Projeto MC458: Implementação e Análise de Estruturas para Matrizes Esparsas

MC458 - Projeto e Análise de Algoritmos I

Prof. Santiago V. Ravelo

Autores:

Isabel Cristina Marras Salles
Victor Luigi Roquetto
Rafael Feltrin Lamen Rocha

Código-fonte e Anexos (GitHub):
Link

Novembro de 2025

Conteúdo

Lista de Símbolos e Abreviações

Número total de elementos não nulos de uma matriz	k
Quantidade de linhas que possuem pelo menos um elemento	l
Densidade média (número médio de elementos por linha não nula)	d
Números de buckets de um hash	m
Número de elementos não nulos em um hash	n
Fator de carga da tabela Hash ($\lambda = n/m$)	λ
Notação Big-O (Limite superior assintótico)	$O(\cdot)$
Custo amortizado da i-ésima operação	$\hat{c_i}$
Função potencial para análise amortizada	Φ
Árvore rubro negra esquerdista (Left-Leaning Red-Black Tree)	LLRBT

1 Introdução

Uma matriz esparsa é por definição uma matriz em que a maioria dos elementos é igual a zero.

Mais formalmente, uma matriz $A \in R^{n \times m}$ é considerada esparsa se o número de elementos não nulos k satisfaz a seguinte condição:

$$k \ll n \times m$$

Isto é, o número de elementos não nulos (k) é muito menor que o número total de posições possíveis ($n \times m$).

Matrizes esparsas aparecem frequentemente nos mais diversos campos da computação e matemática, por vezes representando processos reais que dependem de eficiência e otimização. Desse modo, o seguinte relatório visa, por meio da ótica experimental e teórica, implementar e comparar o uso de duas estruturas distintas que promovem o uso mais eficiente de memória e poder computacional.

2 Estruturas de Dados Adotadas

2.1 Estrutura 1: Hash

Para a primeira estrutura foi escolhido o Hash especialmente pela sua capacidade de encontrar um elemento a partir de uma chave em tempo esperado $O(1)$.

A implementação do hash foi projetada para que fosse mais eficiente recuperar todos os elementos de uma mesma linha, algo essencial para a realização da multiplicação de matrizes. Para isso, a estrutura foi organizada da seguinte forma:

- **Hash Externo (Linhas):** um vetor de buckets utilizado para o mapeamento das linhas. A partir do índice i da coordenada (i, j), a função de hash calcula em qual posição do vetor a linha deve ser armazenada. Para garantir a facilidade em encontrar todos os elementos de uma mesma linha no caso de colisões, foi utilizada uma lista ligada em que cada nó representa uma única linha e contém seu próprio hash interno.

- **Hash Interno (Colunas):** Cada nó que representa uma linha no hash externo tem sua própria tabela hash. A posição nesse hash interno é calculada a partir do índice j da coordenada (i, j). Em caso de colisões, os nós dos elementos são armazenados como uma lista ligada.

Para garantir o acesso esperado em $O(1)$ e uso de memória $O(k)$, implementou-se um redimensionamento dinâmico dos vetores das tabelas Hash, baseado no Fator de Carga $\lambda = \frac{k}{m}$, onde k representa a quantidade de itens armazenados (elementos não nulos na tabela interna ou linhas não nulas na tabela externa) e m o número total de buckets de cada tabela hash. Quando $\lambda > 0.75$, a tabela (seja ela a interna ou a externa) dobra de tamanho, tornando os elementos mais dispersos e diminuindo a probabilidade de colisão, a fim de manter o acesso eficiente. Quando $\lambda < 0.25$, o tamanho da tabela é reduzido na metade, assegurando que a estrutura não ocupe memória excessiva após operações que alterem o valor de elementos para 0.

Além do rehash, para garantir que o uso de memória seja $O(k)$, elementos com valor igual a 0 não são armazenados. Além disso, sempre que um elemento tem seu valor atualizado para 0, seu nó é apagado. Caso essa remoção torne uma linha vazia, seu hash interno é deletado e a linha removida da tabela externa.

O acesso ao elemento na coordenada (i, j) é realizado em duas etapas:

- **Busca na tabela externa (linhas):** A partir do índice i, a função de Hash calcula qual será a posição (bucket) da linha no vetor do hash externo. Caso haja mais de uma linha nessa mesma posição, a lista ligada armazenada nesse bucket é percorrida até que o nó correspondente a linha i seja encontrado.

- **Busca na tabela interna (colunas):** Após a linha ser localizada, é calculada pela função de hash qual a posição na tabela interna do elemento com coluna j. No caso de colisões, a lista ligada armazenada na posição é percorrida até encontrar o elemento com coordenada (i, j).

Caso nenhum nó com essas coordenadas seja localizado, o valor retornado é 0, afinal os elementos nulos são os que não estão armazenados na estrutura.

Apesar de ser necessário percorrer alguns elementos em casos de colisão, a dinâmica adotada com o rehash minimiza a probabilidade de que muitos elementos sejam armazenados na mesma posição. Assim, o número médio de elementos a ser percorrido é baixo, garantindo que a complexidade de tempo esperada para a operação de acesso seja $O(1)$.

A inserção/atualização segue uma lógica similar ao acesso a um elemento na posição (i, j) para encontrar a posição que deve ser modificada. Ela se dá em duas partes:

- Primeiro é localizada a posição da linha i. Caso não haja nenhuma correspondência e o valor a ser inserido seja diferente de 0, significa que a linha era nula e o elemento a ser adicionado é o primeiro, portanto um novo nó para a linha e seu hash interno deve ser criado e adicionado no hash externo. Se nenhuma correspondência tiver sido encontrada e o valor a ser inserido/atualizado seja 0, não há nada que precise ser feito.

- Caso haja uma linha correspondente, a coluna j é buscada no hash interno, similarmente ao acesso a um elemento. Se o nó para a posição (i, j) ainda não existir e o valor a ser adicionado for diferente de 0, um novo nó é criado e adicionado a lista ligada, mas se o valor for igual a 0, não é preciso criar um novo nó. Caso já exista uma correspondência, o valor deve ser atualizado se for diferente de 0, ou o nó deve ser removido caso seja nulo. Se a atualização tornar uma linha nula, ela deve ser removida, conforme dito anteriormente.

Após essas operações, as quantidades de valores armazenados são atualizadas e verifica-se a necessidade de redimensionar os vetores.

Para que fosse possível retornar a matriz transposta em um tempo constante, o armazenamento dela é feito em paralelo ao da matriz normal. Ao inserir um elemento na posição (i, j), ele é inserido na posição (j, i) da transposta. Se a matriz for multiplicada por um escalar, a transposta também será. A duplicidade não altera a complexidade assintótica das operações da estrutura e nem do armazenamento. O retorno é feito retornando o ponteiro para a matriz transposta, o que é $O(1)$.

A soma de matrizes é feita em duas partes:

- **Adição Matriz A:** São percorridas todas as buckets externas da matriz A. Caso a bucket não esteja vazia, cada linha armazenada nela tem seu hash interno percorrido, iterando-se sobre todos os buckets internos e percorrendo todos os elementos dentro de cada bucket não vazio. Os valores dos elementos encontrados são adicionados na matriz C.

- **Adição Matriz B:** A matriz B é percorrida da mesma forma que a matriz A, mas, para cada elemento encontrado, primeiro busca-se em C o valor armazenado na mesma coordenada (i, j), e soma-se com o valor do elemento em B, atualizando o valor em (i, j) da matriz C para o resultado da soma.

Pelo redimensionamento dinâmico, espera-se que o número de buckets seja proporcional ao número de elementos armazenados e que não haja muitas colisões, portanto é esperado que o custo assintótico de percorrer toda a estrutura seja equivalente a percorrer os elementos não nulos de cada matriz, ou seja, percorrer k_a e depois k_b. Além disso, é esperado que o acesso ao valor que está em C e a inserção dos novos valores sejam feitos em tempo constante $O(1)$, logo a complexidade esperada da soma é determinada por $O(k_a + k_b)$.

A multiplicação por escalar, $\alpha.A$, segue o mesmo princípio de percorrer todas as buckets externas e internas, tendo cada valor não nulo multiplicado pelo escalar α. Caso o valor do escalar seja 0, para manter a característica da estrutura em não armazenar elementos nulos, todos os nós são removidos. É esperado que o número de buckets seja proporcional ao número de elementos armazenados, por conta do redimensionamento dinâmico baseado no fator de carga, fazendo com que a complexidade esperada seja de $O(k)$.

A multiplicação de duas matrizes $C = A \times B$ foi implementada com base na definição algébrica:

$$C_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$$

Toda a matriz A é percorrida (iterando sobre os buckets externos e internos), visitando todos os elementos não nulos. Para cada elemento a_{ik} encontrado em A, busca-se a linha k de B. Essa busca, pela forma como os hashs foram estruturados com separação de linhas e colunas, é feita em $O(1)$. Após a localização da linha k, todos os elementos b_{kj} são multiplicados por a_{ik}, sendo acumulados em C na posição (i, j). Para cada elemento não nulo em A, espera-se percorrer o número médio de elementos não nulos por linha não nula em B, sendo então a complexidade esperada inferior a $O(k_a \times k_b)$.

As estruturas descritas acima foram projetadas para atender às complexidades exigidas na especificação do projeto. A análise teórica e as provas formais desses custos serão detalhadas na Seção 3.

2.2 Estrutura 2: Árvore Rubro-Negra

A escolha da estrutura de dados para a representação da matriz esparsa foi guiada principalmente pelos requisitos de complexidade assintótica impostos pelo projeto. Após análise dos requisitos, optou-se pela implementação de uma **Árvore Rubro-Negra Esquerdista (LLRBT)**. A seguir, detalham-se as motivações e as características arquiteturais dessa escolha.

2.2.1 Motivação e Seleção da Estrutura

Uma análise imediata do problema já revelou que estruturas lineares simples (como listas encadeadas) ou árvores de busca binária (BST) tradicionais não seriam suficientes para garantir o desempenho no pior caso. Uma BST não balanceada, por exemplo, poderia degenerar para uma complexidade linear $O(K)$ em cenários de inserção ordenada.

Consequentemente, a necessidade de uma Árvore de Busca Binária Balanceada (BBST) tornou-se evidente. Dentre as opções disponíveis, foi escolhida a **Árvore Rubro-Negra Esquerdista (LLRBT)**. Essa escolha se deu principalmente pelo fato de essa estrutura ser eficiente e pela familiaridade prévia da equipe com o conceito. Inclusive, a base teórica e prática para esta implementação fundamentou-se no material didático da disciplina **MC202 (Estruturas de Dados)**, ministrada no segundo semestre de 2024.

2.2.2 Organização da Estrutura

A estrutura central consiste em nós que armazenam a tripla (i, j, valor), além de metadados de controle (cor do nó e ponteiros para filhos esquerdo e direito).

Um desafio intrínseco ao uso de árvores para matrizes é o mapeamento de coordenadas bidimensionais (i, j) para uma estrutura de ordenação linear. Para solucionar isso, adotou-se uma ordenação lexicográfica. A lógica de comparação define que uma posição (i_1, j_1) é "menor" que (i_2, j_2) se:

$$i_1 < i_2 \quad \text{ou} \quad (i_1 = i_2 \text{ e } j_1 < j_2)$$

Essa abordagem permite que a árvore mantenha os elementos logicamente ordenados por linha e, dentro da mesma linha, por coluna. Isso facilita não apenas a busca e inserção pontual, mas também operações que exigem a iteração ordenada ou a recuperação de linhas inteiras, essenciais para a otimização da multiplicação de matrizes.

3 Análise Teórica de Complexidade

3.1 Análise da Estrutura 1: Hash

Antes de analisar operação por operação, vai ser necessário fazer uma análise geral e fundamental para poder definir a complexidade do hash.

```
1 int funcaoHash(int i)
2     {
3         return (31 * i) % this->n_buckets;
4     }
```

Primeiramente, a função hash usada acima, pelo Princípio da Casa dos Pombos, permite a ocorrência de colisões. Já que para um $n\_buckets$ menor que o número i de linhas da matriz, em alguma operação vai ocorrer uma colisão.

Desse modo, vamos analisar o ocorrência dessas colisões com base nas seguintes premissas e definições. Seja:

- m: O número de *buckets* na tabela hash (no código, `n_buckets`).
- n: O número de elementos armazenados no hash (no código, `qtd_armazenada`).
- Fator de Carga (λ): A relação entre elementos e *buckets*, $\lambda = \frac{n}{m}$.
- Hipóteses de Alocação: Assuma que:
  1. Cada chave tem a mesma probabilidade $(1/m)$ de ser alocada em qualquer um dos m *buckets*.
  2. A escolha do *bucket* para cada chave é independente das outras chaves.
- Fator de Carga Constante: A implementação de `rehash()` garante que o fator de carga λ é sempre mantido abaixo de uma constante (no código, 0.75). Se $\lambda > 0.75$, m é dobrado (e n fica igual), reduzindo λ para ≈ 0.375. Portanto, podemos afirmar que $\lambda \leq 0.75$, ou seja $O(1)$.

Queremos aqui descobrir qual é o tamanho da lista ligada associada a uma posição no hash. Em outras palavras, queremos descobrir qual o número de colisões que acontece em um bucket aleatório no hash.

Seja B_j um bucket aleatório. Seja X_i uma variável aleatória que representa um elemento cuja chave é i, onde $i \in 1, ..., n$.

$$X_i = \begin{cases} 1 & \text{se a } i\text{-ésima chave mapeia para o bucket } B_j \\ 0 & \text{caso contrário} \end{cases}$$

Dessa forma, para descobrir $Len(B_j)$, quantos elementos estão em B_j, basta fazer:

$$Len(B_j) = \sum_{i=1}^{n} X_i$$

Pela linearidade da esperança:

$$E[Len(B_j)] = E\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} E[X_i]$$

O valor esperado de uma variável aleatória indicadora X_i é simplesmente a probabilidade de ela ser 1. Pela Hipótese da Alocação:

$$E[X_i] = (1 \cdot P(X_i = 1)) + (0 \cdot P(X_i = 0)) = P(X_i = 1) = \frac{1}{m}$$

Substituindo isso de volta na equação da esperança:

$$E[Len(B_j)] = \sum_{i=1}^{n} \frac{1}{m} = n \cdot \frac{1}{m} = \frac{n}{m}$$

Portanto:

$$E[Len(B_j)] = \frac{n}{m} = \lambda$$

Como sabemos que λ é constante, concluímos que o número de elementos em um bucket é em média $O(1)$.

Desse modo, cada `while` que percorre os elementos de um bucket vai ser considerado $O(1)$ nas demonstrações a seguir.

Uma observação é que para cada operação realizada, uma equivalente será feita na matriz transposta que está sendo armazenada e atualizada juntamente com a normal. Entretanto, isso não afeta a complexidade assintótica das demonstrações a seguir visto que é somente acrescido um fator constante de vezes ao número de operações.

3.1.1 Memória

Como visto na descrição da estrutura, existe um hash duplo.

- O primeiro hash (mais externo) mapeia a linha i de um elemento não nulo.
- O segundo hash (interno) mapeia a coluna j desse elemento.

Esse primeiro hash mais externo terá um número l de linhas não nulas armazenadas. Dentro de cada uma dessas linhas, haverá o segundo hash, com um número médio de d elementos, que pode ser definido como $d = k/l$, onde k é o número de elementos não nulos da matriz.

Por conta de o número de buckets dos hashs se expandir a uma taxa fixa de 2×antigo_n_buckets+ 1, à medida que a quantidade de elementos não nulos é inserida, podemos afirmar que o custo de armazenamento dos hashs é $O(l \times d) = O(k)$.

3.1.2 Acessar elemento

Abaixo está a função `acharElemento` da classe `Matriz`:

```
int acharElemento(int i, int j)
{
    int n = funcaoHash(i); ----------------------  O(1)
    p_no_linha aux = tabela_linhas[n];
    while (aux != nullptr) ----------------------  O(1)
    {
        if (aux->i == i)
        {
            return aux->valores->acharElemento(i, j); -- O(1)
        }
        aux = aux->prox;
    }
    return 0;
}
```

Abaixo está a função `acharElemento` da classe `Hash_colunas`:

```
int acharElemento(int i, int j)
{
    int n = funcaoHash(j); -------------------------- O(1)
    p_no_coluna aux = this->tabela[n];
    while (aux != nullptr) -------------------------- O(1)
    {
        if (aux->i == i && aux->j == j)
        {
            return aux->valor;
        }
        aux = aux->prox;
    }
    return 0;
}
```

Seja A, uma matriz. Queremos acessar o elemento $A[i, j]$. A `funcaoHash` é $O(1)$. Primeiro, vamos analisar o custo de `acharElemento` de `Hash_colunas`. Como visto na demonstração dos números de elementos de um bucket, sabemos que esse `while` itera $O(1)$ vezes, como o conteúdo dentro do `while` é constante, temos que o custo total de `acharElemento` de Hash_colunas é $O(1)$.

Agora, vejamos `acharElemento` de `Matriz`. Novamente, sabemos que o número de iterações desse `while` é $O(1)$, como vimos que os elementos e funções internas também são constantes, temos que `acharElemento` de `Matriz` é $O(1)$.

3.1.3 Inserir/atualizar elemento

Abaixo está a função `inserir` da classe `Matriz`:

```
void inserir(int valor, int i, int j)
    {
        if (this->qtd_armazenada > 0.75 * this->n_buckets)
        {
            rehash(); -------------------- O(n) \\ n sendo o numero de elementos
        }                                        \\ dentro do hash atualmente
        int n = funcaoHash(i);
        p_no_linha aux = tabela_linhas[n];
        while (aux != nullptr && aux->i != i) -------  O(1)
        {
            aux = aux->prox;
        }

        if (aux == nullptr)
        {
            aux = new NoLinha(i);
            this->qtd_armazenada++;

            aux->prox = tabela_linhas[n];
```

```
            tabela_linhas[n] = aux;
        }

        aux->valores->inserir(valor, i, j); ---------- O(1)
    }
```

Abaixo está a função `Inserir` da classe `Hash_colunas`:

```
void inserir(int valor, int i, int j)
    {
        if (this->qtd_armazenada > 0.75 * this->n_buckets)
        {
            rehash(); ----------- O(n) \\ numero de elementos dentro do hash
        }
        int n = funcaoHash(j); ------------------------------------------  O(1)
        this->tabela[n] = inserir_lista(this->tabela[n], i, j, valor); -----  O(1)
    }
```

```
void rehash(bool encolher)
    {
        int antigo_n_buckets = this->n_buckets;
        if(encolher == true){
            this->n_buckets = max(5, antigo_n_buckets/2);
        }else{
            this->n_buckets = antigo_n_buckets * 2 + 1;
        }
        if (this->n_buckets == antigo_n_buckets) return;

        vector<p_no_coluna> tabela_nova(this->n_buckets, nullptr);

        int n_hash;
        for (int i = 0; i < antigo_n_buckets; i++) ----------------- O(n)
        {
            p_no_coluna aux = this->tabela[i];

            while (aux != nullptr)
            {
                p_no_coluna prox = aux->prox;
                n_hash = funcaoHash(aux->j);
                aux->prox = tabela_nova[n_hash];
                tabela_nova[n_hash] = aux;
                aux = prox;
            }
        }
        this->tabela = tabela_nova;
    }
```

```
p_no_coluna inserir_lista(p_no_coluna lista, int i, int j, int valor)
    {
        if (lista == NULL)
        {
            p_no_coluna novo = new NoColuna(i, j, valor);
            this->qtd_armazenada++;
            return novo;
        }
        p_no_coluna aux = lista;

        if (aux->i == i && aux->j == j)
        {
            aux->valor = valor;
            return lista;
        }
        while (aux->prox != NULL) ------------------------------------ O(1)
        {
            if (aux->prox->i == i && aux->prox->j == j)
            {
                aux->prox->valor = valor;
                return lista;
            }
            aux = aux->prox;
        }
        p_no_coluna novo = new NoColuna(i, j, valor);
        aux->prox = novo;
        this->qtd_armazenada++;
        return lista;
```

```
29      }
```

A função `inserir_lista` é $O(1)$, pois apresenta verificações constantes e um `while` que como demonstrado anteriormente é $O(1)$.

A função `rehash`, itera os buckets do hash fazendo as atualizações necessárias. O `while` interno é $O(1)$, e o número de buckets é $O(l)$ para o `rehash Matriz` e $O(d)$ para o `rehash Hash_colunas`, onde l é o número de linhas não nulas e d o número médio de elementos por linha não nula.

Desse modo, o custo das funções inserir dependem se ocorre ou não o rehash. Assim, faz-se necessário uma análise amortizada de custo para determinar o caso médio dessas funções.

A análise a seguir será feita para `inserir Hash_colunas`, mas ela é equivalente para `inserir Matriz` que é o nosso objetivo.

O objetivo é provar que o custo amortizado da função `inserir` é $O(1)$, mesmo considerando o custo $O(n)$ da operação `rehash` (onde n é `qtd_armazenada`).

Definição do Problema

O objetivo é provar que o custo amortizado $\hat{c_i}$ da i-ésima operação (inserção ou remoção) é $\mathcal{O}(1)$, mesmo que o custo real c_i seja $\mathcal{O}(n)$ quando um `rehash` ocorre.

Seja n a `qtd_armazenada` e m o `n_buckets`. As regras de rehash adotadas são:

- Expansão: se $\alpha = n/m > 0.75$, faz-se $m \leftarrow 2m + 1$.
- Contração: se $\alpha = n/m < 0.25$, faz-se $m \leftarrow \max(5, \lfloor m/2 \rfloor)$.

O custo real c_i de uma operação é:

- $c_i \leq c_0$ (constante) se não houver `rehash`;
- $c_i \leq c_0 + c_1 n_{i-1}$ se houver `rehash`, onde c_1 é o custo constante por elemento realocado.

Usaremos o método do potencial. Definimos o custo amortizado

$$\hat{c_i} = c_i + \Phi(D_i) - \Phi(D_{i-1}).$$

Definimos nossa função potencial $\Phi(D_i)$ como:

$$\Phi(D) \;=\; C \cdot \begin{cases} 2n - m, & \text{se } n \geq \dfrac{m}{2}, \\ \dfrac{m}{2} - n, & \text{se } n < \dfrac{m}{2}, \end{cases}$$

onde $C > 0$ será escolhido em função de c_1. Verifique que em cada ramo $\Phi(D) \geq 0$:

- se $n \geq m/2$ então $2n - m \geq 0$;
- se $n < m/2$ então $m/2 - n > 0$.

Além disso, com as regras de rehash acima mantemos sempre $0.25 \leq \alpha \leq 0.75$, isso garante que a forma do potencial aplicada nos estados antes e depois do rehash caiba nos casos abaixo.

Caso A: Inserção sem `rehash`

Se a inserção não provoca rehash então $n_i = n_{i-1} + 1$ e $m_i = m_{i-1}$ e $c_i \leq c_0$.

- Subcaso A1: estado anterior com $n_{i-1} \geq m_{i-1}/2$.
  Neste ramo $\Phi(D) = C(2n - m)$. Assim

$$\begin{aligned}\Delta\Phi &= \Phi(D_i) - \Phi(D_{i-1}) = C\big(2(n_{i-1}+1) - m_{i-1}\big) - C\big(2n_{i-1} - m_{i-1}\big) \\ &= C \cdot 2.\end{aligned}$$

  Logo o custo amortizado

$$\hat{c}_i \leq c_0 + 2C = O(1).$$

- Subcaso A2: estado anterior com $n_{i-1} < m_{i-1}/2$ e a inserção não cruza $m/2$.
  Aqui $\Phi(D) = C(m/2 - n)$. Então

$$\Delta\Phi = C\big(\tfrac{m_{i-1}}{2} - (n_{i-1}+1)\big) - C\big(\tfrac{m_{i-1}}{2} - n_{i-1}\big) = -C.$$

  Portanto

$$\hat{c}_i \leq c_0 - C = O(1),$$

Caso B: Inserção com `rehash`

Suponha a inserção i provoca expansão. No estado anterior temos $n_{i-1} \approx 0.75m_{i-1}$ e após a inserção $n_i = n_{i-1} + 1$, $m_i = 2m_{i-1} + 1 \approx 2m_{i-1}$.

Avaliando a diferença de potencial, temos que antes do rehash, como $n_{i-1} \geq m_{i-1}/2$, usamos $\Phi(D_{i-1}) = C(2n_{i-1} - m_{i-1})$. Aproximando $n_{i-1} = 0.75m_{i-1}$ obtemos

$$\Phi(D_{i-1}) \approx C(2 \cdot 0.75m_{i-1} - m_{i-1}) = C(0.5m_{i-1}).$$

Depois do rehash, temos $m_i \approx 2m_{i-1}$ e $n_i \approx 0.75m_{i-1} + 1$, logo temos que $n_i < m_i/2$. Assim, temos que:

$$\Phi(D_i) = C\Big(\frac{m_i}{2} - n_i\Big) \approx C\big(m_{i-1} - 0.75m_{i-1} - 1\big) = C\big(0.25m_{i-1} - 1\big).$$

A diferença de potencial é

$$\Phi(D_i) - \Phi(D_{i-1}) \approx C\big(0.25m_{i-1} - 1\big) - C\big(0.5m_{i-1}\big) = -C\big(0.25m_{i-1} + 1\big).$$

Ou seja, há uma diminuição de potencial de $C \cdot 0.25m_{i-1}$.

Para comparar com n_{i-1}, usamos $n_{i-1} \approx 0.75m_{i-1}$, logo

$$0.25m_{i-1} = \frac{1}{3}\, n_{i-1}.$$

Portanto a diminuição de potencial é aproximadamente $C \cdot \frac{1}{3}n_{i-1}$.

Agora o custo amortizado:

$$\begin{aligned}\hat{c}_i &\leq (c_0 + c_1 n_{i-1}) + \big(\Phi(D_i) - \Phi(D_{i-1})\big) \\ &\approx c_0 + c_1 n_{i-1} - C \cdot \tfrac{1}{3} n_{i-1}.\end{aligned}$$

Escolhendo C tal que $C \cdot \frac{1}{3} \geq c_1$, temos que $\hat{c}_i \leq c_0 + O(1) = O(1)$. Logo, com $C \geq 3c_1$ o gasto do rehash é coberto pelo decréscimo de potencial.

Caso C: Remoção sem `rehash`

Se a remoção não provoca rehash então $n_i = n_{i-1} - 1$ e $m_i = m_{i-1}$.

- Subcaso C1: estado anterior com $n_{i-1} \geq m_{i-1}/2$.
  Aqui $\Phi(D) = C(2n - m)$ e

$$\Delta\Phi = C\big(2(n_{i-1} - 1) - m_{i-1}\big) - C\big(2n_{i-1} - m_{i-1}\big) = -2C,$$

  portanto $\hat{c}_i \leq c_0 - 2C = O(1)$.

- Subcaso C2: estado anterior com $n_{i-1} < m_{i-1}/2$ e remoção não cruza $m/2$.
  Aqui $\Phi(D) = C(m/2 - n)$ e

$$\Delta\Phi = C\big(\tfrac{m_{i-1}}{2} - (n_{i-1} - 1)\big) - C\big(\tfrac{m_{i-1}}{2} - n_{i-1}\big) = C,$$

  portanto $\hat{c}_i \leq c_0 + C = O(1)$.

Caso D: Remoção com `rehash`

Suponha a remoção i provoca contração. No estado anterior $n_{i-1} \approx 0.25m_{i-1}$. Após a remoção, $n_i = n_{i-1} - 1$ e $m_i \approx m_{i-1}/2$.

Antes do rehash, como $n_{i-1} < m_{i-1}/2$, usamos $\Phi(D_{i-1}) = C(m_{i-1}/2 - n_{i-1})$. Aproximando $n_{i-1} = 0.25m_{i-1}$ obtemos

$$\Phi(D_{i-1}) \approx C\big(0.5m_{i-1} - 0.25m_{i-1}\big) = C(0.25m_{i-1}).$$

Após a contração, $m_i \approx m_{i-1}/2$ e $n_i = n_{i-1} - 1$, como $n_{i-1} \approx 0.25m_{i-1}$, logo $n_i < 0.25m_{i-1}$ e consequentemente $n_i < m_i/2$. Desse modo, usamos $\Phi(D_i) = C(m_i/2 - n_i)$.

$$\Phi(D_i) \approx C(m_i/2 - n_i) \approx C(m_{i-1}/4 - n_{i-1} + 1) \approx C(m_{i-1}/4 - m_{i-1}/4 + 1) \approx C.$$

Logo, a diferença de potencial é

$$\Phi(D_i) - \Phi(D_{i-1}) \approx C - C(0.25m_{i-1}) = C(1 - 0.25m_{i-1}) \approx -C(0.25m_{i-1}).$$

Usando $n_{i-1} \approx 0.25m_{i-1}$, a queda de potencial equivale a

$$C \cdot 0.25m_{i-1} = C \cdot \frac{0.25}{0.25} n_{i-1} = C \cdot n_{i-1}.$$

O custo amortizado fica

$$\hat{c}_i \leq (c_0 + c_1 n_{i-1}) - Cn_{i-1} + O(1) = c_0 + (c_1 - C)n_{i-1} + O(1).$$

Portanto, escolhendo $C \geq c_1$ garante-se $\hat{c}_i = O(1)$.

Escolha de C e conclusão

Juntando as restrições obtidas, escolhemos $C = 3c_1$ para cobrir tanto a contração quanto a expansão.

Com essa escolha a contribuição linear em n desaparece nos casos com rehash, e todos os casos têm custo amortizado $\hat{c}_i = O(1)$. Logo, para qualquer sequência de k operações

$$\sum_{i=1}^{k} \hat{c}_i = \sum_{i=1}^{k} c_i + \Phi(D_k) - \Phi(D_0) = O(k),$$

o que implica que o custo amortizado por operação é $O(1)$.

3.1.4 Retornar transposta

```
struct MatrizComTransposta{
    Matriz* normal;
    Matriz* transposta;

    MatrizComTransposta(){
        normal = new Matriz(11);
        transposta = new Matriz(11);
    }

    ~MatrizComTransposta() {
        delete normal;
        delete transposta;
    }

    void inserir(int valor, int i, int j){
        normal->inserir(valor, i, j);
        transposta->inserir(valor, j, i);
    }

    int acharElemento(int i, int j){
        return normal->acharElemento(i, j);
    }

    Matriz* getTransposta(){
        return this->transposta;
    }

    void multiplicarEscalar(int escalar){
        normal->multiplicaEscalar(escalar);
        transposta->multiplicaEscalar(escalar);
    }
};
```

Acima está a estrutura da matriz como hash. Como pode ser visto, a matriz transposta é armazenada juntamente com a normal, e toda operação que é feita na normal também é feita na transposta, de modo que a transposta está sempre atualizada a cada operação. Desse modo, para retornar a transposta é só entregar o ponteiro para a transposta o que é feito em $O(1)$.

3.1.5 Soma de matrizes

```
void somaMatrizes(Matriz* a, Matriz* b, Matriz* c)
{
    for (int i = 0; i < a->n_buckets; i++)
    {
        p_no_linha aux_linha = a->tabela_linhas[i];
        while (aux_linha != nullptr)
        {
            Hash_colunas *hash_colunas = aux_linha->valores;
            for (int k = 0; k < hash_colunas->n_buckets; k++)
            {
                p_no_coluna aux_valor = hash_colunas->tabela[k];
                while (aux_valor != nullptr)
                {
                    c->inserir(aux_valor->valor, aux_valor->i, aux_valor->j);
                    aux_valor = aux_valor->prox;
                }
            }
            aux_linha = aux_linha->prox;
        }
    }


    for (int j = 0; j < b->n_buckets; j++)
    {
        p_no_linha aux_linha = b->tabela_linhas[j];
        while (aux_linha != nullptr)
        {
            Hash_colunas *hash_colunas = aux_linha->valores;
            for (int m = 0; m < hash_colunas->n_buckets; m++)
            {
```

```
                p_no_coluna aux_valor = hash_colunas->tabela[m];
                while (aux_valor != nullptr)
                {
                    int valor_c = c->acharElemento(aux_valor->i, aux_valor->j);
                    c->inserir(valor_c + aux_valor->valor, aux_valor->i, aux_valor->j);
                    aux_valor = aux_valor->prox;
                }
            }
            aux_linha = aux_linha->prox;
        }
    }
}
```

Com base nas demonstrações anteriores, podemos definir que percorrer um bucket do hash é $O(1)$ e que a função inserir é $O(1)$.

Dessa forma vamos analisar o primeiro for de dentro para fora. Dentro do `while` mais interno, temos operações constantes e a função inserir que é $O(1)$, dessa forma esse `while` é $O(1)$. Temos então um `for` que itera por `hash_colunas->n_buckets` que como visto anteriormente é d_a, o número médio de elementos por linha não nula de A, logo o custo desse for é $O(d_a)$. Em seguida vem mais um `while` $O(1)$, que dá $O(1) \times O(d_a) = O(d_a)$. Por fim, mais um `for` que itera por `a->n_buckets` que é l_a, o número de linhas não nulas de A. Dessa forma o custo fica $O(l_a) \times O(d_a) = O(k_a)$.

A analise do segundo for é idêntica a essa só que para matriz B. Dessa forma, o custo total dessa função é $O(k_a) + O(k_b) = O(k_a + k_b)$.

3.1.6 Multiplicação por escalar

```
void multiplicaEscalar(int escalar)
    {
        for (int i = 0; i < this->n_buckets; i++)
        {
            p_no_linha aux_linha = this->tabela_linhas[i];
            while (aux_linha != nullptr)
            {
                Hash_colunas *hash_colunas = aux_linha->valores;
                for (int k = 0; k < hash_colunas->n_buckets; k++)
                {
                    p_no_coluna aux_valor = hash_colunas->tabela[k];
                    while (aux_valor != nullptr)
                    {
                        aux_valor->valor = aux_valor->valor * escalar;
                        aux_valor = aux_valor->prox;
                    }
                }
                aux_linha = aux_linha->prox;
            }
        }
    }
```

Com base nas demonstrações anteriores, podemos definir que percorrer um bucket do hash é $O(1)$.

Dessa forma vamos analisar o `for` de dentro para fora. Dentro do `while` mais interno, temos operações constantes que é $O(1)$, dessa forma esse `while` é $O(1)$. Temos então um `for` que itera por `hash_colunas->n_buckets` que como visto anteriormente é d_a, o número médio de elementos por linha não nula de A, logo o custo desse for é $O(d_a)$. Em seguida vem mais um `while` $O(1)$, que dá $O(1) \times O(d_a) = O(d_a)$. Por fim, mais um `for` que itera por `a->n_buckets` que é l_a, o número de linhas não nulas de A. Dessa forma o custo fica $O(l_a) \times O(d_a) = O(k_a)$.

Dessa forma o custo total dessa função é $O(k_a)$.

3.1.7 Multiplicação de matrizes

```
void multiplicarMatrizes(Matriz* a, Matriz* b, Matriz *c){
    for(int i = 0; i < a->n_buckets; i++){    ---------------------------------- O(l_a)
        p_no_linha aux_linha_a = a->tabela_linhas[i];
        while (aux_linha_a != nullptr){      ---------------------------------- O(1)
            Hash_colunas *hash_colunas_a = aux_linha_a->valores;
            for (int k = 0; k < hash_colunas_a->n_buckets; k++) --------------- O(d_a)
            {
                p_no_coluna aux_valor_a = hash_colunas_a->tabela[k];
                while (aux_valor_a != nullptr)  ------------------------------- O(1)
                {
                    int k_a = aux_valor_a->j;
                    int n_hash_linha_b = b->funcaoHash(k_a);
                    p_no_linha aux_linha_b = b->tabela_linhas[n_hash_linha_b];

                    while(aux_linha_b != nullptr && aux_linha_b->i != k_a ){  - O(1)
                        aux_linha_b = aux_linha_b->prox;
                    }

                    if(aux_linha_b != nullptr){
                        Hash_colunas *hash_colunas_b = aux_linha_b->valores;
                        for(int m = 0; m < hash_colunas_b->n_buckets; m++){  -- O(d_b)
                            p_no_coluna aux_valor_b = hash_colunas_b->tabela[m];
                            while (aux_valor_b != nullptr)  -----------------   O(1)
                            {
                                int produto = aux_valor_a->valor * aux_valor_b->valor;
                                int valor_c = c->acharElemento(aux_valor_a->i, aux_valor_b->j)
                                    ;
                                c->inserir(valor_c + produto, aux_valor_a->i, aux_valor_b->j);
                                aux_valor_b = aux_valor_b->prox;
                            }
                        }
                    }
                    aux_valor_a = aux_valor_a -> prox;
                }
            }
            aux_linha_a = aux_linha_a->prox;
        }
    }
}
```

Considere os `whiles` como $O(1)$, e as entradas como as matrizes A e B, com l_a o número de linhas não nulas de A, d_a o número médio de elementos por linha não nula de A e d_b o número médio de elementos por linha não nula de B.

Fica evidente que analisando a função de dentro para fora, é iterado no primeiro `for` $O(d_b)$, e depois no segundo `for` $O(d_a)$ e por fim no `for` mais externo $O(l_a)$. Isso é equivalente a $O(l_a) \times O(d_a) \times O(d_b) = O(k_a) \times O(d_b) = O(k_a \times d_b)$.

Portanto, a complexidade final da função é $O(k_a \times d_b)$.

3.2 Análise da Estrutura 2: Arvore Rubro-Negra

Para todas as demonstrações a seguir, fica estipulado que estamos trabalhando sobre uma arvore balanceada.

3.2.1 Memória

A estrutura dá arvore permite armazenar os elementos não nulos em nós, dessa forma se uma matriz A, possui k elementos não-nulos, a árvore apresentara k nós balanceados o que vai resultar em uma complexidade de memória de $O(k)$.

3.2.2 Acessar elemento

```
double acessar_rec (p_no no_atual, int i, int j) const{
            if (no_atual == nullptr){
                return 0;
            }
```

```
            if (i < no_atual->i) {
                return acessar_rec(no_atual->esq, i, j);
            }
            else if (i > no_atual->i) {
                return acessar_rec(no_atual->dir, i, j);
            }
            else {
                if (j < no_atual->j) {
                    return acessar_rec(no_atual->esq, i, j);
                }
                else if (j > no_atual->j) {
                    return acessar_rec(no_atual->dir, i, j);
                }
                else {
                    return no_atual->x;
                }
            }
        }
```

A função acima recebe o índice $i = x, j = y$ do elemento desejado e realiza uma busca em uma arvore de k elementos não-nulos. Como a arvore esta balanceada, cada nível descido reduz a quantidade de elementos na metade, desse modo a recorrência dessa função pode ser modelada como:

$$T(k) = \begin{cases} O(1) & \text{se } i = x \text{ e } j = y \\ T(k/2) + O(1) & \text{se } i \neq x \text{ ou } j \neq y \end{cases}$$

Pensando no pior caso, do elemento estar em uma folha ou o elemento procurado ser nulo, temos que $T(k) = T(k/2) + O(1)$ é a recursão feita até chegar ao fim da arvore. Pelo Teorema Mestre, temos que o custo disso é $O(\log k)$. Dessa forma, o custo dessa função é no pior caso $O(\log k)$.

3.2.3 Inserir/atualizar elemento

```
p_no inserir_rec(p_no raiz, int i, int j, double x) {
            if (raiz == nullptr) {
                return new no(i, j, x);
            }

            if (i < raiz->i){
                raiz-> esq = inserir_rec(raiz->esq, i, j, x);
            }
            else if (i > raiz->i){
                raiz-> dir = inserir_rec(raiz->dir, i, j, x);
            }
            else {
                if (j < raiz->j){
                    raiz-> esq = inserir_rec(raiz->esq, i, j, x);
                }
                else if (j > raiz->j){
                    raiz-> dir = inserir_rec(raiz->dir, i, j, x);
                }
                else {
                    raiz->x = x; // ATUALIZA VALOR
                    return raiz;
                }
            }
            if (ehVermelho(raiz->dir) && !ehVermelho(raiz->esq)) {  ---------------- O(1)
                raiz = rotaciona_para_esquerda(raiz);  ------------------------------ O(1)
            }
            if (ehVermelho(raiz->esq) && ehVermelho(raiz->esq->esq)) { ------------- O(1)
                raiz = rotaciona_para_direita(raiz); -------------------------------- O(1)
            }
            if (ehVermelho(raiz->esq) && ehVermelho(raiz->dir)) { ------------------ O(1)
                sobe_vermelho(raiz); ------------------------------------------------ O(1)
            }

            return raiz;
        }
```

```
p_no rotaciona_para_direita(p_no raiz){
            p_no nova_raiz = raiz->esq;
            raiz->esq = nova_raiz->dir;
            nova_raiz->dir = raiz;
            nova_raiz->cor = raiz->cor;
            raiz->cor = VERMELHO;
            return nova_raiz;
        }
```

```
        void sobe_vermelho(p_no raiz){
            raiz->cor = VERMELHO;
            raiz->esq->cor = PRETO;
            raiz->dir->cor = PRETO;
        }
```

Primeiramente, é fácil ver que as funções auxiliares(ehVermelho, rotaciona_para_esquerda, rotaciona_para_esquerda e sobe_vermelho) são todas $O(1)$, por terem sempre um número constante de operações.

Desse modo, a função inserir_rec apresenta função de recorrência idêntica a de acessar um elemento:

$$T(k) = \begin{cases} O(1) & \text{se } i = x \text{ e } j = y \\ T(k/2) + O(1) & \text{se } i \neq x \text{ ou } j \neq y \end{cases}$$

E como demonstrado acima, o pior caso quando um elemento é nulo ou está nas folhas, pelo Teorema Mestre, é $O(\log k)$, sendo k o número de elementos da árvore.

3.2.4 Retornar transposta

```
    MatrizEsparsaArvoreBalanceada transposta() {
        return MatrizEsparsaArvoreBalanceada(raiz_transposta, raiz); -----  O(1)
    }
```

```
        MatrizEsparsaArvoreBalanceada(p_no nova_normal, p_no nova_transposta) : raiz(
            nova_normal), raiz_transposta(nova_transposta) {}
```

A árvore que representa a matriz transposta é mantida atualizada durante as inserções, sendo armazenada em paralelo a matriz original e já estando disponível na memória quando a função transposta é chamada.

Ao chamar a função `transposta()`, um novo objeto é instanciado e ocorre a troca dos ponteiros das raízes: o endereço de memória da raiz transposta do objeto original é atribuído à raiz do novo objeto. Como esse processo envolve apenas alocação e atribuição de endereços, sem cópia dos dados, independe do número de elementos armazenados (k), resultando em uma complexidade constante $O(1)$.

3.2.5 Soma de matrizes

```
    void somar_rec(p_no no_atual, const MatrizEsparsaArvoreBalanceada &matriz2,
        MatrizEsparsaArvoreBalanceada &resultado){
        if (no_atual == nullptr) { ------------------------------------------------  O(1)
            return; ------------------------------------------------------------  O(1)
        }
        if (resultado.acessar(no_atual->i,no_atual->j) != 0.0){
            return;
        }
        double valor_no_equivalente = matriz2.acessar(no_atual->i, no_atual->j); --- O(log k_2
            )
        double valor_soma = no_atual->x + valor_no_equivalente; --------------------  O(1)

        if (valor_soma != 0.0){
            resultado.inserir(no_atual->i, no_atual->j, valor_soma); ---------------  O(log
                k_c)
        }
```

```
14
15          somar_rec(no_atual->esq, matriz2, resultado);
16          somar_rec(no_atual->dir, matriz2, resultado);
17      }
```

```
1       MatrizEsparsaArvoreBalanceada somar(const MatrizEsparsaArvoreBalanceada &matriz2) {
2           MatrizEsparsaArvoreBalanceada resultado;
3           somar_rec(raiz, matriz2, resultado);
4           somar_rec(matriz2.raiz, *this, resultado);
5
6           return resultado;
7       }
```

A soma de duas matrizes é feita somando-se todos os itens da matriz A e da matriz B na matriz resultado, chamando a função `somar_rec` duas vezes. Portanto deve ser analisado a complexidade de `somar_rec`.

A função `somar_rec` recebe um nó atual, uma matriz2 que será utilizada para soma e uma matriz resultado para armazenar as somas. Em uma iteração de `somar_rec` temos:

- Verificação se o nó é nulo: como é uma comparação, o tempo é constante $O(1)$.
- Acesso ao elemento na posição (i, j) da matriz resultado: Como provado, isso leva tempo $O(\log k_c)$, onde k_c é o número de elementos não nulos na matriz resultado.
- Encontrar o valor equivalente na matriz 2: é o tempo de acessar o elemento na posição (i, j), que foi provado ser $O(\log k_2)$, onde k_2 é o número de elementos não nulos na matriz2.
- Somar o valor do nó com o valor encontrado: operação de soma simples que leva tempo constante $O(1)$.
- Se o valor for diferente de 0, inserir na matriz resultado: inserção como provado tem complexidade $O(\log k_c)$, onde k_c é o número de elementos não nulos na matriz resultado.

Assim, o custo operacional de visitar um nó, denotado por C_{op}, é dominado pelas operações logarítmicas:

$$C_{op} \approx O(\log k_C) + O(\log k_2)$$

A recorrência para o custo $T(k)$ de percorrer uma subárvore de k nós é dada por:

$$T(k) = \begin{cases} O(1) & \text{se } k = 0 \\ T(k_{esq}) + T(k_{dir}) + C_{op} & \text{se } k > 0 \end{cases}$$

Onde:

- k é o número de nós na subárvore atual.
- k_{esq} e k_{dir} são o número de nós nas subárvores esquerda e direita.
- C_{op} é o custo das operações realizadas em cada nó (buscas e inserções).

Análise da Primeira Chamada (Matriz A): Nesta etapa, iteramos sobre os k_A nós de A. Para cada nó, realizamos uma busca em B e uma inserção em C (que começa vazia).

- Custo de inserção em C: $\sum_{i=1}^{k_A} \log i = \log(k_A!) \approx k_A \log k_A$
- Custo de busca em B: $k_A \cdot \log k_B$.

$$\text{Custo}_A \approx O(k_A \log k_A + k_A \log k_B)$$

Análise da Segunda Chamada (Matriz B): Nesta etapa, iteramos sobre os k_B nós de B. A árvore C já possui k_A elementos e crescerá até k_C (onde $k_C \leq k_A + k_B$).

- O custo de cada operação (busca ou inserção em C) é limitado superiormente por $\log k_C$.
- Custo total: $\sum_{j=1}^{k_B} \log k_C = k_B \log k_C$.

$$\text{Custo}_B \approx O(k_B \log k_C)$$

Complexidade Total: Somando as duas etapas e considerando que $k_A \leq k_C$ e $k_B \leq k_C$, podemos simplificar os termos logarítmicos pelo limite superior $\log k_C$:

$$T_{total} = O(k_A \log k_C) + O(k_B \log k_C) = O((k_A + k_B) \log k_C)$$

3.2.6 Multiplicação por escalar

```
1  void multiplicar_por_escalar(double escalar){
2      if (escalar == 0.0) {
3          limpar(); ------------------------------------------------------------------ O(k)
4          return;
5      }
6      multiplicar_por_escalar_rec(raiz, escalar);
7  }
```

```
1  void multiplicar_por_escalar_rec(p_no no_atual, double escalar) {
2      if (no_atual == nullptr) { ------------------------------------------------------ O(1)
3          return;
4      }
5      no_atual->x *= escalar; ---------------------------------------------------------- O(1)
6      multiplicar_por_escalar_rec(no_atual->esq, escalar);
7      multiplicar_por_escalar_rec(no_atual->dir, escalar);
8  }
```

```
1  void limpar() {
2      liberar_no(raiz); ---------------------------------------------------------------- O(k)
3      liberar_no(raiz_transposta); ----------------------------------------------------- O(k)
4      raiz = nullptr; ------------------------------------------------------------------ O(1)
5      raiz_transposta = nullptr;------------------------------------------------------- O(1)
6  }
```

```
1  void liberar_no(p_no no_atual){
2      if (no_atual == nullptr){ -------------------------------------------------------- O(1)
3          return;
4      }
5      liberar_no(no_atual->esq);
6      liberar_no(no_atual->dir);
7      delete no_atual; ----------------------------------------------------------------- O(1)
8  }
```

A análise de multiplicação por escalar será dividida em dois casos: $\alpha = 0$ ou $\alpha \neq 0$ Caso o escalar seja diferente de 0, a complexidade é dada pela função `multiplicar_por_escalar_rec`. Uma chamada dessa função realiza as seguintes operações:

- Verifica se o nó é nulo e retorna: comparação em tempo constante, $O(1)$
- Multiplicação por escalar: multiplica o valor do nó pelo escalar. Como é uma operação de multiplicação simples, tem tempo constante.
- Chamada recursiva para a sub-árvore da esquerda e da direita

A relação de recorrência é dada então por:

$$T(k) = \begin{cases} O(1) & \text{se } k = 0 \\ T(k_{esq}) + T(k_{dir}) + O(1) & \text{se } k > 0 \end{cases}$$

Considerando que a soma dos nós das subárvores mais a raiz é igual ao total de nós ($k_{esq}+k_{dir}+1 = k$), a função visita cada nó exatamente uma vez, resultando em:

$$\sum_{i=1}^{k} O(1) = O(k)$$

Caso o escalar seja igual a 0, a complexidade é dada pela função limpar. A função limpar chama `liberar_no` para a raiz e para a transposta. A função `liberar_no` verifica se o nó atual é null, chama recursivamente para a direita e para a esquerda e por fim deleta o nó atual. Ela tem a mesma relação de recorrência que `multiplicar_por_escalar_rec`, que vimos resultar em $O(k)$. A função é chamada duas vezes em `limpar`, juntamente com a atribuição de null a raiz e a raiz_transposta, tendo complexidade total de:

$$T(k) = 2 \cdot O(k) + O(1) = O(k)$$

Portanto, independentemente do valor do escalar, a multiplicação por escalar tem complexidade $O(k)$.

3.2.7 Multiplicação de matrizes

```
void multi_rec(p_no no_atual, const MatrizEsparsaArvoreBalanceada &matriz2,
    MatrizEsparsaArvoreBalanceada &resultado){
    if (no_atual == nullptr){
        return;
    }
    int i = no_atual->i;
    int k = no_atual->j;
    vector<p_no> nos_linha_k;

    elementos_linha_i(matriz2.raiz, k, nos_linha_k); ---------- O(log k_b + d_b)

    \\ Realiza a multiplicacao e acumula os resultados
    for (size_t w = 0; w < nos_linha_k.size(); w++){ ---------- O(d_b)
        p_no no_matriz2 = nos_linha_k[w];
        int j = no_matriz2->j;
        double produto = no_atual->x * no_matriz2->x;

        resultado.acumular(i, j, produto); -------------------- O(log k_c)
    }
    multi_rec(no_atual->esq, matriz2, resultado);
    multi_rec(no_atual->dir, matriz2, resultado);
}
```

```
void elementos_linha_i(p_no no_atual, int i, vector<p_no> &nos_encontrados){
    if (no_atual == nullptr){
        return;
    }
    if (i < no_atual->i){
        elementos_linha_i(no_atual->esq, i, nos_encontrados);
    }
    else if (i > no_atual->i){
        elementos_linha_i(no_atual->dir, i, nos_encontrados);
    }
    else {
        elementos_linha_i(no_atual->esq, i, nos_encontrados);
        nos_encontrados.push_back(no_atual);
        elementos_linha_i(no_atual->dir, i, nos_encontrados);
    }
}
```

```
void acumular(int i, int j, double x) {
    if (x == 0.0) return;
    raiz = inserir_ou_somar_rec(raiz, i, j, x); -------------- O(log k)
    raiz->cor = PRETO;
}
```

```
p_no inserir_ou_somar_rec(p_no raiz, int i, int j, double valor_a_somar) {
            if (raiz == nullptr) {
                return new no(i, j, valor_a_somar);
            }

            if (i < raiz->i){
                raiz->esq = inserir_ou_somar_rec(raiz->esq, i, j, valor_a_somar);
            }
            else if (i > raiz->i){
                raiz->dir = inserir_ou_somar_rec(raiz->dir, i, j, valor_a_somar);
            }
            else {
                if (j < raiz->j){
                    raiz->esq = inserir_ou_somar_rec(raiz->esq, i, j, valor_a_somar);
                }
                else if (j > raiz->j){
                    raiz->dir = inserir_ou_somar_rec(raiz->dir, i, j, valor_a_somar);
                }
                else {
                    //soma direto no no existente
                    raiz->x += valor_a_somar;
                    if (raiz->x == 0.0) {
                        // Se o valor somado resultar em zero, remove o no
                        return remover_rec(raiz, i, j);
                    }
                    return raiz;
                }
            }

            if (ehVermelho(raiz->dir) && !ehVermelho(raiz->esq)) {
                raiz = rotaciona_para_esquerda(raiz);
            }
            if (ehVermelho(raiz->esq) && ehVermelho(raiz->esq->esq)) {
                raiz = rotaciona_para_direita(raiz);
            }
            if (ehVermelho(raiz->esq) && ehVermelho(raiz->dir)) {
                sobe_vermelho(raiz);
            }

            return raiz;
        }
```

```
p_no remover_rec(p_no raiz, int i, int j) {
            if (raiz == nullptr) {
                return nullptr;
            }

            if (i < raiz->i  (i == raiz->i && j < raiz->j)) {
                if (raiz->esq == nullptr) {
                    return raiz;
                }
                if (!ehVermelho(raiz->esq) && !ehVermelho(raiz->esq->esq)) { ----- O(1)
                    raiz = descer_vermelho_esq(raiz); ---------------------------- O(1)
                }
                raiz->esq = remover_rec(raiz->esq, i, j);
            }
            else {
                if (ehVermelho(raiz->esq)) { ------------------------------------- O(1)
                    raiz = rotaciona_para_direita(raiz); ------------------------- O(1)
                }
                if (i == raiz->i && j == raiz->j) {
                    if (raiz->dir == nullptr) {
                        delete raiz ;
                        return nullptr;
                    }

                    if (!ehVermelho(raiz->dir) && !ehVermelho(raiz->dir->esq)) { - O(1)
                        raiz = descer_vermelho_dir(raiz); ---------------------- O(1)
                    }

                    p_no sucessor = min(raiz->dir); ------------------------------ O(logk)
                    raiz->i = sucessor->i;
                    raiz->j = sucessor->j;
                    raiz->x = sucessor->x;

```

```
34                  raiz->dir = deletar_min(raiz->dir); ------------------------- O(logk)
35              }
36              else {
37                  if (raiz->dir == nullptr) {
38                      return raiz;
39                  }
40                  if (!ehVermelho(raiz->dir) && !ehVermelho(raiz->dir->esq)) {
41                      raiz = descer_vermelho_dir(raiz); ---------------------------- O(1)
42                  }
43                  raiz->dir = remover_rec(raiz->dir, i, j);
44              }
45          }
46
47          return balancear(raiz); ------------------------------------------- O(1)
48      }
```

Essa função é complexa, e para demonstra-la com exatidão será necessário provar a complexidade de todas as funções auxiliares que estão integradas no código. Para evitar que a demonstração se alongue demais, serei mais breve ao analisar as funções auxiliares.

- `elementos_linha_i`
  Essa função localiza os elementos de uma determinada linha $i = x$ na árvore B e pode ser definida pela seguinte recorrência:

$$T(k_b) = \begin{cases} T(k_b/2) + O(1) & \text{se } i \neq x \\ 2T(k_b/2) + O(1) & \text{se } i = x \end{cases}$$

  Sendo assim, $T(k_b) = T(k_b/2)+O(1)$ é usada até achar a linha desejada e tem complexidade no pior caso, pelo Teorema Mestre, de $O(\log k_b)$. A segunda recorrência, $T(k_b) = 2T(k_b/2)+O(1)$ vai passar por todos os elementos da linha, como já estamos por definição usando d_b como a quantidade média de elementos não nulos por linha não nula, podemos dizer que a complexidade dessa parte é $O(d_b)$.
  Portanto, a complexidade total dessa função é $O(\log k_b + d_b)$.

- `remover_rec`
  As funções auxiliares `min` e `deletar_min` são recursivas com uma recorrência simples de $T(k) = T(k/2) + O(1)$ que como já visto resulta em complexidade $O(\log k)$. Entretanto é válido perceber que essa função será chamada em subárvores e dificilmente terá esse custo.

  Agora, analisando a `remover_rec`, podemos formular sua recorrência como:

$$T(k) = \begin{cases} T(k/2) + O(1) & \text{se } i \neq x \text{ ou } j \neq y \\ O(\log k) & \text{se } i = x \text{ e } j = y \end{cases}$$

  Desse modo, pelo Teorema Mestre, sabemos que $T(k) = T(k/2) + O(1)$ tem complexidade $O(\log k)$. Somando a complexidade de `min` e `deletar_min` no final, temos que a complexidade geral é $O(\log k + \log k) = O(\log k)$.

- `inserir_ou_somar_rec`
  Essa função recebe $i = x, j = y$ e pode ser definida pela seguinte recorrência caso o elemento não sege nulo:

$$T(k) = \begin{cases} T(k/2) + O(1) & \text{se } i \neq x \text{ ou } j \neq y \\ O(1) & \text{se } i = x \text{ e } j = y \end{cases}$$

  Pelo Teorema Mestre, essa função é no pior caso $O(\log k)$. Quando a soma resulta em um elemento nulo, é chamada a função `remover_rec` que como visto anteriormente é $O(\log k)$. Assim, no pior caso a complexidade é $O(\log k) + O(\log k) = O(\log k)$.

Dessa forma, podemos agora analisar a complexidade de `multi_rec`. Temos que a recorrência dessa função é:

$$T(k_a) = 2T(k_a/2) + O(\log k_b + d_b \times \log k_c + d_b)$$

Como k_a não depende das variáveis de B ou C, podemos simplificar a análise para $T(k_a) = 2T(k_a/2) + O(1)$, essa complexidade, pelo Teorema Mestre, vai ser $O(k_a)$. Multiplicando, temos que o custo total é:

$$O(k_a(\log k_b + d_b \times \log k_c + d_b)) = O(k_a(\log k_b + d_b \times \log k_c))$$

.

3.3 Análise da Matriz em Arranjo Bidimensional

Aqui está uma analise brevíssima da complexidade de uma estrutura de matriz tradicional. Como a análise é trivial e extremamente documentada na literatura, optamos por omitir as demonstrações e só deixamos as complexidades. Estamos usando como referencia uma matriz $A_{n\times m}$ e uma matriz $B_{m\times p}$.

- **Memória:** $O(n \times m)$
- **Acessar elemento:** $O(1)$

```
double acessar(int i, int j) const {
        return dados[indice(i, j)]; ----------------- O(1)
}
```

- **Inserir/atualizar elemento:** $O(1)$

```
void inserir(int i, int j, double valor) {
    dados[indice(i, j)] = valor;  ------------------ O(1)
}
```

- **Retornar Transposta:** $O(n \times m)$

```
MatrizNormal transposta() const {
        MatrizNormal resultado(colunas, linhas, 0.0);
        for (int i = 0; i < linhas; ++i) { --------------------------------- O(n)
                for (int j = 0; j < colunas; ++j) { ----------------------- O(m)
                        resultado.dados[j * linhas + i] = dados[indice(i, j)];
                }
        }
        return resultado;
}
```

- **Soma de matrizes:** $O(n \times m)$

```
MatrizNormal somar(const MatrizNormal &outra) const {
        MatrizNormal resultado(linhas, colunas, 0.0);

        for (size_t idx = 0; idx < dados.size(); ++idx) { --------- O(n.m)
                resultado.dados[idx] = dados[idx] + outra.dados[idx];
        }
        return resultado;
}
```

- **Multiplicação por escalar:** $O(n \times m)$

```
void multiplicar_por_escalar(double escalar) {
        for (double &valor : dados) { ---------------------------- O(n.m)
                valor *= escalar;
        }
}
```

- **Multiplicação de matrizes:** $O(n \times m \times p)$

```
MatrizNormal multiplicar(const MatrizNormal &outra) const {
    MatrizNormal resultado(linhas, outra.colunas, 0.0);

    for (int i = 0; i < linhas; ++i) { ---------------------- O(n)
        for (int j = 0; j < outra.colunas; ++j) { ------------ O(p)

            double soma = 0.0;

            for (int k = 0; k < colunas; ++k) { -------------- O(m)
                double valor_a = dados[i * colunas + k];
                double valor_b = outra.dados[k * outra.colunas + j];

                soma += valor_a * valor_b;
            }

            resultado.dados[i * outra.colunas + j] = soma;
        }
    }
    return resultado;
}
```

4 Análise Experimental

Para validar as complexidades teóricas e avaliar o desempenho prático das estruturas de dados implementadas, foi desenvolvido um ambiente de testes automatizado utilizando a linguagem C++. O código fonte completo dos testes e das estruturas encontra-se disponível no repositório do GitHub.

4.1 Análise do tempo de execução

4.1.1 Metodologia

Nesta seção, detalhamos as ferramentas, o processo de geração de dados e os protocolos de medição utilizados.

Ambiente de Implementação e Bibliotecas Os algoritmos foram implementados visando eficiência e controle de memória. Para a medição de tempo e manipulação de dados, foram utilizadas as seguintes bibliotecas padrão do C++:

- `<chrono>`: Utilizada para a cronometragem de alta precisão (*High Resolution Clock*). O tempo de execução foi capturado em microssegundos (μs) para garantir sensibilidade suficiente em operações rápidas que são realizadas nas baterias de testes.
- `<random>`: Para a geração de massas de dados sintéticos foi utilizada a Mersenne Twister (`std::mt19937`). Ela garante uma distribuição uniforme de alta qualidade e um período de repetição suficientemente longo para evitar vieses estatísticos em matrizes de grandes dimensões ($N \geq 10^5$).

Geração de Instâncias de Teste Para a avaliação das estruturas, foi implementado um gerador de matrizes esparsas sintéticas. O gerador recebe como parâmetros a dimensão da matriz (N) e o percentual de esparsidade desejado. O processo de geração garante que:

1. O número exato de elementos não nulos (K) respeite a fórmula $K = \lfloor N^2 \times \text{esparsidade} \rfloor$.
2. As coordenadas (i, j) sejam distribuídas uniformemente no espaço $N \times N$, sem duplicatas.
3. Os valores armazenados sejam números reais aleatórios (`double`) entre 1.0 e 100.0.

Protocolo de Execução e Medição Na bateria de testes, com o objetivo de reduzir a diferença entre as execuções dos testes foram adotadas as seguintes medidas:

1. **Repetição:** Cada operação (inserção, soma, multiplicação entre matrizes, multiplicação por escalar e obter a transposta) para um dado par de parâmetros $(N, \text{Esparsidade})$ foi executada 10 vezes consecutivas. O tempo registrado para análise é a média aritmética dessas 10 execuções. Isso diminui o impacto de *outliers* e oferece uma representação mais fiel do custo computacional médio da estrutura.

2. **Reset de Estado:** Para operações de inserção, a estrutura de dados foi instanciada e destruída a cada iteração do loop de repetição, garantindo que não houvesse contaminação de memória ou cache entre testes.

3. **Manutenção do ambiente:** Todos os testes foram realizados no mesmo computador com as mesmas condições de ambiente interno.

Os cenários de teste foram divididos em duas baterias principais:

- **Cenário de Pequena Escala ($N \in \{100, 1.000\}$):** Comparação entre Matriz Normal (Array), Hash Table e LLRBT.
- **Cenário de Larga Escala ($N \in \{10^4, 10^5, 10^6\}$):** Comparação exclusiva entre Hash Table e LLRBT, visto que a alocação de uma Matriz Normal excederia a memória RAM disponível para $N \geq 10^5$.

Coleta e Saída de Dados Os resultados brutos foram exportados automaticamente para um arquivo em formato **CSV**, facilitando o processamento posterior para geração de gráficos. Os dados das duas baterias de testes. A estrutura do arquivo de saída segue o padrão:

```
Estrutura, Operacao, Dimensao(N), Esparsidade(%), K_Real, Tempo(us)
```

Onde:

- **Estrutura:** Identificador da implementação (Normal, Hash, Árvore).
- **Operação:** Tipo de teste realizado (Inserção, Soma, Multiplicação).
- **K_Real:** Número absoluto de elementos não nulos processados.
- **Tempo(us):** Tempo médio de execução em microssegundos.

4.1.2 Desempenho em diferentes graus de esparsidade

A análise inicial investiga o impacto da densidade de elementos não nulos (K) sobre a eficiência computacional. Para isolar essa variável, fixou-se a dimensão das matrizes em $N = 10^3 \times 10^3$, variando a esparsidade entre 1%, 5%, 10% e 20% de elementos não nulos.

O cenário de **alta esparsidade (1%)**, apresentado na **Figura 1**, ilustra a principal motivação para o uso de estruturas esparsas. Neste contexto, onde $K \ll N^2$ (definição de uma matriz esparsa) as implementações baseadas em Hash e LLRBT superam drasticamente a abordagem densa na maioria das operações. A multiplicação, por exemplo, se beneficia do fato de muitas linhas serem nulas e de que estruturas esparsas manipulam apenas os elementos existentes, sendo ordens de magnitude mais rápida nas versões esparsas.

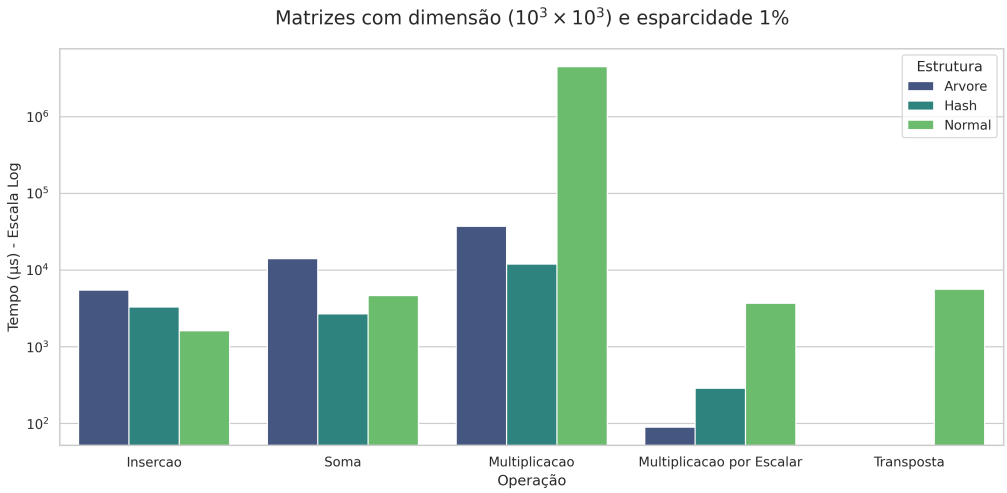

Figura 1: Tempo de execução da Multiplicação ($N = 10^3$) em escala logarítmica. O gráfico evidencia o ponto onde a estrutura densa se torna mais eficiente que a LLRBT (aprox. 10%).

No entanto, ao analisarmos a evolução do tempo conforme a densidade aumenta, notamos comportamentos distintos entre as operações. A **Figura 2** exibe a operação de **Soma**. Observa-se que o tempo da Matriz Normal permanece constante (linha reta), pois sua complexidade depende apenas de N ($O(N^2)$), beneficiando-se da alocação contígua e previsibilidade de cache, características de um computador real que estão ocultas na análise assintótica. Em contrapartida, as estruturas esparsas apresentam um crescimento linear evidente, pois sua complexidade é dependente de K ($O(K)$). Mesmo com poucos elementos, o acesso não contíguo já torna as estruturas esparsas menos competitivas nesta operação específica para a dimensão testada.

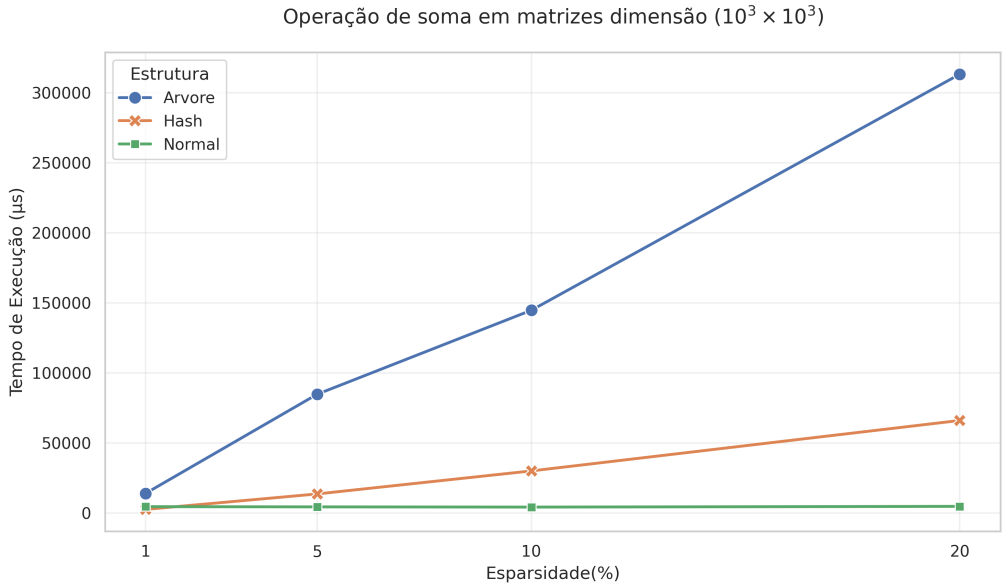

Figura 2: Tempo de execução da operação de Soma ($N = 10^3$) variando a esparsidade. Note o crescimento linear das estruturas esparsas versus o tempo constante da Matriz Normal.

O ponto crítico da análise reside na operação de **Multiplicação**, detalhada na **Figura 3**. Em escala logarítmica, percebe-se o "ponto de inflexão". Inicialmente, a vantagem das esparsas é massiva. Contudo, ao atingir 10% de esparsidade, a curva da LLRBT cruza a da Matriz Normal, indicando que o custo logarítmico de busca ($O(\log K)$) somado às constantes da estrutura supera o custo cúbico ($O(N^3)$) da multiplicação densa.

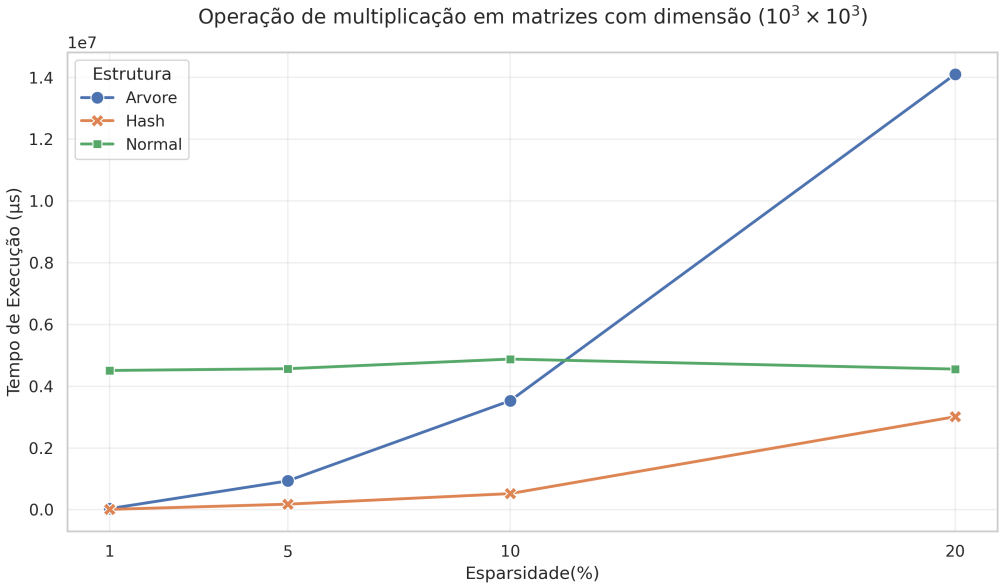

Figura 3: Tempo de execução da Multiplicação ($N = 10^3$) em escala logarítmica. O gráfico evidencia o ponto onde a estrutura densa se torna mais eficiente que a LLRBT (aprox. 10%).

Esse fenômeno se consolida no cenário de **baixa esparsidade (20%)**, ilustrado na **Figura 4**. Aqui, a quantidade de elementos não nulos é grande o suficiente para saturar as estruturas dinâmicas. A Matriz Normal vence a Árvore em todas as operações e empata ou vence a Hash na inserção e soma. Isso confirma que, para esta dimensão, esparsidades acima de 20% justificam o retorno à representação matricial tradicional (densa), devido à eliminação do *overhead* de ponteiros e tratamento de colisões, sendo bem mais eficiente na prática.

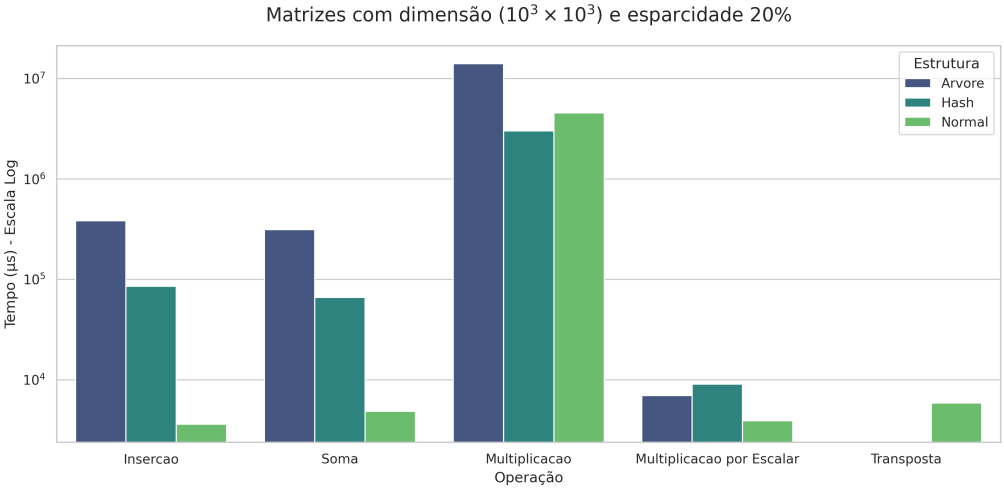

Figura 4: Comparativo de desempenho com 20% de esparsidade ($N = 10^3$). O aumento de K favorece a Matriz Normal, reduzindo ou eliminando a vantagem das estruturas esparsas.

4.1.3 Comparativo Direto: Tabela Hash vs. Árvore Rubro-Negra

Nesta etapa, restringimos a análise às estruturas esparsas (Tabela Hash e LLRBT) aplicadas a matrizes do maior porte possível presente nos testes ($N = 10^6 \times 10^6$). A Matriz Normal foi desconsiderada devido à inviabilidade de alocação de memória ($O(N^2)$) para estas dimensões ($N \geq 10^4 \times 10^4$). O objetivo é confrontar a complexidade esperada da Hash com a complexi-

dade garantida da Árvore, conforme definido nos requisitos do projeto e que foram provados anteriormente.

A **Figura 5** apresenta o desempenho global das estruturas. Observa-se uma predominância clara da Tabela Hash nas operações principais. Este resultado corrobora perfeitamente com a análise teórica: a complexidade esperada de acesso da Hash é $O(1)$, o que assintoticamente supera a complexidade garantida $O(\log K)$ da LLRBT, e essa dominância teórica no caso médio ocorre nas operações de acesso, inserção, soma e multiplicação de matrizes. Mesmo considerando o pior caso da Hash (colisões), o custo amortizado se mostrou consistentemente inferior ao custo de navegação na altura da árvore para a quantidade de elementos testada.

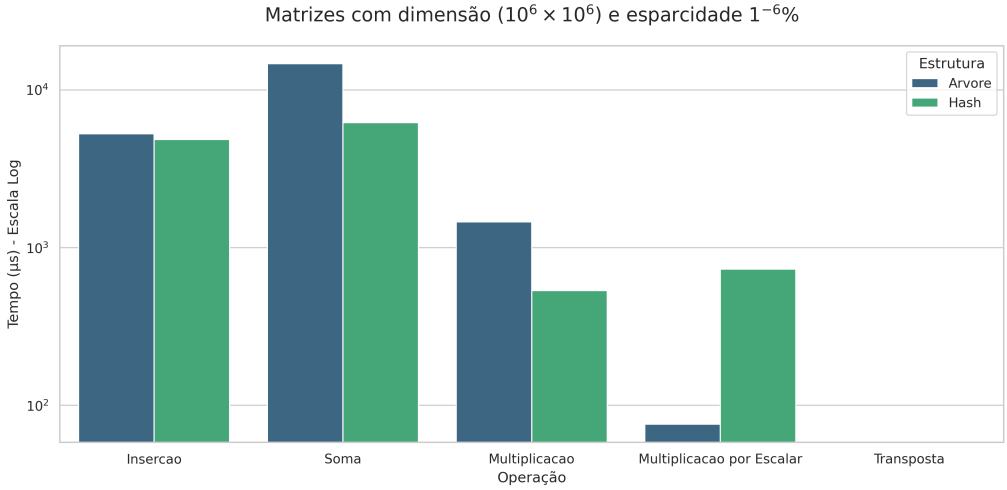

Figura 5: Comparativo de desempenho para $N = 10^6$ (Baixa Esparsidade). A Hash vence nas operações de acesso ($O(1)$ vs $O(\log K)$), mas perde na multiplicação por escalar onde as complexidades teóricas se igualam ($O(K)$).

Contudo, a operação de **Multiplicação por Escalar** apresenta uma inversão de desempenho, onde a LLRBT supera a Tabela Hash. Este fenômeno é explicado ao revisitarmos novamente a tabela de complexidades: esta é a única operação, em conjunto com "retornar transposta", onde ambas as estruturas possuem a mesma complexidade assintótica, $O(K)$.

Dessa forma, o caso médio do hash não é tão melhor quanto o pior caso garantido da árvore. Essa diferença se deve justamente a arquitetura de ambas as estruturas:

1. **Na LLRBT:** A travessia recursiva ($O(K)$) visita estritamente os nós existentes, sem desperdício de ciclos.

2. **Na Tabela Hash:** A implementação exige iterar sobre a estrutura de buckets. Como a matriz é esparsa, o algoritmo gasta tempo verificando posições nulas na tabela de dispersão, adicionando um custo fixo de iteração que não existe na árvore.

A **Figura 6** aprofunda essa análise mostrando a evolução temporal da multiplicação por escalar em função de K. O comportamento linear das duas curvas confirma a predição teórica de $O(K)$ para ambas. Entretanto, a inclinação menor da reta da Árvore evidencia sua superioridade de tempo.

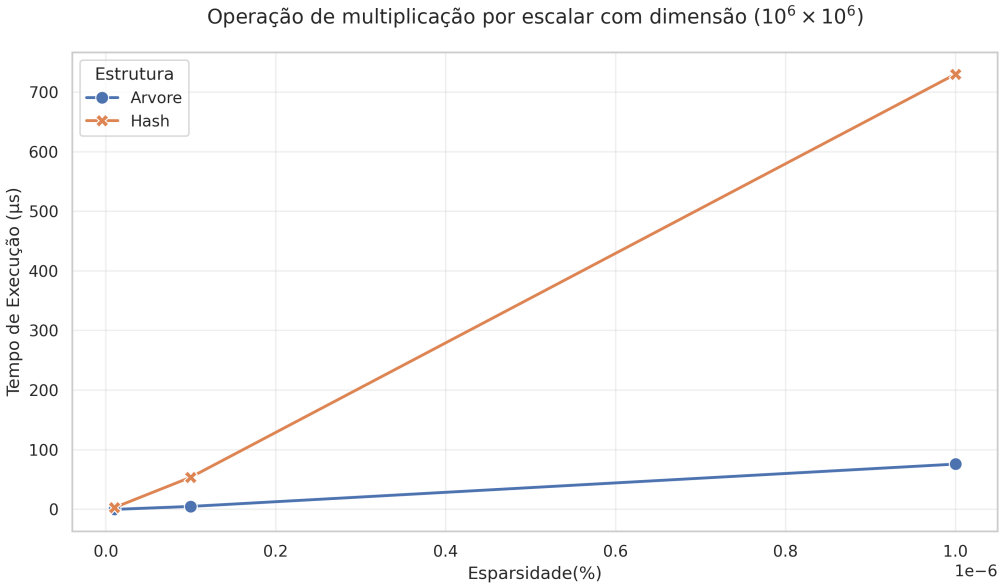

Figura 6: Tempo de execução da Multiplicação por Escalar ($N = 10^6$). Ambas as estruturas apresentam crescimento linear ($O(K)$), mas a Árvore possui melhor desempenho prático devido à ausência de iteração em buckets vazios.

4.2 Análise de memória

4.2.1 Metodologia

A avaliação do consumo de memória baseia-se em uma modelagem analítica das estruturas de dados implementadas em C++. Diferente da análise temporal, sujeita a flutuações de escalonamento do processador, esta abordagem permite determinar com precisão o custo espacial intrínseco por elemento e o comportamento assintótico exato de cada solução. Para os cálculos, assumiu-se uma arquitetura de hardware de 64 bits:

Parâmetros do Sistema

- **Ponteiro (P):** 8 bytes.
- **Alinhamento:** Estruturas são preenchidas (*padding*) para alinhar membros de 8 bytes em endereços múltiplos de 8.

Matriz Normal (Vetor Linearizado) Estrutura contígua em memória.

- **Componentes:** Um vetor de `double` com tamanho $N \times N$.
- Cálculo:

$$Mem_{Matriz}(N) = (N^2 \times 8 \text{ bytes})$$

$$\boxed{Mem_{Matriz} \approx 8N^2}$$

Árvore Rubro-Negra (Nó) Estrutura baseada em nós dispersos. O consumo é dominado pelo tamanho da `struct` somado ao *overhead* de alocação para cada elemento K.

- **Análise da Struct (`no`):**
  - Dados (`i`, `j`, `cor`): $4 + 4 + 4 = 12$ bytes.

- *Padding* (para alinhamento): 4 bytes.
- Dados (`x`): 8 bytes.
- Ponteiros (`esq`, `dir`): $8 + 8 = 16$ bytes.
- **Total Struct:** 40 bytes.

- Cálculo por Elemento:
$$C_{no} = \text{sizeof(no)} \approx 40$$
- Fórmula Final:
$$\boxed{Mem_{RB}(K) = 40K}$$

3. Tabela Hash Nessa estrutura do hash vamos considerar o **pior caso de consumo** (logo após a operação de rehash, onde o *Load Factor* ≈ 0.375).

Definições:

- l: Quantidade de linhas não nulas (Hash Externo).
- K: Quantidade total de elementos (Hashes Internos).

Custos Unitários:

1. **Custo do Nó Encadeado (C_{link}):**
   - Struct (`chave` + `padding` + `valor` + `prox`): 24 bytes.
2. **Custo de Bucket (C_{bucket}):**
   - Considerando a expansão $2 \times n + 1$ no rehash (fator $0.75 \to 0.375$).
   - Custo médio de ponteiros no vetor por elemento armazenado: 8 bytes/0.375 $\approx$ **21.33** bytes.

Fórmula Final: O consumo total é a soma do custo estrutural das linhas (l) e do custo de armazenamento dos dados (K).

$$Mem_{Hash}(K, l) = \underbrace{l \cdot (C_{link} + C_{bucket})}_{\text{Custo das Linhas}} + \underbrace{K \cdot (C_{link} + C_{bucket})}_{\text{Custo dos Elementos}}$$

Substituindo os valores:

$$Mem_{Hash}(K, l) = (l + K) \cdot (40 + 21.33) \approx (l + K) \cdot (62)$$

Como a matriz é esparsa, podemos considerar um caso médio onde cada elemento está numa linha diferente ($l = K$), dessa forma:

$$\boxed{Mem_{Hash} \approx 2K \cdot 62 = 124K}$$

4.2.2 Limites Físicos e Inviabilidade da Representação Densa

A projeção teórica do consumo de memória, ilustrada na **Figura 7**, evidencia a limitação crítica da representação densa (Matriz Normal). Embora esta estrutura possua a maior eficiência de armazenamento *por elemento* individual (apenas 8 bytes, como mostrado na seção anterior), sua dependência quadrática da dimensão ($O(N^2)$) impõe uma barreira física intransponível para problemas de larga escala.

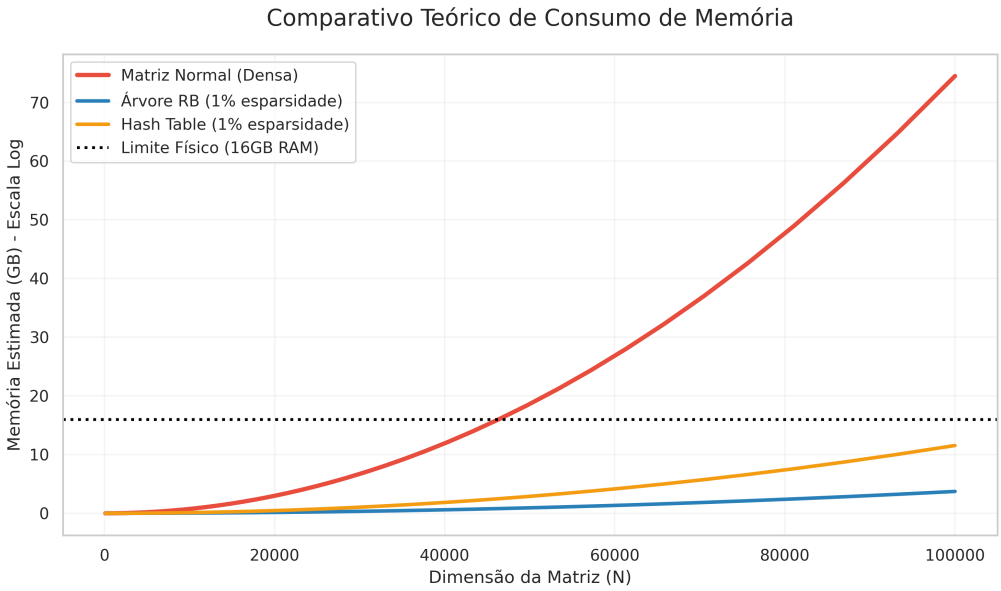

Figura 7: Projeção de consumo de memória (escala logarítmica). O crescimento quadrático ($O(N^2)$) inviabiliza a Matriz Normal em grandes dimensões, enquanto as estruturas esparsas mantêm escalabilidade linear ($O(K)$)

Ao analisarmos o comportamento assintótico, percebemos que o "custo de entrada" das estruturas esparsas — calculado anteriormente como 40 bytes para a LLRBT e ≈ 124 bytes para o pior caso da Hash — torna-se irrelevante à medida que N cresce. Para uma matriz de dimensão $N = 10^5$, a representação densa exigiria a alocação contígua de 10^{10} elementos. Considerando 8 bytes por `double`, isso resulta em uma demanda de aproximadamente **74,5 GB de RAM** (8×10^{10} bytes). Tal requisito excede a capacidade da maioria dos computadores convencionais.

Portanto, o gráfico demonstra que a "ineficiência" estrutural das representações esparsas (ponteiros e *padding*) é um preço "barato" a se pagar pela viabilidade de armazenar matrizes de alta dimensionalidade, onde armazenamos apenas o necessário. A Matriz Normal é estritamente limitada a problemas onde N é pequeno (tipicamente $N < 10^4$), independentemente da esparsidade dos dados.

4.2.3 Eficiência Espacial: Hash vs. Árvore Rubro-Negra

Restringindo a análise às estruturas viáveis para grandes dimensões, a **Figura 8** apresenta o comparativo de consumo de memória entre a Tabela Hash e a LLRBT a em função do número de elementos (K).

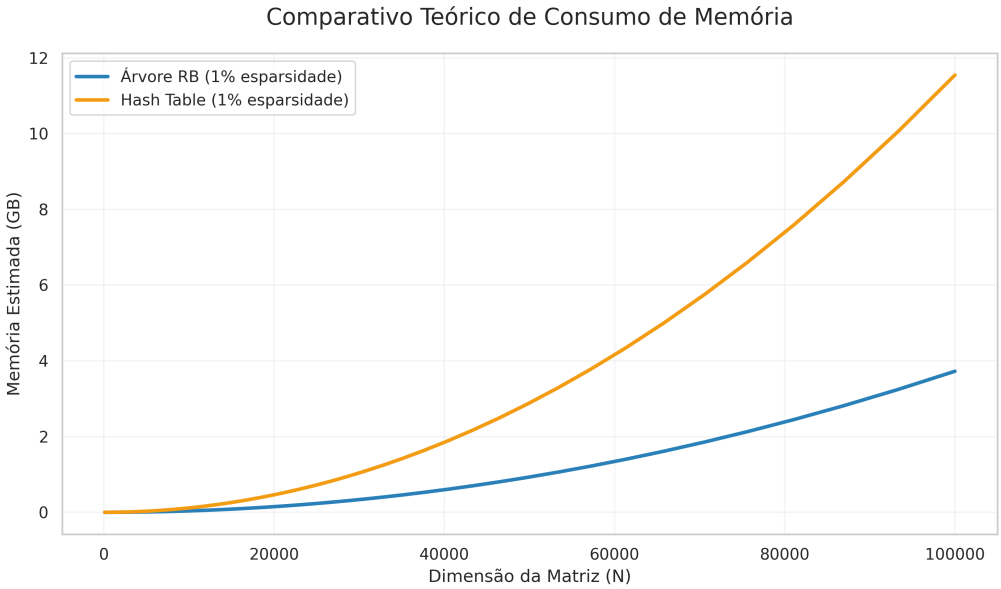

Figura 8: Comparativo de eficiência espacial ($O(K)$). A LLRBT consome menos memória por realizar alocação exata de nós, enquanto a Tabela Hash exige alocação excedente para sustentar sua performance de acesso.

Embora ambas as estruturas satisfaçam o requisito de complexidade espacial linear $O(K)$, a inclinação das retas revela uma diferença significativa na constante de consumo. A LLRBT demonstra ser, consistentemente, a estrutura mais eficiente em termos de memória. Isso decorre de sua natureza de alocação exata: cada nó alocado corresponde a um dado inserido, com um *overhead* fixo de ponteiros e cor (calculado em ≈ 40 bytes). Não há memória reservada "por precaução".

Em contraste, a Tabela Hash apresenta um consumo aproximadamente 3 vezes superior no modelo teórico (≈ 124 bytes por elemento no pior caso analisado). Essa diferença é intrínseca ao mecanismo de *hashing*.

5 Discussão

O desenvolvimento e análise experimental deste projeto permitem estabelecer direcionamentos sobre a aplicabilidade de cada estrutura de dados para a representação de matrizes. Os resultados confirmam que não existe uma estrutura "mágica", e sim que a escolha ideal depende estritamente das restrições de memória do sistema e das operações predominantes na aplicação. Assim, como em quase todas as estruturas de dados, o que determina a escolha é o equilíbrio entre eficiência espacial e temporal.

A implementação tradicional (Matriz Normal) provou ser a mais eficiente em tempo de execução para matrizes de pequenas dimensões ($N \leq 10^3$) e com baixa esparsidade ($\geq 20\%$). No entanto, sua complexidade espacial $O(N^2)$ a torna proibitiva para cenários reais de alta dimensionalidade. Para $N \geq 10^4$, considerando um computador convencional, o uso de estruturas esparsas já não é apenas uma questão de otimização, mas uma necessidade física para evitar o esgotamento da memória RAM.

Entre as estruturas esparsas, a Tabela Hash consolidou-se como a melhor escolha para cenários onde a velocidade é prioritária. Seu acesso $O(1)$ amortizado na prática garantiu tempos de inserção, busca, soma e multiplicação entre matrizes consistentemente inferiores aos da árvore. Entretanto, essa alta eficiência temporal cobra um preço no consumo de memória (representando um custo 3 vezes maior no pior caso) e na eficiência de operações que exigem

iteração completa (como a multiplicação por escalar), onde o custo de verificar *buckets* vazios degrada o desempenho.

A Árvore Rubro-Negra destacou-se como a estrutura mais robusta e econômica. Embora seu acesso $O(\log K)$ seja mais lento que o da Hash, ela é estritamente previsível (complexidade garantida). Essa previsibilidade garante segurança para sistemas críticos, que não podem correr o risco de depender de uma análise amortizada. Por mais que raro, se o pior caso ocorrer na Hash, todo o sistema pode ser comprometido. Além disso, uma de suas principais vantagens reside na sua eficiência de memória, uma vez que realiza alocação exata nó a nó (sem vetores pré-alocados). Com isso, ela consome cerca de 3 vezes menos memória que a Hash. Ademais, sua estrutura favorece operações de travessia completa, superando a Hash na multiplicação por escalar.

6 Conclusão

O presente relatório analisou com sucesso as complexidades assintóticas de ambas estruturas como também validou experimentalmente quais as circunstancias em que é preferível a escolha e uma em detrimento da outra. Como visto, apesar de mais eficientes quando o nível de esparsidade é baixo, a estrutura simples de matriz é inviável para computadores comuns em instancias muito grandes(maiores que $10^4 \times 10^4$)), tanto em memória quanto em tempo.

Além disso, observamos que mesmo assintoticamente melhor em muitas demonstrações, o Hash em alguns casos performou de maneira similar a Arvore Rubro-Negra, o que pode ser atribuído a uma constante multiplicativa maior no hash e a interferência de casos cuja o número de colisões foi significativo, para a qual os testes realizados não foram suficientemente diversos, por limitações em equipamento, para enxergar clara disparidade computacional.

Portanto, tendo em vistas os resultados obtidos, fica claro que não existe estritamente uma resposta correta. Cabe ao leitor definir suas prioridades e escolher a estrutura que mais o convém, tendo em vistas suas necessidades. Enquanto a Tabela Hash oferece o melhor desempenho médio para acesso e manipulação, a LLRBT garante maior economia de memória e segurança no pior caso, independentemente do fator de colisão.