Comparação de Desempenho entre as Estruturas de Dados Hash Table e Árvore Balanceada

Alexandre Ferreira e Silva
Programa de Pós Graduação de Ciência da Computação
Universidade Federal de São Paulo
São José dos Campos, São Paulo, Brasil
alexandregoodmann@gmail.com

***Abstract*—This study presents a comparative analysis of the performance of Hash Table and Balanced Tree (AVL) data structures applied to a large-scale, real-world dataset. The research examines theoretical and experimental aspects of these structures, measuring and comparing insertion and search times. The evaluations focused on data manipulation efficiency using approximately 300,000 records. The results highlight the advantages and disadvantages of each structure, emphasizing the most appropriate contexts for their use based on computational complexity and measured performance.**

***Index Terms*—Hash Table, Árvore AVL, Data Structure Performance, Insertion Efficiency, Hashing Techniques, UUID, Large-Scale Data Processing.**

I. Introdução

O processamento de informações é um problema comum dentro da computação, pois vários aspectos precisam ser levados em consideração, não somente os dados que serão processados, mas também a estrutura de dados em que essas informações precisam estar organizadas. Este trabalho propõe comparar o desempenho entre as estruturas de dados Hash Table e Árvore Balanceada aplicadas à uma base de dados real e de grande proporções.

Este artigo aborda temas como estrutura de dados, especificamente sobre Hash Table e estrutura em árvore balanceada (AVL), faz uma breve comparação entre essas duas estruturas, ao ponto de demonstrar resultados a certa da performance de inserção e busca para cada uma delas. Toca em pontos específicos de cada estrutura com massa de dados real e algorítimos. Explica qual estrutrua deverá ser utilizada para cada necessidade específica, comparando e sugerindo vantagens de desvantagens para cada uma delas.

A. Hash Table Encadeada

A tabela Hash, também conhecida como Hash Table é uma estrutura de dados que facilita a armazenagem de dados, ou seja, proporciona uma inserção de maneira rápida [2]. Consiste basicamente em uma chave e valor, usando o que se chama de função hash para mapear as chaves, que poderão ser de tipos variados, como número, string ou outro. Para este trabalho em questão será adotado chave do tipo inteiro. Essas chaves informam o índice onde se encontra o valor a ser armazenado. Por tratar-se da chave ser o próprio índice do array onde o valor está armazenado, isso traz uma grande performance na busca, sem a necessidade em percorrer toda a estrutura de dados, levando em consideração que esta estrutura está alocada em memória. A função Hash recebe a chave informada, neste caso um número inteiro e transforma em um índice da tabela, faz uma distribuição das chaves para os índices, com isso colisões ocorrem, quando um mesmo índice é utilizado para várias chaves. Basicamente há dois tipos de colisão; uma de endereçamento aberto, quando o algorítimo procura um próximo índice disponível e a segunda colisão chamada de encadeamento, que utiliza de listas ligadas para armazenar múltiplos valores, este último será abordado neste trabalho. A vantagem em se utilizar uma tabela hash está no seu rápido acesso, seja inserção, remoção ou busca, pois sabendo-se o índice em que se encontra o dado a complexidade é constante $O(1)$. As desvantagens se encontram nas colisões que são formadas devido ao tamanho da tabela, reduzindo a eficiência. Para que estas colisões sejam minimizadas faz-se necessário aumentar o tamanho da tabela para que haja espaço suficiente para os índices, o que poderá ocasionar em um uso ineficiente da memória. A Hash Table é amplamente utilizada na computação, sobretudo em áreas que exigem um acesso rápido aos dados, como sistemas de cach, banco de dados e dicionários.

B. Árvore Balanceada

A árvore AVL é um um tipo de árvore binária que mantém a estrutura balanceada, isto significa que seus lados possuirão o mesmo tamanho ou no máximo a diferença de um elemento. Esta particularidade garante uma melhor eficiência para operações de busca, inserção e remoção. Em 1962 dois matemáticos russos, Adelson-Velsky e Landis, se tornaram os criadores deste modelo de estrutura de dados. Como se trata de uma estrutura de árvore balanceada, sua altura máxima cresce na ordem de $O(\log n)$, onde n é o número de nós. Com isso garante-se que as operações de de busca, inserção e exclusão também sejam de mesma ordem. Uma ação necessária para este modelo de estrutura é a rotação, que consiste no rebalanceamento da árvore, se dá sempre que uma inserção ou remoção é feita, causando um desbalanceamento, esta ação garante que a árvore permanecerá balanceada, havendo quatro tipos de rotação, à direta, à esquerda, dupla à direta e dupla à esquerda. A vantagem em se utilizar árvore balanceada está na eficiência em relação às árvores desbalanceadas (binárias), outra vantagem está no seu grande desempenho em acesso aos

dados. A desvantagem deste modelo está no fato de que poderá ser necessário efetuar a operação de rotação no momento de inserção ou exclusão frequentes, ocasionando em um baixo desempenho; o que deverá considerar-se neste modelo.

C. UUID

Com o crescente número de sistemas, acompanhado de um crescimento exponencial de dados, houve a necessidade em identificar essas informações com uma chave única, e que em sua contrução seja possível de identificar um marco temporal [4] [7] e de localização na geração desta chave. Para tanto, neste trabalho, foi utilizada a **UUID** (Universally Unique Identifier) para identificação única. Este recursos é amplamente utilizado em sistemas de banco de dados distribuídos, em APIs para garantir que cada acesso seja único a um servidor central. Suas principais características estão no formato de 128 bits, representado de forma hexadecimal; foi projetado para ser único; possui cinco versões diferentes, onde cada uma possui características específicas; por último, é portável, podendo ser utilizado em qualquer linguagem de programação. Para este trabalho será utilizada a versão 1, que é baseada no timestamp (60 bits) e no endereço MAC (48 bits) do hardware onde será gerado.

II. Metodologia

Para critérios de comparação [1] entre as duas estruturas de dados, Hash Table e Árvore AVL, foi comparado o tempo de inserção dos dados na estrutura e tempo de busca por chaves escolhidas de forma arbitrária. Este estudo adota uma abordagem experimental com foco quantitativo, utilizando medições controladas para avaliar a eficiência de manipulação das estruturas de dados.

A. Equipamento Utilizado

- Processador Intel(R) Core(TM) i7-8550U CPU 1.80GHz 1.99 GHz
- RAM instalada 16,0 GB (utilizável: 15,9 GB)

B. Software Utilizado

- Tipo de sistema Sistema operacional de 64 bits, processador baseado em x64 Windows 10 Home Single Language
- Linguagem de programação Python na versão 3.13.0
- Ambiente de desenvolvimento integrado VSCode na versão 1.95.3

C. Base de Dados

Para fins de cálculo da receita corrente líquida por parte dos entes conforme estabelecido no § 16 do art. 166 da Constituição Federal, são disponibilizados os demonstrativos das Emendas Parlamentares Individuais e de Bancada em nossa página na internet [8]. Esta série de dados abertos foi criada para divulgar de forma mais detalhada as informações referentes às emendas parlamentares individuais e de bancada. Foi feito download da base de dados em formato CSV, onde sua última atualização por parte do Ministério da Fazenda foi feita em 22 de novembro de 2024.

D. Descrição dos Procedimentos

Utilizando a base de dados adquirida a partir da base de dados abertos do governo federal. Onde, para cada CNPJ (cadastro nacional de pessoa jurídica), foi gerado uma chave UUID única para servir como base de teste e como próprio dado usado nas estruturas de dados propostas. Este conjunto de dados contém 295935 registros, no formato de linha e separados por ponto e vírgula; um arquivo CSV.

Foi feito um algorítimo em Python para manipulação dessas informações da seguinte forma:

1) Abrir o arquivo CSV, e para cada linha, adquirir o valor do CNPJ que encontra-se na nona posição. Com base neste CNPJ criar uma chave UUID;
2) Durante a interação do arquivo, escolher cinco chaves para que nas etapas seguintes sejam usadas para busca. Essas chaves estão nas posições 60000, 12000, 18000, 240000 e última posição, respectivamente.
3) Com base no vetor de chaves UUID criado, um loop neste vetor é feito e então cria-se a árvore balanceada, utilizando um outro algorítimo próprio. Foi utilizado o critério de valor *time low* que é a primeira parte da composição do UUID, pois é um marco temporal único; usando, naturalemnte, a mesma massa de dados e contando o tempo de inserção. Marcando o tempo inicial e final de toda inserção de dados na árvore AVL.
4) Para criação da Hash Table um terceiro algorítimo próprio foi usado. Para este trabalho, a função Hash exite que o valor seja um número, retornando uma posição na tabela com base em um cálculo, onde o mesmo *time low* foi utilizado na estrutura anterior. Foram criadas nove tabelas Hash, onde a primeira contém 10000 posições de índex e a última 9000; esta decisão foi tomada para observar o tempo de inserção e de busca para tabelas hashs em diferentes tamanhos. Durante esta etapa de criação da tabela Hash, foi contado o tempo para de completa inserção de dados.
5) Posteriormente, a busca de cinco chaves escolhidas de maneira arbitrária, que se encontram nas posições citadas acima. Esta escolha se deu com o intuito em observar o resultado das pesquisas, na hipótese em que a posição poderá influenciar no tempo de busca para cada uma das estruturas de dados propostas. Para cada uma destas buscas foi registrado o tempo de pesquisa.
6) Por último, duas tabelas contendo os resultados das inserções e buscas foram criadas e mostradas em log de console do projeto.

III. Resultados

Para obtenção dos resultados na comparação feita entre a as estruturas de dados Hash Table e árvore balanceada, foi utilizada a linguagem de programação Python, na versão 3. Para construção da tabela Hash e árvore AVL foi marcado o tempo de execução, ou seja, um tempo zero no início da inserção e marcado o tempo final do último registro. Os seguintes resultados foram encontrados:

A. Inserção

Para os experimentos de inserção feitas nas estruturas de dados, observamos os resultados na Tabela 1, contendo a árvore AVL e as tabelas. Da mesma forma gráfica na Fig. 1. Tempo em segundos.

TABLE I
TEMPO DE INSERÇÃO

Tamanho	Tempo(s)
AVL	3,3182
10000	1,0580
20000	0,9761
30000	1,0079
40000	1,0547
50000	1,1572
60000	0,8976
70000	1,2523
80000	0,8471
90000	1,0504

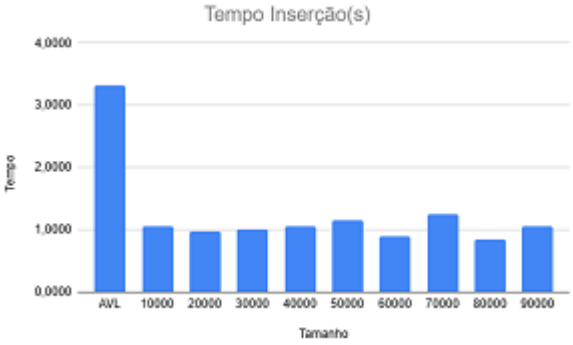

Fig. 1. Gráfico de tempo de inserção dos dados.

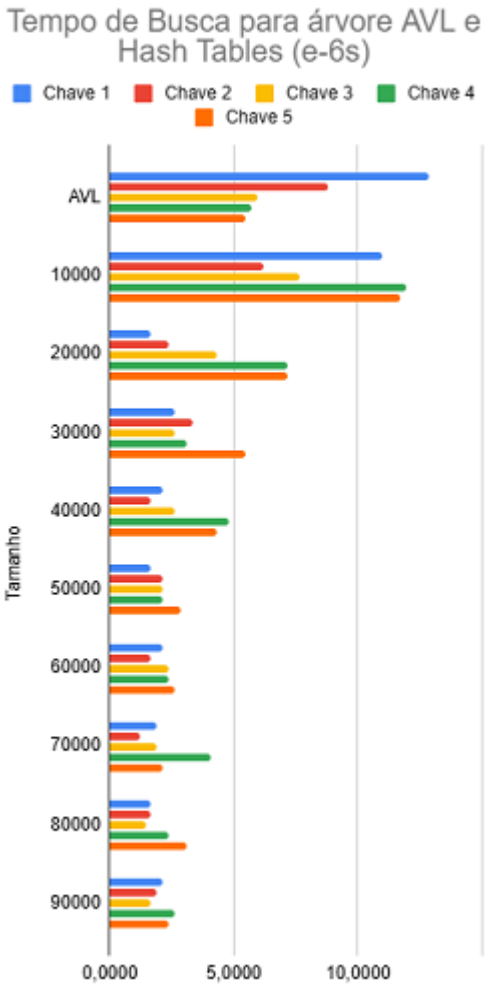

Fig. 2. Gráfico de tempo de busca.

B. Busca

Os tempos de busca para as duas estruturas de dados estão dispostas na Tabela 2 e Fig. 2. Com o intuito em tornar a leitura dos resultados mais fáceis os números foram truncados. No entanto, é válido mencionar que todos estes resultados encontram-se na ordem de e-6. Onde o tempo é medido em segundos:

TABLE II
TEMPO DE PESQUISA PARA ÁRVORE E TABELAS

Tamanho	Chave 1	Chave 2	Chave 3	Chave 4	Chave 5
AVL	12,8746	8,8215	5,9605	5,7220	5,4836
10000	10,9673	6,1989	7,6294	11,9209	11,6825
20000	1,6689	2,3842	4,2915	7,1526	7,1526
30000	2,6226	3,3379	2,6226	3,0994	5,4836
40000	2,1458	1,6689	2,6226	4,7684	4,2915
50000	1,6689	2,1458	2,1458	2,1458	2,8610
60000	2,1458	1,6689	2,3842	2,3842	2,6226
70000	1,9073	1,1921	1,9073	4,0531	2,1458
80000	1,6689	1,6689	1,4305	2,3842	3,0994
90000	2,1458	1,9073	1,6689	2,6226	2,3842

C. Complexidade

Abaixo, na Tabela 3, observamos a complexidade de inserção, exclusão e busca para cada uma das estruturas de dados propostas:

TABLE III
COMPLEXIDADE DAS ESTRUTURAS DE DADOS

Ação	AVL	Hash Table
Inserção	$O(\log n)$	$O(1)$
Busca	$O(\log n)$	$O(1)$
Exclusão	$O(\log n)$	$O(1)$

IV. DISCUSSÃO

Tratando-se da operação de inserção, há uma grande diferença entre as duas estruturas. Onde a inserção chega a ser três vezes mais rápida para tabelas Hash em relação à árvore AVL [5]. Outro aspecto interessante para observar é o fato de que não houve grande mudança no tempo de inserção paras as nove tabelas hashs criadas. Houve uma pequena variação, no máximo em vinte por cento, para as tabelas.

Os resultados mostram que para uma mesma massa de dados, neste caso cerca de 300 mil registros no formato UUID a tabela Hash se mostrou mais eficiente a medida que seu tamanho era incrementado, isto é, o número de índices. A estrutura de dados de árvore balanceada, que também possui uma grande eficiência, se mostrou equiparada somente à primeira tabela, onde há 10 mil posições para mesma massa de dados. Todas essas observações tratam-se a cerca da busca de dados. Também percebeu-se que não há grande diferença na busca de chaves para tabelas acima de 50000 posições; assim podendo levar em consideração uma tabela menor com a mesma eficiência e consumo moderado de memória. Bem como não houve diferenças consideráveis pelo valor das chaves e suas posições no array; isso implica que não é relevante se a chave está no início ou fim do array, evidenciando a complexidade $O(1)$ para tabelas Hash.

V. Conclusão

Os experimentos confirmaram que as Hash Tables são mais eficientes em operações de inserção e busca devido à sua complexidade constante $O(1)$, especialmente em cenários onde o aumento do tamanho da tabela minimiza colisões. Já as Árvores Balanceadas (AVL) demonstraram um desempenho consistente, mas inferior às Hash Tables, principalmente em situações de grandes volumes de dados. Contudo, a AVL mantém sua relevância em cenários onde a ordenação dos dados é fundamental. Portanto, a escolha entre essas estruturas deve ser orientada pelos requisitos específicos de cada aplicação, equilibrando eficiência e adequação funcional.

References

[1] Neyer, Mark P. (2009). *A Comparison of Dictionary Implementations*. Publisher: April.

[2] Saxena, Akshay; Anand, Harsh; Pradhan, Tribikram; Mishra, Satya Ranjan (2015). *A Hybrid Chaining Model with AVL and Binary Search Tree to Enhance Search Speed in Hashing*. International Journal of Hybrid Information Technology, Vol. 8, No. 3, pp. 185–194.

[3] Saini, Lalit Kumar; Verma, Deepak (2019). *Algorithmic Analysis of Advancement in Chaining Method With BST & AVL ISSN*. Publisher: JETIR.

[4] Agarwal, Arushi; Pathak, Sashakt; Agarwal, Sakshi (2020). *MN Hashing: Search Time Optimization with Collision Resolution Using Balanced Tree*. In Proceedings of Futuristic Trends in Networks and Computing Technologies: Second International Conference, FTNCT 2019, Chandigarh, India, November 22–23, 2019, Revised Selected Papers 2, pp. 196–209. Springer.

[5] Robenek, Daniel; Platos, Jan; Snasel, Vaclav (2013). *Efficient In-memory Data Structures for n-grams Indexing*. In Proceedings of DATESO, pp. 48–58. Citeseer.

[6] Tapia-Fernández, Santiago; García-García, Daniel; García-Hernandez, Pablo (2022). *Key Concepts, Weakness and Benchmark on Hash Table Data Structures*. Algorithms, Vol. 15, No. 3, Article 100. MDPI.

[7] Zou, Ligeng; Zhang, Zuping; Long, Jun (2015). *A fast incremental algorithm for constructing concept lattices*. Expert Systems with Applications, Vol. 42, No. 9, pp. 4474–4481. Elsevier.

[8] Ministério da Fazenda, "Emendas Parlamentares Individuais e de bancada", Portal do Governo, 2024. [Online]. Disponível em: https://dados.gov.br/dados/conjuntos-dados/emendas-parlamentares. [Acessado: 27 nov. 2024].