UFC

Alunos: Antônio Erick Freitas Ferreira & João Pedro Soares Matias.

Professor: Dr Me Atílio Gomes Luiz.

Estrutura de dados.

26 de outubro de 2022.
Quixadá-Ce.

Matriz Esparsa.

VISÃO GERAL

Uma matriz esparsa é aquela onde a maioria dos elementos possui o valor zero. Existem diversas aplicações que fazem uso de matrizes esparsas na engenharia e na matemática, como o método das malhas para resoluções de circuitos elétricos ou sistemas de equações lineares. Essas matrizes também são comuns na computação, como as matrizes de adjacência de grafos, e mapas de bits (bitmap) em imagens digitais.

OBJETIVO

1. Implementar um código na linguagem de programação C + + para a construção de uma matriz esparsa alocando apenas valores diferentes de zero utilizando os conhecimentos de listas circulares.
2. Seguir todas as instruções passadas no documento.

Do que é “feito” de fato essa matriz?

Bom, essa matriz é feita de vários nós (que são structs), onde cada nó tem as características de linha, coluna e valor. Cada nó também contém a habilidade de apontar para mais 2 nós, o nó abaixo, formando uma lista na vertical, e o nó à direita, formando uma lista na horizontal.

```
 9  struct Node{
10      Node *direita{nullptr};
11      Node *abaixo{nullptr};
12      int linha=0;
13      int coluna=0;
14      double valor=0;
```

Essa struct também precisa de um construtor, para que sempre que eu necessite criar um novo nó, alguns valores sejam ajustados.

```
16      Node(double valor, Node *proxDireita, Node *proxAbaixo, int l, int c){
17          this->valor = valor;
18          direita = proxDireita;
19          abaixo = proxAbaixo;
20          linha = l;
21          coluna = c;
22      }
```

Ok, mas apenas isso não será suficiente para a criação da matriz esparsa, por isso, se torna necessário a utilização de uma outra classe para gerenciar esses nós.

A classe SparseMatrix.

Essa classe é responsável por organizar esses nós de forma a criar uma matriz esparsa. Cada SparseMatrix tem suas próprias características: O primeiro sentinela (ou sentinela principal), a quantidade de linhas e a quantidade de colunas.

```
10  class SparseMatrix{
11      private:
12      Node *head {nullptr};
13      int quantLinhas;
14      int quantColunas;
```

Assim como a struct Node, esta classe também precisará de um construtor, para que seja formado um “esqueleto”.

O construtor.

Logo de cara, eu preciso ter conhecimento se os valores para a criação da matriz são válidos, ora, não é possível criar uma matriz com linhas <= 0 ou colunas <=0, não é mesmo?

Agora sim, vamos a criação do esqueleto. A primeira coisa a ser feito é salvar o tamanho dessa matriz nas variáveis já citadas. Após isso, a partir do sentinela principal, usando um laço de repetição para linha e outro para coluna, é criado todos os sentinelas de linhas e colunas, todos esses sentinelas "secundários" também são nós, e por padrão, seus valores sempre serão 0, criando assim uma fila na horizontal e uma na vertical. Os sentinelas secundários das linhas, terão sempre coluna = 0 e linha correspondente, o mesmo pensamento para os das colunas, tendo sempre linha = 0 e coluna correspondente.

```
11  SparseMatrix::SparseMatrix(int m, int n){ //m linhas e n colunas.
12      if(m > 0 && n > 0){
13
14          quantLinhas = m;
15          quantColunas = n;
16
17          head = new Node(0,nullptr,nullptr,0,0);
18          head->direita = head;
19          head->abaixo = head;
20          for(int i = 0; i < m; i++){//Criando sentinelas de linhas.
21              Node *andador = head;
22              while(andador->abaixo != head){
23                  andador = andador->abaixo;
24              }
25
26              Node *novoSentinela = new Node(0,nullptr,nullptr,i+1,0);
27
28              novoSentinela->direita = novoSentinela;
29              novoSentinela->abaixo = head;
30              andador->abaixo = novoSentinela;
31          }
32
33
34          for(int i = 0; i < n; i++){//Criando sentinelas das colunas.
35              Node *andador = head;
36              while(andador->direita != head){
37                  andador = andador->direita;
38              }
39
40              Node *novoSentinela = new Node(0,nullptr,nullptr,0,i+1);
41              novoSentinela->direita = head;
42              novoSentinela->abaixo = novoSentinela;
43              andador->direita = novoSentinela;
44          }
45      }else{
46          throw range_error("Error! Valores invalidos para uma matriz!");
47      }
48  }
```

(Imagem reduzida para que mantenha a mesma página).

A função insert.

Esta função possui responsabilidade quádrupla e é de fato a mais complexa de ser implementada, além de ser a função primordial de todo o código.

Primeiro me preocupo em saber se estou trabalhando com algo dentro dos limites da matriz. Por exemplo, se eu tenho uma matriz 5x5, não será possível realizar um insert na linha 6 ou coluna 6 e linha e coluna <= 0.

```
63      //Primeiro, verifico se a linha e a coluna buscada existe de fato nessa matriz.
64      if(i > quantLinhas || j > quantColunas || i < 1 || j < 1){
65          throw range_error("ERROR!");
66      }
```

Estando seguro que estou manipulando algum lugar possível, a segunda preocupação se torna saber se eu já tenho um nó com linha = i e coluna = j, ou seja, se já existe esse nó. Existindo esse nó, existem dois caminhos. Caso value = 0, esse nó precisa ser desalocado, caso value != 0, esse nó precisa ter seu valor atualizado.

```
68      if(get(i,j) != 0 && value != 0){
69          Node *caminha = head;
70          while(caminha->linha != i){
71              caminha = caminha->abaixo;
72          }
73          while(caminha->coluna != j){
74              caminha = caminha->direita;
75          }
76          if(caminha->linha == i && caminha->coluna == j)
77              caminha->valor = value;
78          }
```

A função get (será explicada mais a frente) me diz que já existe esse nó e como value != 0, o que eu faço é um ponteiro de nó caminhar pela estrutura, primeiro pela linha, depois pela coluna, em seguida atualizo o valor desse nó.

A terceira preocupação é caso esse nó exista e value = 0, pois aqui, esse nó deve ser desalocado e a fila não pode ser desfeita, se fazendo necessário a organização de linhas e colunas.

```
}else if(get(i,j) != 0 && value == 0){
    //Movimentação pela linha.
    Node*fixLin = head;
    Node*searchLin = nullptr;
    Node*prevLin = nullptr;

    while(fixLin->linha != i){
        fixLin = fixLin->abaixo;
    }

    //Arrumo os outros 2 ponteiros para também ficarem na linha que eu quero.
    prevLin = fixLin; //A ideia do prev é fazer com que ele ande sempre 1 nó atrás do searchLin.
    searchLin = fixLin->direita;

    while(searchLin->coluna != j && searchLin->direita != fixLin){
        prevLin = searchLin;
        searchLin = searchLin->direita;
    }

    Node*fixCol = head;
    Node*searchCol = nullptr;
    Node*prevCol = nullptr;

    while(fixCol->coluna != j){
        fixCol = fixCol->direita;
    }
    prevCol = fixCol;
    searchCol = fixCol->abaixo;

    while(searchCol->linha != i && searchCol->abaixo != fixCol){
        searchCol = searchCol->abaixo;
    }
    if(searchLin->linha == i && searchCol->coluna == j){
        prevCol->abaixo = searchCol->abaixo;
        prevLin->direita = searchLin->direita;
        delete searchLin;
    }
```

O que é de fato feito é: 6 ponteiros para nós. Dois fixos na linha e coluna, dois para chegar no nó que deve ser desalocado e dois para estarem sempre um nó antes. Quando eu tiver a certeza que realmente estou no nó, corrijo a fila pela linha e pela coluna e deleto o nó em questão.

A quarta (e mais difícil) preocupação é quando esse nó ainda não existe. A lógica desenvolvida agora foi manter os 6 sentinelas e mesmo, sabendo que esse nó ainda não existe, fazer com que os procuradores e os que ficam antes deles andassem, para que assim, eu pudesse saber onde esse nó deve ser inserido.

Para os casos de inserção, conseguimos visualizar 4 possíveis casos: quando a linha e a coluna estão vazias, quando a linha já tem nó mas a coluna não, quando a linha não tem nó mas a coluna tem e quando a linha e a coluna já tem algum nó. Essa análise foi o que possibilitou que a estrutura das linhas e colunas seguissem como listas horizontais e verticais, respectivamente. Por essa parte do código ter mais de 150 linhas, não irei colocar as prints aqui. :(

A função get.

Esta função deve retornar o valor do nó que possua linha = i e coluna = j. Assim como todos os outros, a lógica inicial é sempre testar se esses valores de i e j são reais nessa matriz.

```
252     if((i > quantLinhas || j > quantColunas) || (i == 0 && j == 0)){
253         throw range_error("ERROR!");
```

Agora sabendo que esse nó procurado pode está dentro da matriz, a lógica usada é: criar dois ponteiros para nó começando da cabeça e indo até o sentinela secundário de linha = i, depois faço o segundo ponteiro se movimentar para a direita procurando esse nó, caso ele o encontre, esse valor é retornado, mas caso esse ponteiro caminhe por toda lista e não encontre esse nó, o valor retornado será 0.

```
254     }else{
255         Node*search = head;
256         while(search->linha != i && search->abaixo != head){
257             search = search->abaixo;
258         }
259         Node*fix = search;
260         while(search->coluna != j && search->direita != fix){
261             search = search->direita;
262         }
263 
264         if(search->linha == i && search->coluna == j){
265             return search->valor;
266         }else{
267             return 0;
268         }
269     }
270 }
```

Agora que já inserimos, alteramos valores e até mesmo deslocamos nós, a próxima preocupação é como desalocar tudo isso que foi criado.

O destrutor.

Como agora estamos falando em apenas destruir quem existe, não há a necessidade de se preocupar com os ponteiros. Portanto, a lógica adotada foi: criar um ponteiro para nó apontando para o sentinela principal, criar um laço com a condição de parada ser chegar ao fim da lista(se movendo para baixo), mas enquanto esse ponteiro caminha, é criado +2 ponteiros para nó. O primeiro aponta para a direita do ponteiro que caminha na linha, o segundo estará sempre um nó atrás desse. É feito outro laço com a condição de parada ser o primeiro chegar ao fim da linha pela direita, faço o segundo apontar pro primeiro, o primeiro andar para a direita e por fim, dou um delete no segundo. Dessa forma, estarei apagando linha a linha por completo até o último nó.

```
272  SparseMatrix::~SparseMatrix(){
273      Node*auxLin = head;
274      while(auxLin->abaixo != head){
275          auxLin = auxLin->abaixo;
276          Node*prevLin = nullptr;
277          Node*ProxLin = auxLin->direita;
278
279          while(ProxLin->direita != auxLin){
280              prevLin = ProxLin;
281              ProxLin = ProxLin->direita;
282              delete prevLin;
283          }
284      }
```

Agora o que me restou foi apenas o esqueleto que falei sobre no construtor, preciso deletá-lo também. Faço meu ponteiro que caminha pelas linhas ficar um elemento abaixo do sentinela principal, crio um laço de repetição até que ele chegue no último elemento, enquanto isso, um outro ponteiro para nó andará um nó antes e irá destruindo nó a nó. A mesma lógica se aplica para os nós coluna. E por fim, deleto o head.

```
285
286     auxLin = head->abaixo;
287     while(auxLin->abaixo != head){
288         Node*prev = auxLin;
289         auxLin = auxLin->abaixo;
290         delete prev;
291     }
292
293     Node*auxCol = head->direita;
294     while(auxCol->direita != head){
295         Node*prev = auxCol;
296         auxCol = auxCol->direita;
297         delete prev;
298     }
299     delete head;
300     cout << "Matriz Liberada!" << endl;
301 }
```

A função print.

Esta função tem uma lógica bem parecida com a do destrutor (quando estou destruindo linha a linha), a diferença é que agora estou imprimindo esses valores na tela.

```cpp
void SparseMatrix::print(){
    Node*auxLin = head->abaixo;
    Node*anda = nullptr;
    //O que eu nao consigo entender nesse for é o fato dele nao me permitir usar i & k = 0,
    // e depois usar no if i+1 ou k+1;
    for(int i = 1; i <= quantLinhas; i++){
        anda = auxLin->direita;
        for(int k = 1; k <= quantColunas; k++){
            if(anda->linha == i && anda->coluna == k){
                cout << anda->valor << " ";
                anda = anda->direita;
            }else{
                cout << "0 ";
            }

            if(auxLin->direita->linha == 0 || auxLin->direita->linha == i){
                auxLin = auxLin->abaixo;
            }
        }
        cout << endl;
    }
}
```

Funções auxiliares.

Como solicitado pelo professor, foi implementado algumas outras funções. Uma para ler um arquivo.txt, uma para somar e outra para multiplicar matrizes. Como acredito que esse não seja o grande foco desse trabalho, tendo em vista que soma e multiplicação já tínhamos implementado a lógica na atividade TAD Matriz e a função para a leitura de arquivos também é algo bem simples, não comentarei sobre elas.

Mas para que a soma e multiplicação funcionasse eu precisei implementar duas funções: getquantLinhas() e getquantColunas() para que eu conseguisse acessar de fora o tamanho das matrizes passadas, mas também nada de outro mundo.

Análise de complexidade.

Função Get

Analisando a Função Get, o seu pior caso, seria pegar o valor da última linha e última coluna e retorná-lo. Desse modo, no pior caso temos:

- Uma condição If para verificar valores inválidos, representada por uma constante C1;

A condição else será automaticamente aceita após a não atribuição da condição if; E a partir da condição else temos:

● Um while que faz uma atribuição à search, que roda “i” vezes, essa atribuição será representada por uma constante C2;

● Um while que faz uma atribuição à search, que roda “j” vezes, essa atribuição será representada por uma constante C3;

● Uma condição IF representada por uma constante C4;

O tempo gasto nesse algoritmo é representado pela soma dessas constantes de acordo com a quantidade de vezes que cada uma aparecerá. Desse modo temos que o tempo é:

C1 + C2(i) + C3(j) + C4;

De acordo com a instrução de que o máximo que a matriz insere são 30.000 (trinta mil) linhas e 30.000 (trinta mil) colunas, no pior caso, linhas e colunas teriam valores iguais e portanto podem ser representados pela mesma variável (no caso, n) portanto:

C1 + C2(n) + C3(n) + C4;

Posso colocar n em evidência: C1 + C4 + n(C2 + C3);

Chamarei (C1 + C4) de A e (C2 + C3) de B: A + n(B)

Dessa forma, assumo como verdade que A + n(B) é menor ou igual a A(n) + B(n):

A + n(B) <= A(n) + B(n);

A + n(B) <= n(A + B);

Chamarei (A + B) de C:

A + n(B) <= C(n)

Definindo que o grau de complexidade dessa função é uma grandeza **O(n)**.

Função Sum

Na função Sum, seu papel é somar duas matrizes seguindo as regras de soma já existentes, portanto, seu pior caso será, atribuído o valor máximo a linhas e colunas, somar duas matrizes 30.000 x 30.000;

Desse modo, no pior caso nós temos:

● Uma condição if para determinar se as matrizes são inválidas, representadas por uma constante C1;

● Uma atribuição representada por uma constante C2, que acontecerá “j” vezes, representando o n° de colunas, e “i” vezes, representando o n° de linhas, portanto a constante será atribuída i x j (i vezes j);

Como i e j possuem o mesmo valor (30.000) eles podem ser representados pela mesma variável, no caso, **n**;

O tempo gasto nesse algoritmo é representado pela soma dessas constantes de acordo com a quantidade de vezes que cada uma aparecerá. Assim, nosso somatório terá essa formação:

C1 + C2(n*n);

C1 + C2(n^2);

Chamarei C1 de A e C2 de B: A + B(n^2);

Dessa forma, assumo como verdade que A + B(n^2) é menor ou igual a An^2 + Bn^2:

A + B(n^2) <= An^2 + Bn^2;

A + B(n^2) <= n^2(A + B);

Chamarei (A+B) de C e o colocarei em evidência:

A + B(n^2) <= C(n^2);

Seu grau de complexidade será então uma grandeza **O(n^2)**;

Função Insert

A função Insert insere o elemento na linha e coluna desejada, portanto, seu pior caso seria, atribuído o valor máximo a linha e coluna,inserir um valor (!= de 0) na última linha e coluna, ambas representadas por 30.000, e quando não há um nó já definido. Desse modo, no pior caso nós temos:

● Uma condição If para valores inválidos, representado por uma constante C1;

● Uma condição else if para caso o valor atribuído seja igual a 0, representada por uma constante C2;

Como queremos o pior caso, entraremos automaticamente na condição else, onde temos:

● Uma atribuição a “fixLin” que acontecerá “i” vezes, e será representada por uma constante C3;

● Uma condição If, caso alguém exista na linhas, representada por uma constante C4;

Mas como queremos o pior caso, entraremos na condição else, quando o nó não exista, nessa condição temos:

● Uma atribuição a “fixCol” que acontecerá “j” vezes e será representada por uma constante C5;

● Duas atribuições a “prevLin” e “searchLin”, respectivamente, que acontecerá “i” vezes e será representada por C6 e C7 respectivamente;

● Duas atribuições a “prevCol” e “searchCol”, respectivamente, que acontecerá “j” vezes e será representada por C8 e C9 respectivamente;

● Uma condição If para saber se linha e coluna estão vazias, representada por uma constante C10;

● Uma condição else if para saber se já existe um nó na linha mas não na coluna, representado por uma constante C11;

● Semelhantemente uma condição else if para saber se já existe um nó na coluna mas não na linha, representada por uma constante C12;

● Uma condição para quando já existe um nó na linha e na coluna, representada por uma constante C13;

Adentrando nessa condição temos:

● Uma condição if para adicionar um nó, representado por uma constante C14;

● Nesse if temos 4 atribuições, representadas por C15, C16, C17 e C18, respectivamente;

O tempo gasto nesse algoritmo é representado pela soma dessas constantes de acordo com a quantidade de vezes que cada uma aparecerá. Assim, nosso somatório de tempo é definido como sendo:

C1 + C2 + C3(i) + C4 + C5(j) + (C6 + C7)i + (C8 + C9)j + C10 + C11 + C12 + C13 + C14 + C15 + C16 + C17 + C18;

Chamarei (C1 + C2 + C4 + C10 + C11 + C12 + C13 + C14 + C15 + C16 + C17 + C18) de A:

A + C3i + C5j + C6i + C7i + C8j + C9j

Como quero inserir um elemento na última linha e última colunas, ambas representadas por 30.000, i e j assumem o mesmo valor, e ambas podem ser representadas por uma mesma variável, no caso, **n**:

A + C3n + C5n + C6n + C7n + C8n + C9n

A + n(C3 + C5 + C6 + C7 + C8 + C9)

Chamarei (C3 + C5 + C6 + C7 + C8 + C9) de B:

A + n(B)

Dessa forma, assumo como verdade que A + n(B) é menor ou igual a A(n) + B(n):

A + n(B) <= A(n) + B(n);

A + n(B) <= n(A + B);

Chamarei (A + B) de C: A + n(B) <= C(n)

Definindo que o grau de complexidade dessa função é uma grandeza **O(n)**

Nossas considerações.

Antes de começarmos de fato a “codar”, sentamos e pensamos muito a respeito de como seria toda essa estrutura, desenhamos e imaginamos. Não houve separação de partes nesse trabalho. Um grande facilitador para a implementação do código foi o construtor da SparseMatrix, que estava conseguindo manter o esqueleto com o formato ideal para as necessidades futuras.

Após a implementação do construtor, já fomos direto para a insert, essa foi de longe a mais demorada e dolorida. Escrevemos e reescrevemos várias vezes a fim de tentar cobrir o maior número de casos possíveis e de deixar o código mais simples. Como demoramos muito na insert e se fez necessário mais estudo para implementação da mesma, quando acabamos ela, as outras funções pareciam bem mais simples e fáceis de implementar. E de fato, foi isso que aconteceu.

Por fim, a função de soma e multiplicação como já tínhamos pela TAD matriz, só precisamos replicar a lógica em cima deste código. O que houve de “lançamento” foi a leitura de arquivo, que também não foi nada de extraordinário, com uma busca de 5 minutos no YouTube e uma breve leitura no site do Cplusplus foi totalmente possível de implementar.

A main não iterativa foi um jeito de simplificar um pouco mais o nosso lado. Mas há vários casos de teste para a comprovação de que esse código funciona. No mais, caso mantenha-se insatisfeito com os casos que lá tem, sinta-se à vontade para alterar o arquivo Main.cpp.

Quando o código for compilado, sugerimos que retorne ao início do terminal e acompanhe o que se passa no terminal e no código ao mesmo tempo.

E como solicitado pelo professor, este código é executável também em sistemas **diferentes** de Windows.

Referências.

Para a implementação das listas a **ÚNICA** referência que usamos foi o material disponibilizado pelo próprio professor (ForwardList, CircularList) e muitos (mas muitos mesmo) testes.

Para a implementação da função com arquivo:

https://www.youtube.com/watch?v=2UNw5-jfqhc&t=505s&ab_channel=CFBCursos

https://cplusplus.com/reference/fstream/fstream/?kw=fstream