Grafos - Formas de Implementação

Prof. Luiz Gustavo Almeida Martins

Grafos: formas de implementação

Matriz de Adjacências

Grafos: matriz de adjacências

Estrutura de representação:

Quantidade de vértices e arestas (opcional)

Grafos: matriz de adjacências

Estrutura de representação:

Quantidade de vértices e arestas (opcional)

Informações dos **vértices** (se existir)

Ex: grau do vértice

Grafos: matriz de adjacências

Estrutura de representação:

Quantidade de vértices e arestas (opcional)

Informações dos **vértices** (se existir)

Ex: grau do vértice

Informação das **arestas** (**matriz de adjacências**)

Grafos: matriz de adjacências

Estrutura de representação:

Quantidade de vértices e arestas (opcional)

Informações dos **vértices** (se existir)

Ex: grau do vértice

Informação das **arestas** (**matriz de adjacências**)

```
struct grafo {
    int qtde_vertices, qtde_arestas;
    int *grau;      // Vetor com o grau dos vértices
    int **aresta;   // Matriz de adjacências
};
```

Grafos: matriz de adjacências

Estrutura de representação:

Quantidade de vértices

Informações dos **vértices** (se existir)

Ex: grau do vértice

Informação das **arestas** (matriz de adjacências)

```
struct grafo {
    int qtde_vertices, qtde_arestas;
    int *grau;      // Vetor com o grau dos vértices
    int **aresta;   // Matriz de adjacências
};

typedef struct grafo Grafo;
```

Grafos: matriz de adjacências

Estrutura de representação:

Quantidade de vértices

Informações dos **vértices** (se existir)

Ex: grau do vértice

Informação das **arestas** (matriz de adjacências)

grafo.c

```
struct grafo {
    int qtde_vertices, qtde_arestas;
    int *grau;      // Vetor com o grau dos vértices
    int **aresta;   // Matriz de adjacências
};
```

grafo.h

```
typedef struct grafo Grafo;
```

Especificação do TAD Dígrafo

Operação ***cria_grafo:***

Entrada: a quantidade de vértices

Pré-condição: quantidade de vértices ser válida

Processo: alocar a área para representar o grafo, se necessário, e colocá-lo na **condição de vazio**

Saída: o endereço do grafo se operação bem sucedida ou ***NULL*** se houve algum erro

Pós-condição: nenhuma

Grafos: matriz de adjacências

*Grafo * **cria_grafo** (int nro_vertices)*

FIM

Grafos: matriz de adjacências

```
Grafo * cria_grafo (int nro_vertices)
        SE nro_vertices inválido ENTÃO
            Retorna NULL
        FIM_SE

FIM
```

Grafos: matriz de adjacências

*Grafo * **cria_grafo** (int nro_vertices)*

SE nro_vertices inválido ENTÃO

Retorna NULL

FIM_SE

Aloca memória para o grafo;

FIM

Grafos: matriz de adjacências

```
Grafo * cria_grafo (int nro_vertices)
        SE nro_vertices inválido ENTÃO
            Retorna NULL
        FIM_SE
        Aloca memória para o grafo;
        SE alocação bem sucedida ENTÃO
            Preenche os campos qtde_vertices e qtde_arestas do grafo;




        FIM_SE

FIM
```

Grafos: matriz de adjacências

```
Grafo * cria_grafo (int nro_vertices)
        SE nro_vertices inválido ENTÃO
            Retorna NULL
        FIM_SE
        Aloca memória para o grafo;
        SE alocação bem sucedida ENTÃO
            Preenche os campos qtde_vertices e qtde_arestas do grafo;
            Aloca memória para o vetor com o grau de vértices e
            inicializa todos os seus elementos com ZERO;



        FIM_SE

FIM
```

Grafos: matriz de adjacências

```
Grafo * cria_grafo (int nro_vertices)
        SE nro_vertices inválido ENTÃO
            Retorna NULL
        FIM_SE
        Aloca memória para o grafo;
        SE alocação bem sucedida ENTÃO
            Preenche os campos qtde_vertices e qtde_arestas do grafo;
            Aloca memória para o vetor com o grau de vértices e
            inicializa todos os seus elementos com ZERO;
            Aloca memória para a matriz de adjacências e
            inicializa todos os seus elementos com ZERO;
        FIM_SE

FIM
```

Grafos: matriz de adjacências

```
Grafo * cria_grafo (int nro_vertices)
        SE nro_vertices inválido ENTÃO
            Retorna NULL
        FIM_SE
        Aloca memória para o grafo;
        SE alocação bem sucedida ENTÃO
            Preenche os campos qtde_vertices e qtde_arestas do grafo;
            Aloca memória para o vetor com o grau de vértices e
            inicializa todos os seus elementos com ZERO;
            Aloca memória para a matriz de adjacências e
            inicializa todos os seus elementos com ZERO;
        FIM_SE
        Retorna o endereço do grafo;
FIM
```

Grafos: matriz de adjacências

```
Grafo * cria_grafo(int nro_vertices) {
    if (nro_vertices <= 0) return NULL;          // Verifica se qtde adequada

    Grafo * G = (Grafo *) malloc(sizeof(Grafo));
    if (G == NULL) return NULL;

    G->qtde_vertices = nro_vertices;
    G->qtde_arestas = 0;

    G->grau = (int *) calloc(nro_vertices, sizeof(int));
    if (G->grau == NULL) {
        free (G);
        return NULL;
    }
    •••
```

Grafos: matriz de adjacências

```
Grafo * cria_grafo(int nro_vertices) {
    if (nro_vertices <= 0) return NULL;

    Grafo * G = (Grafo *) malloc(sizeof(Grafo));    // Aloca o Grafo
    if (G == NULL) return NULL;                     // Verifica sucesso

    G->qtde_vertices = nro_vertices;
    G->qtde_arestas = 0;

    G->grau = (int *) calloc(nro_vertices, sizeof(int));
    if (G->grau == NULL) {
        free (G);
        return NULL;
    }
    ...
```

Grafos: matriz de adjacências

```
Grafo * cria_grafo(int nro_vertices) {
    if (nro_vertices <= 0) return NULL;

    Grafo * G = (Grafo *) malloc(sizeof(Grafo));
    if (G == NULL) return NULL;

    G->qtde_vertices = nro_vertices;          // Preenche campos de qtde
    G->qtde_arestas = 0;

    G->grau = (int *) calloc(nro_vertices, sizeof(int));
    if (G->grau == NULL) {
        free (G);
        return NULL;
    }
    ...
```

Grafos: matriz de adjacências

```
Grafo * cria_grafo(int nro_vertices) {
    if (nro_vertices <= 0) return NULL;

    Grafo * G = (Grafo *) malloc(sizeof(Grafo));
    if (G == NULL) return NULL;

    G->qtde_vertices = nro_vertices;
    G->qtde_arestas = 0;

    G->grau = (int *) calloc(nro_vertices, sizeof(int));    // Aloca vetor com o grau
    if (G->grau == NULL) {                                  // Verifica sucesso
        free (G);
        return NULL;
    }
    ...
```

Grafos: matriz de adjacências

```
    •••
    G->aresta = (int **) malloc(nro_vertices * sizeof(int *));  // Aloca linha matriz adj.
    if (G->aresta == NULL) {                                    // Verifica sucesso
        free(G->grau);  free(G);  return NULL;
    }
    int i, k;
    for(i=0; i < nro_vertices; i++) {
       G->aresta[i] = (int*) calloc(nro_vertices, sizeof(int));
        if (G->aresta[i]  == NULL) {
              for (k=0; k < i; k++) free(G->aresta[k]);
              free(G->aresta);  free(G->grau);  free(G);  return NULL;
         }
    }
    return G;
}
```

Grafos: matriz de adjacências

```
    •••
    G->aresta = (int **) malloc(nro_vertices * sizeof(int *));
    if (G->aresta == NULL) {
        free(G->grau);  free(G);  return NULL;
    }
    int i, k;
    for(i=0; i < nro_vertices; i++) {                    // Aloca colunas da matriz adj.
       G->aresta[i] = (int*) calloc(nro_vertices, sizeof(int));
        if (G->aresta[i]  == NULL) {                     // Verifica sucesso
              for (k=0; k < i; k++) free(G->aresta[k]);
              free(G->aresta);  free(G->grau);  free(G);  return NULL;
         }
    }
    return G;
}
```

Grafos: matriz de adjacências

```
    •••
    G->aresta = (int **) malloc(nro_vertices * sizeof(int *));
    if (G->aresta == NULL) {
        free(G->grau);  free(G);  return NULL;
    }
    int i, k;
    for(i=0; i < nro_vertices; i++) {
       G->aresta[i] = (int*) calloc(nro_vertices, sizeof(int));
        if (G->aresta[i]  == NULL) {
              for (k=0; k < i; k++) free(G->aresta[k]);
              free(G->aresta);  free(G->grau);  free(G);  return NULL;
         }
    }
    return G;                                          // Retorna end. do grafo
}
```

Especificação do TAD Dígrafo

Operação ***insere_aresta:***

Entrada: o endereço do grafo, os identificadores do par de vértices (V_i e V_j), e o peso da aresta (P)

Pré-condição: o grafo existir e os vértices serem válidos e a aresta não existir

Processo: inserir uma aresta do vértice de origem (V_i) para o vértice de destino (V_j) com peso P

Saída: 1 se sucesso, 0 se aresta já existe ou -1 se grafo inconsistente

Pós-condição: o grafo de entrada com uma nova aresta

Grafos: matriz de adjacências

int ***insere_aresta*** *(Grafo * G, int V1, int V2, int P)*

FIM

Grafos: matriz de adjacências

```
int insere_aresta (Grafo * G, int V1, int V2, int P)
        SE grafo não existe OU V1 ou V2 inválidos ENTÃO
            Retorna -1;
        FIM_SE

FIM
```

Grafos: matriz de adjacências

```
int insere_aresta (Grafo * G, int V1, int V2, int P)
        SE grafo não existe OU V1 ou V2 inválidos ENTÃO
            Retorna -1;
        FIM_SE

        SE aresta[V1,V2] ≠ 0 ENTÃO
            Retorna 0;
        FIM_SE
```

FIM

Grafos: matriz de adjacências

```
int insere_aresta (Grafo * G, int V1, int V2, int P)
        SE grafo não existe OU V1 ou V2 inválidos ENTÃO
            Retorna -1;
        FIM_SE

        SE aresta[V1,V2] ≠ 0 ENTÃO
            Retorna 0;
        FIM_SE

        Atribui P à aresta[V1,V2];



FIM
```

Grafos: matriz de adjacências

int ***insere_aresta*** *(Grafo * G, int V1, int V2, int P)*

SE grafo não existe OU V1 ou V2 inválidos ENTÃO
Retorna -1;
FIM_SE

SE aresta[V1,V2] ≠ 0 ENTÃO
Retorna 0;
FIM_SE

Atribui P à aresta[V1,V2];
Incrementa o campo qtde_arestas;

FIM

Grafos: matriz de adjacências

int ***insere_aresta*** *(Grafo * G, int V1, int V2, int P)*

SE grafo não existe OU V1 ou V2 inválidos ENTÃO
Retorna -1;
FIM_SE

SE aresta[V1,V2] ≠ 0 ENTÃO
Retorna 0;
FIM_SE

Atribui P à aresta[V1,V2];
Incrementa o campo qtde_arestas;
Incrementa o grau dos vértices V1 e V2;

FIM

Grafos: matriz de adjacências

*int **insere_aresta** (Grafo * G, int V1, int V2, int P)*

SE grafo não existe OU V1 ou V2 inválidos ENTÃO
Retorna -1;
FIM_SE

SE aresta[V1,V2] ≠ 0 ENTÃO
Retorna 0;
FIM_SE

Atribui P à aresta[V1,V2];
Incrementa o campo qtde_arestas;
Incrementa o grau dos vértices V1 e V2;
Retorna 1;

FIM

Especificação do TAD Dígrafo

Operação ***verifica_aresta:***

Entrada: o endereço do grafo e os identificadores do par de vértices (V_i e V_j)

Pré-condição: o grafo existir e os vértices serem válidos

Processo: verifica se existe aresta entre o vértice V_i (origem) e V_j (destino)

Saída: 1 se aresta existe, 0 se aresta não existe ou -1 se grafo inconsistente

Pós-condição: nenhuma

Grafos: matriz de adjacências

*int **verifica_aresta** (Grafo * G, int V1, int V2)*

FIM

Grafos: matriz de adjacências

int ***verifica_aresta*** *(Grafo * G, int V1, int V2)*

SE grafo não existe OU V1 ou V2 inválidos ENTÃO

Retorna -1;

FIM_SE

FIM

Grafos: matriz de adjacências

```
int verifica_aresta (Grafo * G, int V1, int V2)
        SE grafo não existe OU V1 ou V2 inválidos ENTÃO
            Retorna -1;
        FIM_SE

        SE aresta[V1,V2] = 0 ENTÃO
            Retorna 0;
        SENÃO
            Retorna 1;
        FIM_SE
FIM
```

Especificação do TAD Dígrafo

Operação ***remove_aresta:***

Entrada: o endereço do grafo e os identificadores do par de vértices (V_i e V_j)

Pré-condição: o grafo existir, os vértices serem válidos e a aresta desejada existir

Processo: remover a aresta existente entre os vértices V_i (origem) e V_j (destino)

Saída: 1 se sucesso, 0 se aresta não existe ou -1 se grafo inconsistente

Pós-condição: grafo de entrada com uma aresta a menos

Grafos: matriz de adjacências

*int **remove_aresta** (Grafo * G, int V1, int V2)*

FIM

Grafos: matriz de adjacências

*int **remove_aresta** (Grafo * G, int V1, int V2)*

SE grafo não existe OU V1 ou V2 inválidos ENTÃO

Retorna -1;

FIM_SE

FIM

Grafos: matriz de adjacências

```
int remove_aresta (Grafo * G, int V1, int V2)
        SE grafo não existe OU V1 ou V2 inválidos ENTÃO
            Retorna -1;
        FIM_SE

        SE aresta[V1,V2] = 0 ENTÃO
            Retorna 0;
        FIM_SE




FIM
```

Grafos: matriz de adjacências

*int **remove_aresta** (Grafo * G, int V1, int V2)*

SE grafo não existe OU V1 ou V2 inválidos ENTÃO
Retorna -1;
FIM_SE

SE aresta[V1,V2] = 0 ENTÃO
Retorna 0;
FIM_SE

*Atribui **ZERO** à aresta[V1,V2];*

FIM

Grafos: matriz de adjacências

*int **remove_aresta** (Grafo * G, int V1, int V2)*

SE grafo não existe OU V1 ou V2 inválidos ENTÃO
Retorna -1;
FIM_SE

SE aresta[V1,V2] = 0 ENTÃO
Retorna 0;
FIM_SE

*Atribui **ZERO** à aresta[V1,V2];*
Decrementa o campo qtde_arestas;

FIM

Grafos: matriz de adjacências

*int **remove_aresta** (Grafo * G, int V1, int V2)*

SE grafo não existe OU V1 ou V2 inválidos ENTÃO
Retorna -1;
FIM_SE

SE aresta[V1,V2] = 0 ENTÃO
Retorna 0;
FIM_SE

*Atribui **ZERO** à aresta[V1,V2];*
Decrementa o campo qtde_arestas;
Decrementa o grau dos vértices V1 e V2;

FIM

Grafos: matriz de adjacências

*int **remove_aresta** (Grafo * G, int V1, int V2)*

SE grafo não existe OU V1 ou V2 inválidos ENTÃO
Retorna -1;
FIM_SE

SE aresta[V1,V2] = 0 ENTÃO
Retorna 0;
FIM_SE

*Atribui **ZERO** à aresta[V1,V2];*
Decrementa o campo qtde_arestas;
Decrementa o grau dos vértices V1 e V2;
Retorna 1;

FIM

Especificação do TAD Dígrafo

Operação ***consulta_aresta*:**

Entrada: o endereço do grafo, os identificadores do par de vértices (V_i e V_j), e o endereço da variável de retorno do peso da aresta

Pré-condição: o grafo existir, os vértices serem válidos e a aresta desejada existir

Processo: atribuir o peso da aresta existente entre os vértices V_i e V_j para a variável de retorno

Saída: 1 se sucesso, 0 se aresta não existe ou -1 se grafo inconsistente

Pós-condição: nenhuma

Grafos: matriz de adjacências

int ***consulta_aresta*** *(Grafo * G, int V1, int V2, int * P)*

FIM

Grafos: matriz de adjacências

```
int consulta_aresta (Grafo * G, int V1, int V2, int * P)
        SE grafo não existe OU V1 ou V2 inválidos ENTÃO
            Retorna -1;
        FIM_SE




FIM
```

Grafos: matriz de adjacências

```
int consulta_aresta (Grafo * G, int V1, int V2, int * P)
        SE grafo não existe OU V1 ou V2 inválidos ENTÃO
            Retorna -1;
        FIM_SE

        SE aresta[V1,V2] = 0 ENTÃO
            Retorna 0;
        FIM_SE




FIM
```

Grafos: matriz de adjacências

int ***consulta_aresta*** *(Grafo * G, int V1, int V2, int * P)*

SE grafo não existe OU V1 ou V2 inválidos ENTÃO

Retorna -1;

FIM_SE

SE aresta[V1,V2] = 0 ENTÃO

Retorna 0;

FIM_SE

Atribui o valor da aresta[V1,V2] à variável de retorno P;

FIM

Grafos: matriz de adjacências

```
int consulta_aresta (Grafo * G, int V1, int V2, int * P)
        SE grafo não existe OU V1 ou V2 inválidos ENTÃO
            Retorna -1;
        FIM_SE

        SE aresta[V1,V2] = 0 ENTÃO
            Retorna 0;
        FIM_SE

        Atribui o valor da aresta[V1,V2] à variável de retorno P;
        Retorna 1;
FIM
```

Especificação do TAD Dígrafo

Operação ***libera_grafo:***

Entrada: o endereço do endereço do grafo

Pré-condição: o grafo existir

Processo: liberar a área ocupada pelo grafo

Saída: nenhuma

Pós-condição: grafo inexistente

Grafos: matriz de adjacências

libera_grafo *(Grafo ** G)*

FIM

Grafos: matriz de adjacências

libera_grafo *(Grafo ** G)*

PARA cada linha i da matriz de adjacência FAÇA

Libera memória alocada para aresta[i];

FIM_PARA

FIM

Grafos: matriz de adjacências

libera_grafo *(Grafo ** G)*

PARA cada linha i da matriz de adjacência FAÇA

Libera memória alocada para aresta[i];

FIM_PARA

*Libera memória alocada para a matriz de adj. (**aresta**);*

FIM

Grafos: matriz de adjacências

libera_grafo *(Grafo ** G)*

PARA cada linha i da matriz de adjacência FAÇA

Libera memória alocada para aresta[i];

FIM_PARA

*Libera memória alocada para a matriz de adj. (**aresta**);*

*Libera memória alocada para o vetor com o **grau** de vértices;*

FIM

Grafos: matriz de adjacências

libera_grafo *(Grafo ** G)*

PARA cada linha i da matriz de adjacência FAÇA

Libera memória alocada para aresta[i];

FIM_PARA

*Libera memória alocada para a matriz de adj. (**aresta**);*

*Libera memória alocada para o vetor com o **grau** de vértices;*

Libera memória alocada para o grafo;

FIM

Grafos: matriz de adjacências

libera_grafo *(Grafo ** G)*

PARA cada linha i da matriz de adjacência FAÇA

Libera memória alocada para aresta[i];

FIM_PARA

*Libera memória alocada para a matriz de adj. (**aresta**);*

*Libera memória alocada para o vetor com o **grau** de vértices;*

Libera memória alocada para o grafo;

*Atribui **NULL** ao endereço do grafo (G*);*

FIM

Especificação do TAD Dígrafo

Operação ***mostra_adjacentes:***

Entrada: o endereço do grafo e o identificador de um vértice *V*

Pré-condição: o grafo existir e o vértice ser válido

Processo: apresentar o conjunto dos vértices adjacentes ao vértice *V*

Saída: nenhuma

Pós-condição: nenhuma

Grafos: matriz de adjacências

mostra_adjacentes *(Grafo * G, int V)*

FIM

Grafos: matriz de adjacências

```
mostra_adjacentes (Grafo * G, int V)
        SE grafo não existe OU V inválido ENTÃO
            Escreva ("Grafo inexistente ou vértice inválido");

        FIM_SE
FIM
```

Grafos: matriz de adjacências

```
mostra_adjacentes (Grafo * G, int V)
        SE grafo não existe OU V inválido ENTÃO
           Escreva ("Grafo inexistente ou vértice inválido");
        SENAO

           PARA cada coluna i da matriz de adjacência FAÇA
              SE aresta[V,i] ≠ 0 ENTÃO  // Aresta existe
                 Escreva (V, "->", i, "=", aresta[V, i]);
              FIM_SE
           FIM_PARA



        FIM_SE
FIM
```

Grafos: matriz de adjacências

```
mostra_adjacentes (Grafo * G, int V)
        SE grafo não existe OU V inválido ENTÃO
           Escreva ("Grafo inexistente ou vértice inválido");
        SENAO
           Cria uma variável auxiliar CONT e a inicializa com ZERO;
           PARA cada coluna i da matriz de adjacência FAÇA
              SE aresta[V,i] ≠ 0 ENTÃO  // Aresta existe
                 Escreva (V, "->", i, "=", aresta[V, i]);   Incrementa CONT;
              FIM_SE
           FIM_PARA
           SE CONT = 0 ENTÃO
              Escreva ("O vértice ", V, " não possui vértices adjacentes.");
           FIM_SE
        FIM_SE
FIM
```

Especificação do TAD Dígrafo

Operação ***mostra_grafo:***

Entrada: o endereço do grafo

Pré-condição: o grafo existir

Processo: apresentar cada vértice do grafo e seu conjunto de vértices adjacentes

Saída: nenhuma

Pós-condição: nenhuma

Grafos: matriz de adjacências

mostra_grafo *(Grafo * G)*

FIM

Grafos: matriz de adjacências

```
mostra_grafo (Grafo * G)
        SE grafo não existe ENTÃO
            Escreva ("Grafo inexistente");




        FIM_SE
FIM
```

Grafos: matriz de adjacências

```
mostra_grafo (Grafo * G)
        SE grafo não existe ENTÃO
            Escreva ("Grafo inexistente");

        SENÃO SE qtde_arestas = 0 ENTÃO
            Escreva ("Grafo vazio");









        FIM_SE
FIM
```

Grafos: matriz de adjacências

```
mostra_grafo (Grafo * G)
        SE grafo não existe ENTÃO
            Escreva ("Grafo inexistente");

        SENÃO SE qtde_arestas = 0 ENTÃO
            Escreva ("Grafo vazio");

        SENÃO
            PARA cada linha i da matriz de adjacência FAÇA
                Escreva ("Vértice ", i, ":");
                mostra_adjacentes(G, i);
            FIM_PARA

        FIM_SE
FIM
```

Grafos: formas de implementação

Listas de Adjacências

Grafos: listas de adjacências

Representada por **2 estruturas aninhadas:**

Grafos: listas de adjacências

Representada por **2 estruturas aninhadas:**

Estrutura do **nó**:

Identificador do **vértice** adjacente

Peso associado à **aresta**

Endereço do **próximo nó** de adjacência

Grafos: listas de adjacências

Representada por **2 estruturas aninhadas:**

Estrutura do **nó**:

- Identificador do **vértice** adjacente
- Peso associado à **aresta**
- Endereço do **próximo nó** de adjacência

Estrutura do **grafo**:

- **Quantidade** de vértices e arestas
- Vetor com o **grau** dos vértices
- **Lista** de adjacências dos vértices
  - **Endereço do 1º nó da lista**

Grafos: listas de adjacências

Implementação em C:

```
struct no {
    int vertice;
    int peso;
    struct no * prox;
};
```

Grafos: listas de adjacências

Implementação em C:

```
struct no {
    int vertice;
    int peso;
    struct no * prox;
};

struct grafo {
    int qtde_vertices, qtde_arestas;
    int * grau;
    struct no ** aresta;  // vetor de end. 1º nó
};
```

Grafos: listas de adjacências

Implementação em C:

```
struct no {
    int vertice;
    int peso;
    struct no * prox;
};

struct grafo {
    int qtde_vertices, qtde_arestas;
    int * grau;
    struct no ** aresta;  // vetor de end. 1º nó
};
typedef struct grafo Grafo;
```

Grafos: listas de adjacências

Implementação em C:

```
struct no {
    int vertice;
    int peso;
    struct no * prox;
};

typedef struct no No;  // Opcional
struct grafo {
    int qtde_vertices, qtde_arestas;
    int * grau;
    No ** aresta;  // vetor de end. 1º nó
};
typedef struct grafo Grafo;
```

Grafos: listas de adjacências

Implementação em C:

grafo.c

```
struct no {
    int vertice;
    int peso;
    struct no * prox;
};

typedef struct no No;  // Opcional
struct grafo {
    int qtde_vertices, qtde_arestas;
    int * grau;
    No ** aresta;  // vetor de end. 1º nó
};
```

grafo.h

```
typedef struct grafo Grafo;
```

Especificação do TAD Dígrafo

Operação ***cria_grafo:***

Entrada: a quantidade de vértices

Pré-condição: quantidade de vértices ser válida

Processo: alocar a área para representar o grafo, se necessário, e colocá-lo na **condição de vazio**

Saída: o endereço do grafo se operação bem sucedida ou ***NULL*** se houve algum erro

Pós-condição: nenhuma

Grafos: listas de adjacências

*Grafo * **cria_grafo** (int nro_vertices)*

FIM

Grafos: listas de adjacências

```
Grafo * cria_grafo (int nro_vertices)
        SE nro_vertices inválido ENTÃO
           Retorna NULL;
        FIM_SE
```

FIM

Grafos: listas de adjacências

```
Grafo * cria_grafo (int nro_vertices)
        SE nro_vertices inválido ENTÃO
            Retorna NULL;
        FIM_SE
        Aloca memória para o grafo;

FIM
```

Grafos: listas de adjacências

```
Grafo * cria_grafo (int nro_vertices)
        SE nro_vertices inválido ENTÃO
            Retorna NULL;
        FIM_SE
        Aloca memória para o grafo;
        SE alocação bem sucedida ENTÃO
            Preenche os campos qtde_vertices e qtde_arestas do grafo;









        FIM_SE

FIM
```

Grafos: listas de adjacências

```
Grafo * cria_grafo (int nro_vertices)
        SE nro_vertices inválido ENTÃO
            Retorna NULL;
        FIM_SE
        Aloca memória para o grafo;
        SE alocação bem sucedida ENTÃO
            Preenche os campos qtde_vertices e qtde_arestas do grafo;
            Aloca memória para o vetor com o grau dos vértices e
            inicializa seus elementos com ZERO;
        FIM_SE
FIM
```

Grafos: listas de adjacências

*Grafo * **cria_grafo** (int nro_vertices)*

SE nro_vertices inválido ENTÃO

Retorna NULL;

FIM_SE

Aloca memória para o grafo;

SE alocação bem sucedida ENTÃO

Preenche os campos qtde_vertices e qtde_arestas do grafo;

Aloca memória para o vetor com o grau dos vértices e

inicializa seus elementos com ZERO;

*Aloca memória para o vetor de listas de adjacências (**vetor de ponteiros**)*

FIM_SE

FIM

Grafos: listas de adjacências

```
Grafo * cria_grafo (int nro_vertices)
        SE nro_vertices inválido ENTÃO
            Retorna NULL;
        FIM_SE
        Aloca memória para o grafo;
        SE alocação bem sucedida ENTÃO
            Preenche os campos qtde_vertices e qtde_arestas do grafo;
            Aloca memória para o vetor com o grau dos vértices e
            inicializa seus elementos com ZERO;
            Aloca memória para o vetor de listas de adjacências  (vetor de ponteiros)
            SE alocação bem sucedida ENTÃO
                PARA cada elemento do vetor FAÇA
                    Inicializa elemento (ponteiro) com NULL;
                FIM_PARA
            FIM_SE
        FIM_SE

FIM
```

Grafos: listas de adjacências

```
Grafo * cria_grafo (int nro_vertices)
        SE nro_vertices inválido ENTÃO
            Retorna NULL;
        FIM_SE
        Aloca memória para o grafo;
        SE alocação bem sucedida ENTÃO
            Preenche os campos qtde_vertices e qtde_arestas do grafo;
            Aloca memória para o vetor com o grau dos vértices e
            inicializa seus elementos com ZERO;
            Aloca memória para o vetor de listas de adjacências  (vetor de ponteiros)
            SE alocação bem sucedida ENTÃO
                PARA cada elemento do vetor FAÇA
                    Inicializa elemento (ponteiro) com NULL;
                FIM_PARA
            FIM_SE
        FIM_SE
        Retorna o endereço do grafo;
FIM
```

Especificação do TAD Dígrafo

Operação ***insere_aresta:***

Entrada: o endereço do grafo, os identificadores do par de vértices (V_i e V_j), e o peso da aresta (P)

Pré-condição: o grafo existir e os vértices serem válidos e a aresta não existir

Processo: inserir uma aresta do vértice de origem (V_i) para o vértice de destino (V_j) com peso P

Saída: 1 se sucesso, 0 se aresta já existe ou -1 se grafo inconsistente

Pós-condição: o grafo de entrada com uma nova aresta

Grafos: listas de adjacências

int ***insere_aresta*** *(Grafo * G, int V1, int V2, int P)*

Grafos: listas de adjacências

```
int insere_aresta (Grafo * G, int V1, int V2, int P)
        SE grafo não existe OU V1 ou V2 inválidos ENTÃO
            Retorna -1;
        FIM_SE
```

Grafos: listas de adjacências

*int **insere_aresta** (Grafo * G, int V1, int V2, int P)*

SE grafo não existe OU V1 ou V2 inválidos ENTÃO

Retorna -1;

FIM_SE

// Percorre a lista a procura da aresta

*Atribui a um ponteiro **aux** o endereço apontado pela lista (**aresta[V1]**);*

ENQUANTO aux ≠ NULL E aux->vertice ≠ V2 FAÇA

*Faz **aux** apontar para o próximo nó de adjacência (**avança aux**);*

FIM_ENQUANTO

Grafos: listas de adjacências

*int **insere_aresta** (Grafo * G, int V1, int V2, int P)*

SE grafo não existe OU V1 ou V2 inválidos ENTÃO

Retorna -1;

FIM_SE

// Percorre a lista a procura da aresta

*Atribui a um ponteiro **aux** o endereço apontado pela lista (**aresta[V1]**);*

ENQUANTO aux ≠ NULL E aux->vertice ≠ V2 FAÇA

*Faz **aux** apontar para o próximo nó de adjacência (**avança aux**);*

FIM_ENQUANTO

SE aux ≠ NULL ENTÃO

Retorna 0; // Aresta já existe

FIM_SE

•••

Grafos: listas de adjacências

...

Aloca memória para um novo nó;

FIM

Grafos: listas de adjacências

...

Aloca memória para um novo nó;

SE alocação bem sucedida ENTÃO // Preenche o novo nó e o insere na lista

Atribui ao campo vertice do novo nó o valor de V2;

Atribui ao campo peso do novo nó o valor de P;

Atribui ao campo prox do novo nó o valor da lista (aresta[V1]);

Atribui à lista o endereço do novo nó;

FIM_SE

FIM

Grafos: listas de adjacências

```
...
Aloca memória para um novo nó;
SE alocação bem sucedida ENTÃO // Preenche o novo nó e o insere na lista
    Atribui ao campo vertice do novo nó o valor de V2;
    Atribui ao campo peso do novo nó o valor de P;
    Atribui ao campo prox do novo nó o valor da lista (aresta[V1]);
    Atribui à lista o endereço do novo nó;
FIM_SE

// Atualiza as informações do grafo
Incrementa qte_arestas;
Incrementa o grau dos vértices V1 e V2;

FIM
```

Grafos: listas de adjacências

```
...
Aloca memória para um novo nó;
SE alocação bem sucedida ENTÃO // Preenche o novo nó e o insere na lista
    Atribui ao campo vertice do novo nó o valor de V2;
    Atribui ao campo peso do novo nó o valor de P;
    Atribui ao campo prox do novo nó o valor da lista (aresta[V1]);
    Atribui à lista o endereço do novo nó;
FIM_SE

// Atualiza as informações do grafo
Incrementa o campo qte_arestas;
Incrementa o grau dos vértices V1 e V2;
Retorna 1;
FIM
```

Especificação do TAD Dígrafo

Operação ***verifica_aresta:***

Entrada: o endereço do grafo e os identificadores do par de vértices (V_i e V_j)

Pré-condição: o grafo existir e os vértices serem válidos

Processo: verifica se existe aresta entre o vértice V_i (origem) e V_j (destino)

Saída: 1 se aresta existe, 0 se aresta não existe ou -1 se grafo inconsistente

Pós-condição: nenhuma

Grafos: listas de adjacências

*int **verifica_aresta** (Grafo * G, int V1, int V2)*

FIM

Grafos: listas de adjacências

```
int verifica_aresta (Grafo * G, int V1, int V2)
        SE grafo não existe OU V1 ou V2 inválidos ENTÃO
            Retorna -1;
        FIM_SE
```

FIM

Grafos: listas de adjacências

*int **verifica_aresta** (Grafo * G, int V1, int V2)*

SE grafo não existe OU V1 ou V2 inválidos ENTÃO

Retorna -1;

FIM_SE

// Percorre a lista a procura da aresta

*Atribui a um ponteiro **aux** o endereço apontado pela lista (**aresta[V1]**);*

ENQUANTO aux ≠ NULL E aux->vertice ≠ V2 FAÇA

*Faz **aux** apontar para o próximo nó de adjacência (**avança aux**);*

FIM_ENQUANTO

FIM

Grafos: listas de adjacências

*int **verifica_aresta** (Grafo * G, int V1, int V2)*

SE grafo não existe OU V1 ou V2 inválidos ENTÃO

Retorna -1;

FIM_SE

// Percorre a lista a procura da aresta

*Atribui a um ponteiro **aux** o endereço apontado pela lista (**aresta[V1]**);*

ENQUANTO aux ≠ NULL E aux->vertice ≠ V2 FAÇA

*Faz **aux** apontar para o próximo nó de adjacência (**avança aux**);*

FIM_ENQUANTO

SE aux = NULL ENTÃO

Retorna 0; // Aresta não existe

SENÃO

Retorna 1; // Aresta já existe

FIM_SE

FIM

Especificação do TAD Dígrafo

Operação ***remove_aresta:***

Entrada: o endereço do grafo e os identificadores do par de vértices (V_i e V_j)

Pré-condição: o grafo existir, os vértices serem válidos e a aresta desejada existir

Processo: remover a aresta existente entre os vértices V_i (origem) e V_j (destino)

Saída: 1 se sucesso, 0 se aresta não existe ou -1 se grafo inconsistente

Pós-condição: grafo de entrada com uma aresta a menos

Grafos: listas de adjacências

int ***remove_aresta*** *(Grafo * G, int V1, int V2)*

Grafos: listas de adjacências

```
int remove_aresta (Grafo * G, int V1, int V2)
    SE grafo não existe OU V1 ou V2 inválidos ENTÃO
        Retorna -1;
    FIM_SE
```

Grafos: listas de adjacências

*int **remove_aresta** (Grafo * G, int V1, int V2)*

SE grafo não existe OU V1 ou V2 inválidos ENTÃO

Retorna -1;

FIM_SE

// Percorre a lista a procura da aresta

*Atribui a um ponteiro **aux** o end. apontado pela lista (**aresta[V1]**) ;*

*Atribui **NULL** a um ponteiro **ant**;*

Grafos: listas de adjacências

int ***remove_aresta*** *(Grafo * G, int V1, int V2)*

SE grafo não existe OU V1 ou V2 inválidos ENTÃO

Retorna -1;

FIM_SE

// Percorre a lista a procura da aresta

Atribui a um ponteiro ***aux*** *o end. apontado pela lista (****aresta[V1]*** *) ;*

Atribui ***NULL*** *a um ponteiro* ***ant****;*

ENQUANTO aux ≠ NULL E aux->vertice ≠ V2 FAÇA

Atribui a ***ant*** *o valor atual de* ***aux****;*

Faz ***aux*** *apontar para o próximo nó da lista (****avança aux****);*

FIM_ENQUANTO

•••

Grafos: listas de adjacências

...

SE ***aux*** *= NULL ENTÃO* *// Aresta não existe*

Retorna 0;

FIM_SE

FIM

Grafos: listas de adjacências

...

SE ***aux*** *= NULL ENTÃO* *// Aresta não existe*

Retorna 0;

FIM_SE

SE ant = NULL ENTÃO *// Verifica se é o nó apontado pela lista (1º nó)*

*Faz a lista (****aresta[V1]****) apontar para o sucessor de* ***aux****;*

FIM_SE

FIM

Grafos: listas de adjacências

...

SE ***aux*** *= NULL ENTÃO* *// Aresta não existe*

Retorna 0;

FIM_SE

SE ant = NULL ENTÃO *// Verifica se é o nó apontado pela lista (1º nó)*

*Faz a lista (****aresta[V1]****) apontar para o sucessor de* ***aux****;*

SENÃO

Faz o nó apontado por ***ant*** *apontar para o sucessor de* ***aux****;*

FIM_SE

FIM

Grafos: listas de adjacências

...

SE ***aux*** *= NULL ENTÃO* *// Aresta não existe*

Retorna 0;

FIM_SE

SE ant = NULL ENTÃO *// Verifica se é o nó apontado pela lista (1º nó)*

*Faz a lista (****aresta[V1]****) apontar para o sucessor de* ***aux****;*

SENÃO

Faz o nó apontado por ***ant*** *apontar para o sucessor de* ***aux****;*

FIM_SE

Libera memória alocada para o nó apontado por ***aux****;*

FIM

Grafos: listas de adjacências

...

*SE **aux** = NULL ENTÃO // Aresta não existe*

Retorna 0;

FIM_SE

SE ant = NULL ENTÃO // Verifica se é o nó apontado pela lista (1º nó)

*Faz a lista (**aresta[V1]**) apontar para o sucessor de **aux**;*

SENÃO

*Faz o nó apontado por **ant** apontar para o sucessor de **aux**;*

FIM_SE

*Libera memória alocada para o nó apontado por **aux**;*

Decrementa o campo qtde_arestas;

Decrementa o grau dos vértices V1 e V2;

FIM

Grafos: listas de adjacências

```
...
SE aux = NULL ENTÃO   // Aresta não existe
    Retorna 0;
FIM_SE

SE ant = NULL ENTÃO   // Verifica se é o nó apontado pela lista (1º nó)
    Faz a lista (aresta[V1]) apontar para o sucessor de aux;
SENÃO
    Faz o nó apontado por ant apontar para o sucessor de aux;
FIM_SE
Libera memória alocada para o nó apontado por aux;
Decrementa o campo qtde_arestas;
Decrementa o grau dos vértices V1 e V2;
Retorna 1;
FIM
```

Especificação do TAD Dígrafo

Operação ***consulta_aresta*:**

Entrada: o endereço do grafo, os identificadores do par de vértices (V_i e V_j), e o endereço da variável de retorno do peso da aresta

Pré-condição: o grafo existir, os vértices serem válidos e a aresta desejada existir

Processo: atribuir o peso da aresta existente entre os vértices V_i e V_j para a variável de retorno

Saída: 1 se sucesso, 0 se aresta não existe ou -1 se grafo inconsistente

Pós-condição: nenhuma

Grafos: listas de adjacências

int ***consulta_aresta*** *(Grafo * G, int V1, int V2, int * P)*

FIM

Grafos: listas de adjacências

```
int consulta_aresta (Grafo * G, int V1, int V2, int * P)
        SE grafo não existe OU V1 ou V2 inválidos ENTÃO
           Retorna -1;
        FIM_SE
```

FIM

Grafos: listas de adjacências

int ***consulta_aresta*** *(Grafo * G, int V1, int V2, int * P)*

SE grafo não existe OU V1 ou V2 inválidos ENTÃO

Retorna -1;

FIM_SE

// Percorre a lista a procura da aresta

Atribui a um ponteiro ***aux*** *o endereço apontado pela lista (****aresta[V1]*** *);*

ENQUANTO aux ≠ NULL E aux->vertice ≠ V2 FAÇA

Faz ***aux*** *apontar para o próximo nó de adjacência (****avança aux****);*

FIM_ENQUANTO

FIM

Grafos: listas de adjacências

```
int consulta_aresta (Grafo * G, int V1, int V2, int * P)
        SE grafo não existe OU V1 ou V2 inválidos ENTÃO
            Retorna -1;
        FIM_SE
        // Percorre a lista a procura da aresta
        Atribui a um ponteiro aux o endereço apontado pela lista (aresta[V1] );
        ENQUANTO aux ≠ NULL E aux->vertice ≠ V2 FAÇA
            Faz aux apontar para o próximo nó de adjacência (avança aux);
        FIM_ENQUANTO
        SE aux = NULL ENTÃO
            Retorna 0;   // Aresta não existe




        FIM_SE
FIM
```

Grafos: listas de adjacências

```
int consulta_aresta (Grafo * G, int V1, int V2, int * P)
        SE grafo não existe OU V1 ou V2 inválidos ENTÃO
            Retorna -1;
        FIM_SE
        // Percorre a lista a procura da aresta
        Atribui a um ponteiro aux o endereço apontado pela lista (aresta[V1] );
        ENQUANTO aux ≠ NULL E aux->vertice ≠ V2 FAÇA
            Faz aux apontar para o próximo nó de adjacência (avança aux);
        FIM_ENQUANTO
        SE aux = NULL ENTÃO
            Retorna 0;   // Aresta não existe
        SENÃO
            Atribui o peso do nó apontado por aux à variável de retorno (*P);
            Retorna 1;
        FIM_SE
FIM
```

Especificação do TAD Dígrafo

Operação ***libera_grafo:***

Entrada: o endereço do endereço do grafo

Pré-condição: o grafo existir

Processo: liberar a área ocupada pelo grafo

Saída: nenhuma

Pós-condição: grafo inexistente

Grafos: listas de adjacências

libera_grafo *(Grafo ** G)*

FIM

Grafos: listas de adjacências

libera_grafo *(Grafo ** G)*

PARA cada linha i do vetor de listas de adjacências FAÇA

*Atribui a **aux** o endereço apontado pela lista (**aresta[i]**);*

ENQUANTO aux ≠ NULL FAÇA

*Atribui a **aux2** o valor de **aux**;*

*Atribui a **aux** o endereço de seu sucessor (**avança aux**);*

*Libera memória do nó apontado por **aux2**;*

FIM_ENQUANTO

FIM_PARA

FIM

Grafos: listas de adjacências

libera_grafo *(Grafo ** G)*

PARA cada linha i do vetor de listas de adjacências FAÇA

*Atribui a **aux** o endereço apontado pela lista (**aresta[i]**);*

ENQUANTO aux ≠ NULL FAÇA

*Atribui a **aux2** o valor de **aux**;*

*Atribui a **aux** o endereço de seu sucessor (**avança aux**);*

*Libera memória do nó apontado por **aux2**;*

FIM_ENQUANTO

FIM_PARA

*Libera memória alocada para o vetor de listas (**aresta**);*

FIM

Grafos: listas de adjacências

libera_grafo *(Grafo ** G)*

PARA cada linha i do vetor de listas de adjacências FAÇA

*Atribui a **aux** o endereço apontado pela lista (**aresta[i]**);*

ENQUANTO aux ≠ NULL FAÇA

*Atribui a **aux2** o valor de **aux**;*

*Atribui a **aux** o endereço de seu sucessor (**avança aux**);*

*Libera memória do nó apontado por **aux2**;*

FIM_ENQUANTO

FIM_PARA

*Libera memória alocada para o vetor de listas (**aresta**);*

*Libera memória alocada para o vetor com o **grau** dos vértices;*

FIM

Grafos: listas de adjacências

libera_grafo *(Grafo ** G)*

PARA cada linha i do vetor de listas de adjacências FAÇA

*Atribui a **aux** o endereço apontado pela lista (**aresta[i]**);*

ENQUANTO aux ≠ NULL FAÇA

*Atribui a **aux2** o valor de **aux**;*

*Atribui a **aux** o endereço de seu sucessor (**avança aux**);*

*Libera memória do nó apontado por **aux2**;*

FIM_ENQUANTO

FIM_PARA

*Libera memória alocada para o vetor de listas (**aresta**);*

*Libera memória alocada para o vetor com o **grau** dos vértices;*

*Libera memória alocada para o **grafo**;*

FIM

Grafos: listas de adjacências

libera_grafo *(Grafo ** G)*

PARA cada linha i do vetor de listas de adjacências FAÇA

*Atribui a **aux** o endereço apontado pela lista (**aresta[i]**);*

ENQUANTO aux ≠ NULL FAÇA

*Atribui a **aux2** o valor de **aux**;*

*Atribui a **aux** o endereço de seu sucessor (**avança aux**);*

*Libera memória do nó apontado por **aux2**;*

FIM_ENQUANTO

FIM_PARA

*Libera memória alocada para o vetor de listas (**aresta**);*

*Libera memória alocada para o vetor com o **grau** dos vértices;*

*Libera memória alocada para o **grafo**;*

*Atribui **NULL** ao endereço do grafo (G*);*

FIM

Grafos: listas de adjacências

libera_grafo *(Grafo ** G)*

PARA cada linha i do vetor de listas de adjacências FAÇA

*Atribui a **aux** o endereço apontado pela lista (**aresta[i]**);*

ENQUANTO aux ≠ NULL FAÇA

*Atribui a **aux2** o valor de **aux**;*

*Atribui a **aux** o endereço de seu sucessor (**avança aux**);*

*Libera memória do nó apontado por **aux2**;*

FIM_ENQUANTO

FIM_PARA

*Libera memória alocada para o vetor de listas (**aresta**);*

*Libera memória alocada para o vetor com o **grau** dos vértices;*

*Libera memória alocada para o **grafo**;*

*Atribui **NULL** ao endereço do grafo (G*);*

FIM

Especificação do TAD Dígrafo

Operação ***mostra_adjacentes*:**

Entrada: o endereço do grafo e o identificador de um vértice *V*

Pré-condição: o grafo existir e o vértice ser válido

Processo: apresentar o conjunto dos vértices adjacentes ao vértice *V*

Saída: nenhuma

Pós-condição: nenhuma

Grafos: listas de adjacências

mostra_adjacentes *(Grafo * G, int V)*

FIM

Grafos: listas de adjacências

```
mostra_adjacentes (Grafo * G, int V)
        SE grafo não existe OU V inválido ENTÃO
            Escreva ("Grafo inexistente ou vértice inválido");

        FIM_SE
FIM
```

Grafos: listas de adjacências

```
mostra_adjacentes (Grafo * G, int V)
        SE grafo não existe OU V inválido ENTÃO
           Escreva ("Grafo inexistente ou vértice inválido");
        SENAO
           Atribui a aux o end. apontado pela lista do vértice V (aresta[V] );
           SE aux = NULL ENTÃO
              Escreva ("O vértice ", V, " não possui vértices adjacentes.");









           FIM_SE
        FIM_SE
FIM
```

Grafos: listas de adjacências

```
mostra_adjacentes (Grafo * G, int V)
        SE grafo não existe OU V inválido ENTÃO
           Escreva ("Grafo inexistente ou vértice inválido");
        SENAO
           Atribui a aux o end. apontado pela lista do vértice V (aresta[V] );
           SE aux = NULL ENTÃO
              Escreva ("O vértice ", V, " não possui vértices adjacentes.");
           SENÃO
              ENQUANTO aux ≠ NULL FAÇA
                 Escreva (V, "->", aux->vertice, "=", aux->peso);
                 Atribui a aux o endereço de seu sucessor (avança aux);
              FIM_ENQUANTO
           FIM_SE
        FIM_SE
FIM
```

Especificação do TAD Dígrafo

Operação ***mostra_grafo:***

Entrada: o endereço do grafo

Pré-condição: o grafo existir

Processo: apresentar cada vértice do grafo e seu conjunto de vértices adjacentes

Saída: nenhuma

Pós-condição: nenhuma

Grafos: listas de adjacências

mostra_grafo *(Grafo * G)*

FIM

Grafos: listas de adjacências

```
mostra_grafo (Grafo * G)
        SE grafo não existe ENTÃO
            Escreva ("Grafo inexistente");
        FIM_SE
FIM
```

Grafos: listas de adjacências

```
mostra_grafo (Grafo * G)
        SE grafo não existe ENTÃO
            Escreva ("Grafo inexistente");

        SENÃO SE qtde_arestas = 0 ENTÃO
            Escreva ("Grafo vazio");









        FIM_SE
FIM
```

Grafos: listas de adjacências

```
mostra_grafo (Grafo * G)
        SE grafo não existe ENTÃO
           Escreva ("Grafo inexistente");

        SENÃO SE qtde_arestas = 0 ENTÃO
           Escreva ("Grafo vazio");

        SENÃO
           PARA cada linha i do vetor de listas de adjacências FAÇA
              Escreva ("Vértice ", i, ":");
              mostra_adjacentes(G, i);
           FIM_PARA

        FIM_SE
FIM
```

Grafos: listas de adjacências

```
mostra_grafo (Grafo * G)
        SE grafo não existe ENTÃO
           Escreva ("Grafo inexistente");

        SENÃO SE qtde_arestas = 0 ENTÃO
           Escreva ("Grafo vazio");

        SENÃO
           PARA cada linha i do vetor de listas de adjacências FAÇA
              Escreva ("Vértice ", i, ":");
              mostra_adjacentes(G, i);
           FIM_PARA

        FIM_SE
FIM
```

NÃO MUDA em relação à matriz de adjacências

Exercícios

1. Implemente as operações básicas do TAD dígrafo (grafo direcionado) usando matriz de adjacências e listas de adjacências.

2. Altere o exercício anterior de modo a implementar as operações básicas do TAD grafo não direcionado.

3. Altere a implementação do TAD dígrafo usando listas de adjacências, de modo a adotar a seguinte forma de representação:

```
struct no {
    int vertice;
    int peso;
    struct no * prox;
};
```

```
struct vertice {
    int grau;
    struct no * aresta;
};
```

```
struct grafo {
    int qtde_vertices;
    int qtde_arestas;
    struct vertice * prox;
};
```

Bibliografia

Slides adaptados do material da Profa. Dra. Denise Guliato.

BACKES, A. Linguagem C Descomplicada: portal de vídeo-aulas para estudo de programação. Disponível em: https://programacaodescomplicada.wordpress.com/indice/estrutura-de-dados/

CORMEN, T.H. et al. Algoritmos: Teoria e Prática, Campus, 2002

ZIVIANI, N. Projeto de algoritmos: com implementações em Pascal e C (2ª ed.), Thomson, 2004

MORAES, C.R. Estruturas de Dados e Algoritmos: uma abordagem didática (2ª ed.), Futura, 2003