UNIVERSIDADE ESTADUAL DA PARAÍBA

BACHARELADO EM CIÊNCIAS DA COMPUTAÇÃO

ERIC NATAN BATISTA TORRES

JASMINE MARIA DE ALMEIDA RIBEIRO

SONALLY CECÍLIA DANTAS SALES

THALES HENRIQUE DE SOUZA TELES

YASMIN PEREIRA TRAVASSOS RAMOS

GIOVANNA DONATO SANTIAGO CARRAZZONI

RELATÓRIO PROJETO DE ESTRUTURA DE DADOS SOBRE TABELAS HASH

CAMPINA GRANDE

2024

SUMÁRIO

INTRODUÇÃO

As Tabelas Hash são uma estrutura de dados amplamente utilizada na ciência da computação para armazenar e recuperar dados de maneira eficiente. Diferentemente das estruturas de dados lineares, como pilhas e filas, as tabelas hash utilizam uma função(função hash) que, através do conteúdo de uma informação faz o mapeamento desse elemento diretamente para um índice de um array, permitindo o acesso imediato aos elementos. É nessa característica que se encontra a vantagem da tabela hash, pois seu desempenho de busca, inserção e remoção é, idealmente, constante, ou seja, O(1), independente do tamanho dos dados.

Apesar de, na teoria, a tabela hash ser eficiente, colisões(quando dois elementos recebem o mesmo índice) são inevitáveis, pois frequentemente o número de elementos é maior que o número de espaços na tabela. Essas colisões são tratadas através de métodos como o encadeamento(usada no endereçamento fechado) ou a busca de novas posições para os elementos na tabela(endereçamento aberto). Além de lidar com essas colisões, a tabela hash pode se redimensionar, aumentando sua capacidade, caso esteja muito cheia, em uma operação denominada rehashing, que aumenta a capacidade da tabela para manter o desempenho ideal.

Essas propriedades tornam as tabelas hash especialmente úteis em aplicações que exigem acesso rápido e eficiente aos dados, como sistemas de banco de dados, caches, tabelas de simbolização em compiladores e dicionários. Contudo, seu desempenho depende de diversos fatores, como a qualidade da função de hash, o método de resolução de colisões e a manutenção do fator de carga. Comparativamente a outras estruturas, como árvores ou listas, as tabelas hash são menos indicadas quando há necessidade de percorrer ou ordenar os elementos, mas sua eficiência para operações diretas as torna uma escolha primordial em muitos cenários computacionais.

METODOLOGIA

A metodologia deste relatório consiste em analisar o desempenho de Tabelas Hash com endereçamento fechado e endereçamento aberto, considerando as estratégias de sondagem linear, dupla e quadrática. Como critérios de avaliação, foram utilizados o tempo de inserção de elementos na tabela, o tempo de busca, o número de colisões (quando duas ou mais chaves geram o mesmo endereço na tabela) e a quantidade de operações de rehashing realizadas nas tabelas de endereçamento aberto.

Para a análise, utilizamos um conjunto de dados com 300.000 elementos únicos no intervalo de 0 a 1.000.000 para os testes de inserção. Já para os testes de busca, utilizamos um conjunto de 500.000 elementos, incluindo repetições, no mesmo intervalo.

Os experimentos foram realizados em um notebook Samsung com as seguintes especificações: processador 11th Gen Intel(R) Core(TM) i3-1115G4 @ 3.00GHz, 12 GB de memória RAM DDR4 (8 GB + 4 GB) a 2666 MHz, SSD de 256 GB, gráficos Intel(R) UHD Graphics com DirectX 12 e sistema operacional Windows 11, 64 bits. Essas especificações garantem uma base estável para a execução dos testes e coleta de resultados.

O código-fonte utilizado para os experimentos está disponível no repositório indicado na seção de referências deste relatório. Para realizar o perfilamento dos algoritmos, utilizamos o IntelliJ IDE, que oferece ferramentas detalhadas para a análise de desempenho. Essa ferramenta foi essencial para identificar gargalos e avaliar cada operação de forma precisa, permitindo uma análise detalhada do comportamento das Tabelas Hash.

TABELA HASH

- **Conceito**

Uma Tabela Hash é uma estrutura de dados que associa chaves a valores, permitindo buscas, inserções e remoções eficientes. A estrutura utiliza uma **função de hash**, que mapeia as chaves para endereços específicos de uma tabela, onde os valores correspondentes são armazenados. A principal característica da Tabela Hash é que a função de hash distribui as chaves de forma a minimizar colisões, ou seja, o número de vezes que duas chaves diferentes mapeiam para o mesmo endereço.

Apesar da eficiência média no acesso aos dados ser *O(1)*, a estrutura não é imune a colisões, que ocorrem quando duas ou mais chaves são mapeadas para o mesmo índice.

Em comparação com outras estruturas como a **Tabela de Acesso Direto**, a Tabela Hash é mais eficiente no uso de memória. Em vez de ocupar $|U|$ posições, sendo U o universo de todas as chaves possíveis, a Tabela Hash utiliza um espaço proporcional ao número de chaves armazenadas ($|K|$), ou seja, $O(|K|)$. Essa eficiência é alcançada porque o elemento com chave *k* é guardado na posição *h(k)*, onde *h* é a função hash.

A função hash desempenha um papel crucial na eficácia da Tabela Hash, devendo ser determinística, para que sempre gere o mesmo índice para uma mesma chave, e eficiente, para distribuir as chaves de maneira uniforme. Por essas características, as Tabelas Hash são amplamente utilizadas em diversas aplicações, como implementações de dicionários, caches, bancos de dados e até em sistemas de compilação, onde acesso rápido por chave é fundamental

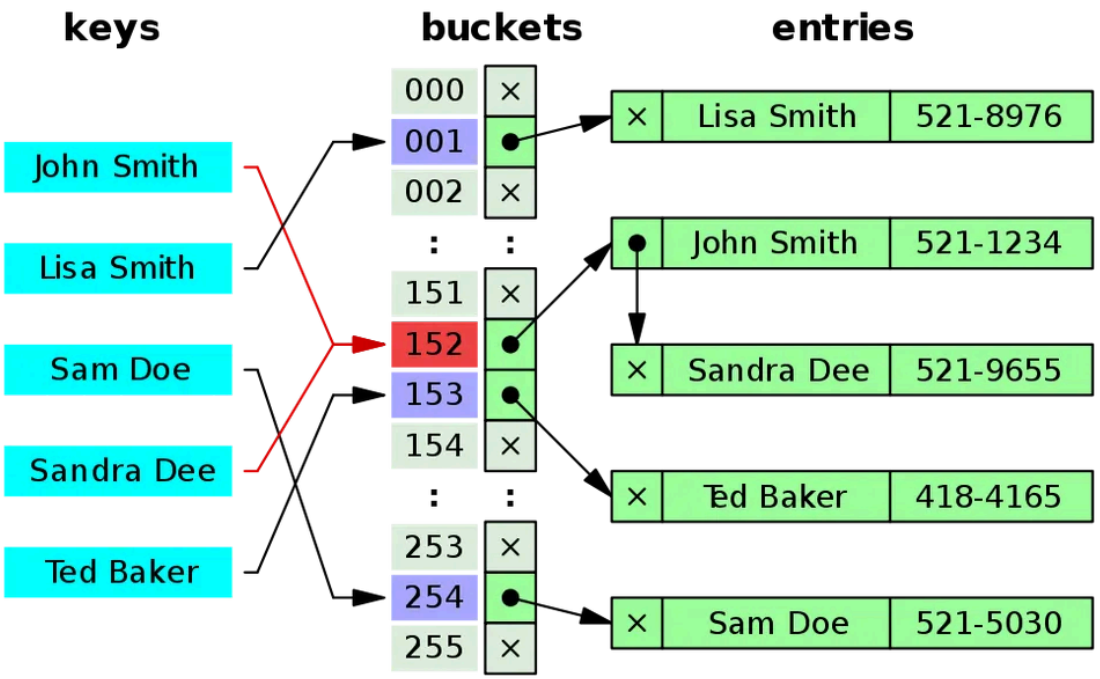


Figura 1 - Exemplo de tabela hash. Fonte: (https://www.techtudo.com.br/noticias/2012/07/o-que-e-hash.ghtml, 2024).

- **Estrutura**

Uma Tabela Hash consiste em um array onde cada índice pode armazenar uma entrada (chave-valor). Cada entrada é associada a um índice específico calculado por uma função de hash. A função de hash é crucial para a eficiência da Tabela Hash, pois ela determina como as chaves serão mapeadas nos índices do array. A tabela de hash é inicialmente configurada com um tamanho fixo, mas pode ser redimensionada conforme necessário, especialmente em implementações que priorizam alta eficiência e a minimização de colisões. Esse redimensionamento geralmente ocorre quando o **fator de carga** da tabela (isto é, a razão entre o número de elementos armazenados e o número de slots disponíveis) atinge um limiar crítico. Esse processo, conhecido como **rehashing**, envolve aumentar o tamanho da tabela, expandindo o array para o próximo número primo maior que o dobro do seu tamanho atual, além de redistribuir os elementos na nova tabela para garantir a manutenção da eficiência nas operações de inserção, busca e remoção.

A estrutura básica de uma tabela hash consiste em:

- **Array (Vetor)**: Um array onde os elementos serão armazenados. Cada posição do array é chamada de **slot** ou **balde**.
- **Função de Hash**: Uma função que recebe uma chave (geralmente uma string ou número) e a transforma em um índice do array. Esse índice determina onde o valor correspondente será armazenado no array.

```
private int hash(int key) {
    return key % M;
}
```

Figura 2 - Exemplo de função modular tabela hash. Fonte: (https://www.ime.usp.br/~pf/estruturas-de-dados/aulas/st-hash.html, 2024).

- **Gestão de Colisões:** Como diferentes chaves podem gerar o mesmo índice (chamado de colisão), é necessário um método eficaz para lidar com essas situações, garantindo que todos os elementos sejam armazenados corretamente e possam ser recuperados. Uma tabela hash ideal seria aquela sem colisões, mas isso é impossível quando o número de chaves possíveis ($|U|$) é maior do que o tamanho da tabela (m), já que duas ou mais chaves podem ser mapeadas para o mesmo índice.

Existem duas abordagens principais para resolver colisões:

1. **Encadeamento (Chaining):** Também conhecido como endereçamento fechado, cada slot do array aponta para uma lista ligada que armazena todos os elementos que possuem o mesmo índice gerado pela função de hash. Quando uma colisão ocorre, o novo elemento é simplesmente adicionado à lista correspondente. Geralmente, os elementos são inseridos na cabeça da lista, com base na ideia de que itens recentemente inseridos têm maior probabilidade de serem acessados em breve, o que pode melhorar o desempenho em alguns casos.

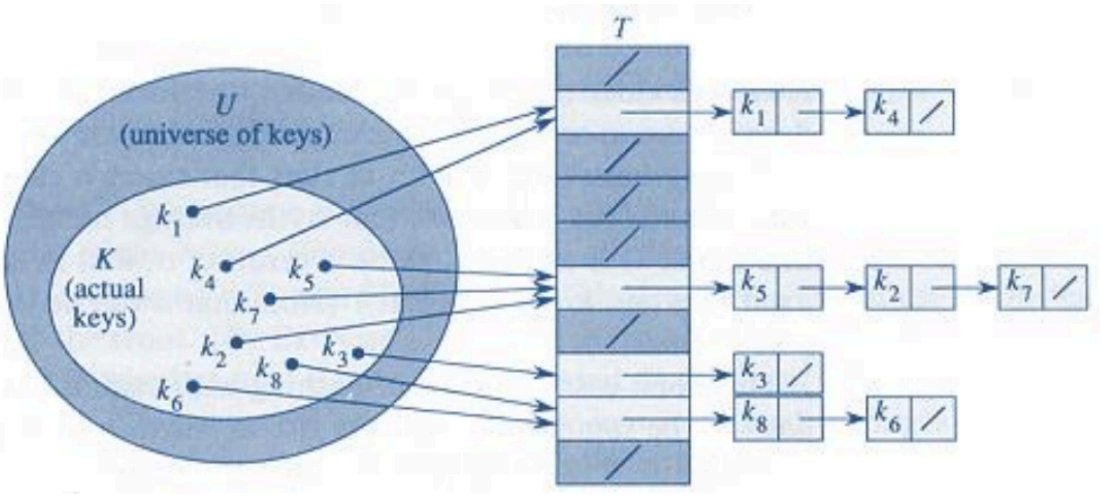


Figura 3 - Exemplo de endereçamento fechado usando lista encadeada Fonte: (https://suap.uepb.edu.br/media/edu/material_aula/EDA-10_TabelasHash-cca0acba6ce9453ab53bcc8dfb7c0009.pdf, 2024).

2. **Endereçamento Aberto (Open Addressing):** Nesse método, em vez de usar listas ligadas, a tabela busca outro slot livre na própria tabela seguindo uma sequência predefinida. Existem diferentes estratégias para determinar essa sequência:
    - **Sondagem Linear:** O próximo slot é verificado de forma sequencial até encontrar um espaço vazio.
    - **Sondagem Quadrática:** O deslocamento cresce de forma quadrática (por exemplo, 1, 4, 9...) para evitar o agrupamento de colisões.
    - **Duplo Hashing:** Uma segunda função de hash é utilizada para calcular o deslocamento em caso de colisão, oferecendo maior dispersão e reduzindo o número de colisões consecutivas.

- **Funcionamento**

O funcionamento da Tabela Hash depende de uma função de hash que, ao receber uma chave, retorna um índice onde o valor será armazenado. A função de hash é fundamental para o desempenho da tabela, pois sua qualidade determina a distribuição das chaves e, consequentemente, a eficiência nas operações de busca, inserção e remoção.

Para a inserção, a chave é passada para a função de hash, que retorna um índice na tabela, onde o valor correspondente será armazenado. Caso ocorra uma colisão (quando duas ou mais chaves diferentes mapeiam para o mesmo índice), a tabela pode lidar com isso de diferentes formas:

Encadeamento: Neste método, cada posição da tabela contém uma lista de entradas, onde todas as chaves que mapeiam para o mesmo índice são armazenadas.

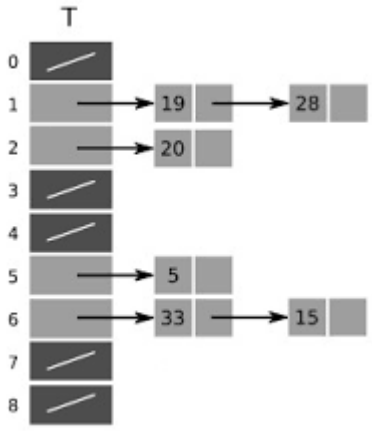


Figura 4 - Exemplo do uso do método de encadeamento tabela hash. Fonte: (https://www2.unifap.br/furtado/files/2016/11/Aula7.pdf, 2024).

Sondagem Linear: Se um índice já estiver ocupado, a tabela procura o próximo índice livre de forma sequencial. Embora simples, esse método pode levar a aglomerações, onde várias posições consecutivas são ocupadas, aumentando o tempo de busca.

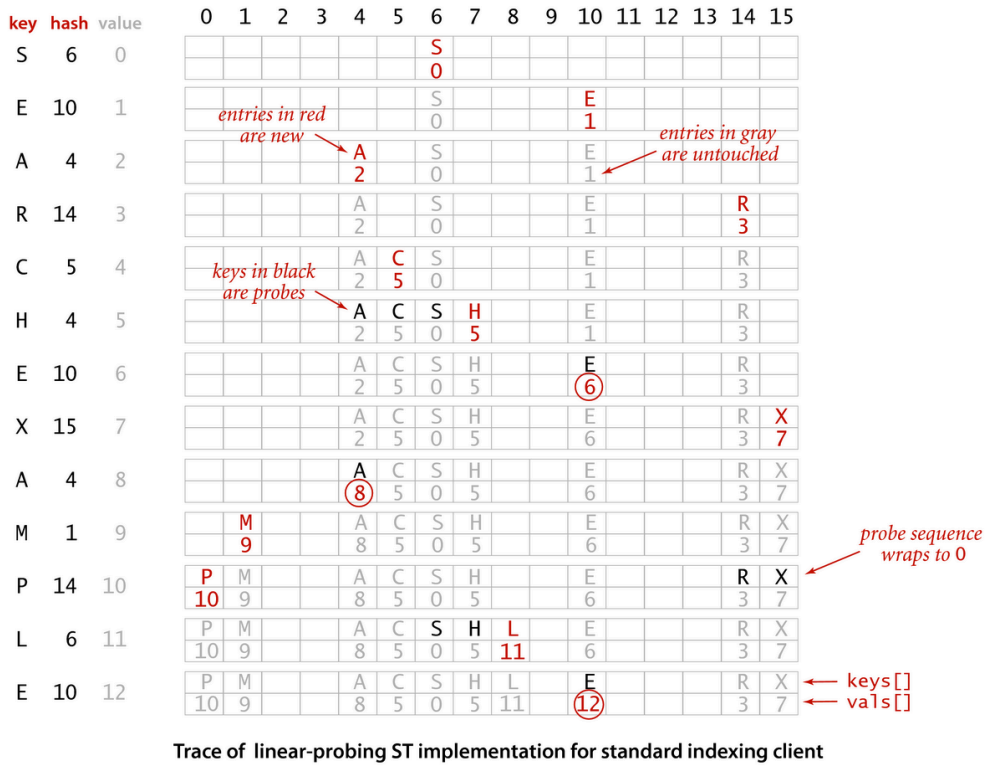


Figura 5 - Exemplo do uso do método sondagem linear tabela hash. Fonte: (https://www.ime.usp.br/~pf/estruturas-de-dados/aulas/st-hash.html, 2024).

- **Função Hash**

A escolha de uma boa função de hash é essencial para minimizar o número de colisões e garantir um desempenho uniforme. A função deve satisfazer a hipótese de hashing uniforme, ou seja, qualquer chave deve ter a mesma probabilidade de ser mapeada para qualquer um dos mmm índices da tabela. Embora ideal, essa uniformidade é difícil de alcançar na prática, mas existem métodos que ajudam a aproximar essa distribuição:

➔ **Método da Divisão:**

- A função é definida como $h(k) = k \bmod m$, onde k é a chave e m o número de posições na tabela.
- É simples e rápido, exigindo apenas uma operação de divisão.
- Recomendações:

- Escolha *m* como um número primo para reduzir a ocorrência de padrões nas distribuições das chaves.
- Evite valores de *m* que sejam potências de 2, pois isso pode levar a distribuições desbalanceadas.

➔ **Método da Multiplicação:**

- ◆ A função é definida como $h(k) = \lfloor m \cdot (k \cdot A \bmod 1) \rfloor$, onde A é uma constante entre 0 e 1, geralmente escolhida como $A \approx (\sqrt{5}-1)/2 \approx 0{,}6180339887$.
- ◆ Esse método é menos sensível à escolha de *m*, permitindo que seja uma potência de *2(m=2^p)*.
- ◆ Embora um pouco mais lento que o método da divisão, ele tende a gerar distribuições mais uniformes.

Ambos os métodos assumem que as chaves podem ser representadas como números, o que é prático, pois strings e outros tipos de dados podem ser facilmente convertidos para valores numéricos. A escolha entre os métodos depende do caso de uso, sendo o da divisão preferido para implementações mais rápidas e o da multiplicação para maior uniformidade em cenários com distribuições imprevisíveis de chaves.

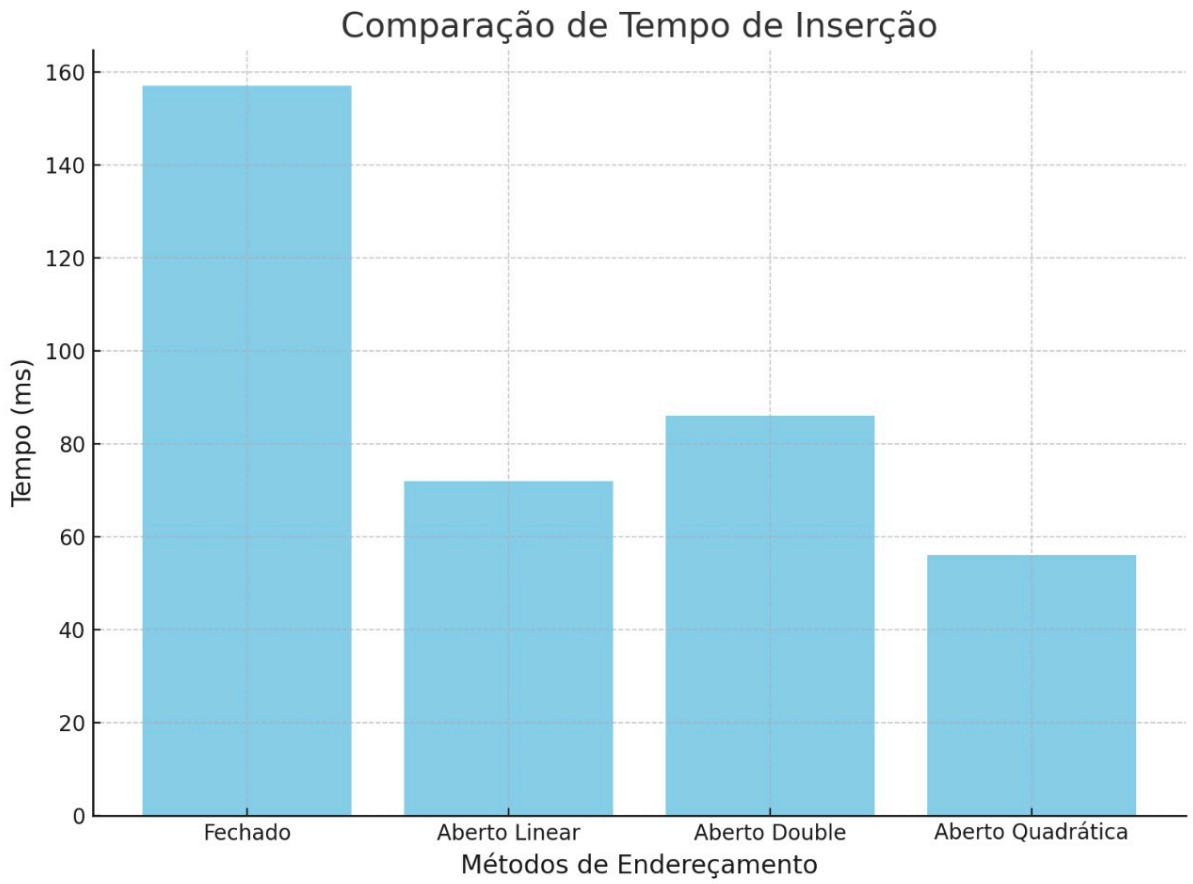
ANÁLISES

Quadro 1 - Comparação do tempo de inserção entre os algoritmos de hash.

Fechado	Aberto Linear	Aberto Double	Aberto Quadrática
157	72	86	56

Fonte: Elaboração própria com os dados analisados. (2024)

Gráfico 1 - Comparação do tempo de inserção entre os algoritmos de hash.



Fonte: Elaboração própria com os dados analisados. (2024)

Analisando o gráfico e o quadro 1, podemos perceber que o tempo de inserção segue os resultados esperados, ou seja, houve um tempo maior na tabela de endereçamento fechado, apresentando um resultado de 157 milissegundos. Isso ocorre porque, quando há uma colisão, ou seja, quando duas ou mais chaves geram o mesmo endereço na tabela, é necessário adicionar o novo elemento à lista vinculada no índice correspondente, o que implica em operações adicionais, como a alocação de memória para os valores colididos. Por outro lado, as tabelas de endereçamento aberto mantêm todos os dados dentro da própria tabela, reduzindo os custos das operações adicionais(as operações adicionais serão os próximos deslocamentos dentro do próprio vetor).

Também podemos analisar o desempenho das tabelas de endereçamento aberto.

A tabela de endereçamento duplo foi que apresentou o segundo maior tempo de inserção entre eles, mostrando que com a função de hash duplo, o algoritmo deve que percorrer mais posições no array para conseguir inserir o elemento na tabela, isso pode ter acontecido por conta dos valores-chave utilizados nos teste.

A tabela de endereçamento linear apresentou o segundo maior tempo de inserção, o que faz sentido já que a função hash percorrerá cada célula da tabela de maneira linear, no pior caso, adicionar um elemento de hash 0 numa tabela com *n* posições ocupadas, o algoritmo teria que percorrer *n* posições para conseguir adicionar o elemento.

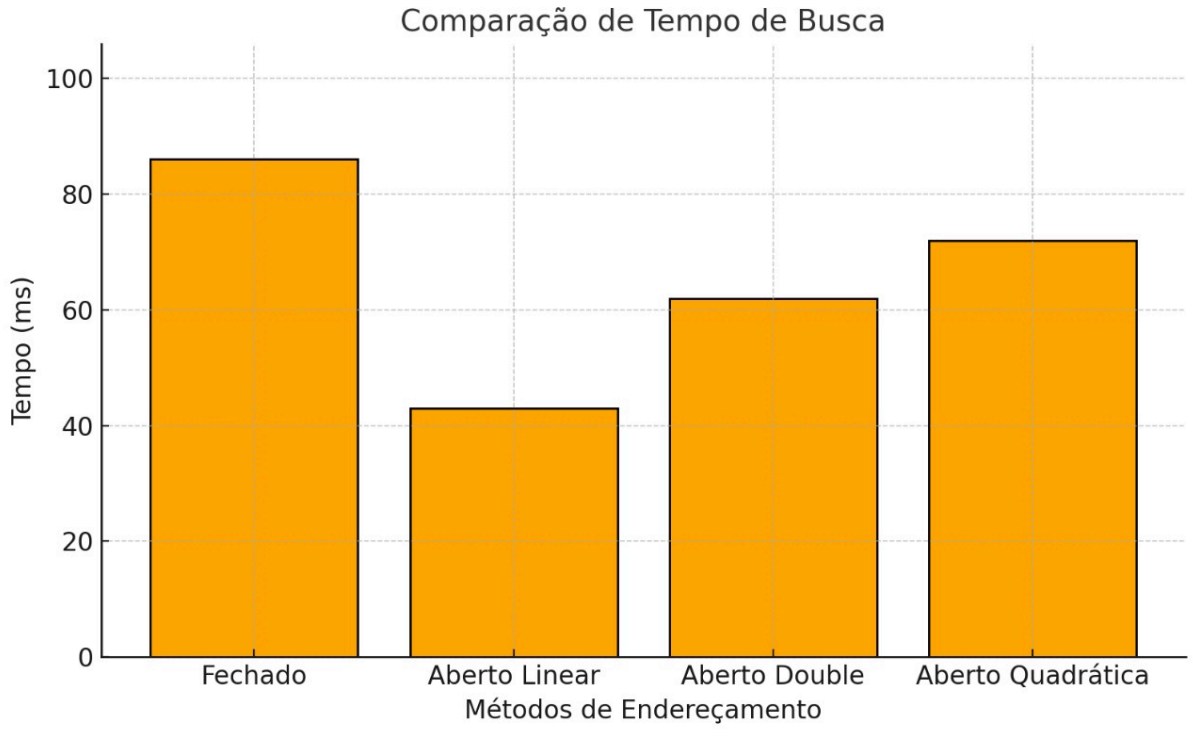
Agora nas tabelas de endereçamento duplo e fechado, a tabela de endereçamento quadrático foi a que teve o melhor desempenho, mostrando que a distribuição utilizando uma função quadrática foi a que sofreu menos colisões, o que foge do esperado, tendo em vista que o número de colisões no endereçamento quadrático foi maior do que o de endereçamento duplo. Como é mostrado no quadro 2 e no gráfico 2 que estão exibidos abaixo.

Quadro 2 - Comparação do tempo de busca entre os algoritmos de hash.

Fechado	Aberto Linear	Aberto Double	Aberto Quadrática
86	43	62	72

Fonte: Elaboração própria com os dados analisados. (2024)

Gráfico 2 - Comparação do tempo de busca entre os algoritmos de hash.



Fonte: Elaboração própria com os dados analisados. (2024)

Analisando o gráfico 2, percebemos que o Endereçamento Fechado apresentou um tempo de busca elevado comparado aos de Endereçamento Aberto, mesmo não sendo tão alto (aprox. 600 ms) quando comparamos com uma busca linear numa lista com o mesmo tamanho da tabela hash. Isso é justificado pelo algoritmo de Endereçamento Fechado utilizar a busca numa lista encadeada para encontrar os elementos que foram mapeados para as mesmas posições, tornando um pouco mais custosa uma varredura na lista encadeada do que num vetor.

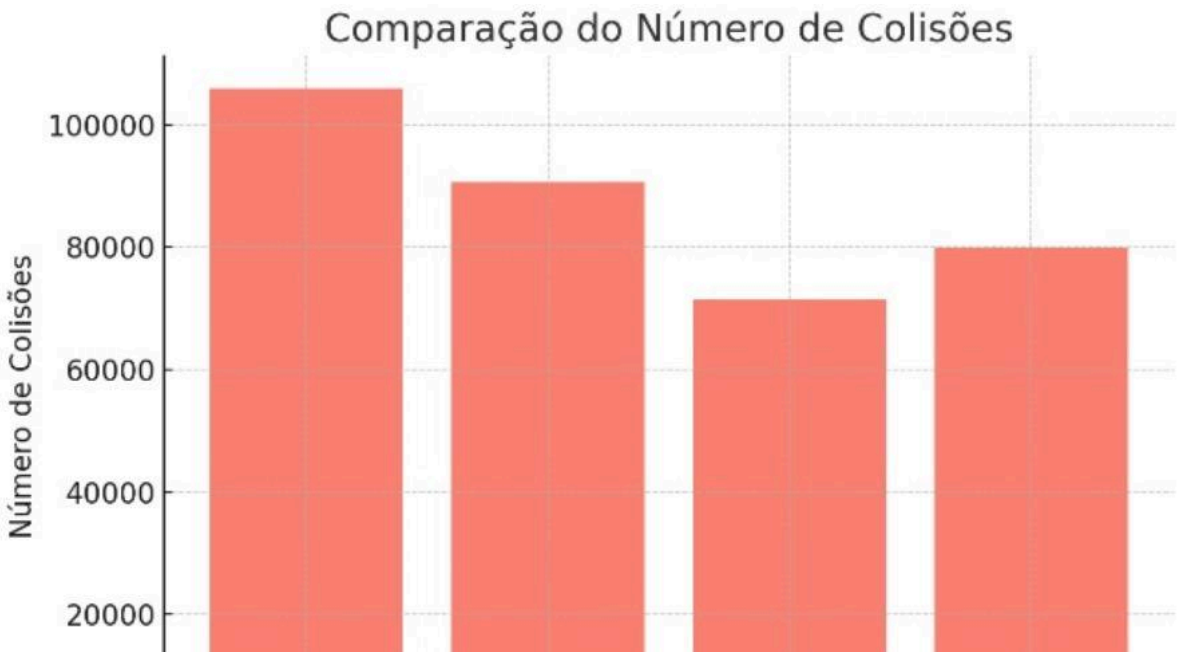
Comparando os algoritmos de Endereçamento Aberto, o Linear e o Duplo apresentaram os menores tempos de busca o endereçamento linear com 43 milissegundos, o que vai de acordo com os resultados esperados, pois o endereçamento linear realiza menos operações devido à sua simplicidade e eficiência na localização de itens. Isso ocorre porque, quando um colapso de hash ocorre, o algoritmo de busca linear verifica sequencialmente as entradas da tabela, o que resulta em uma média de tempo de busca linear próximo ao tempo constante ideal.

Quadro 3 - Quantidade de colisões de cada algoritmos.

Fechado	**Aberto Linear**	**Aberto Duplo**	**Aberto Quadrático**
275000	89684	70462	78068

Fonte: Elaboração própria com os dados analisados. (2024)

Gráfico 3 - Comparação do número de colisões entre os algoritmos de hash.



Fonte: Elaboração própria com os dados analisados. (2024)

Com base no gráfico 3 e no quadro 3, percebemos que o algoritmo que apresentou o maior número de colisões foi de Endereçamento Fechado. Este resultado é o esperado, pois como temos um fator de carga igual a 12 e 300.000 elementos foram inseridos, o número de células na tabela será de 25.000.
Resolvendo uma equação simples, conseguimos descobrir o número de células necessárias na tabela:

- $\alpha = \frac{n}{m} \Rightarrow 12 = \frac{300.000}{m} \Rightarrow m = \frac{300.000}{12} \Rightarrow m = 25.000$
- $\alpha: fator\ de\ carga = 12$,
- n: $número\ de\ chaves = 300.000$,
- m: $número\ de\ células\ na\ tabelas = 25.000$

Portanto, após os primeiros 25.000 elementos inseridos, os 275.000 restantes, obrigatoriamente, sofreram colisões. Com isso, percebemos que o fator de carga escolhido impacta diretamente no número de colisões de um algoritmo de Endereçamento Fechado. Caso fosse escolhido um fato de carga menor, o número de colisões também seria menor.

Comparando as colisões dos Endereçamentos Abertos percebemos que o algoritmo de Endereçamento Duplo apresentou o menor número de colisões, mostrando que utilizando uma segunda função de hashing para aumentar o tamanho do “passo” na inserção de um elemento na tabela, o número de colisões foi mínimo para os dados utilizados nos testes, porém o tempo de inserção, quando comparado aos outros algoritmos de endereçamento, foi o dobro.

CONSIDERAÇÕES FINAIS

Portanto, podemos concluir que na escolha entre o Endereçamento Aberto e Fechado, alguns pontos devem ser considerados, a simplicidade de implementação e velocidade do endereçamento fechado é interessante quando possuímos um pequeno número de chaves, tornando a busca dentro das listas encadeadas próximas de O(1). Porém, como a base de dados utilizada possui 300.000 elementos e o fator de carga foi relativamente alto, os algoritmos de Endereçamento Aberto, mostraram-se mais interessantes ao lidarem com um grande número de dados e colisões. Apesar de algumas pequenas divergências dos resultados esperados, o Endereçamento Duplo e o Quadrático foram os melhores ao lidarem com colisões, inserções e busca de elementos.

REFERÊNCIAS

---

1. **SCHONS, L.** Tabelas Hash. Disponível em: https://medium.com/@sschonss/tabelas-hash-1f1a85a83795. Acesso em: 19 de nov. 2024.
2. **PISA, P.** O que é Hash?. Disponível em: https://www.techtudo.com.br/noticias/2012/07/o-que-e-hash.ghtml. Acesso em: 19 de nov. 2024.
3. **FEOFILLOF, P.** Hashing. Disponível em: https://www.ime.usp.br/~pf/estruturas-de-dados/aulas/st-hash.html. Acesso em: 19 de nov. 2024.
4. **BRUNET, J. A.** Tabelas Hash. Disponível em: https://joaoarthurbm.github.io/eda/posts/hashtable/#:~:text=Tabela%20hash%20é%20uma%20estrutura,em%20tempo%20constante%20e%20uniforme. Acesso em: 19 de nov. 2024.

5. **WIKIPEDIA.** Tabela de dispersão. Disponível em: https://pt.wikipedia.org/wiki/Tabela_de_dispersão. Acesso em: 19 de nov. 2024.
6. **JASON, J.** Tabela de dispersão. Disponível em: https://suap.uepb.edu.br/media/edu/material_aula/EDA-10_TabelasHash-cca0acba6ce9453ab53bcc8dfb7c0009.pdf. Acesso em: 19 de nov. 2024.
7. Código com as implementações do algoritmo. Disponível em: https://github.com/sonallycecilia/projeto4-EDA