Análise Comparativa de Desempenho de Estruturas de Dados para Dicionários

Iago de Oliveira Lô - 565321

[1] Universidade Federal do Ceará - Campus Quixadá (UFC)
Quixadá – CE – Brazil

iagooliveiralo07@alu.ufc.br

Abstract. *This work presents the implementation and comparative performance analysis of four data structures for dictionaries: AVL Tree, Red-Black Tree, Chained Hash Table, and Open Addressing Hash Table with Double Hashing. The structures were applied to the problem of word frequency counting. Key performance metrics such as execution time, key comparisons, and structure-specific operations (rotations, color changes, collisions) were collected and analyzed.*

Resumo. *Este trabalho apresenta a implementação e a análise de desempenho comparativa de quatro estruturas de dados para dicionários: Árvore AVL, Árvore Rubro-Negra, Tabela Hash com Encadeamento e Tabela Hash com Endereçamento Aberto por Hash Duplo. As estruturas foram aplicadas ao problema de contagem de frequência de palavras. Métricas de desempenho chave como tempo de execução, comparações de chave e operações específicas de cada estrutura (rotações, trocas de cor, colisões) foram coletadas e analisadas.*

1. Introdução

A organização e o acesso eficiente a grandes volumes de dados são problemas fundamentais em projetos de software. A escolha da estrutura de dados correta é um fator crítico que impacta diretamente o desempenho de um sistema. Este trabalho foca na implementação e na análise de desempenho comparativa de quatro estruturas de dados distintas, aplicadas ao problema de contagem de frequência de palavras em um corpus textual. As estruturas analisadas são: a Árvore AVL, a Árvore Rubro-Negra, a Tabela Hash com Encadeamento Separado e a Tabela Hash com Endereçamento Aberto por Hash Duplo.

Este relatório está estruturado da seguinte forma: a Seção 2 detalha a implementação de cada estrutura de dados, destacando suas características e as decisões de projeto. A Seção 3 descreve a metodologia utilizada para os testes e a coleta de métricas. A Seção 4 apresenta os resultados numéricos obtidos, que são interpretados na Seção 5. Por fim, a Seção 6 apresenta a conclusão do trabalho, sumarizando os achados e discutindo os trade-offs de cada abordagem.

2. Implementação das Estruturas de Dados

Esta seção visa apresentar como cada estrutura foi pensada e construída, mostrando suas individualidades.

2.1. Árvore AVL

A Árvore AVL (Adelson-Velsky e Landis) é uma estrutura de árvore de busca binária (ABB) auto-balanceada. Sua propriedade fundamental é a manutenção de um fator de balanceamento para cada nó, definido como a diferença entre a altura da subárvore direita e a da subárvore esquerda. A propriedade da AVL estipula que este fator deve pertencer ao conjunto $\{-1, 0, 1\}$. Esta restrição garante que a altura da árvore permaneça logarítmica, $O(\log n)$, em relação ao número de nós, assegurando um desempenho eficiente para as operações. O rebalanceamento é mantido através de rotações simples ou duplas, acionadas quando uma inserção ou remoção viola a propriedade de altura.

A estrutura foi implementada como uma classe template em C++, `AVL<Key, Value>`, para garantir generalidade. Cada nó armazena um par chave-valor (`std::pair<Key, Value>`), ponteiros para os filhos esquerdo e direito, e um inteiro para sua altura. Na aplicação final deste projeto, o par será utilizado para armazenar uma palavra (chave) e sua frequência (valor).

As operações de inserção e remoção foram implementadas de forma recursiva. Após a modificação em uma subárvore, as chamadas recursivas retornam, permitindo a atualização da altura de cada nó ancestral. Durante este retorno, o fator de balanceamento é verificado. Caso um desbalanceamento seja detectado (fator -2 ou +2), a operação de rebalanceamento apropriada é executada para restaurar a invariante da AVL.

2.2. Árvore Rubro-Negra

A Árvore Rubro-Negra é uma ABB auto-balanceada que utiliza um sistema de coloração de nós (vermelho ou preto) para garantir seu balanceamento. O equilíbrio é mantido pela aplicação de cinco propriedades fundamentais, que, entre outras coisas, estipulam que a raiz é preta e que nenhum nó vermelho pode ter um filho vermelho. A propriedade mais importante é a da ”altura negra”: todo caminho de um nó até uma de suas folhas descendentes deve conter o mesmo número de nós pretos. Isso garante a altura logarítmica da árvore, $O(\log n)$.

A implementação da classe `RB<Key, Value>` utiliza um nó sentinela, `TNULL`, que representa todas as folhas (NIL) da árvore e também serve como pai da raiz. Esta abordagem de projeto simplifica o código ao evitar verificações de ponteiros nulos. Cada nó armazena o par chave-valor, sua cor, e ponteiros para os filhos esquerdo, direito e para o pai.

As operações de inserção e remoção são seguidas pela chamada de funções de correção (`insertFix` e `deleteFix`). Estas funções são responsáveis por restaurar as cinco propriedades da árvore, que podem ser violadas durante uma modificação, através de uma série de rotações e trocas de cor que se propagam pela árvore conforme necessário.

2.3. Tabela Hash com Encadeamento Separado

A Tabela Hash é uma estrutura que mapeia chaves a valores através de uma função de hash, permitindo, em média, operações em tempo constante. Para o tratamento de colisões — que ocorrem quando chaves distintas são mapeadas para o mesmo índice —, foi utilizada a estratégia de Encadeamento Separado. Nesta abordagem, cada índice da tabela (ou ”bucket”) aponta para uma lista ligada que armazena todos os elementos cujo hash corresponde àquele índice.

A classe `ChainedHashTable<Key, Value>` foi implementada utilizando um `std::vector` de `std::list<std::pair<Key, Value>>`. O mapeamento de chaves para índices é realizado em duas etapas: primeiro, a função de hash padrão do C++, `std::hash<Key>`, gera um código hash; em seguida, o método da divisão (operador módulo %) é aplicado sobre o tamanho da tabela para determinar o bucket.

Para garantir que o desempenho se mantenha próximo de $O(1)$, a tabela implementa um mecanismo de redimensionamento (`rehash`). Quando o fator de carga (definido como o número de elementos dividido pelo tamanho da tabela) excede um limiar pré-definido, a tabela é reconstruída com um tamanho maior — nesta implementação, o próximo número primo maior que o dobro do tamanho anterior — e todos os elementos são re-inseridos.

2.4. Tabela Hash com Endereçamento Aberto (Hash Duplo)

O Endereçamento Aberto é uma estratégia de tratamento de colisões em que todos os elementos são armazenados no próprio vetor da tabela. Quando uma colisão acontece, o algoritmo sonda (prova) posições subsequentes na tabela de forma sistemática até encontrar um slot disponível.

Para esta implementação, foi escolhida a técnica de Hash Duplo. Esta abordagem de sondagem utiliza uma segunda função de hash para calcular o intervalo (ou ”passo”) entre cada prova, o que mitiga eficientemente os problemas de agrupamento (*clustering*) que afetam métodos mais simples, como a sondagem linear.

Um pilar desta implementação é o gerenciamento do estado de cada slot, que pode ser `EMPTY`, `OCCUPIED` ou `DELETED`. O estado `DELETED` é fundamental para o correto funcionamento da operação de remoção. Em vez de remover fisicamente um elemento (o que poderia quebrar uma cadeia de sondagem existente), o slot é apenas marcado como deletado. Isso garante que as operações de busca por chaves que colidiram e passaram por aquele slot continuem a sondar corretamente, sem parar prematuramente. A remoção, portanto, é uma operação ”preguiçosa” que apenas altera o estado do slot.

3. Metodologia de Testes e Análise

Esta seção descreve o experimento realizado para avaliar o desempenho das estruturas de dados implementadas. O procedimento foi projetado para ser claro e replicável, garantindo a validade dos resultados apresentados.

3.1. Ambiente de Teste

Todos os testes de desempenho foram executados em um único ambiente de hardware e software para garantir a consistência e a comparabilidade dos resultados. As especificações do ambiente são detalhadas a seguir:

- **Hardware:**
  - Processador: `Intel(R) Core(TM) i3-4005U CPU @ 1.70GHz`
  - Memória RAM: `3.7 GiB`
- **Software:**
  - Sistema Operacional: `Ubuntu 24.04`
  - Compilador: `GNU G++ versão 13.3.0`

– Flags de Compilação: `-Wall -Wextra -std=c++17`. É importante notar que nenhuma flag de otimização (como `-O2` ou `-O3`) foi utilizada, a fim de medir o desempenho bruto dos algoritmos implementados sem interferência de otimizações do compilador.

3.2. Conjunto de Dados (Corpus)

Para realizar uma análise de desempenho robusta, foi gerado um conjunto de dados sintético que simula um cenário desafiador para as estruturas de dados, especialmente para as tabelas hash. O corpus de teste consistiu em:

- **Quantidade de Elementos:** 50.000 chaves únicas.
- **Tipo de Chave:** `std::string`.
- **Geração dos Dados:** As chaves foram geradas como strings aleatórias de 10 caracteres, compostas por caracteres alfanuméricos (maiúsculos e minúsculos) e dígitos.

A escolha por strings aleatórias, em vez de palavras de um texto real ou inteiros sequenciais, foi deliberada. Este tipo de dado garante uma alta entropia e uma distribuição que não favorece nenhuma estrutura em particular, além de aumentar a probabilidade de colisões na tabela hash, permitindo uma avaliação mais rigorosa de suas estratégias de tratamento de colisão.

3.3. Metodologia do Benchmark

Para mitigar a variabilidade causada por processos do sistema operacional e outros fatores estocásticos, cada benchmark foi executado 5 (cinco) vezes. Os valores apresentados na seção de Resultados correspondem à média aritmética das métricas coletadas ao longo dessas cinco execuções.

O benchmark para cada estrutura de dados foi dividido em duas fases distintas e sequenciais: Inserção e Busca.

1. **Fase de Inserção:** Para cada estrutura, um cronômetro de alta precisão (`std::chrono::high_resolution_clock`) foi iniciado. Em seguida, todos os 50.000 elementos do conjunto de dados foram inseridos na estrutura. Ao final do processo, o cronômetro foi parado, e o tempo total de inserção foi registrado. Durante esta fase, as métricas de comparações e as métricas específicas de cada estrutura (rotações, trocas de cor ou colisões) foram acumuladas.
2. **Fase de Busca:** Imediatamente após a fase de inserção, e com a estrutura já populada, um novo ciclo de medição foi iniciado. O mesmo conjunto de 50.000 chaves foi utilizado para realizar uma operação de busca para cada chave. O tempo total e o número de comparações para esta fase foram registrados separadamente. Este procedimento mede o desempenho de buscas bem-sucedidas no caso médio.

3.4. Métricas Coletadas

Para avaliar e comparar o desempenho de cada estrutura, as seguintes métricas foram coletadas:

- **Tempo de Execução (ms):** Medido em milissegundos, representa o tempo de parede (*wall-clock time*) total para completar todas as operações de uma determinada fase (inserção ou busca).

- **Comparações de Chave:** Um contador que é incrementado a cada vez que uma chave de busca é comparada com uma chave armazenada na estrutura. Esta métrica é um indicador direto do custo computacional do algoritmo de busca da estrutura.
- **Métrica Específica da Estrutura:**
  - **Rotações (Árvores AVL e Rubro-Negra):** Para as árvores auto-balanceadas, este contador mede o número total de operações de rotação realizadas para manter a propriedade de balanceamento durante a fase de inserção. É um indicador do "trabalho" de manutenção da estrutura.
  - **Trocas de Cor (Árvore Rubro-Negra):** Métrica adicional para a Árvore Rubro-Negra, contando o número de vezes que a cor de um nó foi alterada durante as operações de correção.
  - **Colisões (Tabelas Hash):** Para as tabelas hash, este contador mede a frequência com que a função de hash mapeou uma nova chave para um bucket que já estava ocupado (no caso do Encadeamento) ou a frequência com que a sondagem teve que procurar um novo slot após a tentativa inicial (no caso do Endereçamento Aberto).

4. Resultados

Nesta seção, são apresentados os resultados numéricos obtidos a partir da execução dos benchmarks, conforme a metodologia descrita anteriormente. Os valores representam a média aritmética de 5 (cinco) execuções, utilizando um conjunto de dados de 50.000 chaves de strings aleatórias.

A Tabela 1 resume o desempenho de cada estrutura de dados para as fases de inserção e busca.

Table 1. Resultados de Desempenho (Média de 5 Execuções)

Estrutura de Dados	Tempo Inserção (ms)	Tempo Busca (ms)	Comparações (Inserção)	Comparações (Busca)	Métrica Adicional
Árvore AVL	195.07	101.77	1.074.684	1.034.248	Rotações: 34.809
Árvore Rubro-Negra	118.23	68.44	1.080.842	1.139.559	Rotações: 28.859
Tabela Hash Encadeada	91.91	23.07	37.535	74.981	Colisões: 27.576
Tabela Hash End. Aberto	36.25	30.64	69.458	69.458	Colisões: 12.255

5. Análise dos Resultados

Nesta seção, os dados de desempenho apresentados na Tabela 1 são interpretados e discutidos. A análise foca em comparar o desempenho entre as classes de estruturas (árvores vs. tabelas hash) e dentro de cada classe, relacionando sempre os resultados práticos com a teoria de complexidade e as características de cada implementação.

5.1. Análise Geral de Desempenho

A análise dos tempos de execução revela uma distinção de desempenho inequívoca entre as duas classes de estruturas de dados. As Tabelas Hash, tanto a de Encadeamento quanto a de Endereçamento Aberto, demonstraram uma performance vastamente superior à das árvores auto-balanceadas em todas as métricas de tempo. A Tabela Hash com Endereçamento Aberto, em particular, obteve o menor tempo de inserção (36.25 ms), sendo aproximadamente 5.4 vezes mais rápida que a Árvore AVL (195.07 ms).

Este resultado é consistente com a complexidade de tempo teórica. As tabelas hash oferecem uma complexidade média de $O(1)$ para inserção e busca, enquanto as árvores AVL e Rubro-Negra operam com uma complexidade de $O(\log n)$. Para um grande volume de dados ($n = 50.000$), a diferença entre uma operação de tempo constante e uma logarítmica torna-se pronunciada, explicando a vantagem substancial das tabelas hash para esta aplicação.

5.2. Análise Comparativa das Árvores Balanceadas

Ao comparar as duas implementações de árvores auto-balanceadas, observa-se que a Árvore Rubro-Negra (118.23 ms) foi consideravelmente mais rápida na inserção do que a Árvore AVL (195.07 ms). Este fenômeno pode ser explicado pela natureza de suas operações de rebalanceamento. A Árvore AVL mantém um critério de balanceamento mais estrito (fator de balanceamento de no máximo 1), o que a força a realizar um número maior de rotações (34.809) para manter sua invariante. Em contraste, a Árvore Rubro-Negra, com suas regras de coloração mais flexíveis, necessitou de menos rotações (28.859), resultando em uma construção mais rápida.

Na fase de busca, o desempenho de ambas foi mais próximo, o que é esperado, dado que ambas garantem uma altura logarítmica e, portanto, um número de comparações da mesma ordem de magnitude (aproximadamente 1 milhão de comparações para 50.000 buscas). A pequena vantagem da Árvore Rubro-Negra em tempo de busca (68.44 ms vs. 101.77 ms) pode ser atribuída a fatores de localidade de cache ou à menor complexidade constante de suas operações internas.

5.3. Análise Comparativa das Tabelas Hash

A análise das tabelas hash revela nuances importantes. A Tabela Hash com Endereçamento Aberto (usando Hash Duplo) apresentou o melhor tempo de inserção geral (36.25 ms). Isso pode ser atribuído à sua localidade de cache superior: todos os elementos residem em um único vetor contíguo, evitando o custo de alocação de memória dinâmica para os nós de uma lista ligada e os saltos na memória (*cache misses*) que ocorrem ao percorrer o encadeamento.

É notável que a implementação com Hash Duplo resultou em um número de colisões significativamente menor (12.255) do que a Tabela Hash Encadeada (27.576). Isso valida a escolha do Hash Duplo como uma estratégia de sondagem superior para mitigar o agrupamento e distribuir as chaves de forma mais eficaz.

Curiosamente, apesar de ter menos colisões, a Tabela com Endereçamento Aberto realizou mais comparações na inserção (69.458) do que a Encadeada (37.535). Isso ocorre porque cada sondagem no endereçamento aberto conta como uma comparação, enquanto na tabela encadeada, as comparações só ocorrem ao percorrer a lista de um bucket que já sofreu colisão. Na fase de busca, no entanto, a Tabela Hash Encadeada (23.07 ms) mostrou-se ligeiramente mais rápida, sugerindo que o custo de percorrer listas curtas pode, em alguns casos, ser menor que o custo de múltiplos saltos de sondagem do Hash Duplo.

5.4. Relação entre Métricas e Desempenho

É fundamental observar a correlação direta entre as métricas coletadas e o tempo de execução. Nas árvores, o número total de comparações (na ordem de 10^6) é o principal

fator que determina o tempo de busca, refletindo a natureza de sua travessia logarítmica. Na inserção, o custo adicional das rotações contribui para o tempo total.

Nas tabelas hash, o número de colisões durante a inserção é o principal indicador de degradação de desempenho. Cada colisão representa um trabalho adicional — seja percorrendo uma lista (Encadeamento) ou sondando novos slots (Endereçamento Aberto) — que se reflete diretamente no tempo de inserção e no número de comparações. A eficiência superior das tabelas hash é evidenciada pelo número de comparações de busca (próximo de 70.000), que, embora maior que o número de elementos (50.000) devido às colisões, ainda é uma ordem de magnitude menor que o das árvores.

6. Conclusão

Este trabalho realizou a implementação e uma análise de desempenho rigorosa de quatro estruturas de dados fundamentais, aplicadas a um problema de dicionário. Os resultados experimentais confirmaram, em grande parte, as expectativas teóricas, mas também revelaram nuances importantes sobre os trade-offs práticos de cada abordagem.

Os principais achados indicam uma clara superioridade de desempenho das Tabelas Hash em termos de tempo de execução para as operações de inserção e busca, sendo a implementação com Endereçamento Aberto por Hash Duplo a mais rápida na construção. Em contrapartida, as árvores auto-balanceadas, embora mais lentas, demonstraram um comportamento mais previsível, com um custo de comparações logarítmico consistente, independente da distribuição das chaves. A análise das métricas específicas, como rotações e colisões, foi fundamental para explicar as diferenças de tempo observadas.

Conclui-se, portanto, que não existe uma estrutura de dados universalmente ”melhor”, mas sim uma escolha mais adequada dependendo dos requisitos da aplicação. Para cenários onde a velocidade de inserção e busca é o fator crítico, como na contagem de frequência de palavras em massa, as Tabelas Hash são a escolha indiscutível. No entanto, para aplicações que exigem um desempenho de pior caso garantido ou a capacidade de percorrer os elementos de forma ordenada (uma funcionalidade não explorada neste trabalho, mas inerente às árvores), as Árvores AVL ou Rubro-Negra seriam mais apropriadas.

O desenvolvimento deste projeto reforçou a importância de não apenas compreender a complexidade teórica dos algoritmos, mas também de analisar os fatores práticos que influenciam o desempenho, como a localidade de cache e os custos de alocação de memória. Como trabalhos futuros, sugere-se a expansão da análise para incluir outras estratégias de sondagem, como a quadrática, e a aplicação das estruturas em conjuntos de dados com diferentes características, como textos com um vocabulário mais restrito, para observar o impacto na distribuição de chaves e no número de colisões.

7. Referências

References

@bookcormen2009, title=Introduction to algorithms, author=Cormen, Thomas H. and Leiserson, Charles E. and Rivest, Ronald L. and Stein, Clifford, year=2009, publisher=MIT press, edition=3rd

@misccpluspluscom, author = cplusplus.com, title = The C++ Resources Network, howpublished = https://cplusplus.com/reference/, year = 2025, note = Acessado em: 04-07-2025