

# Trabalho de Estrutura de Dados

José Natan dos Santos  
Antonio Elivelton Moura Da Silva

Novembro 2022

## 1 Introdução

O intuito deste trabalho é implementar uma matriz esparsa usando linguagem C++. Este documento detalha os 4 arquivos utilizados para construir todo o programa, são eles: Node.h, SparseMatrix.h, SparseMatrix.cpp e main.cpp. Para cada arquivo será descrito as suas principais partes e suas funções, assim como as ideias por trás da criação das mesmas. Esse relatório também apresenta uma seção com os códigos que podem ser utilizados ao executar o programa. Um detalhe importante é que o relatório possui diversas imagens sobre o código que está a ser descrito, e com as imagens é possível ver os comentários feitos no código que poderão servir de complemento ao que está sendo descrito. Um segundo ponto importante é que algumas funções que possuem partes com implementação semelhante podem ter sido desenvolvidas de forma diferente mesmo tendo o mesmo objetivo, exemplo: utilizando uma função auxiliar para a resolução de uma etapa em uma dada função e em outra com a mesma etapa sendo utilizado uma implementação na própria função.

## 2 Comandos

| Função        | Comando          | Variáveis   |
|---------------|------------------|---|
| Create        | create l c       | (número de linhas)(número de colunas)                 |
| Print         | print id         | (número da matriz)                                    |
| Get           | get id l c       | (número da matriz)(número da linha)(número da coluna) |
| Remove        | remove id l c    | (número da matriz)(número da linha)(número da coluna) |
| Insert        | insert id l c v  | (matriz)(linha)(coluna)(valor a ser inserido)         |
| Mostrar todos | showAll          |   |
| Finalizar     | end              |   |
| Somar         | sum id1 id2      | (número das matrizes)                                 |
| Multiplicar   | multiply id1 id2 | (matriz A)(matriz B)                                  |
| Ler Arquivo   | read titulo.txt  | (nome do arquivo)                                     |

Tabela 1: tabela de comandos

## 3 Classe Node

### 3.1 classe node private

A classe node servirá para a criação dos nós da matriz esparsa, cada nó possui 5 valores: Um valor para a identificar a linha, um valor para identificar a coluna, um valor do tipo double para a posição linha coluna e por fim dois ponteiros que apontarão para os próximos nós na mesma linha e coluna. Essa classe também dará acesso ao seu conteúdo para a classe SparseMatrix que será abordada mais adiante no relatório.

```
class Node{
    friend class SparseMatrix;
private:
    Node *right_pointer;
    Node *below_pointer;
    int row;
    int column;
    double value;
```

Figura 1: definição do private da classe node

### 3.2 Classe node Public

A classe node possui apenas um elemento na sua sessão public, que é o seu construtor. O construtor da classe node receberá valores referentes aos citados na sessão private e os atribuirá ao valores referidos.

```
public:
    //constructor
    Node(int row_index, int column_index, const int &position_value, Node *right, Node *below){
        row = row_index;
        column = column_index;
        value = position_value;
        right_pointer = right;
        below_pointer = below;
    }
```

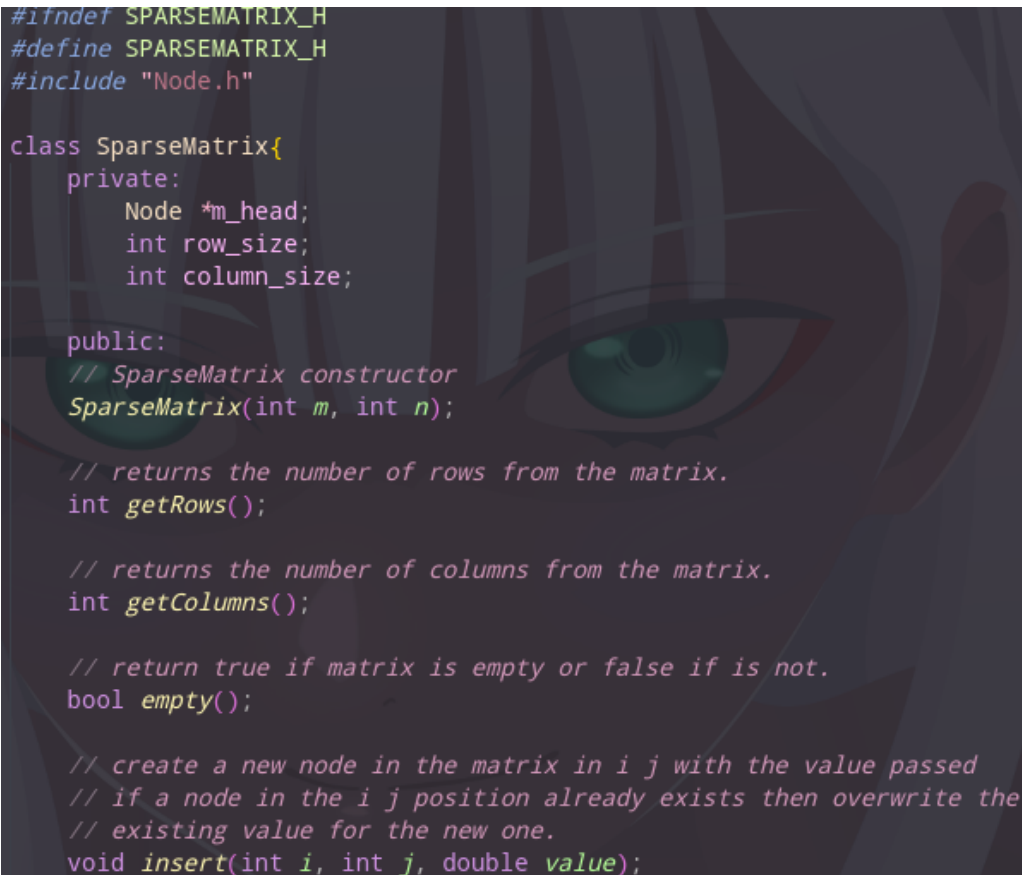
Figura 2: definição do public da classe node

Sem muita complexidade ou dificuldades durante a construção do Node.h após o seu construtor apenas encerramos o arquivo com o 'endif' e os nós estão prontos para serem utilizados.

## 4 SparseMatrix.h

### 4.1 Definição das funções

Este arquivo é também bem simples, possui apenas a definição das funções que serão implementadas no SparseMatrix.cpp, porém é válido comentar que os comentários com as descrições de cada função estão presentes aqui no SparseMatrix.h. A SparseMatrix é constituída de um nó cabeça e a quantidade de linhas e colunas como mostrado na figura 3 abaixo:

The image shows a code editor with a dark background and a faint anime-style character in the background. The code is for the SparseMatrix.h header file. It includes preprocessor directives for conditional compilation, an include for Node.h, and a class definition for SparseMatrix. The class has private attributes for a head pointer, row size, and column size. The public methods include a constructor, getters for row and column counts, an empty check, and an insert method.

```
#ifndef SPARSEMATRIX_H
#define SPARSEMATRIX_H
#include "Node.h"

class SparseMatrix{
private:
    Node *m_head;
    int row_size;
    int column_size;

public:
    // SparseMatrix constructor
    SparseMatrix(int m, int n);

    // returns the number of rows from the matrix.
    int getRows();

    // returns the number of columns from the matrix.
    int getColumns();

    // return true if matrix is empty or false if is not.
    bool empty();

    // create a new node in the matrix in i j with the value passed
    // if a node in the i j position already exists then overwrite the
    // existing value for the new one.
    void insert(int i, int j, double value);
```

Figura 3:

Ao todo 10 funções foram definidas:

itemize

- SparseMatrix
- getRows
- getColumns
- empty
- insert
- get

- validatePosition
- print
- remove
- ~SparseMatrix

```

// return the value in the i j position.
double get(int i, int j);

// returns true if the position already exists or false if not.
bool validatePosition(int i, int j);

// prints the matrix.
void print();

// remove the value in the i j position.
void remove(int i, int j);

// destructor.
~SparseMatrix();

};

#endif

```

Figura 4: definição das funções da sparsematrix

Nessas 10 funções estão inclusas as funções auxiliares, construtor, destrutor e as funções principais. Todas elas serão descritas e explicadas na sessão SparseMatrix.cpp

## 5 SparseMatrix.cpp

Nessa sessão será aprofundaremos na implementação de cada função, explicando a lógica por de trás e apontando as dificuldades encontradas, assim como, uma análise da complexidade do pior caso das funções requisitas pelo PDF de instruções.

### 5.1 Construtor

O construtor da matriz esparsa, assim como o de uma matriz normal, receberá dois valores que definirão a quantidade de linhas e colunas da matriz. Caso os valores passados para linha e coluna sejam abaixo de 0 uma exceção é lançada.

```

SparseMatrix::SparseMatrix(int m, int n){
    // checks if the row and column values are valid.
    if(m < 0 || n < 0)
        throw std::length_error("Valor inserido para linha ou coluna está abaixo do permitido.");

    m_head = new Node(0, 0, 0, nullptr, nullptr);
    row_size = m;
    column_size = n;

    Node *aux = m_head;
    Node *novo;
    // creates sentinel row nodes.
    for(int i=1; i<=m; i++){
        novo = new Node(i, 0, 0, novo, novo);
        aux->below_pointer = novo;
        aux = novo;
        aux->right_pointer = aux;
    }

    aux->below_pointer = m_head;
    aux = m_head;

```

Figura 5:

A ideia aqui é simples, a partir do nó sentinela(mHead) criar i nós para as linhas e j nós para as colunas, que servirão como nós cabeças para as suas respectivas posições. Como as listas da matriz são circulares isso significa que cada novo nó apontará para si mesmo, com o ponteiro da direita ou o ponteiro para baixo, a depender de sua posição na matriz, e o ultimo nó da linha e coluna apontarão para o nó cabeça da matriz.

```

    // creates sentinel column nodes.
    for(int i=1; i<=n; i++){
        novo = new Node(0, i, 0, novo, novo);
        aux->right_pointer = novo;
        aux = novo;
        aux->below_pointer = aux;
    }

    aux->right_pointer = m_head;
}

```

Figura 6: implementação do construtor

Dado ao uso de nós sentinelas em todas as linhas e colunas da matriz as suas posições válidas e que possuirão valores, começarão a partir da posição 1 e não da 0 como geralmente ocorre em outras estruturas de dados semelhantes a array.

## 5.2 Empty

Empty é uma função auxiliar que basicamente dirá se a matriz que a chamou está vazia ou não. A lógica utilizada é simples, é criado um nó auxiliar que percorrerá todos os nós

sentinelas referentes as linhas da matriz, e para cada linha é checado se existe algum nó existente naquela linha que não seja o próprio nó sentinela, ou seja, se existe de fato algum dado salvo na linha, essa checagem é feita olhando as colunas da mesma linha. Se existir um valor na linha isso significa que a matriz não é vazia e a função retorna false, do contrário a função retorna true pois a matriz está vazia.

```
bool SparseMatrix::empty(){
    Node *aux_row = this->m_head->below_pointer;

    // for each row checks if the row is empty
    for(int i=1; i<=this->row_size; i++){
        if(aux_row->right_pointer != aux_row)
            return false;

        aux_row = aux_row->below_pointer;
    }

    return true;
}
```

Figura 7: função auxiliar empty

### 5.3 Insert

A função mais complexa da matriz esparsa, insert por sua vez tem uma missão muito simples, inserir um novo nó na posição i j da matriz e com o valor passado por parâmetro. De início a função verifica se os valores de i e j passados por parâmetro estão dentro da margem da matriz, ou seja, se nenhum deles é maior que o número máximo de linhas e colunas da matriz, caso seja uma mensagem é imprimida na tela e a execução da função é cancelada.

```

void SparseMatrix::insert(int i, int j, double value){
    // checks if the i and j values are valid for the matrix
    if(i > this->row_size || j > this->column_size){
        std::cout << ("Valores de I ou J são inválidos.") << std::endl;
        return;
    }

    Node *aux_row = this->m_head->below_pointer;
    Node *aux_column = this->m_head->right_pointer;
    Node *newest;
    // reach sentinel row equal to I
    while(aux_row->row != i){
        aux_row = aux_row->below_pointer;
    }
    // reach sentinel column equal to J
    while(aux_column->column != j){
        aux_column = aux_column->right_pointer;
    }
}

```

Figura 8:

Após a verificação é criado três nós, sendo dois deles auxiliares, um para as linhas da matriz e outro para as colunas, e o último será o nó a ser adicionado na matriz. Os nós auxiliares irão percorrer a matriz até que estejam um na linha  $i$  e o outro na coluna  $j$ .

Em seguida é feita uma checagem se a matriz está vazia, utilizando a função `empty`. Caso a matriz esteja vazia significa que esse será o primeiro valor a ser inserido nela, então basta criar o novo nó e fazer ele apontar para os auxiliares e os auxiliares apontarem para ele.

Caso a matriz não esteja vazia as coisas ficam um pouco mais complicadas, por conta disso são feitos mais dois novos nós para facilitar a resolução do problema, esses nós servirão como auxiliares dos auxiliares já criados anteriormente, as coisas podem começar a ficar confusas mas é só questão de nomenclatura. Cada novo nó auxiliar receberá os nós que eles vão auxiliar, no caso temos o nó `auxrow` que é o nó auxiliar das linhas da matriz e ele possui agora um auxiliar chamado `auxr1` que é inicializado com `auxrow`, o mesmo vale para o auxiliar das colunas (`auxcolumn`).

```

if(this->empty()){
    newest = new Node(i, j, value, aux_row, aux_column);
    aux_row->right_pointer = newest;
    aux_column->below_pointer = newest;
}else{
    Node *auxr1 = aux_row;
    Node *auxc1 = aux_column;

    // reach node in the position before column j
    while(auxr1->right_pointer != aux_row && auxr1->right_pointer->column < j){
        auxr1 = auxr1->right_pointer;
    }

    // reach node in the position before row i
    while(auxc1->below_pointer != aux_column && auxc1->below_pointer->row < i){
        auxc1 = auxc1->below_pointer;
    }
}

```

Figura 9:

Para quê esses novos auxiliares? Será necessário percorrer pelos nós dentro da matriz e ao mesmo tempo salvar os valores da linha sentinela i e coluna sentinela j, que são respectivamente o auxrow e auxcolumn. Os dois novos auxiliares irão percorrer a matriz, o novo auxiliar das linhas irá até o nó que antecede o nó na coluna j da mesma linha, enquanto o novo auxiliar das colunas irá até a linha que antecede o nó na linha i da mesma coluna.

Depois de salvar a posição dos nós sentinelas i j, e dos nós que estão antes da posição que se deseja inserir um novo nó será necessário fazer o que? Isso mesmo criar mais nós auxiliares! Mas dessa vez serão chamados de temporários para evitar mais dores de cabeça, atente-se ao seguinte da mesma forma que o nó auxr1 é um nó auxiliar do auxrow, o novo nó temporario tempr é um nó auxiliar do auxr1, e o tempc do nó auxc1.

```

Node *tempr = auxr1;
Node *tempc = auxc1;

// checks if the position i j already exists and/or save the next position from it
if(auxr1->right_pointer->column == j){
    tempr = auxr1->right_pointer->right_pointer;
}else{
    tempr = auxr1->right_pointer;
}

if(auxc1->below_pointer->row == i){
    tempc = auxc1->below_pointer->below_pointer;
}else{
    tempc = auxc1->below_pointer;
}

// if the position i j already exists overwrite its value otherwise create a new node in i j
if (auxr1->right_pointer->column == j && auxc1->below_pointer->row == i){
    auxr1->right_pointer->value = value;
}else{
    newest = new Node(i, j, value, tempr, tempc);
    auxr1->right_pointer = newest;
    auxc1->below_pointer = newest;
}
}

```

Figura 10: implementação do insert



A função desses novos nós temporários é salvar o nó que se localiza após a posição  $i, j$ , o motivo é que ao criar um novo nó será necessário fazer ele apontar para os nós seguintes da matriz, esses nós serão os auxiliares temporários, mas também será necessário fazer os nós que antecedem o novo nó apontar para ele, esses nós são os auxiliares dos auxiliares. Confuso? Sim, agora imagine pensar nisso.

Por fim a função faz o que foi dito anteriormente percorre a matriz para salvar nos temporários os nós seguintes aos da posição  $i, j$ , ela faz isso checando se após o nó na posição anterior a  $i, j$  a coluna e linha dele é igual a  $i, j$ , e aqui entra dois casos: Se os valores forem iguais significa que existe um nó naquela posição e o seu valor deve ser sobrescrevido, caso os valores não sejam iguais então não existe um nó na posição  $i, j$ , é criado um novo nó e inserido nessa posição logo após isso a matriz é reajustada com os nós salvos no batalhão de auxiliares criados anteriormente.

### **Análise de complexidade:**

A função `insert` no começo possui dois lados não aninhados que irão percorrer toda a matriz, continuando e considerando o pior caso a função executará mais dois laços que percorrerão toda a matriz novamente, inacreditavelmente a função `insert` no pior caso possui apenas laços não aninhados e não inclui a função auxiliar `empty`, porém a execução de seus laços se dá pelo valor de  $i, j$  que não são necessariamente iguais, logo é correto afirmar que a complexidade do `insert` é  $O(i+j)$ .

## **5.4 Validate Position**

`Validate Position` é mais uma função auxiliar, foi criada por conta da necessidade de verificar se posições  $i, j$  requeridas em certas funções existiam na matriz. Como dito a função verifica a existência da posição  $i, j$ , para isso ela recebe esses valores por parâmetro, e semelhante a função `empty`, será criado um nó auxiliar que percorrerá pelas linhas sentinelas da matriz até chegar na linha  $i$ . Após concluir a primeira parte basta checar se na linha  $i$  existe um nó com o valor da coluna igual a  $j$ , caso ele exista então a função retorna `true`, se não ela retorna `false`.

```
bool SparseMatrix::validatePosition(int i, int j){  
  
    Node *aux_row = this->m_head;  
  
    while(aux_row->below_pointer->row != i){  
        aux_row = aux_row->below_pointer;  
    }  
    aux_row = aux_row->below_pointer;  
  
    // run through the matrix at row i and checks if a node with column j exists  
    for(int i=1; i<=this->column_size; i++){  
        if(aux_row->right_pointer->column == j)  
            return true;  
        aux_row = aux_row->right_pointer;  
    }  
  
    return false;  
}
```

Figura 11: função auxiliar validate position

## 5.5 Get

A função recebe dois valores por parâmetro que serão usados como linha e coluna de uma matriz, e ao fim o valor nessa posição deve ser retornado.

De início é feita uma verificação se a matriz é vazia e se os valores passados são de uma posição válida, caso a matriz não atenda a um desses requisitos é lançada uma exceção. Em seguida é criado um auxiliar, assim como em outras funções ele irá percorrer a matriz até a linha  $i$  e o nó anterior ao nó da coluna  $j$ , vale informar que foi cogitada a criação de uma função auxiliar para fazer esses passos de percorrer o vetor para a posição  $i$  e  $j$ , porém essa ideia se perdeu durante o desenvolvimento.

Por fim o valor na posição  $i$  e  $j$  é retornada.

```
double SparseMatrix::get(int i, int j){
    if(this->empty() || this->row_size < i || this->column_size < j)
        throw std::runtime_error("Matriz Vazia.");

    if(!validatePosition(i, j))
        throw std::runtime_error("Posição Inválida.");

    Node *aux_row = this->m_head->below_pointer;

    // reach the row position before the row i
    while(aux_row->below_pointer->row != i){
        aux_row = aux_row->below_pointer;
    }
    aux_row = aux_row->below_pointer; // move to row i

    // reach the column position before the column j
    while(aux_row->right_pointer->column != j){
        aux_row = aux_row->right_pointer;
    }

    return aux_row->right_pointer->value;
}
```

Figura 12: função get

### Análise de complexidade:

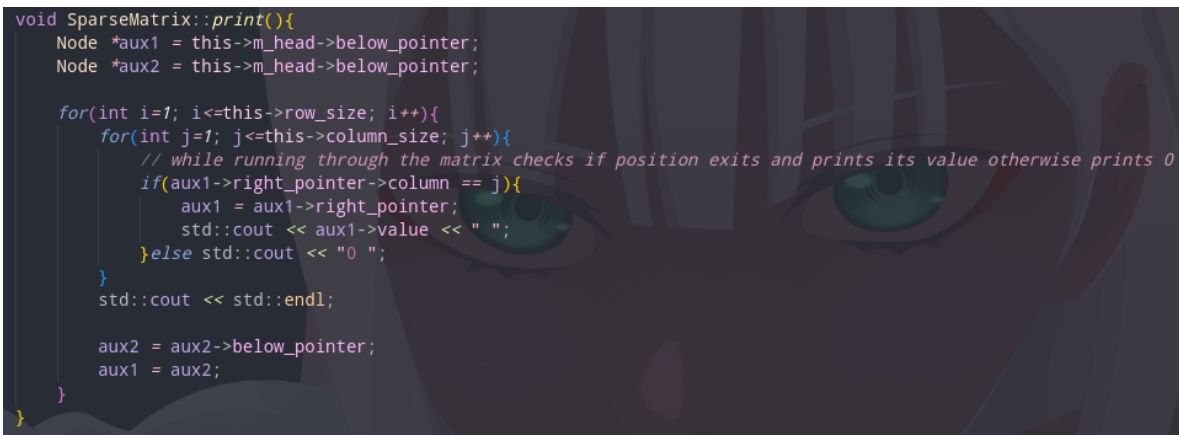
Existem dois laços na função get, o número de vezes que ambos os laços irão executar irá depender dos valores de  $i$  e  $j$ . Como o ponteiro já inicia no next do mhead o pior caso do primeiro laço seria quando o  $i$  fosse 1 e o auxiliar teria que percorrer todas as linhas inclusive voltar para o mhead para então encontrar a linha 1, o que seria equivalente a executar o laço `rowSize`(quantidade de linhas) vezes. Para o segundo laço o pior caso fica ocorre quando  $j$  é igual ao número total de linhas, sendo assim sendo necessário percorrer por todas as colunas da matriz.

Como o número de vezes que os laços executam dependem do valor de entrada na função e os laços não são aninhados a complexidade fica sendo  $O(n)$ .

## 5.6 Print

O objetivo dessa função é imprimir na tela todos os valores da matriz que chamou a função, para uma melhor visualização essa impressão é feita no formato de tabela, com linhas e colunas simulando um desenho de uma matriz de fato. Existe uma restrição porém, a matriz esparsa permite que haja posições sem valores inseridos, o que formaria um espaço em branco na impressão, tendo em vista isso para cada posição  $i$   $j$  na matriz que não possua um nó será imprimido um 0 no lugar.

A função começa criando dois nós auxiliares que percorrerão as linhas da matriz, o primeiro servirá para verificar se existe um nó na coluna  $j$  e na mesma linha que ele, caso exista então o seu valor é impresso do contrário um 0 é imprimido. O segundo nó terá o nó da linha salvo, passará a ter o nó sentinela da linha seguinte e o primeiro auxiliar



```

void SparseMatrix::print(){
    Node *aux1 = this->m_head->below_pointer;
    Node *aux2 = this->m_head->below_pointer;

    for(int i=1; i<=this->row_size; i++){
        for(int j=1; j<=this->column_size; j++){
            // while running through the matrix checks if position exists and prints its value otherwise prints 0
            if(aux1->right_pointer->column == j){
                aux1 = aux1->right_pointer;
                std::cout << aux1->value << " ";
            }else std::cout << "0 ";
        }
        std::cout << std::endl;

        aux2 = aux2->below_pointer;
        aux1 = aux2;
    }
}

```

Figura 13: implementação da função print

receberá esse nó e repetirá o processo até finalizar todas as linhas da matriz. Note-se que seria possível utilizar a função auxiliar Validate Position nessa função, porém dado a simplicidade da função print e como uma forma de mostrar uma resolução alternativa sem a ajuda de funções auxiliares, o print, foi feito dessa maneira.

## 5.7 Remove

Dado uma posição  $i$   $j$  o remove irá deletar o nó na posição indicada. Essa função é um bom exemplo de um bom uso de funções auxiliares, visto que antes de tudo é preciso fazer duas verificações na matriz, a primeira é saber se a matriz é vazia, pois caso seja não haverá nó a ser deletado, a segunda verificar se existe algum nó na posição  $i$   $j$  para ser deletado. Para fazer essas verificações é utilizado as funções auxiliares empty e validate position, observe que mesmo tendo uma grande semelhança as duas possuem propósito e resultado diferentes, e o uso das duas evita de fazer uma função muito extensa e que faça muitas coisas.

Após as verificações é criado um nó auxiliar, ele chegará na linha  $i$  e nessa mesma linha ele irá navegar até o nó que esteja antes do nó da coluna  $j$ , ou seja, o nó que o seu ponteiro da direita aponte para o nó  $i$   $j$ , após isso o nó que é apontado pelo nó  $i$   $j$  é salvo em uma variável temporária e finalmente o nó  $i$   $j$  é deletado e a matriz é reajustada com o auxiliar apontando para o temporário.

## 5.8 Destrutor

Essa função é chamada na hora de deletar uma matriz da memória, para isso dentro da função um auxiliar percorre todos os nós sentinela e deleta um a um até que reste apenas o nó cabeça mhead, tendo apenas o mhead sobrando basta deletá-lo.

# 6 Main

Nesse programa o arquivo main.cpp fica responsável por implementar as funções de leitura de arquivo e soma e multiplicação de matrizes, assim como os comandos utilizados para executar todo o programa e utilizar as funções. Para poder rodar o programa por

```

void SparseMatrix::remove(int i, int j){
    // checks if the matrix is empty
    if(this->empty()){
        std::cout << "ERRO: Matriz vazia" << std::endl;
        return;
    }else{
        // check if a node in the position i j exists
        if(!this->validatePosition(i, j)){
            std::cout << "ERRO: Posição inválida." << std::endl;
            return;
        }

        // reach row i
        Node *aux_row = this->m_head;
        while(aux_row->below_pointer->row <= i){
            aux_row = aux_row->below_pointer;
        }

        // reach node before the node from column j
        while(aux_row->right_pointer->column < j){
            aux_row = aux_row->right_pointer;
        }

        // delete node in position i j and rearranges matrix
        Node *temp = aux_row->right_pointer->right_pointer;
        delete aux_row->right_pointer;
        aux_row->right_pointer = temp;
    }
}

```

Figura 14: implementação da função remove

comandos de forma efetiva é criado um array que armazenará todas as matrizes criadas pelo usuário.

Todos os comandos estão especificados na tabela 1 na sessão 2.

## 6.1 Implementação dos comandos

Comando create utiliza o construtor para criar uma nova matriz e armazena ela no array de matrizes.

O comando print necessita de um id de uma matriz que na prática é o index em que se encontra a matriz desejada no array de matrizes.

```

SparseMatrix::~SparseMatrix(){
    while(this->m_head->right_pointer != nullptr){
        Node *aux = this->m_head->right_pointer;
        this->m_head->right_pointer = aux->right_pointer;
        delete aux;
    }

    while(this->m_head->below_pointer != nullptr){
        Node *aux = this->m_head->below_pointer;
        this->m_head->below_pointer = aux->below_pointer;
        delete aux;
    }
    delete this->m_head;
}

```

Figura 15: implementação do destrutor

```

int main(){
    vector<SparseMatrix*> sparsematrixList;

    while(true){
        string input, action;
        getline(cin, input);

        stringstream ss(input);
        ss >> action;

        cout << "👉:" << ss.str() << endl;
        // create
        if(action == "create"){
            int row, column;
            ss >> row;
            ss >> column;
            SparseMatrix *sp = new SparseMatrix(row, column);
            sparsematrixList.push_back(sp);

            // print matrix X
        }else if(action == "print"){
            int id;
            ss >> id;
            sparsematrixList[id]->print();
        }
    }
}

```

Figura 16: comandos create e print

```
// print position i j from matrix X
}else if(action == "get"){
    int id, row, column;
    ss >> id >> row >> column;
    cout << sparsematrixList[id]->get(row, column) << endl;

// remove position i j from matrix X
}else if(action == "remove"){
    int id, row, column;
    ss >> id >> row >> column;
    sparsematrixList[id]->remove(row, column);

// insert a value on matrix X in position i j
}else if(action == "insert"){
    int id, row, column;
    double value;
    ss >> id >> row >> column >> value;
    sparsematrixList[id]->insert(row, column, value);

// prints all matrix
}else if(action == "showAll"){
    for(unsigned i = 0; i<sparsematrixList.size(); i++){
        cout << "Matrix " << i << endl;
        sparsematrixList[i]->print();
        cout << endl;
    }
}
```

Figura 17: comandos get, insert, remove e showAll

```

// end
}else if(action == "end"){
    for(unsigned i = 0; i<sparsematrixList.size(); i++){
        delete sparsematrixList[i];
    }
    sparsematrixList.clear();
    break;

// soma
}else if(action == "sum"){
    int id1, id2;
    ss >> id1 >> id2;
    SparseMatrix *sp = sum(sparsematrixList[id1], sparsematrixList[id2]);
    sparsematrixList.push_back(sp);
    sp->print();

// multiply
}else if(action == "multiply"){
    int id1, id2;
    ss >> id1 >> id2;
    SparseMatrix *sp = multiply(sparsematrixList[id1], sparsematrixList[id2]);
    sparsematrixList.push_back(sp);
    sp->print();

```

Figura 18: comandos end, sum e multiply

```

// read a matrix from a file
}else if(action == "read"){
    string filename;
    ss >> filename;
    SparseMatrix *sp = readSparseMatrix(filename);
    sparsematrixList.push_back(sp);
    sp->print();
}
else{
    cout << "comand '" << action << "' nonexistent!" << endl;
}
}

return 0;

```

Figura 19: comando read e a exceção

## 6.2 Soma

O objetivo da função é pegar as duas matrizes que são passadas por parâmetro, somar elas e armazenar numa nova matriz que será inserida em um array na main. Para começar a função verifica se as duas matrizes possuem o mesmo tamanho, caso sejam diferentes é lançada uma exceção. Prosseguindo é criada uma matriz de mesmo tamanho que as duas matrizes que serão somadas, para cada posição  $i$   $j$  é verificado através da função auxiliar `validate position` se existe um valor a ser somado nas duas matrizes na posição referenciada, caso haja o valor na posição  $i$   $j$  na primeira matriz é somado com o valor da segunda matriz na mesma posição e armazenado na nova matriz. Caso apenas



uma das duas matrizes não possui um nó na posição  $i, j$  a nova matriz recebe na mesma posição o valor da matriz que possui um nó em  $i, j$ .

```
SparseMatrix *sum(SparseMatrix *A, SparseMatrix *B){
    // checks if both matrix have the same size for rows and columns
    if (A->getRows() != B->getRows() || A->getColumns() != B->getColumns())
        throw std::runtime_error("Matrizes de tamanho diferente");

    SparseMatrix *C = new SparseMatrix(A->getRows(), A->getColumns());

    for (int i = 1; i <= A->getRows(); i++){
        for (int j = 1; j <= A->getColumns(); j++){
            if (A->validatePosition(i, j) && B->validatePosition(i, j)){
                C->insert(i, j, A->get(i, j) + B->get(i, j));
            }
            else if (A->validatePosition(i, j) && B->validatePosition(i, j) == false){
                C->insert(i, j, A->get(i, j));
            }
            else if (A->validatePosition(i, j) == false && B->validatePosition(i, j)){
                C->insert(i, j, B->get(i, j));
            }
        }
    }

    return C;
}
```

Figura 20: função somar matrizes

### Análise de complexidade:

De cara é possível notar que a função utiliza laços aninhados e sabendo que sempre é necessário percorrer toda a matriz para efetuar a soma de todas as posições a função poderia ser definida com complexidade  $O(n^2)$ , porém existe a utilização de uma função auxiliar em todos os casos verificados pela função soma, a função validate position por si só possui uma complexidade de  $O(n)$ , é possível enxergar essa complexidade na sessão que explica a validate position. Sabendo disso a função de soma sempre irá percorrer as duas matrizes sempre irá executar o validate position em ambas as matrizes, resultando em uma complexidade de  $O(n^2 + n)$  ou  $O(n^2) + O(n)$ , algo nesse sentido.

## 6.3 Multiplicação

A função multiply recebe duas matrizes por parâmetro e começa verificando se o número de colunas da primeira matriz é igual ao número de linhas da segunda, já que essa é uma condição para a realização de uma multiplicação entre matrizes. Uma exceção é lançada caso os valores sejam diferentes.

```

SparseMatrix *multiply(SparseMatrix *A, SparseMatrix *B){
    // checks if column and row size from A and B are different
    if (A->getColumns() != B->getRows())
        throw runtime_error("Tamanhos Inválidos");

    SparseMatrix *C = new SparseMatrix(A->getRows(), B->getColumns());

    for (int i = 1; i <= A->getRows(); i++){
        for (int j = 1; j <= B->getColumns(); j++){
            C->insert(i, j, 0);
            for (int k = 1; k <= A->getColumns(); k++){
                if (A->validatePosition(i, k) && B->validatePosition(k, j)){
                    C->insert(i, j, (C->get(i, j) + (A->get(i, k) * B->get(k, j))));
                }
            }
        }
    }
    return C;
}

```

Figura 21: função multiplicar matrizes

Após a verificação é criada uma nova matriz tendo o número de linhas da primeira e o número de colunas da segunda depois os passos padrões para a efetuação de uma multiplicação de matrizes são feitos, basicamente cada linha da primeira matriz é multiplicada pelas colunas da segunda matriz.

## 6.4 Ler arquivo

A função recebe o nome de um arquivo.txt, a partir desse arquivo é feita uma leitura específica nele. Para os dois primeiros valores lidos do arquivo é criada uma nova matriz que tenha o primeiro valor como o número de linhas e o segundo valor do arquivo como número de colunas.

```
SparseMatrix *readSparseMatrix(string filename){

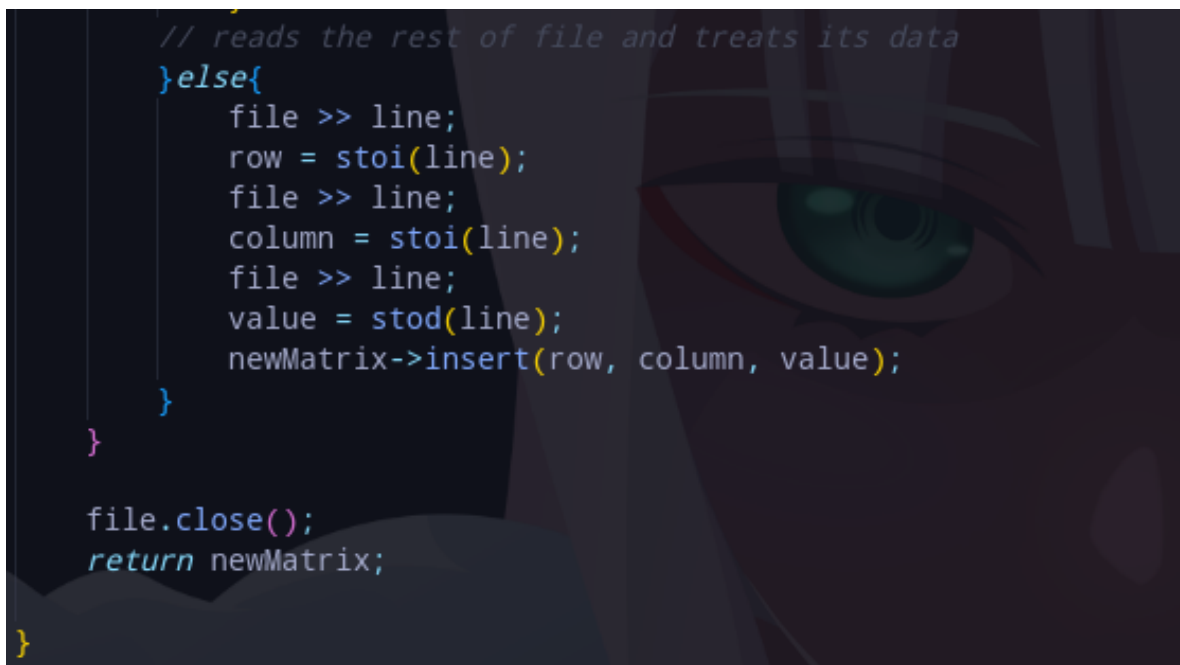
    std::ifstream file(filename); // open file

    string line;
    int row, column;
    double value;
    int counter = 0;
    SparseMatrix *newMatrix;

    while(file.good()){
        // reads the first two values from file
        if(counter < 2){
            if(counter == 0){
                file >> row;
                counter++;
            }else if(counter == 1){
                file >> column;
                newMatrix = new SparseMatrix(row, column);
                counter++;
            }
        }
    }
}
```

Figura 22:

O restante do arquivo é lido e a cada três valores é inserido na nova matriz um nó com os últimos dados lidos sendo eles a linha, coluna e valor do nó respectivamente. Após todo o arquivo ser lido a função fecha o arquivo e retorna a nova matriz, que será armazenada no array de matrizes.



```
// reads the rest of file and treats its data
}else{
    file >> line;
    row = stoi(line);
    file >> line;
    column = stoi(line);
    file >> line;
    value = stod(line);
    newMatrix->insert(row, column, value);
}

file.close();
return newMatrix;
}
```

Figura 23: função de leitura de arquivo

## 7 Divisão das Tarefas

O trabalho foi dividido em blocos, cada um ficou responsável por criar funções que iriam fazer o projeto funcionar, corretamente depois juntamos tudo para fazer pequenas alterações necessárias, a elaboração do relatório foi feita em conjunto.

## 8 Referências

Monitor da disciplina.