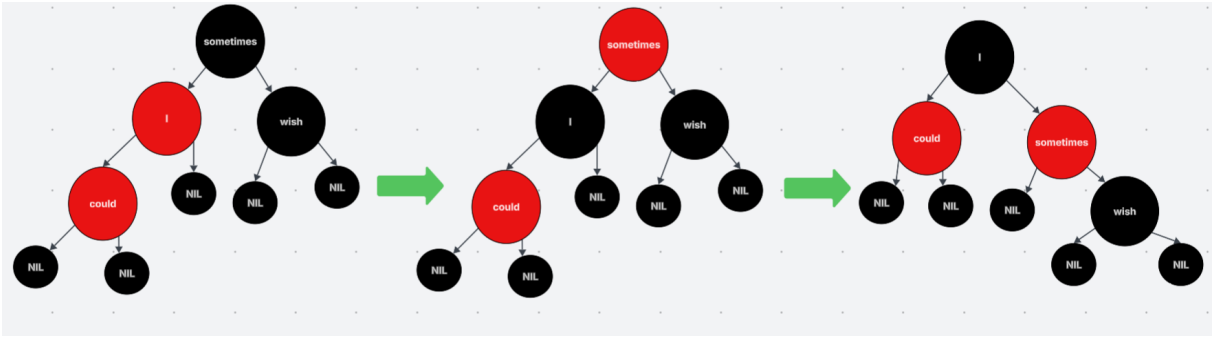

Figura 10: Recoloração e rotação à direita após inserir a palavra “could”.

- **Caso 3:** Tio preto e o nó atual é filho à direita → Rotação no pai para transformar em Caso 2.

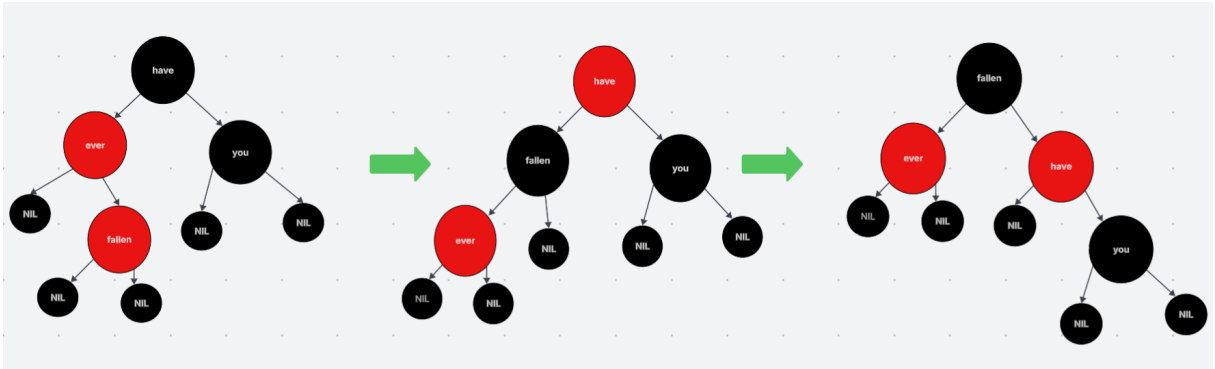

Figura 11: Rotação no pai (“ever”) e preparação para o Caso 2, após a inserção da palavra “fallen”.

Esse mecanismo garante o balanceamento aproximado da árvore com menos rotações que a AVL, mantendo o desempenho eficiente mesmo em cenários de alta quantidade de inserções.

Visualização da Estrutura das Árvores

Para auxiliar na verificação do conteúdo e do balanceamento das árvores, foram implementadas as funções `printTree()` e `printIndex()`.

A função `printTree()` imprime a estrutura da árvore em forma hierárquica no terminal, facilitando a identificação da posição de cada nó, das subárvores esquerda e direita, e a visualização da organização geral da árvore após inserções.

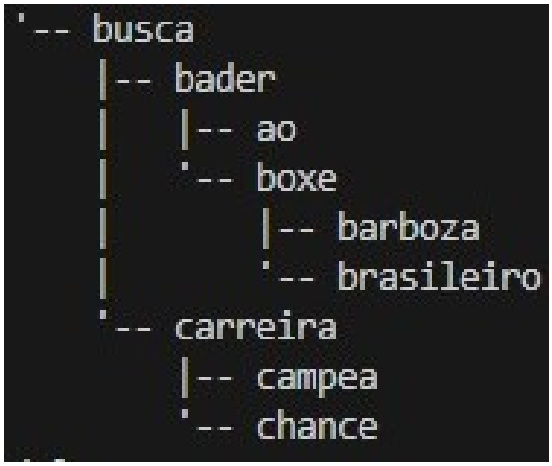

Figura 1: Exemplo de saída da função `printTree()`.
A imagem mostra a estrutura hierárquica da árvore binária de busca após algumas inserções. Essa representação facilita a visualização da profundidade dos nós, a separação entre subárvores esquerda e direita, e o equilíbrio (ou desequilíbrio) da árvore.

Já a função `printIndex()` percorre a árvore em ordem alfabética e exibe as palavras armazenadas, acompanhadas dos identificadores dos documentos em que cada uma aparece. Essa função permite visualizar a indexação atual da árvore e verificar se os dados foram corretamente associados às palavras.

Essas ferramentas são especialmente úteis durante o desenvolvimento, depuração e validação do comportamento das estruturas implementadas.

6 Avaliação Experimental

Esta seção apresenta os resultados experimentais obtidos a partir da comparação entre as estruturas BST, AVL e RBT.

Sobre a base de dados

Para os experimentos, utilizamos uma base composta por mais de 10.000 documentos de texto, fornecida pelo professor da disciplina. Os arquivos estão em formato `.txt` e contêm conteúdo textual em língua portuguesa. Cada documento foi processado para extrair as palavras relevantes, que foram utilizadas na construção do índice invertido.

Durante os testes, as palavras extraídas foram inseridas nas estruturas de dados (BST, AVL e RBT), juntamente com os identificadores dos documentos em que ocorrem. Essa base representa um cenário realista de uso intensivo de inserções e buscas em um grande volume de dados textuais, possibilitando uma avaliação prática da performance de cada estrutura em termos de tempo e número de comparações realizadas.

Foram realizadas medições de desempenho com base em operações de inserção e busca, variando o volume de dados até os 10000 documentos. Os gráficos a seguir ilustram o comportamento de cada estrutura nesses diferentes cenários, permitindo uma análise comparativa quanto à eficiência e escalabilidade de cada árvore.

6.1 Análise da Altura das Árvores

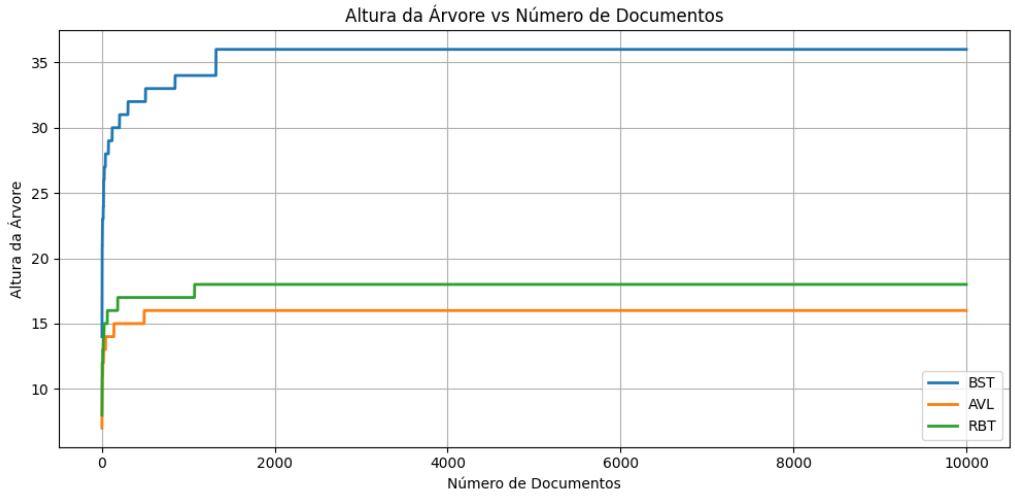

Figura 12: Altura das árvores em função do número de documentos

O gráfico acima ilustra o comportamento das três estruturas em relação à altura máxima conforme o número de documentos indexados cresce.

Uma característica notável é que a altura das três árvores apresenta um crescimento em formato de "escada" à medida que os dados são inseridos. Esse comportamento pode ser explicado por dois fatores principais:

- **Similaridade entre as palavras dos documentos:** Como muitos textos compartilham palavras comuns, a inserção de novos documentos nem sempre resulta na adição de novas palavras ao índice. Nesses casos, apenas listas existentes são atualizadas, sem afetar a estrutura da árvore nem sua altura.

- **Distribuição das palavras e estrutura da árvore:** Mesmo quando novas palavras são inseridas, elas têm alta probabilidade de ser posicionadas em ramos já existentes, preenchendo níveis inferiores intermediários sem provocar aumento de altura. No caso da BST, isso ocorre porque o crescimento da árvore depende diretamente da ordem das inserções. Já em árvores balanceadas (AVL e RBT), mecanismos internos mantêm a altura sob controle, o que torna aumentos súbitos menos frequentes.

De forma mais específica, observamos que a **BST** apresentou o maior crescimento, atingindo altura próxima a 35 com a inserção de todos os documentos. Isso ocorre pela ausência de balanceamento, o que pode resultar em árvores significativamente desbalanceadas. Já a **AVL** manteve a menor altura entre as três estruturas. Seu balanceamento rigoroso assegura árvores mais simétricas e eficientes para buscas. A **RBT**, por sua vez, apresentou altura também menor do que a BST, mas ligeiramente maior do que a altura da AVL, refletindo sua proposta de ter um balanceamento menos rígido.

É importante notar que, embora a BST apresente altura maior que as outras estruturas, ela está longe do cenário degenerado extremo previsto pela teoria, no qual a árvore se comportaria como uma lista encadeada. Isso sugere que os dados possuem uma distribuição que evita o pior caso, resultando em uma árvore com estrutura bem mais eficiente do que uma lista ordenada, por exemplo.

6.2 Análise do Menor Caminho

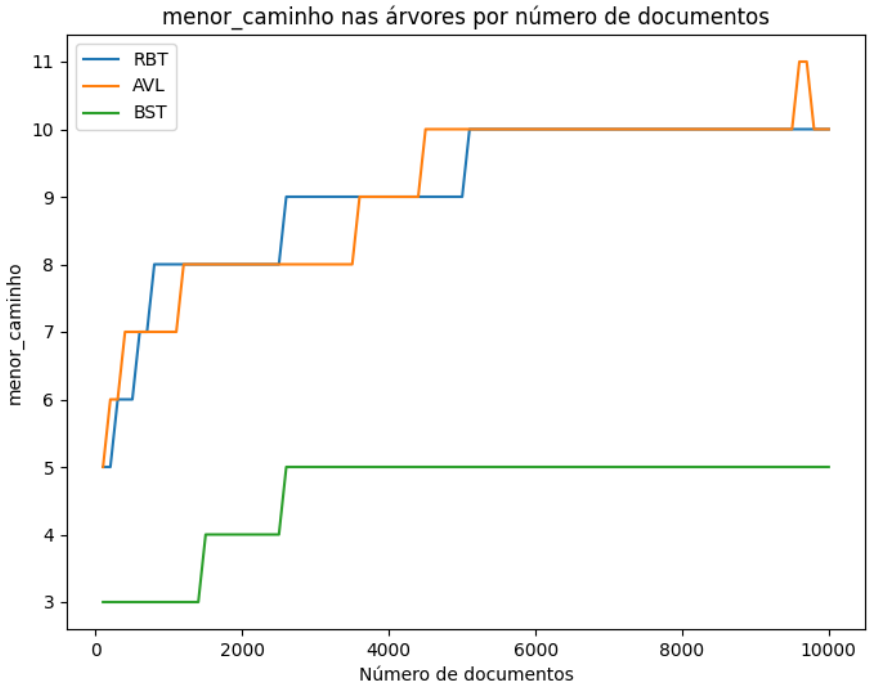

Figura 13: Menor caminho

O gráfico acima ilustra a evolução do menor caminho (a menor distância entre a raiz e qualquer folha) em cada uma das estruturas de árvore à medida que o número de documentos indexados aumenta até 10.000. Observa-se que a BST mantém consistentemente o menor camimho mínimo dentre as 3 árvores, com a distância se estabilizando em 6 rapidamente no começo, e se mantém ate o final, isso se deve devido a ausencia de mecanismos de balanceamento que as outras árvores possuem, permitindo que a árvore cresça de forma assimetrica e desbalanceada, acumulando folhas mais folhas proxima da raiz, especialmente quando as inserções seguem uma ordem que favorece a criação de ramos curtos.

Por outro lado, as árvores (AVL e RBT) apresentam um crescimento mais acentuado do menor caminho, com a AVL exibindo valores ligeiramente superiores aos da RBT em curtos intervalos, não sendo mais notavel essa difeernça quando o numero de arquivos fica em torno de 1000 arqruivos lidos, que ambas as arvores começam a apresentar o mesmo menor caminho.

Em resumo, o menor caminho é um indicador de assimetria da árvore (quando comparado com a altura que é o maior caminho) ja que, se a diferença entre o menor caminho e o maior caminho for exagerada, indica que a árvore esta com seu desempenho afetado, o que é comum de se notar na BST, enquanto as árvores AVL e RBT, por forçar o balanceamento por meio de rotações, apresentam uma distribuição mais uniforme, resultando em um menor caminho um pouco maior, porém dentro de limites controlados.

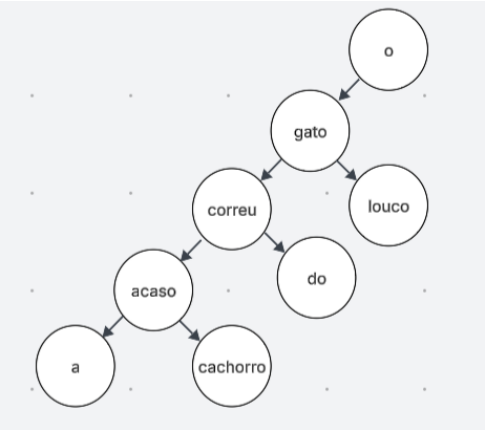

Figura 14: Caso típico de um caminho rígido
Observe ao lado um caso que possívelmente explicaria um número constante para o menor caminho: a palavra “a” é bastante comum, e é pouco provavel que palavras que venham antes dela na ordem lexicográfica (símbolos e números, por exemplo) apareçam nos textos.

6.3 Análise do Tempo de Indexação

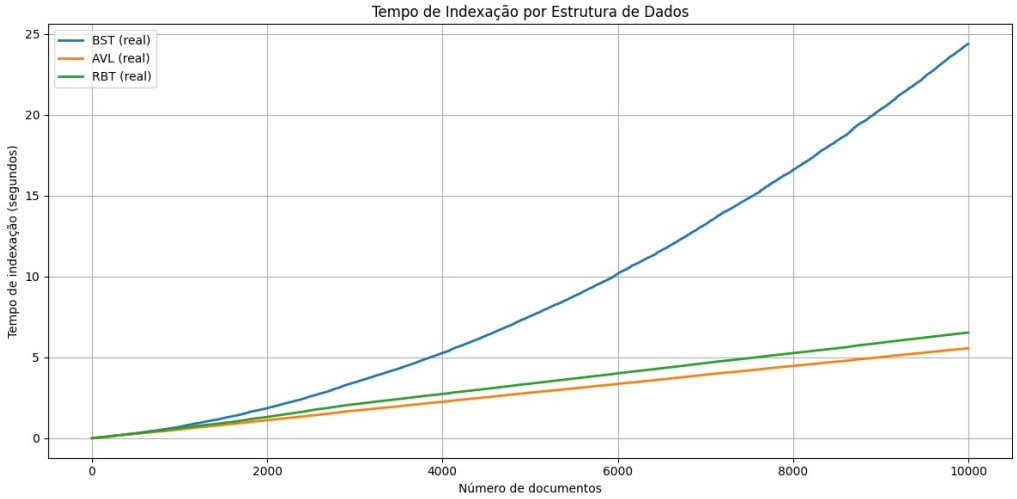

Figura 15: Tempo de Indexação

O gráfico acima apresenta o tempo total de indexação (inserção de todas as palavras de um documento) dos arquivos para cada uma das estruturas analisadas. Observa-se que a **AVL** obteve o menor tempo de inserção, seguida de perto pela Rubro-Negra (RBT), enquanto a **BST** apresentou um desempenho consideravelmente inferior em comparação às duas anteriores.

Esse resultado parece refletir o fato de que, por não aplicar nenhum tipo de balanceamento, a BST pode crescer de maneira desordenada, aumentando o custo das inserções. Já as árvores AVL e RBT, por manterem uma estrutura mais equilibrada, conseguem evitar crescimentos excessivos na altura, o que tende a favorecer o tempo de construção do índice, sendo muito provavelmente essa folha de altura 6 encontrada na BST uma palavra que se eencaixou,

Ainda assim, é importante destacar que as diferenças entre AVL e RBT foram pequenas nesse experimento, e a vantagem observada pode depender de fatores como a ordem dos dados ou o tipo de operação mais frequente em uma aplicação prática.

6.4 Análise do número de comparações em inserção

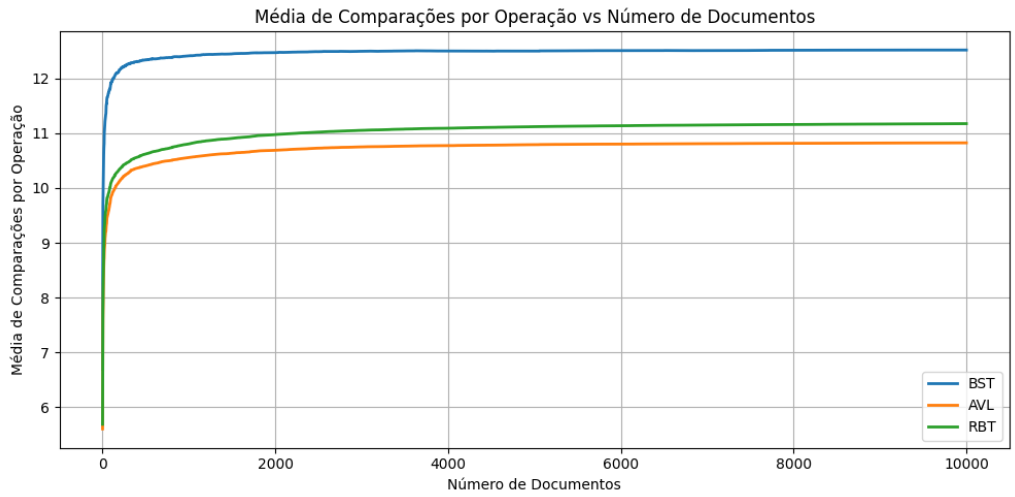

Figura 16: Número de comparações em inserções

O gráfico acima apresenta o número de comparações realizadas durante a inserção de palavras nas diferentes estruturas de árvore. De modo geral, observa-se um crescimento que remete a um comportamento logarítmico, o que é coerente com o funcionamento de estruturas binárias balanceadas. Nessas árvores, a cada nível descido, tende-se a eliminar aproximadamente metade do espaço de busca, o que naturalmente limita o número de comparações necessárias.

Além disso, nota-se que tanto a **AVL** quanto a Rubro-Negra (RBT) realizam, em média, menos comparações do que a **BST**. Essa diferença pode estar relacionada ao fato de que essas duas estruturas aplicam regras explícitas de balanceamento — no caso da AVL, mais rígidas; na RBT, mais flexíveis — o que contribui para manter a altura da árvore controlada. Por outro lado, como a BST não adota nenhum critério de balanceamento, sua altura pode crescer de forma mais acentuada, o que impacta diretamente no custo das inserções.

É importante destacar, no entanto, que embora o número de comparações seja um bom indicativo de desempenho, ele não é o único fator determinante. Aspectos como o custo das rotações (necessárias para manter o balanceamento) e o perfil dos dados inseridos também podem influenciar os resultados observados.

6.5 Análise de Rotações

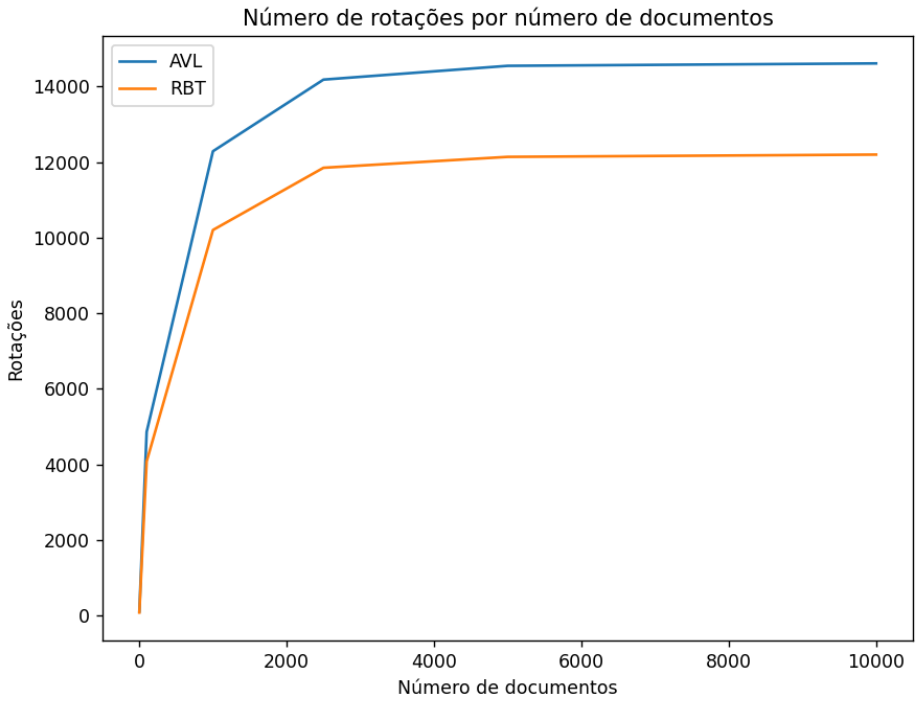

Figura 17: Número de rotações durante a inserção dos documentos

O gráfico acima compara o número de rotações realizadas durante a inserção de documentos nas árvores **AVL** e **RBT**. Observa-se que a árvore rubro-negra efetua significativamente menos rotações em comparação com a AVL, o que se deve à sua estrutura mais flexível em relação ao balanceamento. Essa flexibilidade reduz a necessidade de reestruturações frequentes, podendo impactar positivamente o tempo de inserção de novos `Nodes`, especialmente em cenários com grandes volumes de dados.

7 Conclusão

Com base nas análises apresentadas neste relatório, observamos que tanto a AVL quanto a árvore Rubro-Negra (RBT) oferecem vantagens claras em relação à BST tradicional, especialmente no que diz respeito à manutenção da altura da árvore e ao desempenho das operações de busca e inserção.

A árvore AVL, por ter uma estrutura de balanceamento mais rígida, garante alturas menores e tempos de busca potencialmente mais rápidos. No entanto, esse balanceamento mais agressivo pode resultar em um número maior de rotações durante as inserções, o que pode ser desfavorável em cenários com grandes volumes de dados sendo inseridos dinamicamente.

A árvore Rubro-Negra, por sua vez, adota um balanceamento mais flexível, realizando menos rotações em média e apresentando melhor desempenho de inserção em muitos casos. Ainda assim, isso pode vir com um pequeno custo em termos da altura da árvore e do tempo de busca, em comparação com a AVL.

Portanto, a escolha da estrutura mais adequada depende do cenário de uso. Para aplicações com muitas buscas e poucas inserções, a AVL pode ser mais vantajosa. Já em contextos com inserções frequentes, a RBT tende a oferecer um melhor equilíbrio entre custo de manutenção e desempenho. A BST, embora ineficiente em teoria, possui uma implementação mais simples e um desempenho melhor ou, em casos muito específicos, onde os dados estão ordenados, melhor do que o de uma lista ordenada.

Divisão de Tarefas

- **Adriel Dias** — Implementação e documentação da Árvore Rubro-Negra (RBT), desenvolvimento da função `printTree()` e suas funções auxiliares, criação de testes para RBT além da escrita geral do Relatório do projeto.
- **Gabriel Schuenker** — Implementação dos arquivos de análise de desempenho, do sistema de linha de comando (CLI) específico para a RBT, geração de grande parte das estatísticas e algumas correções e documentações.
- **Matheus Mendes** — Implementação e documentação da Árvore AVL, desenvolvimento do CLI para AVL, escrita da análise experimental no relatório.
- **Nícholas Farrel** — Implementação dos testes unitários, desenvolvimento do escopo do CLI geral e implementação da interface para a Árvore Binária de Busca (BST).
- **Vinícius Tavares** — Estruturação das pastas e arquivos do projeto, implementação e documentação da BST, desenvolvimento da função `printIndex()` e suas funções auxiliares e criação do arquivo `Makefile`.

Dificuldades Encontradas

Durante o desenvolvimento do trabalho, enfrentamos algumas dificuldades relevantes, como:

- **Implementação das Árvores do Zero:** Tivemos desafios na construção das estruturas de dados na linguagem `C++`, o que exigiu um entendimento aprofundado de ponteiros, recursão e gerenciamento de memória.
- **Análise dos Resultados:** A interpretação e comparação dos resultados obtidos nas simulações foi complexa, especialmente para identificar gargalos e justificar o comportamento das operações em diferentes cenários.
- **Trabalho em Equipe:** Apesar do esforço conjunto, houve momentos de dificuldade na coordenação entre os membros, especialmente na divisão equilibrada de tarefas e no alinhamento das implementações individuais.

Referências

[1] Szwarcfiter, Jayme Luiz. *Estrutura de Dados e Seus Algoritmos*. 3ª edição. LTC, 2010.

[2] Werner, Matheus. Curso de Estrutura de Dados — Repositório GitHub do professor da FGV. Disponível em: `https://github.com/matwerner/fgv-ed/tree/main`. Acesso em: junho de 2025.

[3] Sambol, Michael. Canal no YouTube com explicações sobre algoritmos e estruturas de dados. Disponível em: `https://www.youtube.com/@MichaelSambol`. Acesso em: junho de 2025.

Bibliotecas Utilizadas

As seguintes bibliotecas da linguagem C++ foram utilizadas no desenvolvimento do projeto:

- `<iostream>` — para operações de entrada e saída padrão.
- `<fstream>` — para leitura de arquivos contendo os documentos de entrada.
- `<vector>` — para armazenamento dinâmico dos identificadores de documentos.
- `<string>` — para manipulação de palavras e textos.
- `<sstream>` — para processamento de strings e divisão de palavras.