Implementação e Análise de Matrizes Esparsas Utilizando Listas Encadeadas Circulares

Antonio Willian Silva Oliveira[1], **Iago de Oliveira Lo**[2]

[1]Aluno do Campus Quixadá (CE)– Universidade Federal do Ceará (UFC)
Av. José de Freitas Queiroz, 5003 – Cedro – Quixadá – Ceará 63902-580– Brazil

[2]Aluno do Campus Quixadá (CE)– Universidade Federal do Ceará (UFC)
Av. José de Freitas Queiroz, 5003 – Cedro – Quixadá – Ceará 63902-580– Brazil

williansilva@alu.ufc.br, iagooliveiralo07@alu.ufc.br

Abstract. *The adopted structure uses circular linked lists to represent the rows and columns of the matrix, allowing dynamic and efficient access to nonzero elements. Each node points to the next in the same row and to another in the same column, which is in the next row, ensuring the circularity of the matrix. This method prevents memory waste and improves performance compared to conventional matrices. During development, we faced challenges such as maintaining the circularity of the linked lists and managing memory. The tests performed indicated that the implemented structure correctly performs its function as expected and is efficient for large sparse matrices, significantly reducing memory usage. We conclude that using circular linked lists to represent sparse matrices is a viable and advantageous approach, especially for applications that require high efficiency.*

Resumo. *Neste relatório, apresentamos a implementação de uma matriz esparsa circular utilizando listas encadeadas. O objetivo foi desenvolver uma estrutura eficiente para armazenar e manipular matrizes esparsas, economizando espaço e facilitando operações como inserção, soma e multiplicação. A estrutura adotada utiliza listas encadeadas circulares para representar as linhas e colunas da matriz, permitindo um acesso dinâmico e eficiente aos elementos não nulos. Cada nó aponta para o próximo na mesma linha e para outro da mesma coluna, que está na linha seguinte, garantindo a circularidade da matriz. Esse método evita desperdício de memória e melhora o desempenho em comparação com matrizes convencionais. Durante o desenvolvimento, enfrentamos desafios como a manutenção da circularidade das listas encadeadas e o gerenciamento da memória. Os testes realizados indicaram que a estrutura implementada executa corretamente sua função conforme o esperado e é eficiente para matrizes esparsas grandes, reduzindo significativamente o uso de memória. Concluímos que o uso de listas encadeadas circulares para representar matrizes esparsas é uma abordagem viável e vantajosa, especialmente para aplicações que demandam alta eficiência.*

1. Introdução

As matrizes se caracterizam como arranjos bidimensionais de números organizados em linhas e colunas, sendo bastante utilizadas, por exemplo, nas áreas de matemática,

computação e programação. Nesse contexto, é importante evidenciar que o uso de matrizes na computação e na programação enfrenta um grande desafio: quanto maior o número de elementos de uma matriz, maior será a quantidade de dados armazenados na memória. Por isso, uma forma de tentar minimizar os danos causados por esse problema é o uso da Sparse Matrix.

A Sparse Matrix (Matriz Esparsa) é uma matriz que possui como característica o grande número de elementos nulos ou iguais a 0. Por causa disso, seu uso busca armazenar na memória apenas valores não nulos ou diferentes de 0, fazendo com que a quantidade de dados armazenados diminua consideravelmente. Essa característica a torna especialmente útil quando a eficiência de armazenamento e o desempenho computacional são essenciais na aplicação.

Este trabalho tem como objetivo apresentar a Matriz Esparsa, mostrando detalhes da sua implementação e da sua aplicação.

2. Estrutura de Dados Implementada

A implementação da matriz esparsa baseia-se em uma abordagem dinâmica e modular, que utiliza exclusivamente a alocação de memória por nós, em vez do uso tradicional de arrays. Essa estratégia permite representar eficientemente matrizes em que a grande maioria dos elementos é nulo. A matriz esparsa utiliza uma estrutura de dados flexível e escalável que proporciona uma alocação de memória otimizada, evitando desperdícios significativos e promovendo maior eficiência no armazenamento e processamento.

2.1. Representação dos Nós

Cada elemento relevante da matriz é encapsulado em um nó, definido pela estrutura Node. Esse nó contém os seguintes campos:

```
1    struct Node {
2        Node *direita;  // (direcao horizontal)
3        Node *abaixo;   // (direcao vertical)
4        int linha;
5        int coluna;
6        double valor;
7    };
```

Listing 1. Estrutura do Nó em C++

- **Valor:** Representa o dado numérico armazenado na célula.
- **Índices (linha e coluna):** Indicam a posição do elemento dentro da matriz.
- **Ponteiros direita e abaixo:** Estes ponteiros conectam o nó aos seus vizinhos na mesma linha e coluna, respectivamente.

2.2. Estrutura da Matriz

A classe Matriz utiliza um nó especial denominado cabeçalho (ou ”sentinela”), que serve como ponto de referência para toda a estrutura. A partir deste nó, são geradas duas listas circulares:

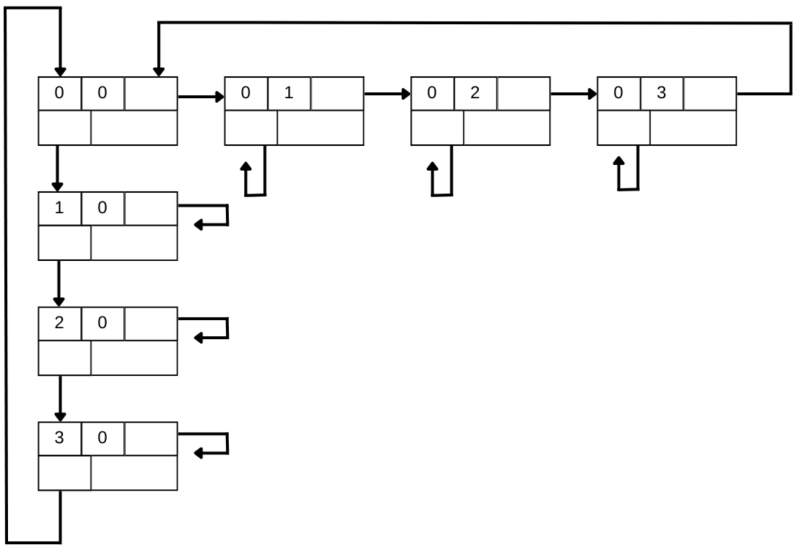

Figure 1. Esquema de uma Matriz Esparsa vazia.

- **Lista de Linhas:** Cada linha da matriz é representada por uma lista circular de nós. Durante a construção, para cada linha, um novo nó sentinela é criado e ligado através do ponteiro abaixo. Ao final do laço, a última linha aponta de volta para o nó cabeçalho, completando a circularidade vertical da estrutura.
- **Lista de Colunas:** Similarmente, cada coluna é organizada como uma lista circular. Os nós sentinelas para as colunas são gerados e conectados através do ponteiro direita, e a última coluna retorna ao cabeçalho, estabelecendo a circularidade horizontal.

2.3. Iterador para a Matriz

A classe IteratorM permite percorrer a matriz de forma sequencial, abstraindo os detalhes de navegação das listas circulares. O iterador é definido com os seguintes atributos:

```
1     class IteratorM {
2         Node *cabecalho;
3         Node *current;
4 }
```

Listing 2. Estrutura do Iterator em C++

- **Ponteiro para o nó cabeçalho:** Permite manter o ponto de referência da estrutura.
- **Ponteiro para o nó atual**: Indica a posição corrente durante a iteração.
- Os operadores sobrecarregados (como ***operator****, ***operator***$\rightarrow$ e ***operator*++**) possibilitam a utilização de construções típicas de iteração (por exemplo, em laços for) para acessar os valores dos nós, garantindo uma interface consistente e intuitiva para o usuário da estrutura.

3. Decisões de Implementação

Durante o desenvolvimento deste projeto, as decisões sobre a implantação foram tomadas após uma análise dos requisitos, considerando as limitações impostas pelo problema específico e as diversas abordagens possíveis. Para cada uma dessas abordagens, avaliamos

aspectos como complexidade computacional, facilidade de manutenção e impacto na performance. O foco principal dessas escolhas foi otimizar fatores essenciais, como a eficiência do sistema, a escalabilidade para suportar crescimento futuro e a robustez da solução em relação a possíveis falhas ou imprevistos. Além disso, buscamos garantir que a implementação atendesse de forma eficaz e equilibrada tanto aos objetivos de desempenho quanto às limitações de uso de memória, visando proporcionar uma solução que fosse não apenas funcional, mas também sustentável e capaz de se adaptar a diferentes cenários de utilização.

3.1. Abstração através de Iteradores

Uma de nossas decisões foi criar um iterador personalizado, chamado **IteratorM**, com o intuito de isolar a complexidade associada à navegação na estrutura de dados. Essa decisão visou estabelecer uma interface de iteração que abstraísse completamente os detalhes internos da organização dos nós, permitindo que os usuários da matriz percorressem os elementos não nulos de forma sequencial e intuitiva, sem a necessidade de compreender a implementação subjacente. O iterador foi desenvolvido para suportar operações fundamentais, como desreferenciação e incremento, seguindo os princípios e paradigmas da biblioteca padrão do C++, garantindo compatibilidade e consistência com as práticas comuns da linguagem.

Além de proporcionar uma navegação eficiente e transparente, essa abstração facilita o uso da estrutura de dados e contribui para a modularização do código. Com a separação das responsabilidades de iteração, o código torna-se mais coeso e de fácil manutenção, pois alterações na implementação interna da estrutura de dados não afetam diretamente a forma como os usuários interagem com ela. Assim, a implementação do **IteratorM** assegura uma solução que é, ao mesmo tempo, flexível, escalável e de fácil adaptação a futuras modificações ou expansões.

4. Divisão do Trabalho

O desenvolvimento deste projeto foi realizado em colaboração, com responsabilidades distribuídas de maneira estratégica após algumas sessões de brainstorming, garantindo uma integração eficaz entre as diferentes fases da implementação. A seguir, detalhamos como o trabalho foi organizado e as principais responsabilidades atribuídas a cada membro da equipe.

4.0.1. Responsabilidades Individuais e Colaborativas

- **Responsável:** Willian Silva
  - Desenvolvimento e Implementação da Interface Visual do Programa (Função Principal - Main);
  - Desenvolvimento dos métodos de construção e de soma entre matrizes;
  - Criação e integração dos nós sentinelas para a formação das listas circulares, assegurando a conexão entre linhas e colunas.
- **Responsável:** Iago de Oliveira:
  - Desenvolvimento das funções de acesso (get), limpeza, liberação de memória e operação de multiplicação de matrizes, com foco na eficiência e na correta manipulação dos nós;

  - Realização de testes unitários e de integração para validar a funcionalidade de cada operação, assegurando a aderência aos requisitos de desempenho e integridade estrutural;
  - Implementação da classe iteradora (IteratorM), que permite a navegação sequencial e a abstração dos detalhes internos da estrutura de dados.
- **Responsabilidade Compartilhada**
  - Implementação do método de Inserção (Insert);
  - Elaboração do relatório técnico, documentando detalhadamente a estrutura de dados implementada, as decisões de implantação, as dificuldades encontradas e os resultados dos testes;
  - Realização da análise de complexidade das funções críticas (insert, get e sum), com a descrição dos casos de pior cenário e justificativas teóricas para os resultados obtidos;
  - Compilação dos resultados dos testes e validação do comportamento da matriz esparsa em cenários variados, garantindo que a solução fosse robusta e escalável.
  - Revisão final do código e da documentação, assegurando a coerência e a clareza de todo o material entregue;
  - Identificação e correção de bugs, além da otimização dos algoritmos de busca e inserção.

5. Dificuldades Encontradas

Durante o desenvolvimento deste projeto, identificamos e enfrentamos diversos desafios técnicos e conceituais. Esses obstáculos exigiram uma análise aprofundada e soluções criativas para garantir a integridade e a eficiência da implementação. A seguir, apresentamos as principais dificuldades encontradas e as estratégias adotadas para superá-las:

5.0.1. Manutenção da Circularidade das Listas Encadeadas

Uma das maiores dificuldades iniciais foi criar listas encadeadas circulares para representar as linhas e colunas. Garantir que o nó cabeçalho estivesse corretamente conectado a ambas as listas (horizontal e vertical) foi essencial para preservar a integridade da estrutura. Durante a fase de testes, constatou-se que pequenas inconsistências na lógica de fechamento das listas poderiam resultar em loops infinitos ou falhas na navegação, comprometendo a execução das operações de inserção, busca e iteração. A resolução desse problema exigiu uma revisão minuciosa dos algoritmos de conexão e a implementação de condições de parada bem definidas, além de uma série de testes rigorosos para validar a consistência da circularidade em diversos cenários.

5.0.2. Gerenciamento de Memória Dinâmica

Outra dificuldade bastante significativa foi o gerenciamento eficiente da memória, pois toda a estrutura da matriz esparsa depende exclusivamente da alocação dinâmica de nós. A ausência de arrays ou estruturas fixas fez com que cada inserção e remoção demandasse atenção especial para evitar vazamentos de memória. Para mitigar esse risco,

foi necessário desenvolver mecanismos robustos de limpeza e liberação de recursos, como a função limpar(), que percorre de forma segura e sistemática toda a estrutura. A implementação exigiu rigor na verificação das conexões entre nós e na manutenção da circularidade, garantindo que nenhum nó permanecesse alocado inadvertidamente após a finalização das operações.

5.0.3. Implementação e Integração do Iterador

A criação da classe iteradora (IteratorM) foi um ponto crítico para facilitar a navegação na estrutura. O iterador precisava abstrair a complexidade interna da matriz esparsa, permitindo que o usuário percorresse os elementos não nulos de forma intuitiva. No entanto, a implementação do operador de incremento apresentou desafios, especialmente em relação à manutenção da circularidade da estrutura, evitando que o iterador entrasse em um loop infinito. Além disso, a necessidade de manter a coerência entre o nó atual e o cabeçalho exigiu uma série de ajustes finos e a implementação de condições de verificação, além de testes para garantir que a interface do iterador fosse robusta e confiável.

Em síntese, as dificuldades enfrentadas durante o desenvolvimento do projeto foram superadas por meio da combinação de análise teórica, desenvolvimento iterativo e rigorosos testes de validação. Esses desafios não apenas contribuíram para o aprimoramento da solução, mas também proporcionaram um aprendizado valioso sobre técnicas avançadas de gerenciamento de memória, manutenção de estruturas circulares e otimização de algoritmos, consolidando a base para futuras implementações.

6. Testes Executados

A criação de um arquivo responsável pela execução de teste foi necessário para validar as operações fundametais da Matriz Esparsa, tais como inserção de valores, soma, multiplicação e análise de desempenho. Dessa forma, os testes buscam garantir a corretude e a eficiência das operações implementadas na matriz.

6.1. Metodologia

Os testes foram implementados em C++ e utilizam as bibliotecas iostream, fstream, stdexcept e chrono. As funções principais testadas são:

6.1.1. Teste de Inserção

Neste teste, uma matriz 5x5 é inicialmente criada com todos os seus elementos definidos como nulos. Em seguida, utiliza-se a função insert() para inserir valores em posições específicas da matriz. Por exemplo, podem ser inseridos:

- O valor 1 na posição (1, 1);
- O valor 2 na posição (5, 5);
- O valor 2 na posição (3, 2).

Após cada inserção, a função get() é utilizada para recuperar o valor armazenado na posição correspondente. A verificação é feita utilizando o assert(), que compara o

valor retornado com o valor esperado. Caso haja divergência, o assert() dispara um erro, indicando que a operação de inserção não foi realizada corretamente.

Esta abordagem garante que cada inserção seja validada individualmente, facilitando a identificação de possíveis problemas na implementação da função insert().

6.1.2. Teste de Soma de Matrizes com mesmo número de linhas e colunas(tamanho)

Neste teste, três matrizes são carregadas a partir de arquivos externos. Duas delas representam os operandos que serão somados utilizando a função `sum()`, enquanto a terceira contém o resultado esperado para essa operação.

Após a execução da função `sum()`, o teste procede à verificação dos resultados. Para isso, é utilizado o método `get()` para acessar cada elemento da matriz resultante e da matriz esperada, comparando os valores correspondentes. Se for detectada qualquer discrepância entre os valores, uma exceção é lançada, indicando que houve falha na implementação da soma.

Esta abordagem garante que tanto as dimensões quanto os valores individuais das matrizes sejam validados, assegurando a correção da operação de soma.

6.1.3. Teste de Multiplicação de Matrizes com mesmo tamanho

Similar ao teste de soma, mas utilizando a função `multiply()`. Três matrizes são carregadas a partir de arquivos, sendo duas as matrizes que serão multiplicadas e a terceira como o resultado do produto.

Após a multiplicação, o método `get()` é chamado para acessar cada elemento da matriz resultante e da matriz esperada, comparando os valores correspondentes. Se for detectada qualquer discrepância entre os valores, uma exceção é lançada, indicando que houve falha na implementação da multiplicação.

6.1.4. Teste de Multiplicação e de Soma com Matrizes de Tamanhos Diferentes

Neste teste, avaliamos o comportamento das funções `sum()` e `multiply()` quando aplicadas a matrizes com dimensões incompatíveis para as operações pretendidas.

Para a soma, a regra é que ambas as matrizes devem possuir o mesmo número de linhas e colunas. Já para a multiplicação, é necessário que o número de colunas da primeira matriz seja igual ao número de linhas da segunda. No cenário testado, as matrizes utilizadas não atendem a nenhum desses critérios, ou seja, suas dimensões são incompatíveis para realizar a soma ou a multiplicação.

Para garantir um tratamento adequado dos erros, as chamadas às funções `sum()` e `multiply()` foram encapsuladas em blocos `try catch(...)`, de forma que quaisquer exceções lançadas devido às incompatibilidades de tamanho sejam capturadas e processadas. O comportamento esperado é que, ao tentar realizar as operações, a aplicação lance exceções informando claramente o problema de dimensão, e assim, o programa se comporta de forma robusta ao detectar e tratar entradas inválidas.

Esta abordagem confirma que o sistema é capaz de identificar e reportar erros quando as operações são solicitadas com matrizes de tamanhos diferentes, evitando resultados inesperados e garantindo a integridade da aplicação.

6.1.5. Teste de Performance

O objetivo deste teste é medir o tempo de execução, em milissegundos (ms), das funções `sum()` e `multiply()` utilizadas na manipulação de matrizes esparsas.

Para a realização do teste, são criadas duas matrizes de dimensão 100x100. A primeira matriz é preenchida com valores definidos pelo padrão i + j, onde i representa o índice da linha e j o índice da coluna. A segunda matriz, por sua vez, é preenchida utilizando o padrão i - j.

A medição do tempo é realizada com o auxílio da biblioteca `chrono`. São registrados os instantes de início e término da execução das operações, permitindo o cálculo da duração de cada função em milissegundos. Ao final, os tempos obtidos são exibidos no console, fornecendo uma avaliação da performance das funções `sum()` e `multiply()`.

7. Análise de Complexidade

7.1. Função `Insert()`

A função `insert()` é uma das operações centrais na implementação da matriz esparsa, responsável por inserir novos elementos (não nulos) na estrutura de dados que representa a matriz.

Por isso, a análise de complexidade dessa função nos permite identificar os pontos críticos e determinar o pior caso teórico da operação. Nesta análise, serão considerados os passos envolvidos na busca da posição correta para inserção—tanto na dimensão das linhas quanto das colunas—e os ajustes dos ponteiros na estrutura de dados.

Além disso, é importante observar que a função insert() primeiramente verifica se existe valor na posição desejada e, caso exista, atualiza o valor. No entanto, caso não exista, é feita a criação de um novo nó.

7.1.1. Verificação inicial:

- A comparação if (value == 0) e a validação dos índices (**if** (posI ≤ 0 || posI $>$ linhas || posJ ≤ 0 || posJ $>$ colunas)) são operações em tempo constante, ou seja, O(1).
- Busca na Lista Horizontal (Percorrendo Linhas) para encontrar a posição desejada:

```
1                   Node *linhaAtual = cabecalho;
2                   while (linhaAtual->linha < posI)
3                   {
4                       linhaAtual = linhaAtual->abaixo;
5                   }
```

- Nesse loop, começamos a partir do nó cabeçalho e vamos descendo pela lista vertical até atingir a linha posI.
- No pior caso, se posI for a última linha, o loop executará linhas iterações, tendo, assim, complexidade O(linhas).

7.1.2. Busca na Lista Vertical (Percorrendo Colunas). Loop para encontrar a coluna desejada:

```
1            Node *colunaAtual = cabecalho;
2            while (colunaAtual->coluna < posJ)
3            {
4                colunaAtual = colunaAtual->direita;
5            }
```

- Aqui, percorremos os nós na lista horizontal para alcançar a coluna posJ.
- No pior caso, se posJ for a última coluna, o loop executará colunas iterações. Complexidade: O(colunas).

7.1.3. Busca na Lista Horizontal para a criação do novo nó:

```
1            Node *aux = linhaAtual;
2            while (aux->direita != linhaAtual && aux->direita->
                coluna < posJ)
3            {
4                aux = aux->direita;
5            }
```

- Caso não haja valor na posição indicada, esse loop percorre os nós na linha para localizar a posição onde o novo nó deve ser inserido.
- No pior caso, se a linha estiver muito "cheia" (ou seja, se houver muitos nós não nulos nessa linha), o número de iterações pode ser proporcional ao número de elementos na linha. Em uma matriz densamente preenchida, isso poderia se aproximar de O(colunas).

7.1.4. Busca na Lista Vertical para a criação do novo nó:

```
1            aux = colunaAtual;
2            while (aux->abaixo != colunaAtual && aux->abaixo->
                linha < posI)
3            {
4                aux = aux->abaixo;
5            }
```

- Esse loop percorre os nós na coluna para inserir o novo nó na posição adequada.
- No pior caso, se a coluna tiver muitos nós, o número de iterações pode ser proporcional ao número de elementos nessa coluna (até O(linhas) no pior cenário).

7.1.5. Inserção do Novo Nó:

- As operações de ajuste dos ponteiros para inserir o novo nó na lista horizontal e vertical são operações em tempo constante $O(1)$.

7.1.6. Conclusão

Portanto, a complexidade total da função insert() é da ordem de O(linhas + colunas) no pior cenário, onde cada dimensão é percorrida independentemente.

7.2. Função `Get()`

A função `get()` tem a finalidade de recuperar o valor de um elemento armazenado na matriz esparsa. Como essa estrutura armazena apenas os elementos não nulos em uma lista encadeada bidimensional (organizada em linhas e colunas), a busca por um elemento específico deve percorrer os nós existentes até encontrar a posição desejada.

Para realizar essa busca, a função utiliza um iterador `IteratorM`, que percorre os elementos da matriz de forma ordenada. A estratégia de busca adotada avança pelo iterador enquanto o nó atual estiver "antes" da posição desejada, garantindo que o acesso seja realizado de maneira eficiente.

7.2.1. A implementação da função `get()` segue os seguintes passos principais:

- **Validação dos Índices**
  Antes de realizar a busca, a função verifica se os índices informados (`posI`, `posJ`) são válidos. Caso contrário, uma exceção é lançada. Essa verificação ocorre em $O(1)$.
- **Inicialização do Iterador**
  A busca é realizada por meio do iterador `IteratorM`, que percorre os elementos da matriz esparsa começando do primeiro elemento armazenado.
- **Busca Sequencial pelo Elemento**
  O loop principal da função percorre os nós armazenados até encontrar a posição desejada ou até ultrapassá-la. Esse processo ocorre no trecho:

```
1     while (it != end() && (it.current->linha < posI || (it.
         current->linha == posI && it.current->coluna < posJ)
         ))
2          {
3               ++it;
4          }
```

  Esse trecho percorre os nós existentes de maneira ordenada, avançando até encontrar a posição desejada.
- **Verificação e Retorno do Valor**
  - Se o nó encontrado corresponder exatamente à posição (`posI`, `posJ`), a função retorna seu valor.
  - Caso contrário, a função retorna 0, indicando que a posição não está armazenada na matriz esparsa.

7.2.2. Análise de Complexidade:

A eficiência da função `get()` depende do número de elementos armazenados na matriz e da posição do elemento buscado.

- **Melhor Caso –** $O(1)$ Se a matriz for muito esparsa e a posição buscada não estiver armazenada, a função pode rapidamente identificar a ausência do elemento e retornar 0 sem percorrer toda a estrutura. Isso acontece quando a busca para logo no início, tornando a complexidade $O(1)$.
- **Caso Médio –** $O(k)$, onde k é o número de elementos não nulos percorridos Em uma matriz esparsa comum, a busca precisa percorrer apenas os nós armazenados, ou seja, os elementos não nulos. O tempo de execução dependerá do número de elementos existentes antes da posição desejada. Dessa forma, esse valor será pequeno, tornando a busca eficiente.
- **Pior Caso –** $O(n)$, onde n é o número total de elementos não nulos No pior cenário, se a matriz estiver quase cheia (com poucos elementos nulos), a busca pode precisar percorrer todos os elementos antes de encontrar a posição desejada. Nesse caso, a complexidade pode chegar a $O(n)$, onde n é o número total de elementos não nulos da matriz.
  No entanto, mesmo nesse cenário, o desempenho ainda pode ser melhor que uma busca em uma matriz densa tradicional, pois a função só percorre elementos realmente armazenados.

7.2.3. Conclusão:

A complexidade da função `get()` varia conforme a estrutura da matriz:

- $O(1)$ no melhor caso (quando o elemento não existe e é rapidamente identificado).
- $O(k)$ no caso médio (onde k é o número de elementos não nulos percorridos).
- $O(n)$ no pior caso (quando precisa percorrer todos os elementos armazenados).

Em uma matriz esparsa típica, a complexidade esperada tende a ser $O(k)$, tornando a busca eficiente. Dessa forma, a estrutura da matriz esparsa permite um acesso relativamente rápido aos elementos armazenados, mesmo sem o uso de uma indexação tradicional.

7.3. Função `Sum()`

A função `sum` tem o objetivo de somar duas matrizes (digamos, matrixA e matrizB) e retornar uma nova matriz com o resultado da soma. A estrutura do código é a seguinte:

7.3.1. Verificação das Dimensões

```
1    if (matrixA.getLinhas() != matrizB.getLinhas() || matrixA.
        getColunas() != matrizB.getColunas())
2    {
3        throw std::invalid_argument("Erro: As matrizes nao
            possuem o mesmo tamanho");
4    }
```

Nesta etapa, são feitas duas comparações simples para verificar se o número de linhas e colunas das matrizes matrixA e matrizB são iguais. Comparar valores inteiros é uma operação de tempo constante.

- Complexidade desta etapa: $O(1)$

Esta etapa é extremamente eficiente e não influencia significativamente na complexidade total da função.

7.3.2. Criação da Nova Matriz

```
Matriz matriz(matrixA.getLinhas(), matrixA.getColunas());
```

A criação da nova matriz é feita através do construtor da classe Matriz. O construtor é responsável por inicializar os cabeçalhos de linha e coluna, criando uma estrutura que facilita o acesso e a inserção de elementos.

- O que o Construtor Faz:
  - Inicializa o cabeçalho principal com um nó que aponta para si mesmo.
  - Cria nós cabeçalho para cada linha: Um laço de 1 até n (onde n é o número de linhas) cria um nó para cada linha.
  - Cria nós cabeçalho para cada coluna: Outro laço de 1 até m (onde m é o número de colunas) cria um nó para cada coluna.
- Complexidade da Criação da Matriz: Como cada linha e coluna é inicializada separadamente a complexidade será $O(n + m)$, sendo n o número de linhas e m o número de colunas

Esta etapa é linear em relação à soma das dimensões da matriz, o que é eficiente considerando o contexto geral.

7.3.3. Iteração Sobre os Elementos

```
    for (int i = 1, linha = matrixA.getLinhas(); i <= linha; i
       ++)
{
    for (int j = 1, coluna = matrixA.getColunas(); j <= coluna;
       j++)
    {
        double valor = matrixA.get(i, j) + matrizB.get(i, j);
        matriz.insert(i, j, valor);
    }
}
```

A função utiliza dois laços aninhados para percorrer todos os elementos da matriz. O laço externo percorre as linhas e o laço interno percorre as colunas.

- **Número total de iterações**: $n \cdot m$
  - Onde n é o número de linhas e m o número de colunas.
- Cada iteração realiza duas operações principais:
  - Acesso aos valores com o método get.
  - Inserção do valor somado com o método insert.

A seguir, detalharemos a complexidade dessas operações.

7.3.4. Acesso aos Valores com o Método `get`

```
1     double valor = matrixA.get(i, j) + matrizB.get(i, j);
```

O método `get` é responsável por acessar o valor na posição (i, j) da matriz. A busca é feita percorrendo a lista ligada até encontrar o nó correspondente.

- Como o `get` Funciona:
  - Começa no nó cabeçalho da matriz.
  - Percorre a lista até encontrar a linha i.
  - Dentro da linha i, percorre até encontrar a coluna j.

Complexidade do `get`: O pior caso ocorre quando o elemento desejado está no final da linha ou coluna, o que exige percorrer todos os nós daquela linha ou coluna.

- **Complexidade por chamada:**

$$O(max(n, m) \tag{1}$$

Como cada iteração da função `sum` faz duas chamadas ao `get` (uma para cada matriz), o custo total por iteração é:

- **Complexidade total do get por iteração:**

$$2 \cdot O(max(n, m) = O(max(n, m) \tag{2}$$

7.3.5. Inserção com o Método `insert`

```
1     matriz.insert(i, j, valor);
```

O método insert insere um novo valor na posição (i, j) da matriz, considerando a estrutura esparsa.

- Como o insert Funciona:
  - Verifica se o valor é zero. Se for, não insere.
  - Percorre a lista de nós na linha i até encontrar a posição correta.
  - Se o nó já existir, atualiza o valor; caso contrário, cria um novo nó.
  - Repete o processo para a coluna j para manter a estrutura consistente.
- Complexidade do `insert`:

$$O(n + m) \tag{3}$$

7.3.6. Complexidade Total da Função `sum`

Agora, combinamos as complexidades das etapas:

1. Verificação das dimensões: $O(1)$
2. Criação da nova matriz: $O(n + m)$
3. Iteração sobre os elementos: Cada uma das $n \cdot m$ iterações realiza operações de custo para o acesso e $O(max(n, m))$ para a inserção. Como $O(n + m)$ domina $O(max(n, m))$, o custo por iteração é: $O(n + m)$

O custo total da iteração é:

$$O(n \cdot m) \cdot O(n + m) \tag{4}$$

Para matrizes quadradas, onde $n = m$:

$$O(n^2) \cdot O(2n) = O(n^3) \tag{5}$$

7.3.7. Conclusão

A análise detalhada demonstra que a função `sum` possui uma complexidade final de $O(n^3)$ para matrizes quadradas. A estrutura esparsa com cabeçalhos de linha e coluna otimiza o acesso e a inserção de elementos, mas o custo ainda cresce de forma cúbica em relação ao tamanho da matriz. Em cenários onde as matrizes são muito grandes, essa complexidade pode impactar o desempenho, sendo necessário considerar otimizações adicionais ou abordagens alternativas para reduzir o tempo de execução.

References

Cplusplus (2025). [Online; acessed 31-January-2025].

John L. Gardenghi (2019). Matrizes esparsas. [Online; accesed 30-January-2025].

N. Ziviani, F. C. B. (2006). *Projeto de Algoritmos com implementaçõoes em Java e C++*. Editora Cengage Learning.

Univesp (2017). Estrutura de Dados - Aula 14 - Matriz esparsa. [Online; accesed 25-January-2025].

Viktor (2021). Matrix Calculator. [Online; accesed 02-January-2025].

Wikipedia contributors (2025). Sparse matrix — Wikipedia, The Free Encyclopedia. [Online; accessed 30-January-2025].

[Cplusplus 2025] [N. Ziviani 2006] [Univesp 2017] [John L. Gardenghi 2019] [Viktor 2021] [Wikipedia contributors 2025]