Estrutura de Dados – Projeto 01
Implementação de uma matriz esparsa com base em listas circularmente encadeadas

Antonio Deivid Santos Costa[1], **Luiz Guilherme Moreira Leite**[2]

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

[2]Curso de Graduação em Ciência da Computação – 2º Semestre
Cadeira de Estrutura de Dados

antoniodeivid@alu.ufc.br, guilhermemoreira@alu.ufc.br

Abstract. *This report aims to describe how the process of developing a sparse matrix was done, implemented through circular linked lists. This document has tools, methods and technologies used for the best optimization of the work, as well as the main difficulties and what was done to overcome them.*

Resumo. *Este relatório tem como obejtivo descrever como se deu o processo de desenvolvimento de matriz esparsa, implementada através de listas encadeadas circulares. Este documento possui ferramentas, métodos e tecnologias utilizadas para a melhor otimização do trabalho, assim como as principais dificuldades e o que foi feito para superar as mesmas.*

1. Introdução

Uma matriz é dita esparsa quando, em maioria, suas posições são preenchidas por uma valor padrão (zeros, por exemplo). Matrizes esparsas podem ser usadas de diversas maneiras dentro da ciência da computação, tendo ainda diferentes protocolos e várias técnicas de análise e armazenamento de dados dependendo do seu uso. Também, possui aplicações em campos como a da física e engenharia.

Tratando de matrizes deste tipo, acaba sendo um grande disperdício armazenar todos os elementos, ou seja, armazenar a matriz completamente. Há a possibilidade de uma grande economia de memória ao armazenar apenas aqueles elementos com valor diferente de zero. Para tal, assim como em outros tipos de armazenamento ultraeficientes, são utilizadas de técnicas baseadas no uso de ponteiros e referências para dados compactados (listas encadeadas), então, tornando possível a implementação de uma matriz esparsa com conexões mais diretas entre dados mais densos (elementos diferentes de zero).

2. Metodologia

A matriz, por completa, foi implementada por meio de alocação encadeada. Cada elemento da matriz é um nó criado a partir de um struct Node. Cada nó possui os atributos para localização – linha, coluna e valor – e também dois ponteiros Node – right e down – para conectar a próxima linha e coluna.

O primeiro passo do desenvolvimento da matriz foi criar uma espécie de ”esqueleto” para ela, com a criação de nós sentinela para cada linha e cada coluna, partindo

do nó head, a cabeça e nó principal da estrutura. Posto isso, todos os métodos foram implementados a partir da localização por meio dos sentinelas. Cada nó é encadeado ao próximo nó para direita e para baixo. Cada último nó de uma linha tem seu ponteiro right apontado para o nó sentinela daquela linha, assim como o ponteiro down do último nó de uma coluna aponta para o sentinela daquela coluna. Desta forma, tornando a matriz circularmente encadeada.

Tendo em vista que esse foi o raciocínio utilizado para construir a estrutura, segue abaixo os métodos criados para auxiliar o bom funcionamento da matriz:

- **SparseMatrix(int, int):** construtor da matriz, recebe quantidade de linhas e colunas;
- **SparseMatrix():** destrutor da matriz;
- **void insert(double, int, int):** método que insere um novo elemento na matriz. Recebe o valor a ser inserido, a linha e a coluna. Caso já haja um elemento nesta posição, seu valor é atualizado;
- **double get(int, int):** recebe linha e coluna e retorna o elemento localizado por estes;
- **int getcolumns():** retorna a quantidade de colunas da matriz;
- **int getlines():** retorna a quantidade de linhas da coluna;
- **bool isEmpty():** retorna true caso a matriz esteja vazia e false, caso contrário;

Para além dos métodos acima, foram criadas também algumas funções extras: **adição**, **multiplicação** e **readSparseMatrix**, para somar matrizes, multiplicar e ler matrizes de um arquivo, respectivamente.

No que tange à divisão de trabalho entre a dupla, a maior parte do projeto foi feito por ambos, em todos os aspectos. O código foi escrito pelos dois, em ligações e por espelhamento de tela, enquanto havia discussão e estudo sobre qual seria o melhor caminho para seguir durante o desenvolvimento. Exceto as funções extras, na main, que foram divididas, onde get e sum ficaram com o aluno Guilherme e readSparseMatrix com o aluno Deivid.

Para o desenvolvimento da estrutura de dados foi utilizado o editor de código-fonte Visual Studio Code. Suas funcionalidades e o ambiente foram muito úteis para todo o processo de criação. O github foi utilizado para versionamento de código entre a dupla.

3. Resultados e Discussão

A estrutura de dados passou em todos os testes propostos pela dupla, para todas as funções, por meio do driver de teste criado. O driver em questão foi criado para ser interativo para o usuário, seguindo os seguintes comandos:

- **create:** adiciona uma matriz ao vector
- **exit:** sai do programa
- **show:** mostra as matrizes adicionadas ao vector
- **sum**: soma duas matrizes
- **multiplies:** multiplica duas matrizes
- **sum aqv:** soma uma matriz com a matriz lida do arquivo
- **multiplies aqv:** multiplica uma matriz com a matriz lida do arquivo
- **add:** adiciona um elemento a uma matriz escolhida

- **add aqv:** adiciona um elemento a uma matriz lida do arquivo
- **get:** mostra o elemento de uma matriz escolhida
- **get aqv:** mostra o elemento de uma matriz lida do arquivo
- **remove:** remove um elemento de uma matriz escolhida
- **remove aqv:** remove um elemento de uma matriz lida do arquivo
  Obs.: para quaisquer dúvidas acerca dos comandos do driver, eles estão listados também no código fonte e nos testes mostrados posteriormente.

Durante todo o processo de desenvolvimento os programadores dedicaram bastante atenção à qualquer possível vazão de memória, sempre a liberando após alocar dinamicamente. Assim sendo, a matriz mostrou bom desempenho e cumpriu perfeitamente aquilo que consta em sua proposta.

As imagens de alguns testes realizados, tal como a análise de complexidade de pior caso das funções de adição, get e insert estarão presentes na seção Imagens, ao final do relatório.

4. Considerações Finais

É certo dizer que o projeto Implementação de uma matriz esparsa com base em listas circularmente encadeadas foi de suma importância para a formação acadêmica e, sim, profissional, dos alunos envolvidos no desenvolvimento dessa estrutura de dados. Não só o aperfeiçoamento de skills já conhecidas como o surgimento de novas durante todo o processo, desde o planejamento inicial da estrutura até o presente relatório.

As maiores dificuldades da dupla se encontraram na implementação do método insert. No entanto, com a orientação do professor Atílio e após diversas discussões e testes, as dificuldades foram superadas. A capacidade de resolver problemas e analisar as coisas de um outro ponto de vista dos envolvidos foi posta a prova durante todo o projeto, visto que várias opções eram cogitadas a todo momento, sempre procurando o melhor caminho e o mais otimizado. O código inteiro foi debugado, procurando possíveis bugs e consertando estes.

Por conseguinte, a experiência, para ambos os programadores, foi bastante proveitosa. Conseguiram obter um ótimo trabalho em equipe e ampliaram bastante sua gama de conhecimentos acerca de programação e estruturas de dados.

5. Imagens

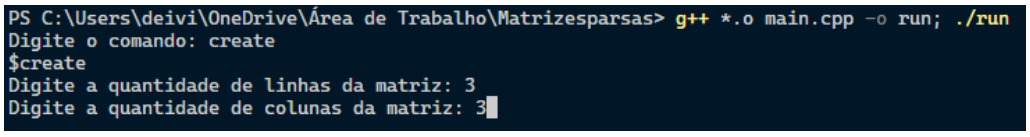


Figure 1. create

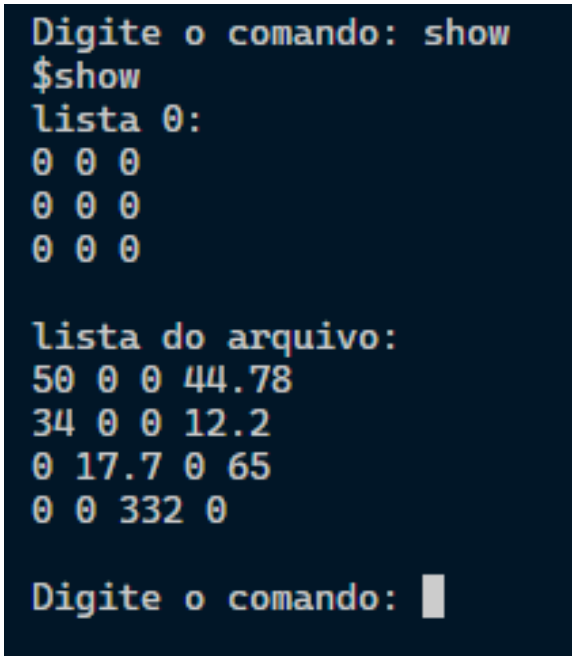


Figure 2. show

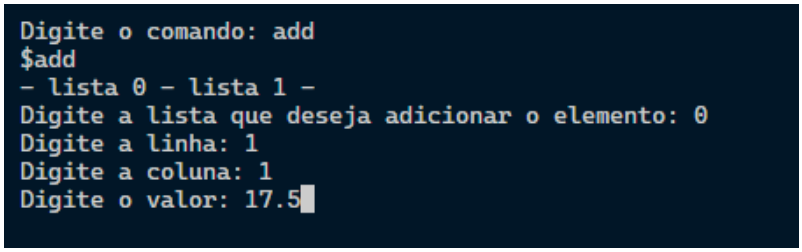


Figure 3. add

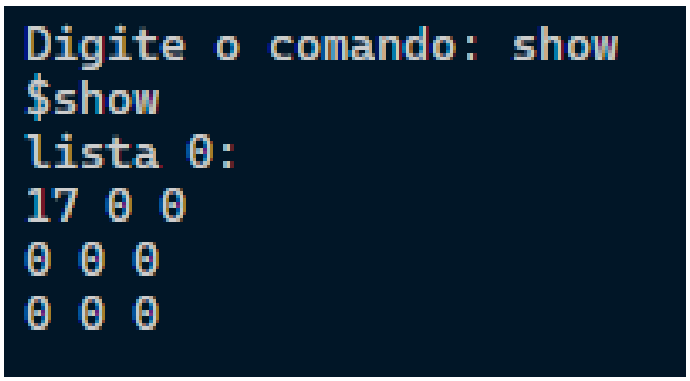


Figure 4. show

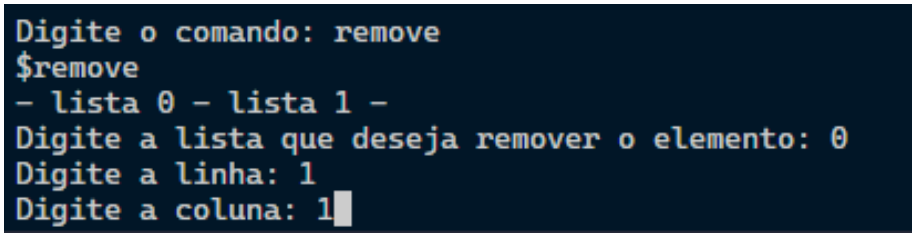


Figure 5. remove

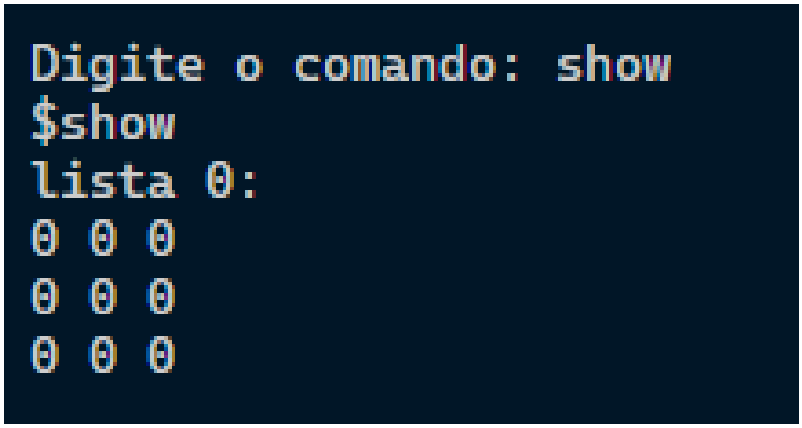


Figure 6. show

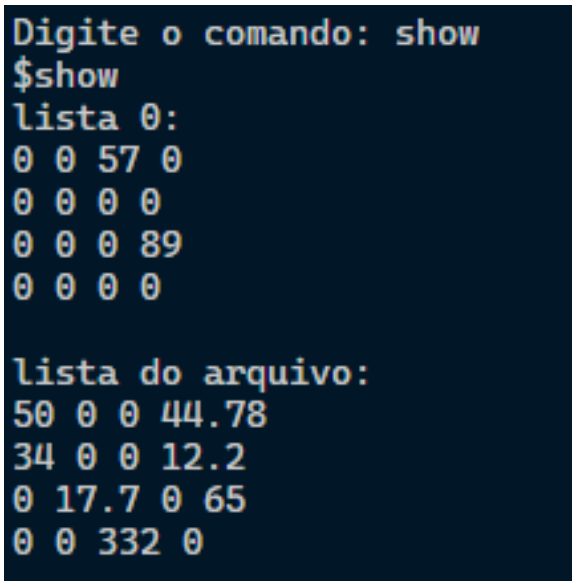


Figure 7. show

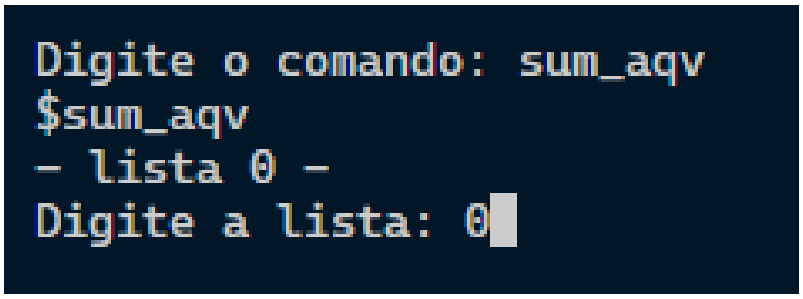


Figure 8. sum aqv

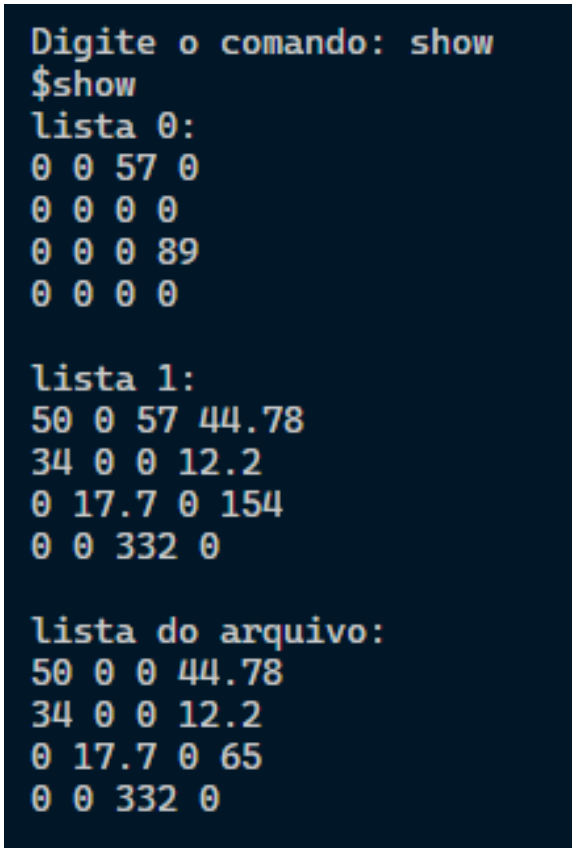
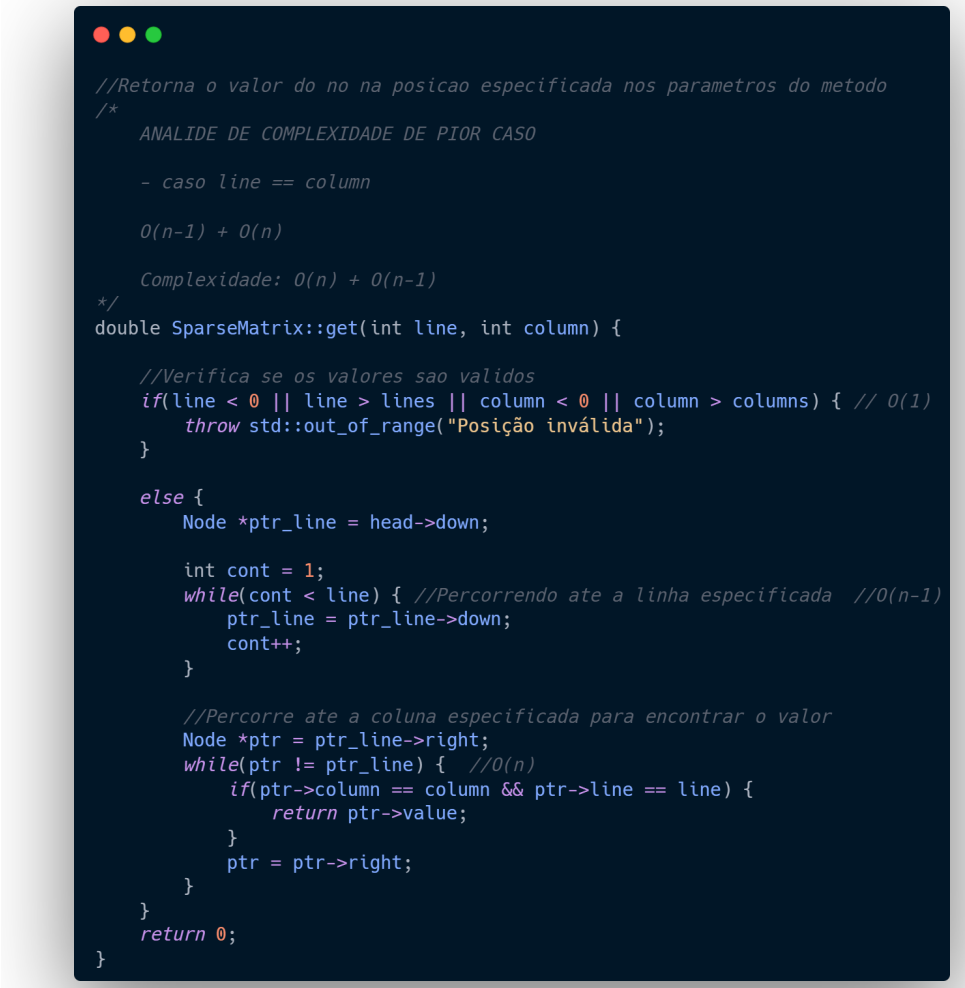


Figure 9. show

```
// soma duas matrizes de tamanhos iguais
/*
    ANALIDE DE COMPLEXIDADE DE PIOR CASO

    O(n) * O(n)

    Complexidade: O(n^(2))
*/
SparseMatrix *adicao(SparseMatrix *a, SparseMatrix *b){
    //caso elas tenham tamanhos diferentes, é lançado uma exeção
    if(a->getlines() != b->getlines() || a->getcolumns() != b->getcolumns()){
        throw std::out_of_range("Matrizes de tamanhos diferentes");
    }
    //cria uma nova matriz resultante
    SparseMatrix *c = new SparseMatrix(a->getlines(),a->getcolumns());

    cout << "Matriz A: " << a->isEmpty() << endl << "Matriz B: " << b->isEmpty() << endl;

    // caso esteja vazia, retorna uma matriz vazia
    if(a->isEmpty() == true && b->isEmpty() == true){
        return c; // O(1)
    }

    // caso uma esteja vazia e a outra não, retorna a matriz não vazia
    else if(a->isEmpty() == false && b->isEmpty() == true){
        return a; // O(1
    }
    else if(a->isEmpty() == true && b->isEmpty() == true){
        return b; // O(1
    }

    // soma as duas matrizes passadas como parametro e insere os resultados na matriz resultante
    for(int i = 1; i <= a->getlines(); i++){ // O(n)
        for(int j = 1; j <= a->getcolumns(); j++){  // O(n)
            double aux = a->get(i,j) + b->get(i,j);
            if(aux != 0){
                c->insert(aux,i,j);
            }
        }
    }
    return c;
}
```

Figure 10. Análise de complexidade da função de adição

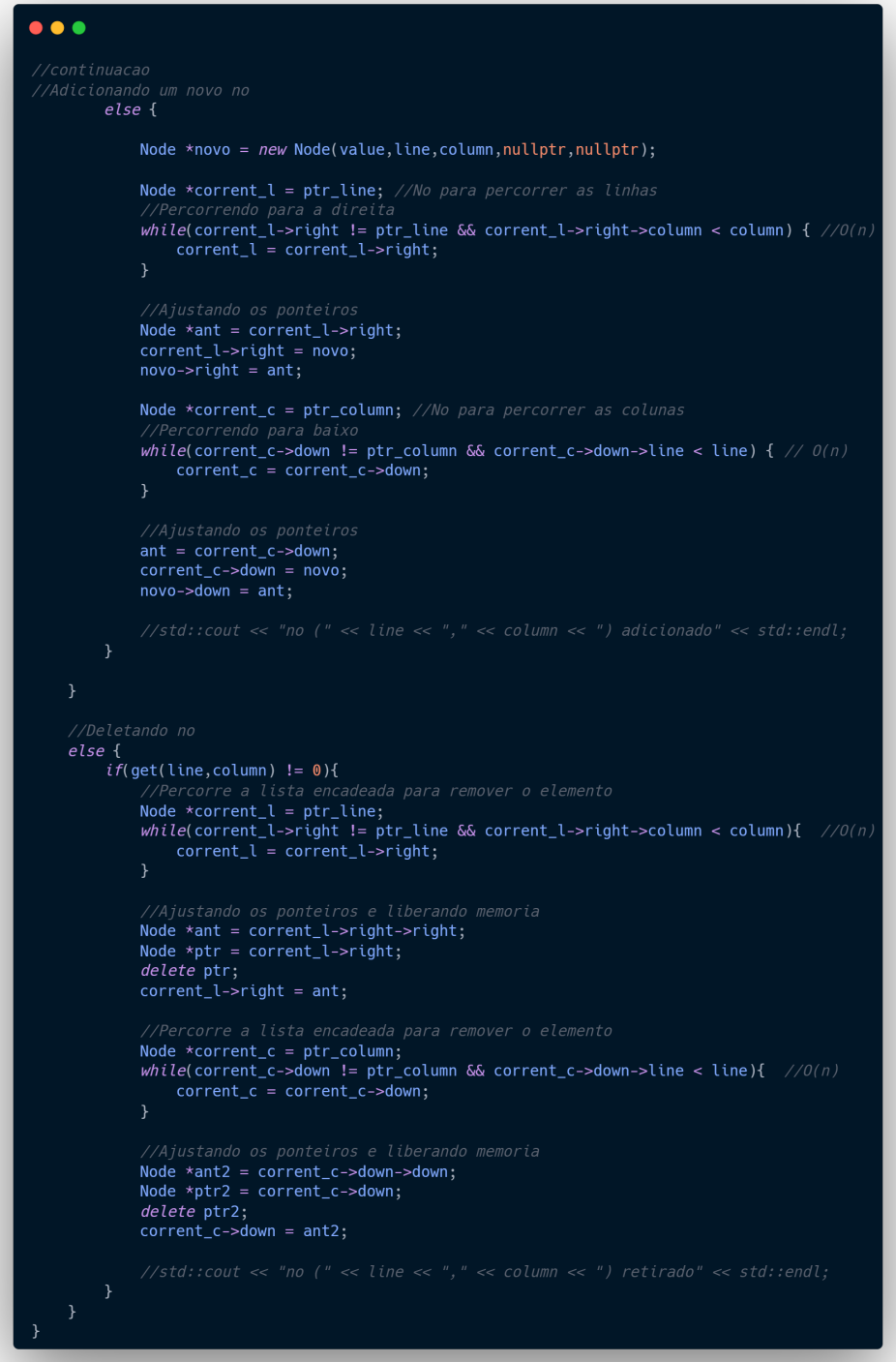


```cpp
//Retorna o valor do no na posicao especificada nos parametros do metodo
/*
    ANALIDE DE COMPLEXIDADE DE PIOR CASO

    - caso line == column

    O(n-1) + O(n)

    Complexidade: O(n) + O(n-1)
*/
double SparseMatrix::get(int line, int column) {

    //Verifica se os valores sao validos
    if(line < 0 || line > lines || column < 0 || column > columns) { // O(1)
        throw std::out_of_range("Posição inválida");
    }

    else {
        Node *ptr_line = head->down;

        int cont = 1;
        while(cont < line) { //Percorrendo ate a linha especificada  //O(n-1)
            ptr_line = ptr_line->down;
            cont++;
        }

        //Percorre ate a coluna especificada para encontrar o valor
        Node *ptr = ptr_line->right;
        while(ptr != ptr_line) {  //O(n)
            if(ptr->column == column && ptr->line == line) {
                return ptr->value;
            }
            ptr = ptr->right;
        }
    }
    return 0;
}
```

Figure 11. Análise de complexidade da função get

```
//Inserir elemento
/*
    ANALIDE DE COMPLEXIDADE DE PIOR CASO

    - caso line == column

    2*O(n) + 2*O(n)
    4*O(n)

    Complexidade: O(n)
*/
void SparseMatrix::insert(double value, int line, int column) {
/*Insere um valor na linha e coluna especificados nos parametros. Se esse valor for 0 e já existir
um no nessa posicao, este sera excluido*/

    //Verifica se valores sao válidos
    if(line < 0 || line > lines || column < 0 || column > columns) { // O(1)
        throw std::out_of_range("Posição inválida");
    }

    Node *ptr_line = head;
    int cont = 0;
    while(cont < line){ //Percorrendo ate a linha especificada  //O(n)
        ptr_line = ptr_line->down;
        cont++;
    }

    Node *ptr_column = head;
    cont = 0;
    while(cont < column){ //Percorrendo ate a coluna especificada  //O(m)
        ptr_column = ptr_column->right;
        cont++;
    }

    if(value != 0) {
        //Caso haja algum no na posicao, apenas o valor eh alterado
        if(get(line,column) != 0){
            Node *corrent_l = ptr_line; //No para percorrer as linhas
            //Percorrendo para a direita
            while(corrent_l->right != ptr_line && corrent_l->right->column < column){  //O(n)
                corrent_l = corrent_l->right;
            }

            corrent_l->right->value = value;
        }
```

Figure 12. Análise de complexidade da função insert



```
//continuacao
//Adicionando um novo no
        else {

            Node *novo = new Node(value,line,column,nullptr,nullptr);

            Node *corrent_l = ptr_line; //No para percorrer as linhas
            //Percorrendo para a direita
            while(corrent_l->right != ptr_line && corrent_l->right->column < column) { //O(n)
                corrent_l = corrent_l->right;
            }

            //Ajustando os ponteiros
            Node *ant = corrent_l->right;
            corrent_l->right = novo;
            novo->right = ant;

            Node *corrent_c = ptr_column; //No para percorrer as colunas
            //Percorrendo para baixo
            while(corrent_c->down != ptr_column && corrent_c->down->line < line) { // O(n)
                corrent_c = corrent_c->down;
            }

            //Ajustando os ponteiros
            ant = corrent_c->down;
            corrent_c->down = novo;
            novo->down = ant;

            //std::cout << "no (" << line << "," << column << ") adicionado" << std::endl;
        }

    }

    //Deletando no
    else {
        if(get(line,column) != 0){
            //Percorre a lista encadeada para remover o elemento
            Node *corrent_l = ptr_line;
            while(corrent_l->right != ptr_line && corrent_l->right->column < column){  //O(n)
                corrent_l = corrent_l->right;
            }

            //Ajustando os ponteiros e liberando memoria
            Node *ant = corrent_l->right->right;
            Node *ptr = corrent_l->right;
            delete ptr;
            corrent_l->right = ant;

            //Percorre a lista encadeada para remover o elemento
            Node *corrent_c = ptr_column;
            while(corrent_c->down != ptr_column && corrent_c->down->line < line){  //O(n)
                corrent_c = corrent_c->down;
            }

            //Ajustando os ponteiros e liberando memoria
            Node *ant2 = corrent_c->down->down;
            Node *ptr2 = corrent_c->down;
            delete ptr2;
            corrent_c->down = ant2;

            //std::cout << "no (" << line << "," << column << ") retirado" << std::endl;
        }
    }
}
```

Figure 13. Análise de complexidade da função insert