Fundação Getulio Vargas
Escola de Matemática Aplicada - EMAp

Índice Invertido e Análise Comparativa de Estruturas de Dados

Gustavo Luciano, Jean Domingueti, Leonardo Verissimo,
Rodrigo da Cruz e Sílvio Estêvão

Rio de Janeiro - RJ

Junho 2025

Sumário

1. Introdução

O projeto consiste em implementar, em C e sem orientação a objetos, um índice invertido — estrutura que associa cada termo à lista de documentos onde ele aparece — utilizando três tipos de árvores de busca: BST, AVL e Rubro-Negra. Cada implementação explora aspectos de alocação dinâmica e manipulação de ponteiros, permitindo comparar diretamente o custo de inserções, rotações e buscas em diferentes estratégias de balanceamento.

2. Organização

O trabalho contou com a colaboração e comprometimento de todos. A seguir, apresentam-se a responsabilidade de cada integrante e de que forma o trabalho foi estruturado.

1. **Gustavo**
   - Responsável pela análise estatística e pela sistematização dos resultados obtidos;
   - Estruturação e organização do relatório final;
   - Confecção de gráficos comparativos e condução da análise global dos dados.
2. **Jean**
   - Implementação da biblioteca `string`;
   - Implementação da leitura de arquivos;
   - Desenvolvimento e execução de testes em geral;
   - Identificação e correção de falhas, além do aprimoramento das rotinas de visualização;
3. **Leonardo**
   - Implementação das operações de inserção, busca, criação e destruição na Árvore de Busca Binária (BST);
   - Elaboração de testes e documentação das funcionalidades relativas à BST;
   - Estruturação e organização do relatório final.
4. **Rodrigo**
   - Implementação das operações de inserção, busca, criação e destruição na Árvore AVL;
   - Desenvolvimento de testes e documentação das funcionalidades referentes à AVL.
5. **Sílvio**
   - Implementação das operações de inserção, busca, criação e destruição na Árvore Rubro-Negra (RBT);
   - Elaboração de testes e documentação das funcionalidades relativas à RBT;
   - Desenvolvimento de funções principais (`main`) individuais para cada tipo de árvore, bem como das correspondentes interfaces de linha de comando (CLI);
   - Identificação e correção de falhas;
   - Revisor geral do código.

Adicionalmente, as bibliotecas `lkdlist`, `tree_utils` e `data` foram desenvolvidas em conjunto por Sílvio e Jean. Todos os integrantes cumpriram rigorosamente suas atribuições.

3. Utilitários

O módulo `tree_utils` oferece funções auxiliares para criação, visualização e exportação de árvores binárias (AVL, BST e RBT) utilizadas no projeto.

Funcionalidades principais

1. Criação:
   - `createNode()`: inicializa um nó com campos padrão e lista de documentos vazia.
   - `createTree()`: cria uma estrutura de árvore com raiz e NIL nulos.
2. Impressão:
   - `printTree(tree)`: mostra a árvore no terminal com formatação gráfica por caracteres Unicode (adaptada a Windows ou Unix).
   - `printIndex(tree)`: exibe as palavras e os IDs de documentos armazenados em cada nó, por ordem "in-order".
3. Análise:
   - `calculateHeight(node)`: calcula a altura de uma subárvore.
   - `getMaxID(node)`: retorna o maior número de IDs de documentos encontrados em um único nó.
   - `calculateMinPath(node, NIL)`: retorna o comprimento do menor caminho da subárvore até uma folha.
4. Exportação:
   - `saveTree(tree)`: gera visualização gráfica da árvore em SVG com cores representando a densidade de documentos em cada nó, usando o Graphviz[1].

Observações

- A visualização textual da árvore facilita depuração e compreensão da estrutura.
- O uso de cores na exportação gráfica permite identificar nós mais "relevantes" visualmente.
- O código trata corretamente ponteiros nulos e usa estruturas portáveis.

4. Como compilar e executar o código

```Shell
make all
./main_<árvore> <search/stats> <número_de_docs> <diretório_dos_docs>
./test_<árvore>
```

Além disso, há um *wrapper* que converte as funções implementadas em `C` para um *namespace* que pode ser utilizado em `C++`. Para utilizá-las, basta importar o arquivo `<árvore>_wrapper.h` no código C++ e usar as funções `insert`, `create`, `destroy` e `search` definidas no *namespace* específico da árvore, esse último sendo:

- BST para a árvore binária simples;
- AVL para a árvore binária balanceada:

[1]Nesse caso, a função `saveTree` usa o programa Graphviz, que deve estar instalado previamente no computador para que a função consiga utilizá-lo. Além disso, o Graphviz parece ter uma tendência a exportar a árvore com os nós de maior valor à esquerda da raiz, que difere da convenção que estamos utilizando nas nossas estruturas.

- RBT para a árvore rubro-negra.

Exemplo: para chamar a função que cria uma RBT, bastaria importar `rbt_wrapper.h` e chamar a função `RBT::create()`.

5. Estruturas

5.1. Árvore Binária de Busca [1]

5.1.1. Definição

Uma Árvore Binária de Busca (BST - *Binary Search Tree*) é uma estrutura de dados do tipo árvore binária em que cada nó obedece à seguinte propriedade de ordenação:

1. Todos os valores na subárvore esquerda de um nó são menores que o valor do nó;
2. Todos os valores na subárvore direita de um nó são maiores que o valor do nó.

Essa propriedade garante que operações como busca, inserção e remoção possam ser realizadas de forma eficiente, geralmente com complexidade proporcional à altura da árvore.

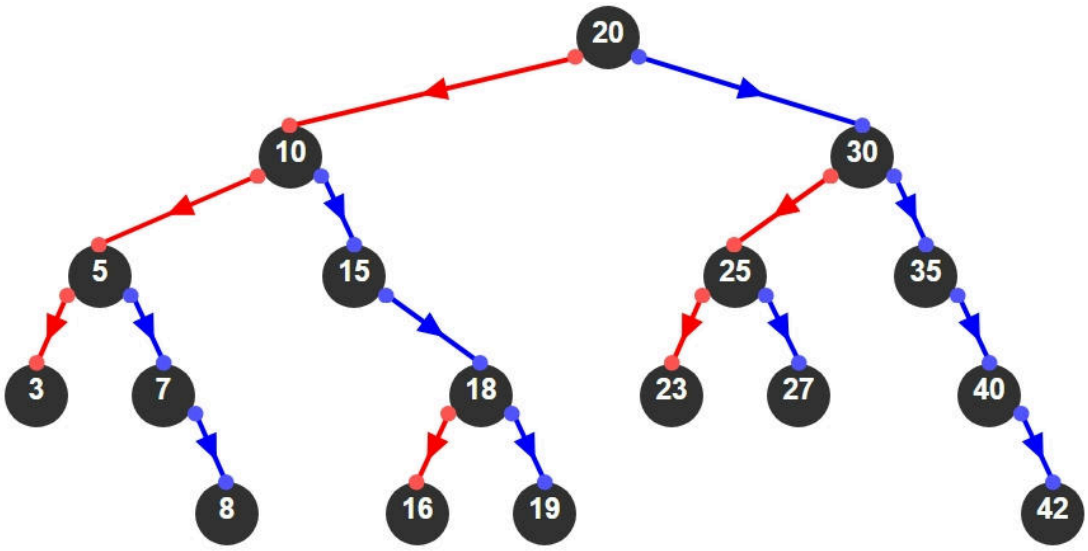


Figura 2: Exemplo de árvore BST com raiz igual a 20 e ordem de inserção:
20, 10, 5, 3, 7, 8, 15, 18, 16, 19, 30, 25, 23, 27, 35, 40, 42. [4]

5.1.2. Implementação

1. **Função `insertBST`**
   - A função se divide em 2 casos:
     1. **Árvore vazia**

        Caso a árvore esteja vazia, um nó é criado por meio da função auxiliar `createNodeWithWord` e, em seguida, esse nó é designado como a raiz da árvore passada como argumento. Nenhuma comparação é necessária nesse caso (`numComparisons = 0`).

     2. **Árvore não vazia**

        Caso a árvore não esteja vazia, é realizada uma busca recursiva usando `searchWord` para encontrar a posição correta de inserção.

        - A função encontra:

- **`currNode`**: nó onde a palavra foi encontrada, caso ela já exista;
- **`lasNode`**: último nó visitado antes de chegar a uma posição NULL, que será utilizado no caso em que a palavra inserida seja nova.

▸ Em seguida, há a divisão em dois subcasos:

- Palavra já registrada **(`currNode` ≠ NULL)**
  O ID do documento é adicionado à lista `documentIds` do nó existente.
- Palavra não registrada **(`currNode` = NULL)**
  Um novo nó é criado e inserido como filho de `lasNode` (esquerda ou direita, baseado na comparação de palavras).

2. **Função `searchBST`**

- Novamente há a utilização da função auxiliar `searchWord` para percorrer a árvore e encontrar a posição relativa a palavra desejada. Em seguida, há a divisão em dois casos:

1. Palavra já registrada **(`currNode` ≠ NULL)**
   O atributo "`found`" da variável "`result`" é atualizado com o valor de 1 e o atributo "`documentIds`" é atualizado com a lista `currNode→documentIds`.
2. Palavra não registrada **(`currNode` = NULL)**
   O atributo "`found`" da variável "`result`" é atualizado com o valor de 0 e o atributo "`documentIds`" é atualizado com NULL.

3. **Função `destroyBST`**

- Libera todos os nós da árvore de forma recursiva:

1. Se o nó atual for NULL, retorna imediatamente.
2. Caso contrário, chama a função `bstFreeRec(tree→root)` para liberar todos os nós.
3. Define `tree→root` = NULL, libera a *tree* e ajusta ponteiro para NULL.

5.1.3. Resultados

A seguir, apresentamos os resultados da execução das operações de inserção e busca na árvore **BST** para 10 000 arquivos:

Resultados encontrados	
Tempo de inserção	260.640
Tempo de busca	1.639000
Número de comparações	49,990,204
Altura da árvore	36
Tamanho do menor galho	5

Distribuição de Comparações nas Buscas

A imagem abaixo mostra um histograma com a frequência de comparações realizadas durante as operações de busca:

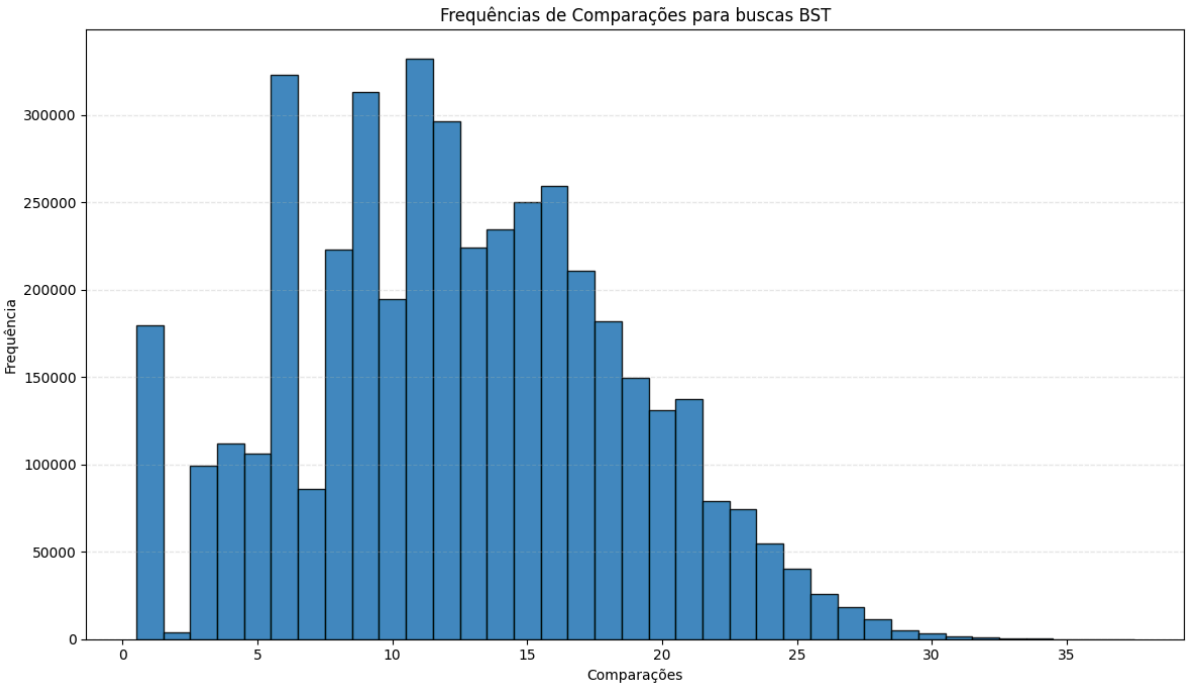


Figura 3: Frequência de comparações para buscas BST.

O gráfico mostra uma distribuição assimétrica, com uma cauda longa à direita. A maior parte das buscas exigiu entre 5 e 15 comparações, com picos em torno de 6, 8 e 11. No entanto, há casos com até 35 comparações, o que indica que algumas partes da árvore cresceram de forma desequilibrada.

Esse comportamento é típico de BSTs que não utilizam mecanismos automáticos de balanceamento. À medida que os dados são inseridos em ordem não aleatória, a árvore pode se tornar desbalanceada, aumentando o número de comparações em algumas buscas.

5.1.3.1. Análise Quantitativa

- Comparações mais frequentes: entre 6 e 12
- Comparação mínima observada: 1
- Comparação máxima observada: 35
- Tendência central (modas locais): 6, 8 e 11

Embora muitas buscas sejam resolvidas com poucas comparações, a ausência de balanceamento pode causar degradação do desempenho em partes específicas da árvore, tornando o custo de algumas buscas significativamente mais alto.

5.2. Árvore AVL [2]

5.2.1. Definição

Uma Árvore AVL — cujas iniciais homenageiam Adelson, Velsky e Landis — é uma árvore binária de busca autobalanceada, que além de satisfazer a propriedade de ordenação de uma BST, mantém um fator de balanceamento restrito em cada nó:

1. Todos os valores na subárvore esquerda de um nó são menores que o valor armazenado nesse nó;
2. Todos os valores na subárvore direita de um nó são maiores que o valor armazenado nesse nó;
3. O fator de balanceamento de cada nó - isto é, a diferença entre as alturas de sua subárvore esquerda e de sua subárvore direita - deve ser −1, 0 ou +1.

Essa garantia assegura que a altura da árvore permaneça em $O(\log n)$, fazendo com que operações de busca, inserção e remoção sejam realizadas em tempo $O(\log n)$ no pior caso.

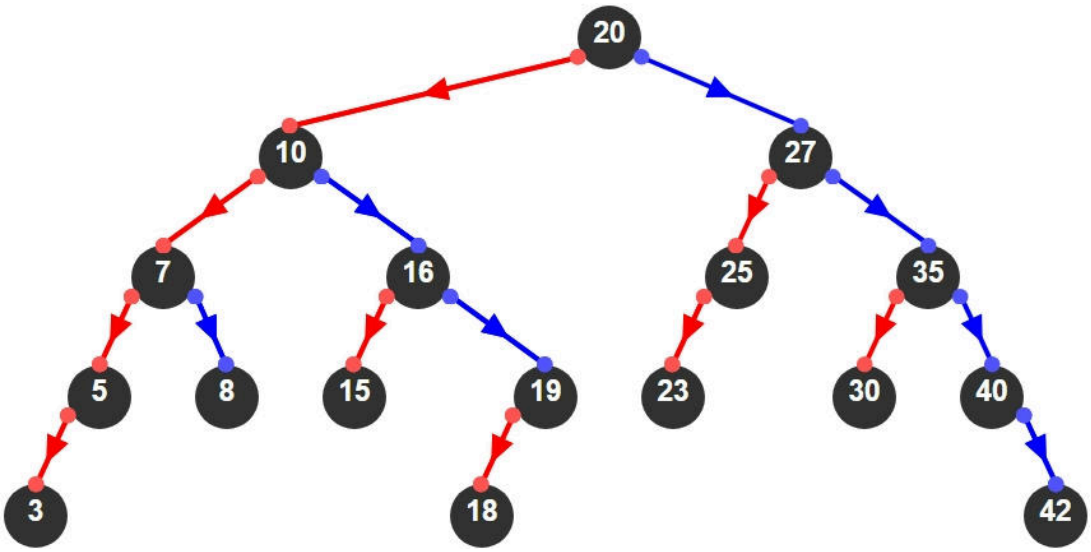


Figura 4: Exemplo de árvore AVL com raiz inicial igual a 20 e sequência de inserção: 20, 10, 25, 7, 16, 30, 5, 8, 15, 27, 40, 3, 19, 23, 35, 42, 18. [4]

5.2.2. Implementação

1. **Função `insertAVL`**

- A função se divide em dois casos:

  1. **Árvore vazia**

     Caso a árvore esteja vazia, um nó é criado por meio da função auxiliar `createNodeAVL`. Esse nó é designado como raiz da árvore passada como argumento. Nenhuma comparação é necessária nesse caso (`numComparisons = 0`).

  2. **Árvore não vazia**

     Caso a árvore não esteja vazia, inicia-se uma busca iterativa para localizar a posição correta de inserção, com contagem do número de comparações realizadas.

- **Durante a busca:**
  - Se a palavra já existir na árvore, o ID do documento é verificado na lista do nó atual por meio de `lookupValue`. Caso o ID ainda não esteja registrado, ele é adicionado à lista `documentIds`.
  - Se a palavra *não existir*, um novo nó é criado e inserido como filho esquerdo ou direito do último nó visitado (`parent`), dependendo do resultado da comparação de palavras.
- Após a inserção de um novo nó, inicia-se o **procedimento de reequilíbrio (balanceamento)** a partir do pai do novo nó, utilizando a função auxiliar `rebalance`.
  - A altura dos nós é atualizada por meio de `updateHeight`.
  - O **fator de balanceamento** de cada nó é calculado com `getBalanceFactor`, e são aplicadas rotações apropriadas:
    1. **Caso Esquerda-Esquerda:** rotação simples à direita.
    2. **Caso Esquerda-Direita:** rotação à esquerda no filho esquerdo, seguida de rotação à direita no nó desbalanceado.

3. **Caso Direita-Direita:** rotação simples à esquerda.
4. **Caso Direita-Esquerda:** rotação à direita no filho direito, seguida de rotação à esquerda no nó desbalanceado.

- Durante as rotações, os ponteiros para os pais são atualizados para manter a integridade da estrutura da árvore. Se a raiz for alterada, o ponteiro `tree.root` é atualizado.

2. **Função `searchAVL`**

A busca na árvore é feita iterativamente a partir da raiz, comparando a palavra procurada com as palavras armazenadas em cada nó.
A função se divide em dois casos:

- **Palavra registrada**

  Se a palavra for encontrada, o campo `found` da estrutura `searchResult` é atualizado para `1`, e a lista de `documentIds` correspondente é retornada.

- **Palavra não registrada**

  Se a palavra não for encontrada até alcançar um nó `null`, o campo `found` é mantido em `0`, e `documentIds` é retornado como `null`. Durante o processo, é contabilizado o número de comparações realizadas e o tempo de execução é medido.

3. **Funções auxiliares de balanceamento**

- `getHeight(node)`: Retorna a altura de um nó ou `0` se for `null`.
- `updateHeight(node)`: Atualiza a altura de um nó com base nas alturas dos filhos.
- `getBalanceFactor(node)`: Calcula o fator de balanceamento como a diferença entre as alturas do filho esquerdo e do direito.
- `rotateLeft(node)` / `rotateRight(node)`: Realizam rotações simples para restaurar o balanceamento da árvore, ajustando também os ponteiros de pais e filhos.
- `rebalance(node)`: Decide qual tipo de rotação aplicar com base no fator de balanceamento, e retorna o novo nó raiz da subárvore após o reequilíbrio.

5.2.3. Resultados

A seguir, apresentamos os resultados da execução das operações de inserção e busca na árvore **AVL** para 10 000 arquivos:

Resultados encontrados	
Tempo de inserção	253.733
Tempo de busca	1.29002
Número de comparações	47,230,297
Altura da árvore	16
Tamanho do menor galho	10

Distribuição de Comparações nas Buscas

A imagem abaixo mostra um histograma representando a distribuição da quantidade de comparações realizadas durante operações de busca:

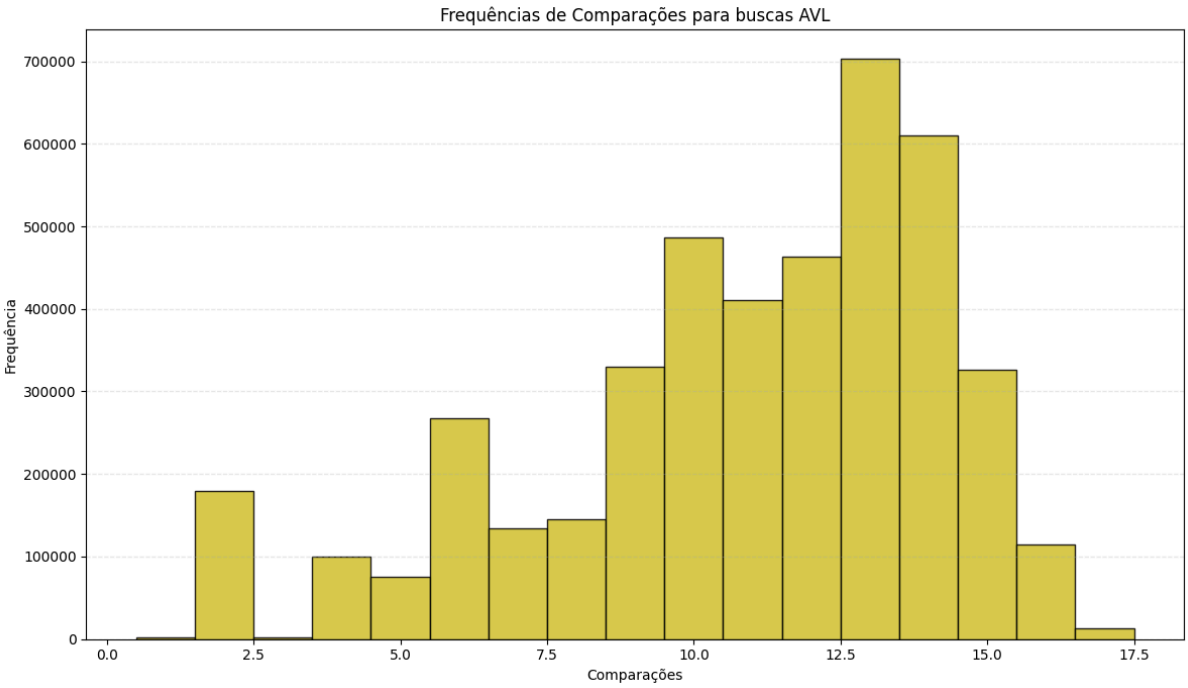


Figura 5: Frequência de Comparações para buscas AVL.

A maior parte das buscas exigiu entre 10 e 14 comparações, com um pico de frequência em torno de 13 comparações. Isso indica que, mesmo com um grande volume de palavras e documentos, a estrutura AVL manteve um desempenho estável e balanceado.

A cauda à esquerda mostra que algumas buscas exigiram poucas comparações (em torno de 2 a 6), provavelmente em casos em que a palavra estava próxima da raiz. Já os casos com maior número de comparações (até 17) representam situações em que a palavra estava em folhas mais profundas — ainda assim, esses casos são menos frequentes, mostrando a eficiência do balanceamento automático da AVL.

5.2.3.1. Análise Quantitativa

- **Comparações mais frequentes**: 12 a 14
- **Comparação mínima observada**: 1
- **Comparação máxima observada**: 17
- **Tendência central (moda)**: aproximadamente 13

Esse resultado está de acordo com o esperado para uma árvore AVL, onde a altura é mantida em `O(log n)` mesmo após muitas inserções. Isso garante que o número de comparações para busca também permaneça logarítmico na maioria dos casos.

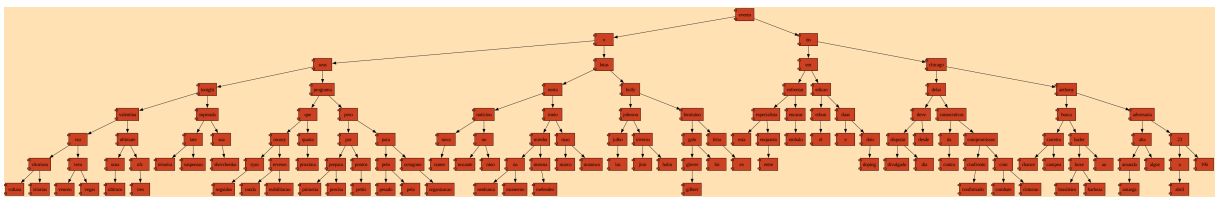


Figura 6: Imagem da AVL gerada com a função saveTree para o primeiro documento.

5.3. Árvore Rubro-Negra [3]

5.3.1. Definição

Uma Árvore Rubro-Negra (RBT - *Red-Black Tree*) é uma árvore binária de busca balanceada que associa a cada nó um atributo de cor (vermelho ou preto) e impõe propriedades que garantem profundidade balanceada após operações de inserção e remoção. Essas regras asseguram que o caminho mais longo da raiz até qualquer folha não exceda o dobro do caminho mais curto, mantendo complexidade assintótica $O(\log n)$ para operações fundamentais.

Uma RBT é uma árvore que satisfaz as seguintes propriedades:

1. Todo nó é ou preto ou vermelho;
2. Todos os nós nulos (que nesse caso consideramos como os nós folha) são pretos;
3. Um nó vermelho não pode ter filhos que sejam também vermelhos;
4. Todo caminho de um dado nó e qualquer uma de suas folhas (os nós nulos de seus descendentes) tem o mesmo número de nós pretos;
5. Por fim, se um nó tem exatamente um filho, esse filho deve ser vermelho.

A última propriedade é, na verdade, uma conclusão lógica que segue das propriedades 2 e 4. Isso porque, se um nó preto tivesse apenas um filho e este, por sua vez, fosse preto, então esse filho (ou seus descendentes) também teriam filhos pretos (pois as folhas de qualquer nó são sempre pretas). Isso, por sua vez, faria com que houvesse um número diferente de nós pretos no caminho entre o nó inicial e cada uma de suas folhas, violando a propriedade 4.

Perceba que a propriedade 3 implica que o caminho de cada nó até sua folha mais distante será sempre menor ou igual ao dobro da distância do caminho entre esse nó e sua folha mais próxima. Isso indica que essas árvores são "aproximadamente balanceadas", uma vez que, apesar das alturas dos nós diferirem em mais do que $|1|$, ela ainda garante tempo logarítmico para suas operações.

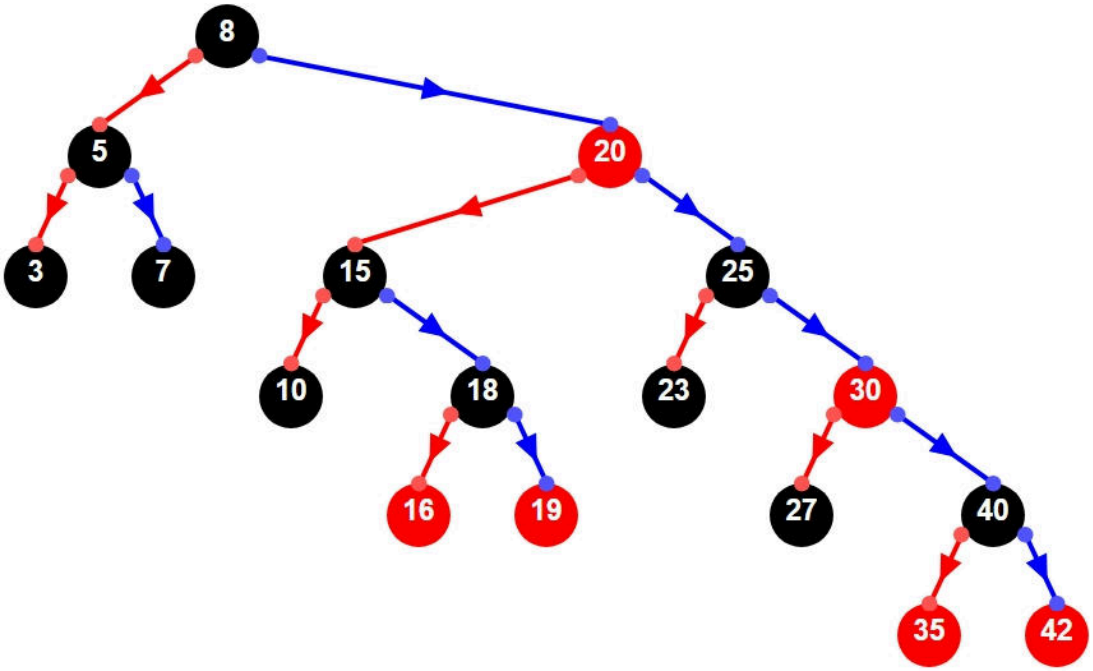


Figura 7: Exemplo de árvore RBT com raiz inicial igual a 20 e sequência de inserção: (20, 10, 30, 5, 3, 7, 8, 15, 18, 19, 16, 25, 23, 27, 35, 40, 42). [4]

5.3.2. Implementação

A implementação da árvore Rubro-Negra compartilha elementos com árvores binárias convencionais. Contudo, as operações de inserção e remoção demandam abordagens específicas, pois as operações de busca não alteram a árvore e a de criação retorna uma árvore vazia.

Considerando o escopo deste trabalho, algoritmos de remoção de nós não foram implementados. Consequentemente, as diferenças entre as funções de inserção da RBT e de uma BST convencional são o centro da análise.

Observa-se que, caso tenhamos uma árvore rubro-negra e insiramos um nó vermelho genérico, ele substituirá uma das folhas (os nós nulos) e passará a ter dois filhos pretos (novamente, os nós nulos). Olhando para seu pai, ele pode ser vermelho ou preto. Se for preto, não há nada que ser feito. Porém, se for vermelho, a terceira propriedade estará sendo quebrada e, portanto, é necessário rebalancear a árvore. Nesse caso, uma das seguintes situações é possível:

1. O tio do nó é vermelho;
2. O tio do nó é preto e a posição do nó relativa ao pai é diferente da posição do pai relativa ao avô.
3. O tio do nó é preto e a posição do nó relativa ao pai é a mesma do pai relativa ao avô (ou seja, se o pai fica à esquerda do avô, o filho fica à esquerda do pai nesse caso);

De fato, é trivial ver que, ao inserir-se um nó vermelho qualquer na árvore, ele sempre recairá em um dos casos acima. Agora, basta ver como rebalancear a árvore a partir desses casos:

1. No primeiro caso, como o pai é vermelho e o tio também, o avô é necessariamente preto (para não violar a propriedade 3), logo é possível repintar os três nós para suas cores contrárias, fazendo com que a "altura preta" da árvore não mude e, além disso, o novo nó inserido também não esteja mais ferindo qualquer propriedade. Porém, como o avô mudou de cor e agora é vermelho, ele pode ser filho de outro nó também vermelho, o que quebra a propriedade 3. Perceba, porém, que esse é apenas o caso que acabamos de resolver, o que indica que podemos apenas fazer o processo descrito recursivamente, até atingir a raiz que, por ser sempre preta, pode ter filhos vermelhos sem problema algum. Assim, a árvore volta a cumprir com as propriedades.
2. Nesse caso, basta fazer a rotação necessária para reduzir o caso atual para o próximo e, então, resolver àquela maneira. Isso é feita com uma rotação à esquerda no pai, caso o filho esteja à sua direita, ou à direita no pai, caso o filho fique à esquerda. Após isso, basta tomar o caso 3.
3. Já no terceiro caso, o pai é vermelho e o tio é preto. Porém, o avô ainda precisa ser preto (pois o pai é vermelho). Logo, se fizermos uma rotação, colocando o avô como filho do pai, a altura preta da sub-árvore à esquerda do pai diminuirá em 1, mas se manterá igual para a sub-árvore da direita (quebrando a propriedade 4). Porém, se pintarmos o antigo avô de vermelho e o antigo pai de preto, não quebraremos a propriedade 3 (pois o tio, novo filho do antigo avô, é preto) e rebalancearemos a altura preta da árvore, além de fazer com que a propriedade 3 deixe de ser violada pelo nó que inserimos. Perceba que após a rotação, o filho do antigo pai que era diferente do nó inserido passa a ser o filho do antigo avô, logo ele também não altera a altura preta da árvore. Esse processo de troca e recolorimento de nós é feito com uma rotação no avô (à direita se o nó inserido ficar à esquerda do pai ou à esquerda caso contrário) e mudando a cor dos nós pai e avô. Com isso, as propriedades são recuperadas.

Com isso, temos a seguinte função de inserção:

1. **função insertRBT**
    1. **Árvore vazia**
       Caso a árvore esteja vazia, basta criar uma da mesma forma que é feito na BST. Porém, pintando

o nó adicionado de preto e inserindo um nó adicional: **NIL**, que representará os nós nulos (folhas) da árvore.

2. **Árvore não vazia**
   Nesse caso, insere-se um nó como na BST. Se o nó pai for vermelho, chama-se a função **`getCase`** para identificar qual dos três cenários de inserção da RBT aplica-se à estrutura. Em seguida, usa-se uma das seguintes funções, a depender do caso retornado pela **`getCase`**:

   1. **`firstHelper`**: muda a cor dos nós pai, avô e tio e chama-se recursivamente, até que chegue na raiz, onde ela para e retorna. Ou seja, trata o caso 1;
   2. **`secondHelper`**: executa a rotação apropriada, a depender da orientação relativa do nó ao seu pai, para que a árvore caia no caso 3. Ou seja, resolve o caso 2;
   3. **`thirdHelper`**: por fim, a **`thirdHelper`** muda a cor do pai e do avô e executa a rotação apropriada no avô, para rebalancear a árvore. Resolve o caso 3.

Além disso, a função de rotação utilizada segue exatamente a mesma lógica da implementada na AVL.

A seguir, temos a função de busca `searchRBT`, que, como não difere da equivalente na BST, foi implementada da mesma forma. Da mesma forma, as funções `createRBT` e `destroyRBT` funcionam exatamente como suas análogas na BST.

5.3.3. Resultados

A seguir, apresentamos os resultados da execução das operações de inserção e busca na árvore **RBT** para 10 000 arquivos:

Resultados encontrados	
Tempo de inserção	260.640
Tempo de busca	1.68
Número de comparações	48444111
Altura da árvore	17
Tamanho do menor galho	10

Distribuição de Comparações nas Buscas

A imagem abaixo mostra um histograma representando a frequência de comparações realizadas durante as operações de busca:

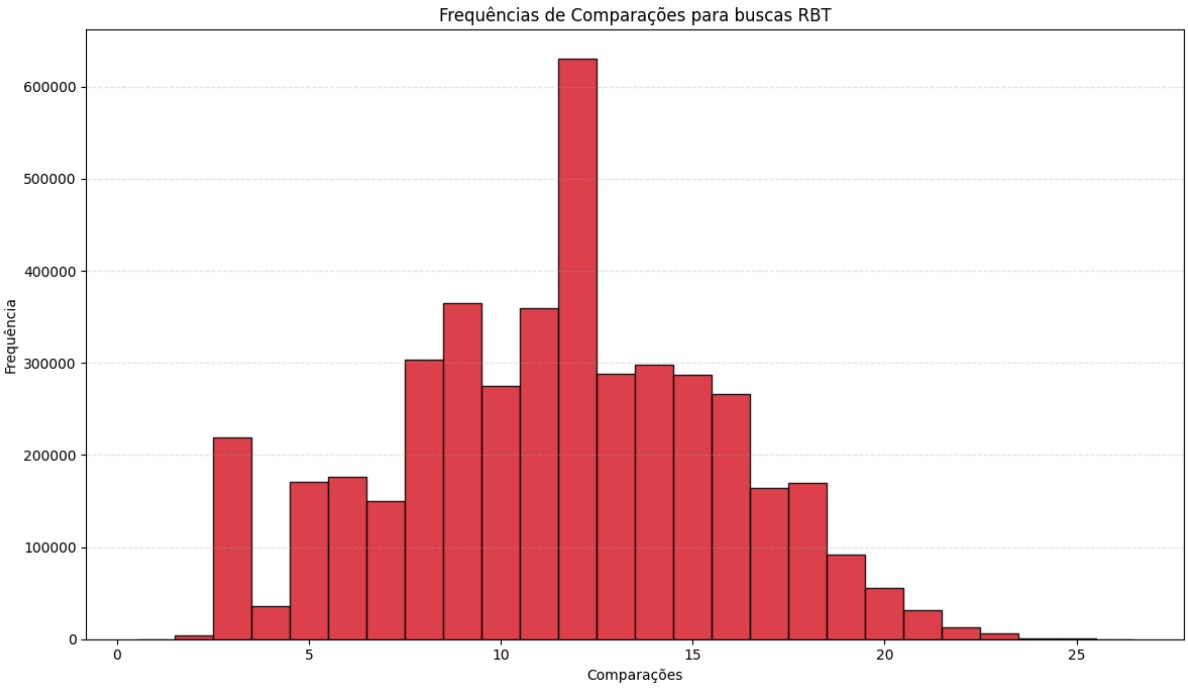


Figura 8: Frequência de Comparações para buscas RBT.

A maior parte das buscas exigiu entre 10 e 15 comparações, com um pico concentrado em torno de 12. A distribuição é relativamente simétrica, com a maior frequência centralizada, mas com casos que se estendem até cerca de 25 comparações.

Esse comportamento é característico de árvores rubro-negras, que mantêm um balanceamento eficiente, mesmo com variações na profundidade entre diferentes caminhos da árvore.

5.3.3.1. Análise Quantitativa

- Comparações mais frequentes: entre 11 e 13
- Comparação mínima observada: 2
- Comparação máxima observada: 25
- Tendência central (moda): aproximadamente 12

O número de comparações se manteve dentro do esperado para essa estrutura, evidenciando que o balanceamento automático proporcionado pelas regras de cores é eficaz para garantir boas condições de busca mesmo com grande volume de dados.

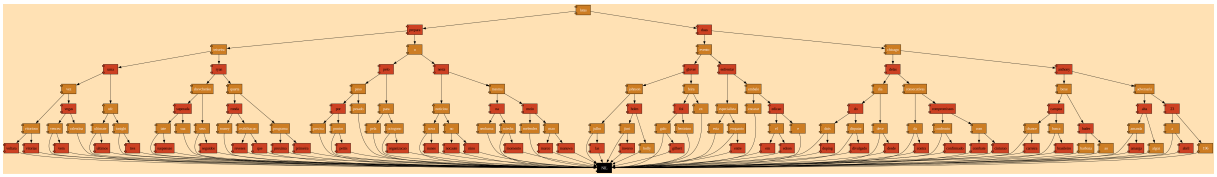


Figura 9: Imagem da BRT gerada com a função saveTree para o primeiro documento.

6. Desafios

6.1. Verificação de duplicatas e complexidade temporal

A implementação da detecção de duplicatas apresentou um dilema envolvendo desempenho e complexidade algorítmica. Diante disso, quatro abordagens diferentes foram consideradas (sendo n o número de palavras únicas e m o maior índice inverso da árvore):

1. **Ignorar a verificação de duplicatas**
   - Descrição: Tratamento de todas as chaves como novas inserções, sem a checagem de repetições.
   - Complexidade: $O(\log n)$.
   - Vantagens: Fácil implementação.
   - Desvantagens: Não impede múltiplas entradas iguais, o que compromete a integridade dos dados.
2. **Comparar com o último elemento inserido**
   - Descrição: Verificação apenas em relação à última chave adicionada.
   - Complexidade: $O(\log n)$.
   - Vantagens: Eficiente para dados já ordenados.
   - Desvantagens: Ineficiente quando os dados não estão ordenados, falhando na detecção de duplicatas.
3. **Varredura sequencial do início ao fim**
   - Descrição: Percorre toda a lista de chaves, do primeiro ao último, buscando duplicatas.
   - Complexidade: $O(m^2 \log n)$.
   - Vantagens: Encontra todas as duplicatas.
   - Desvantagens: Desempenho extremamente baixo para listas muito grandes.
4. **Varredura sequencial do fim ao início**
   - Descrição: Percorre a lista de chaves em ordem inversa, do último ao primeiro.
   - Complexidade $O(m^2 \log n)$.
   - Vantagens: Maior eficiência em cenários parcialmente ordenados; mesmo nível de robustez que a varredura direta.
   - Desvantagens: Custo elevado em casos de dados completamente desordenados.

Solução escolhida

A varredura inversa (do último para o primeiro) foi escolhida por oferecer maior desempenho em conjuntos de dados parcialmente ordenados — como é o caso deste trabalho, já que os documentos são lidos em ordem crescente — sem comprometer a robustez na identificação de duplicatas.

Observação: As complexidades indicadas são estimativas aproximadas.

6.2. Atualizações na raiz após rotações na AVL

A implementação das rotações na AVL exigiu um ajuste extra para manter o ponteiro da raiz sempre correto:

- Problema: Quando a rotação envolvia o nó raiz, o ponteiro `tree→root` continuava apontando para o nó antigo, apesar de a subárvore interna já estar corretamente reestruturada.

- Causa: A função rebalance atua apenas na subárvore desbalanceada e retorna o novo nó de raiz dessa subárvore, mas não atualiza automaticamente o ponteiro principal `tree→root`.
- Solução encontrada: Após chamar `rebalance`, verificamos se o nó retornado difere da raiz atual. Se for o caso, atribuímos manualmente `tree→root = <nó retornado>`, garantindo que o ponteiro principal reflita a nova estrutura.

Com essa correção, garantimos que a raiz da árvore AVL esteja sempre correta, mantendo a consistência da estrutura e o bom funcionamento das operações.

Em virtude das limitações de prazo, não foi possível realizar os testes com a segunda base de dados, o que constitui um dos principais desafios enfrentados.

7. Resultados e comparações gerais

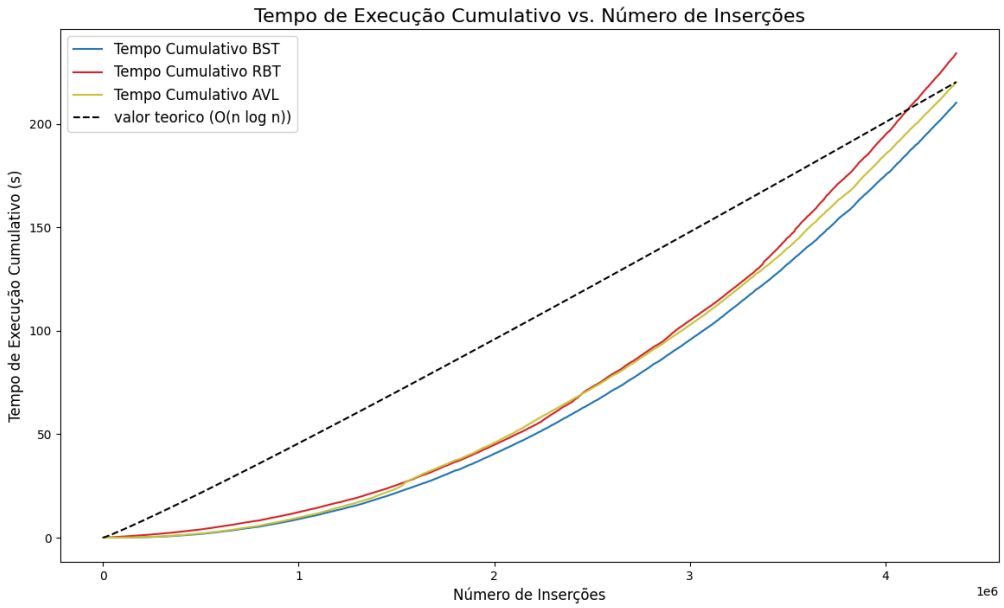
7.1. Complexidade de tempo

Com base na solução previamente discutida, adotou-se uma abordagem robusta cujas implementações apresentam, em média, complexidade próxima de $O(m \log n)$, com ligeiro acréscimo ocasionado pelas seguintes características:

- **Vetores ordenados**
  A ordenação prévia dos vetores reduz o custo de busca na posição adequada.
- **Alta proporção de palavras duplicadas**
  Como muitas chaves já existem no vetor, a inserção geralmente ocorre em tempo constante amortizado, $O(1)$, reservando operações de custo $O(m)$ apenas para casos excepcionais.

Dessa forma, o custo médio de inserção em cada vetor aproxima-se de $O(\log n)$, resultando no comportamento geral mencionado.

O gráfico abaixo apresenta os tempos de execução medidos para todas as árvores:



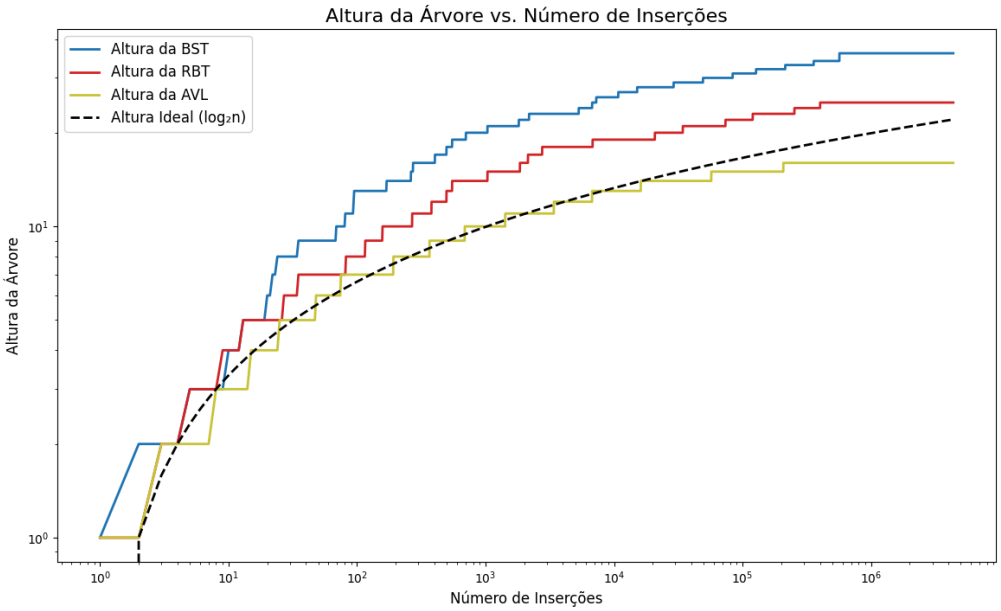
Observa-se que todas as árvores exibem complexidade de tempo muito próxima — o que era de se esperar, já que compartilham a mesma base algorítmica. Em detalhes:

- A **BST** mantém-se como a curva inferior, refletindo sua implementação mais simples.
- A **Árvore Rubro-Negra** e a **Árvore AVL** apresentam desempenho semelhante entre si, pois ambas incorporam rotações que acrescentam um pequeno "*overhead*"[2] em relação à BST.

Assim, embora as três estruturas operem na mesma ordem de grandeza, as diferenças de implementação (principalmente as rotações) explicam o leve acréscimo de custo nas árvores balanceadas. Essa métrica ainda é sensível a fatores aleatórios, e pode não ser replicável em todos os casos.

7.2. Alturas e balanceamento

Devido a baixa replicabilidade dos tempos de execução, podemos comparar os modelos também pelas alturas das árvores. O *plot* abaixo mostra a altura relativa ao número de inserções:



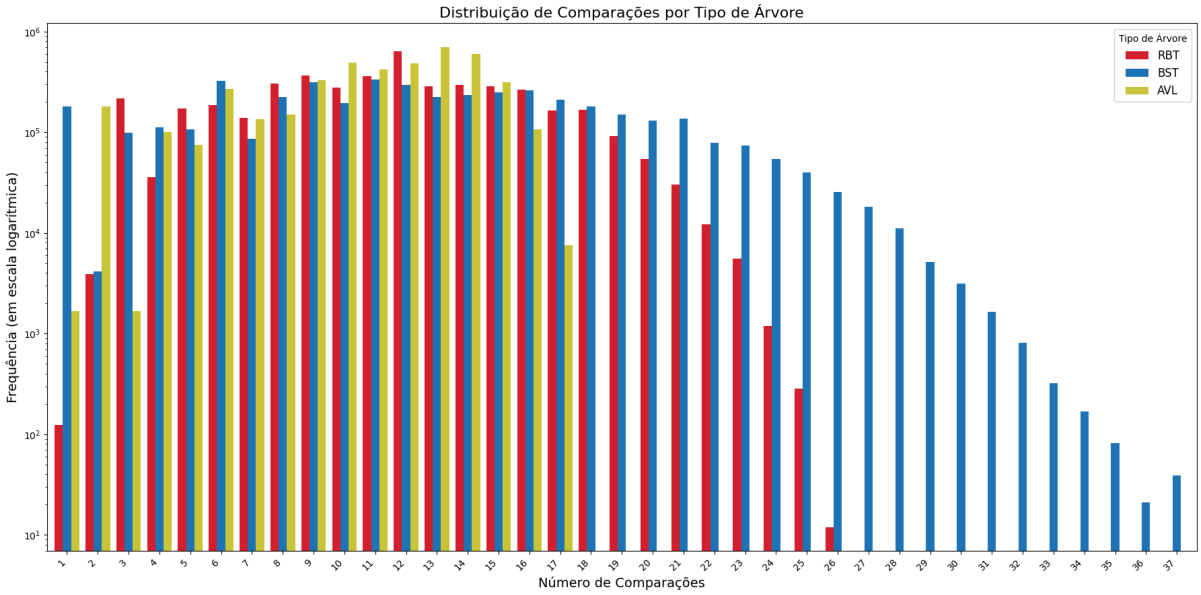
Obtém-se, novamente, os resultados esperados:

- A **AVL** se autobalanceia a cada iteração, mantendo com uma altura semelhante ao valor teórico, sendo uma boa aproximação discreta do valor de $\log n$
- A **RBT** também é uma árvore balanceada, então se aproxima também da curva
- A **BST** não se balanceia, então cresce rapidamente, se mantém mais alta que todas as outras arvores, e não é bem representada pela curva teórica

7.3. Contagem de comparações

Ao combinar as densidades de distribuição do número de comparações de cada estrutura, obtemos o gráfico de barras abaixo:

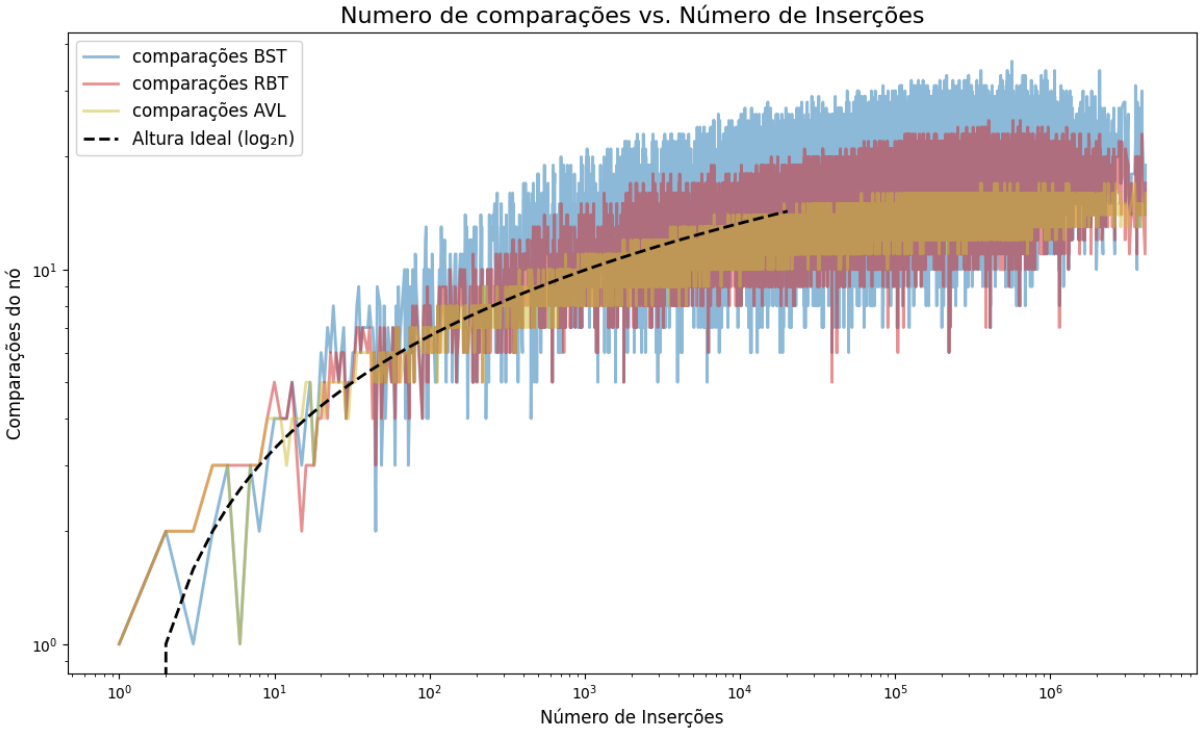
[2] Custo adicional, em termos de tempo, recursos ou complexidade, associado à execução de uma operação, processo, ou sistema.



Os resultados obtidos estão em conformidade com as características esperadas de cada árvore:

- **Árvore AVL (barras amarelas)**: revela uma forte concentração à esquerda, com um menor número de comparações e variação reduzida, o que evidencia seu balanceamento rigoroso;
- **Árvore BST (barras azuis)**: apresenta distribuição mais dispersa e picos ocasionais, demonstrando a ausência de balanceamento e maior flutuação no número de comparações;
- **Árvore Rubro-Negra (barras vermelhas)**: situa-se entre as duas anteriores; por possuir um balanceamento menos severo, apresenta concentração inicial significativa e alguns valores de comparações superiores aos da AVL.

Modificando os dados para analisarmos apenas inserções de palavras novas, podemos observar o comportamento contínuo do número de comparações:



- **Todas as árvores** se aproximam da curva de altura teórica
- A **AVL** mantém-se muito próxima a essa referência, estabilizando-se ligeiramente abaixo dela, com variação baixa, mostrando seu balanceamento;
- A **BST** ultrapassa rapidamente a curva teórica e mantém um crescimento contínuo de altura a partir desse ponto, evidenciando a falta de balanceamento. Além disso, apresenta grande oscilação, com picos mais altos e vales mais baixos que qualquer outra árvore;
- A **RBT** se comporta como uma média das outras árvores, apresentando alguns pontos atípicos, caindo bem abaixo da AVL por vezes, mas se estabilizando com números maiores.

8. Conclusão

A análise comparativa das três estruturas de dados aplicadas ao mesmo conjunto de documentos revelou diferenças significativas de eficiência e comportamento em função do tamanho do *dataset*. A Árvore de Busca Binária apresenta implementação mais simples e desempenho aceitável em volumes reduzidos, mas torna-se instável e vulnerável a desequilíbrios à medida que o número de elementos cresce - suas alturas podem ultrapassar 30, comprometendo buscas e inserções.

Em contraste, a Árvore AVL, apesar da complexidade adicional e do custo computacional imposto pelas rotações constantes, garante o menor crescimento em altura e o menor número total de comparações, justificando plenamente o esforço extra em cenários de grande porte. Já a Árvore Rubro-Negra aparece como uma alternativa intermediária entre as anteriores, uma vez que preserva o balanceamento com menos restrições do que a AVL e, por isso, realiza inserções mais rápidas, mesmo permitindo alturas ligeiramente superiores.

Bibliografia

[1] P. Feofiloff, «Árvores Binárias de Busca». [Online]. Disponível em: https://www.ime.usp.br/~pf/estruturas-de-dados/aulas/st-bst.html

[2] C. G. Fernandes, «Árvores AVL». [Online]. Disponível em: https://www.ime.usp.br/~cris/aulas/09_1_5710/slides/avl.pdf

[3] M. Werner, «Árvore Rubro-Negra». [Online]. Disponível em: https://github.com/matwerner/fgv-ed/blob/main/aulas/Semana%2012%20-%20Rubro%20Negra/22_arvore_rbt.md

[4] Portal do Professor — FCT Unesp, «Árvore Binária». [Online]. Disponível em: https://portaldoprofessor.fct.unesp.br/projetos/cadilag/apps/structs/arv_binaria.php