ACH2024

Aula 5

Implementação de Grafos por Lista de Adjacências

Profa. Ariane Machado Lima

Aula passada...

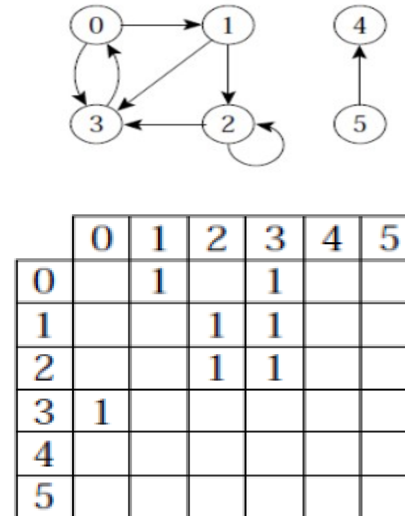
EACH USP
Profa. Ariane Machado Lima

grafo_matrizadj.h

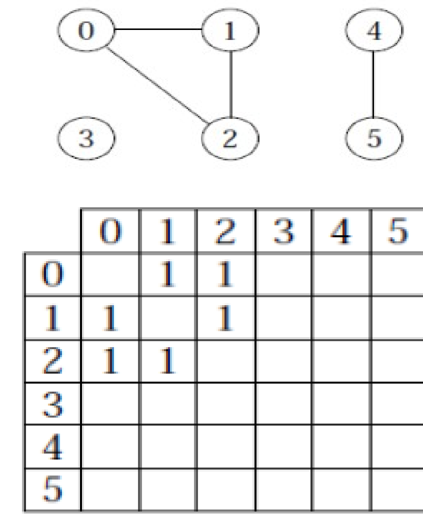
```
#include <stdbool.h>

#define MAXNUMVERTICES 100
#define AN -1                     /* aresta nula (representa ausencia de aresta) */
#define VERTICE_INVALIDO -1  /* vertice inexistente */

typedef int Peso;
typedef struct {
  Peso mat[MAXNUMVERTICES][MAXNUMVERTICES];
  int numVertices;
  int numArestas;
} Grafo;

void inicializaGrafo(Grafo* grafo, int nv);
void insereAresta(int v1, int v2, Peso peso, Grafo *grafo);
bool existeAresta(int v1, int v2, Grafo *grafo);
void removeAresta(int v1, int v2, Peso* peso, Grafo *grafo);
bool listaAdjVazia(int v, Grafo* grafo);
int primeiroListaAdj(int v, Grafo* grafo);
int proxListaAdj(int v, Grafo* grafo, int prox);
void imprimeGrafo(Grafo* grafo);
void liberaGrafo(Grafo* grafo);
```

- Implementação por matriz de adjacência

	Matriz de adj.
inicializaGrafo	$O(v^2)$
imprimeGrafo	$O(v^2)$
insereAresta	O(1)
existeAresta	O(1)
removeAresta	O(1)
listaAdjVazia	O(v)
proxListaAdj	O(v)
liberaGrafo	O(1)



	0	1	2	3	4	5
0		1		1		
1			1	1		
2			1	1		
3	1					
4						
5						



	0	1	2	3	4	5
0		1	1			
1	1		1			
2	1	1				
3						
4						
5						

Aula de hoje

Um outro tipo de implementação de grafos

EACH USP
Profa. Ariane Machado Lima

Matriz de adjacência - Reflexões

Essa representação por matriz adjacência é sempre eficiente?

	Matriz de adj.
inicializaGrafo	$O(v^2)$
imprimeGrafo	$O(v^2)$
insereAresta	O(1)
existeAresta	O(1)
removeAresta	O(1)
listaAdjVazia	O(v)
proxListaAdj	O(v)
liberaGrafo	O(1)

Matriz de adjacência - Reflexões

Essa representação por matriz adjacência é sempre eficiente?

	Matriz de adj.
inicializaGrafo	$O(v^2)$
imprimeGrafo	$O(v^2)$
insereAresta	O(1)
existeAresta	O(1)
removeAresta	O(1)
listaAdjVazia	O(v)
proxListaAdj	O(v)
liberaGrafo	O(1)

- ✓ Acesso instantâneo a uma aresta (tempo constante) – consulta, inserção e remoção

Mesmo que o grafo tenha poucas arestas (**esparso**):

- ✗ Utilização da lista de adjacência em O(v)
- ✗ Espaço $\Omega(v^2)$

Matriz de adjacência - Reflexões

Essa representação por matriz adjacência é sempre eficiente?

	Matriz de adj.
inicializaGrafo	$O(v^2)$
imprimeGrafo	$O(v^2)$
insereAresta	O(1)
existeAresta	O(1)
removeAresta	O(1)
listaAdjVazia	O(v)
proxListaAdj	O(v)
liberaGrafo	O(1)

- ✓ Acesso instantâneo a uma aresta (tempo constante) – consulta, inserção e remoção
- ✓ **Para grafos densos OK !!!**

Mesmo que o grafo tenha poucas arestas (**esparso**):

- ✗ Utilização da lista de adjacência em O(v)
- ✗ Espaço $\Omega(v^2)$

Grafos Completos

- Um grafo completo é um grafo não direcionado no qual todos os pares de vértices são adjacentes.
- Possui quantas arestas ?

Grafos Completos

- Um grafo completo é um grafo não direcionado no qual todos os pares de vértices são adjacentes.
- Possui $(|V|^2 - |V|)/2 = |V|(|V| - 1)/2$ arestas, pois do total de $|V|^2$ pares possíveis de vértices devemos subtrair $|V|$ *self-loops* e dividir por 2 (cada aresta ligando dois vértices é contada duas vezes).

Grafos Completos

- Um grafo completo é um grafo não direcionado no qual todos os pares de vértices são adjacentes.
- Possui $(|V|^2 - |V|)/2 = |V|(|V| - 1)/2$ arestas, pois do total de $|V|^2$ pares possíveis de vértices devemos subtrair $|V|$ *self-loops* e dividir por 2 (cada aresta ligando dois vértices é contada duas vezes).

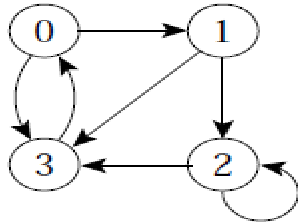
Mas na matriz, como representamos (i,j) e (j,i), só a diagnonal tem AN

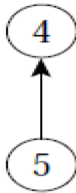
Se o número de arestas é próximo de v^2, ele é denso, e matriz de adjacência pode ser uma boa escolha...

Grafos Completos

- Um grafo completo é um grafo não direcionado no qual todos os pares de vértices são adjacentes.
- Possui $(|V|^2 - |V|)/2 = |V|(|V| - 1)/2$ arestas, pois do total de $|V|^2$ pares possíveis de vértices devemos subtrair $|V|$ *self-loops* e dividir por 2 (cada aresta ligando dois vértices é contada duas vezes).
- O número total de **grafos diferentes** com $|V|$ vértices é $2^{|V|(|V|-1)/2}$ (número de maneiras diferentes de escolher um subconjunto a partir de $|V|(|V| - 1)/2$ possíveis arestas).

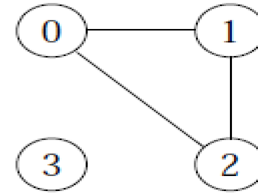
Outra sugestão de implementação para grafos não densos?

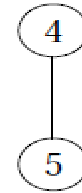




	0	1	2	3	4	5
0		1		1		
1			1	1		
2			1	1		
3	1					
4						
5						

(a)



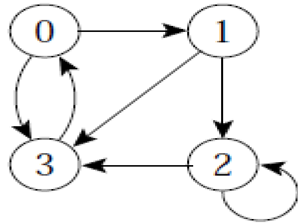


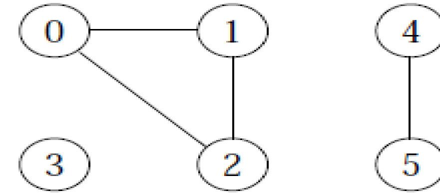
	0	1	2	3	4	5
0		1	1			
1	1		1			
2	1	1				
3						
4						
5						

(b)

Outra sugestão de implementação para grafos não densos?

Matriz esparsa?





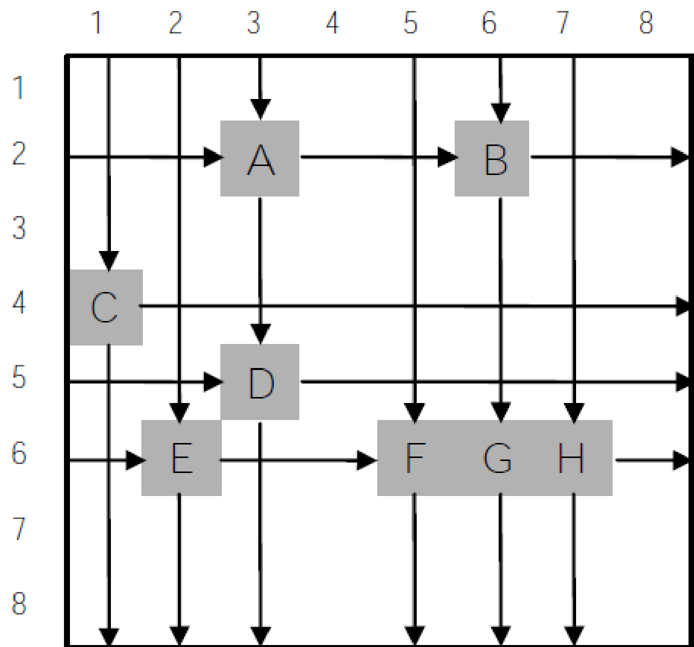
	0	1	2	3	4	5
0		1		1		
1			1	1		
2			1	1		
3	1					
4						
5						

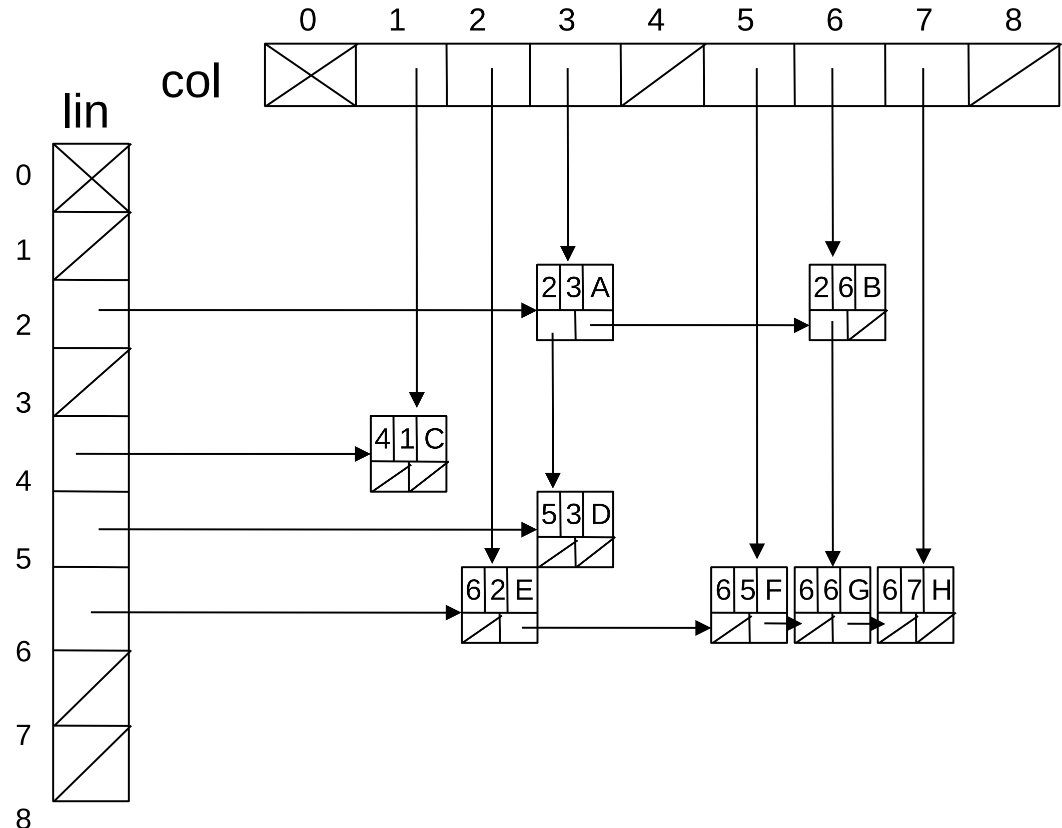
(a)

	0	1	2	3	4	5
0		1	1			
1	1		1			
2	1	1				
3						
4						
5						

(b)

Solução: Implementação de matrizes esparsas por listas cruzadas





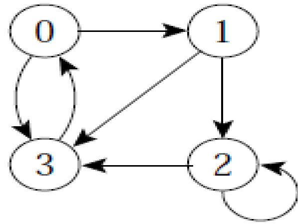
Vantajoso quando:

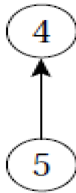
Outra sugestão de implementação para grafos não densos?

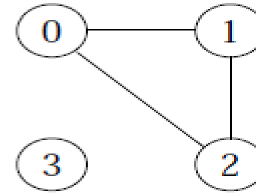
Matriz esparsa?

Mesmo sabendo que eu tenho TODOS os vértices de 1 a numVertices?

Faço muitas buscas tanto por linha quanto por coluna?







	0	1	2	3	4	5
0		1		1		
1			1	1		
2			1	1		
3	1					
4						
5						

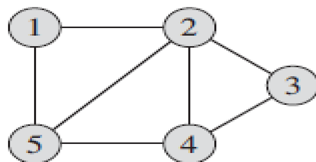
(a)

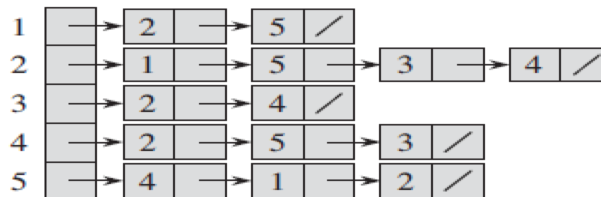
	0	1	2	3	4	5
0		1	1			
1	1		1			
2	1	1				
3						
4						
5						

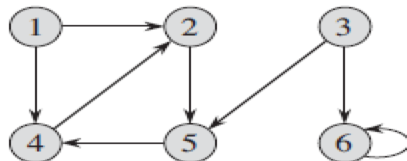
(b)

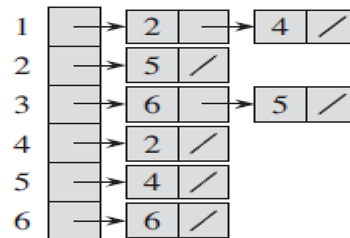
Listas de adjacência

Pois normalmente quero os vértices adjacentes, e não os vizinhos



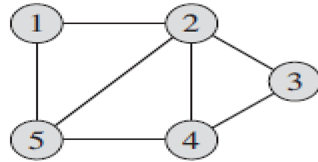


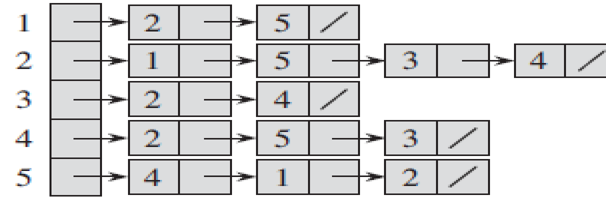


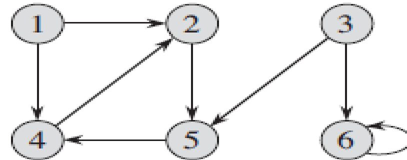


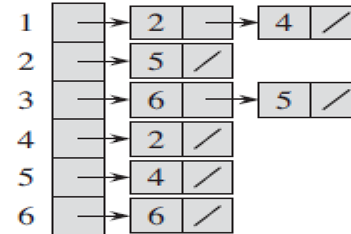
Listas de adjacência

Pois normalmente quero os vértices adjacentes, e não os vizinhos







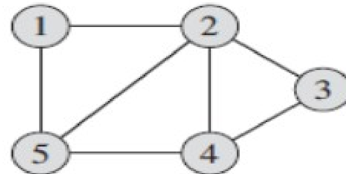


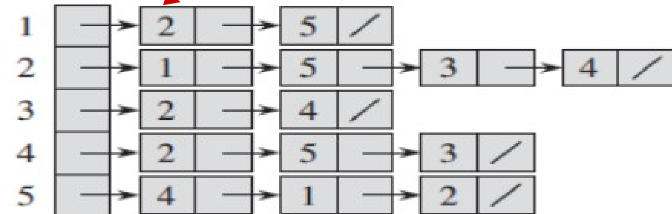
Note que a lista de adjacência não precisa estar ordenada!!!

EACH USP
Profa. Ariane Machado Lima

Arquivo grafo_listaadj.h

```
typedef int Peso;

/* tipo estruturado Aresta : vertice destino, peso, ponteiro p/ prox. aresta */
typedef struct str_aresta {
  int vdest;
  Peso peso;
  struct str_aresta * prox;
} Aresta;
```

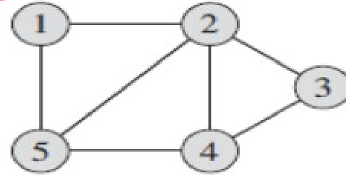


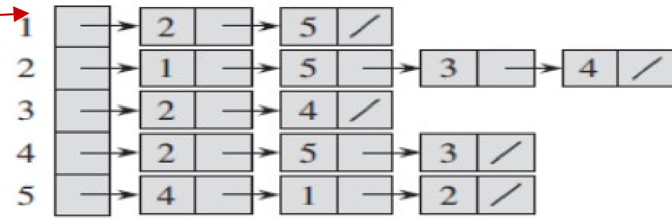


Profa. Ariane Machado Lima

Arquivo grafo_listaadj.h

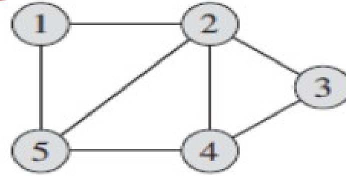
```
typedef int Peso;

/* tipo estruturado Aresta : vertice destino, peso, ponteiro p/ prox. aresta */
typedef struct str_aresta {
  int vdest;
  Peso peso;
  struct str_aresta * prox;
} Aresta;
/*
  tipo estruturado grafo:
      listaAdj: vetor de Arestas ligadas (cada posicao i contem o ponteiro
                para o inicio da lista de adjacencia do vertice i)
*/
typedef struct {
  Aresta **listaAdj;
  int numVertices;
  int numArestas;
} Grafo;
```

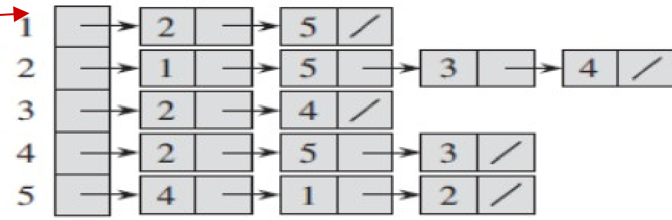




Arquivo grafo_listaadj.h

```
typedef int Peso;

/* tipo estruturado Aresta : vertice destino, peso, ponteiro p/ prox. aresta */
typedef struct str_aresta {
  int vdest;
  Peso peso;
  struct str_aresta * prox;
} Aresta;
/*
  tipo estruturado grafo:
      listaAdj: vetor de Arestas ligadas (cada posicao i contem o ponteiro
                para o inicio da lista de adjacencia do vertice i)
*/
typedef struct {
  Aresta **listaAdj;
  int numVertices;
  int numArestas;
} Grafo;
```

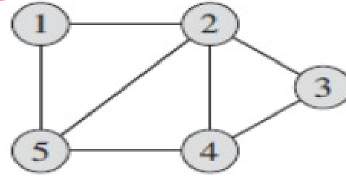


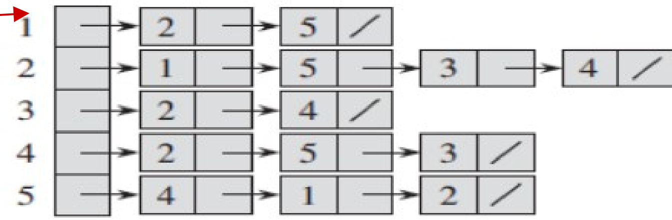


Por que eu não usei alocação estática aqui?

Arquivo grafo_listaadj.h

```
typedef int Peso;

/* tipo estruturado Aresta : vertice destino, peso, ponteiro p/ prox. aresta */
typedef struct str_aresta {
  int vdest;
  Peso peso;
  struct str_aresta * prox;
} Aresta;
/*
  tipo estruturado grafo:
      listaAdj: vetor de Arestas ligadas (cada posicao i contem o ponteiro
                para o inicio da lista de adjacencia do vertice i)
*/
typedef struct {
  Aresta **listaAdj;
  int numVertices;
  int numArestas;
} Grafo;
```

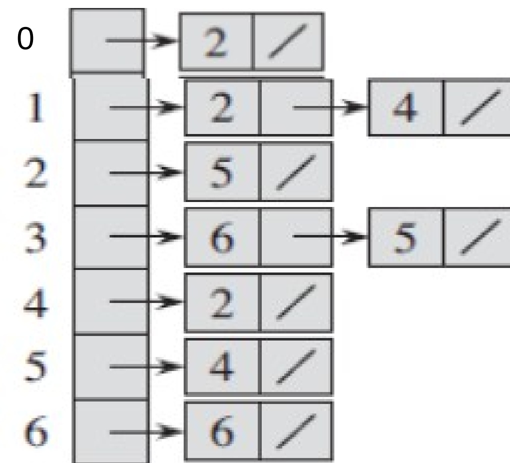




Por que eu não usei alocação estática aqui? Porque essa é uma estrutura de dados para quando estou querendo também economizar espaço

Arquivo grafo_listaadj.h

```
#include <stdbool.h>    /* variaveis bool assumem valores "true" ou "false" */

#define VERTICE_INVALIDO NULL   /* numero de vertice invalido ou ausente */
#define AN -1                   /* aresta nula */

typedef int Peso;

/* tipo estruturado Aresta : vertice destino, peso, ponteiro p/ prox. aresta */
typedef struct str_aresta {
  int vdest;
  Peso peso;
  struct str_aresta* prox;
} Aresta;
/*
  tipo estruturado grafo:
      listaAdj: vetor de Arestas ligadas (cada posicao i contem o ponteiro
                para o inicio da lista de adjacencia do vertice i)
*/
typedef struct {
  Aresta** listaAdj;
  int numVertices;
  int numArestas;
} Grafo;
```

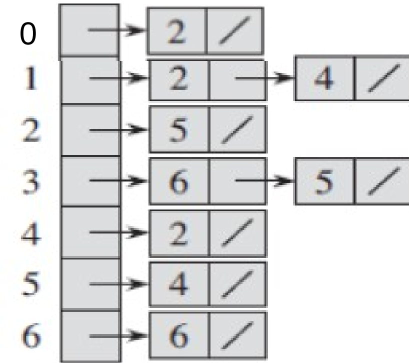



```
/*
  bool inicializaGrafo(Grafo* grafo, int nv): Inicializa um grafo com nv
vértices
  (...)
  Retorna true se inicializou com sucesso e false c.c.
*/
  bool inicializaGrafo(Grafo* grafo, int nv);
```

Arquivo grafo_listaadj.c

```c
#include <stdio.h>
#include <stdlib.h>

/* Leitura do header com estruturas e tipos especiais */
#include "grafo_listaadj.h"

/*
  inicializaGrafo(TipoGrafo* grafo, int nv): Cria um grafo com n vertices.
  Aloca espaco para o vetor de apontadores de listas de adjacencias e,
  para cada vertice, inicializa o apontador de sua lista de adjacencia.
  Retorna true se inicializou com sucesso e false caso contrario.
  Vertices vao de 1 a nv.
*/
bool inicializaGrafo(TipoGrafo *grafo, int nv) {
```

```c
if (grafo == NULL) {
  fprintf(stderr, "ERRO na chamada de inicializaGrafo: grafo == NULL.\n");
  return false;
```




```c
typedef struct str_aresta {
  int vdest;
  Peso peso;
  struct str_aresta* prox;
} Aresta;
```

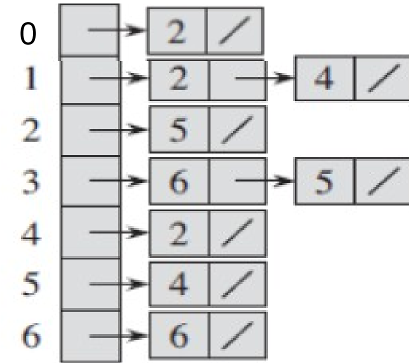
```c
typedef struct {
  Aresta** listaAdj;
  int numVertices;
  int numArestas;
} Grafo;
```

Arquivo grafo_listaadj.c

```c
#include <stdio.h>
#include <stdlib.h>

/* Leitura do header com estruturas e tipos especiais */
#include "grafo_listaadj.h"

/*
  inicializaGrafo(TipoGrafo* grafo, int nv): Cria um grafo com n vertices.
  Aloca espaco para o vetor de apontadores de listas de adjacencias e,
  para cada vertice, inicializa o apontador de sua lista de adjacencia.
  Retorna true se inicializou com sucesso e false caso contrario.
  Vertices vao de 1 a nv.
*/
bool inicializaGrafo(TipoGrafo *grafo, int nv) {
```

```c
if (grafo == NULL) {
  fprintf(stderr, "ERRO na chamada de inicializaGrafo: grafo == NULL.\n");
  return false;
}
```

```c
  if (nv <= 0) {
    fprintf(stderr, "ERRO na chamada de inicializaGrafo: Numero de vertices deve ser positivo.\n");
    return false;
  }

  grafo->numVertices = nv;
  if (!(grafo->listaAdj = (Aresta**) calloc(nv , sizeof(Aresta*)))){
    fprintf(stderr, "ERRO: Falha na alocacao de memoria na funcao inicializaGrafo\n");
    return false;
  }
  grafo->numArestas = 0;
  //calloc ja inicializa com zeros.... nao precisa inicializar grafo->listaAdj[i]
  return true;
}
```




```c
typedef struct str_aresta {
  int vdest;
  Peso peso;
  struct str_aresta* prox;
} Aresta;
```

```c
typedef struct {
  Aresta** listaAdj;
  int numVertices;
  int numArestas;
} Grafo;
```

Arquivo testa_grafo.c

```c
//#include "grafo_matrizadj.h"
#include "grafo_listaadj.h"
#include <stdio.h>

int main()
{
  Grafo g1;
  int numVertices;

  //inicializaGrafo(&g1, 10);

  do {
   printf("Digite o número de vértices do grafo\n");
   scanf("%d", &numVertices);
  } while (!inicializaGrafo(&g1, numVertices));


  //imprimeGrafo(&g1);

  return 0;
}
```

Mudança apenas nessas duas linhas !!!!

Complexidades
(considerando um grafo de v vértices e a arestas)

	Matriz de adj.	Lista de adj.
inicializaGrafo	$O(v^2)$	
imprimeGrafo	$O(v^2)$	
insereAresta	O(1)	
existeAresta	O(1)	
removeAresta	O(1)	
listaAdjVazia	O(v)	
proxListaAdj	O(v)	
liberaGrafo	O(1)	

Complexidades
(considerando um grafo de v vértices e a arestas)

	Matriz de adj.	Lista de adj.
inicializaGrafo	$O(v^2)$	$O(v)$
imprimeGrafo	$O(v^2)$	
insereAresta	$O(1)$	
existeAresta	$O(1)$	
removeAresta	$O(1)$	
listaAdjVazia	$O(v)$	
proxListaAdj	$O(v)$	
liberaGrafo	$O(1)$	

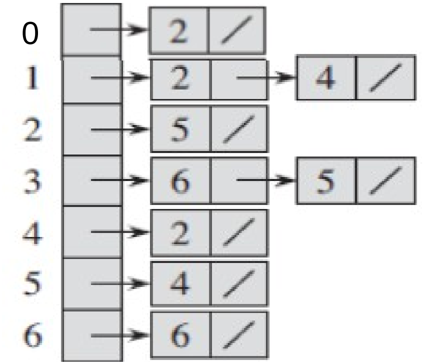
Implementação

Implementar e analisar a complexidade das operações:

- insereAresta
- existeAresta
- removeAresta
- listaAdjVazia: **true** se a lista de adjacentes de um dado vértice é vazia, **false** c.c.
- proxListaAdj: retorna o próximo vértice adjacente de um dado vértice (próximo em relação a um adjacente "atual" passado como parâmetro); na primeira chamada retorna o primeiro; pense no que fazer se não houver próximo...

CONSIDERE QUE O GRAFO É DIRECIONADO!!!!

Verificação se a lista de adjacência de um vértice é vazia

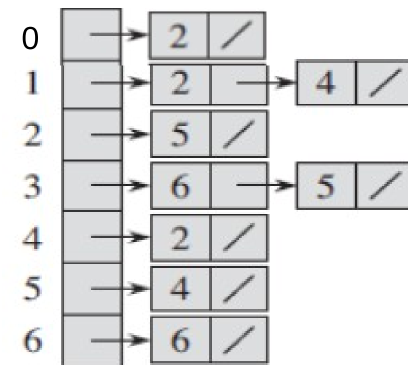



```c
typedef struct str_aresta {
  int vdest;
  Peso peso;
  struct str_aresta* prox;
} Aresta;
```

```c
typedef struct {
  Aresta** listaAdj;
  int numVertices;
  int numArestas;
} Grafo;
```

Verificação se a lista de adjacência de um vértice é vazia

```
/*
   bool listaAdjVazia(int v, Grafo* grafo):
   Retorna true se a lista de adjacencia (de vertices adjacentes)
   do vertice v é vazia, e false caso contrário.
*/
bool listaAdjVazia(int v, Grafo* grafo){
  if (! verificaValidadeVertice(v, grafo))
    return false;
  return (grafo->listaAdj[v]==NULL);
}
```



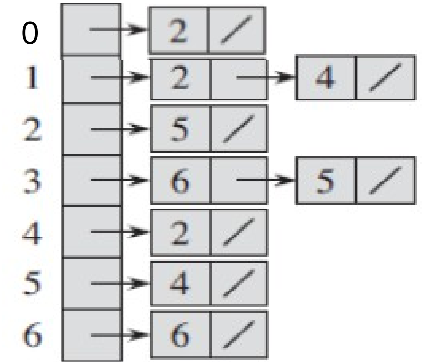

```
typedef struct str_aresta {
  int vdest;
  Peso peso;
  struct str_aresta* prox;
} Aresta;
```

```
typedef struct {
  Aresta** listaAdj;
  int numVertices;
  int numArestas;
} Grafo;
```


EACH USP
Profa. Ariane Machado Lima

Verificação se a lista de adjacência de um vértice é vazia

```c
/*
   bool listaAdjVazia(int v, Grafo* grafo):
   Retorna true se a lista de adjacencia (de vertices adjacentes)
   do vertice v é vazia, e false caso contrário.
*/
bool listaAdjVazia(int v, Grafo* grafo){
  if (! verificaValidadeVertice(v, grafo))
    return false;
  return (grafo->listaAdj[v]==NULL);
}
```




```c
typedef struct str_aresta {
  int vdest;
  Peso peso;
  struct str_aresta* prox;
} Aresta;
```

```c
typedef struct {
  Aresta** listaAdj;
  int numVertices;
  int numArestas;
} Grafo;
```

Protótipo da função na implementação por matriz de adjacência:

```c
bool listaAdjVazia(int v, Grafo* grafo);
```

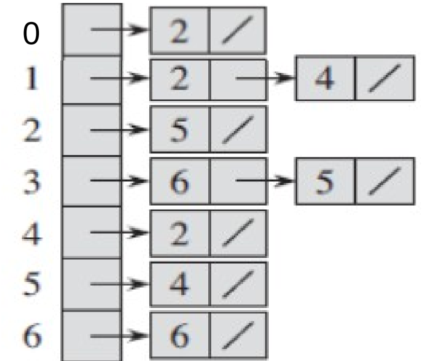
Complexidades
(considerando um grafo de v vértices e a arestas)

	Matriz de adj.	Lista de adj.
inicializaGrafo	$O(v^2)$	$O(v)$
imprimeGrafo	$O(v^2)$	
insereAresta	$O(1)$	
existeAresta	$O(1)$	
removeAresta	$O(1)$	
listaAdjVazia	$O(v)$	
proxListaAdj	$O(v)$	
liberaGrafo	$O(1)$	

Complexidades
(considerando um grafo de v vértices e a arestas)

	Matriz de adj.	Lista de adj.
inicializaGrafo	$O(v^2)$	$O(v)$
imprimeGrafo	$O(v^2)$	
insereAresta	$O(1)$	
existeAresta	$O(1)$	
removeAresta	$O(1)$	
listaAdjVazia	$O(v)$	$O(1)$
proxListaAdj	$O(v)$	
liberaGrafo	$O(1)$	

Próximo da lista de adjacência

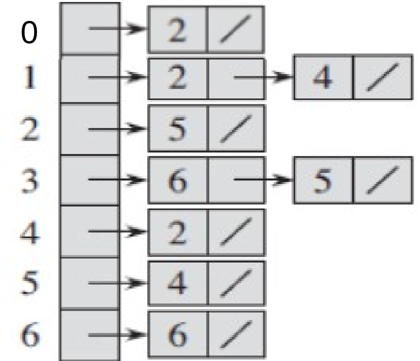



```c
typedef struct str_aresta {
  int vdest;
  Peso peso;
  struct str_aresta* prox;
} Aresta;
```

```c
typedef struct {
  Aresta** listaAdj;
  int numVertices;
  int numArestas;
} Grafo;
```

Próximo da lista de adjacência

```c
/*
   Aresta* proxListaAdj(int v, Grafo* grafo, Aresta* atual):
   Retorna o proximo vertice adjacente a v, partindo do vertice "atual" adjacente a v
   ou NULL se a lista de adjacencia tiver terminado sem um novo proximo.
*/
Aresta* proxListaAdj(int v, Grafo* grafo, Aresta* atual){
  if (atual == NULL) {
    fprintf(stderr, "atual == NULL\n");
    return VERTICE_INVALIDO;
  }
  return(atual->prox);
}
```

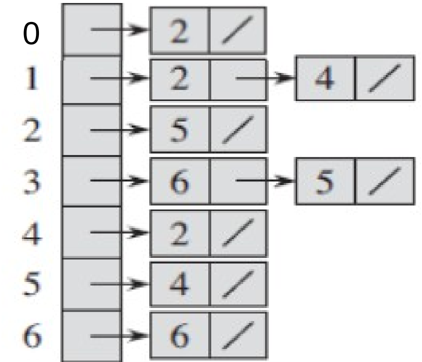



```c
typedef struct str_aresta {
  int vdest;
  Peso peso;
  struct str_aresta* prox;
} Aresta;
```

```c
typedef struct {
  Aresta** listaAdj;
  int numVertices;
  int numArestas;
} Grafo;
```

Próximo da lista de adjacência

```c
/*
   Aresta* proxListaAdj(int v, Grafo* grafo, Aresta* atual):
   Retorna o proximo vertice adjacente a v, partindo do vertice "atual" adjacente a v
   ou NULL se a lista de adjacencia tiver terminado sem um novo proximo.
*/
Aresta* proxListaAdj(int v, Grafo* grafo, Aresta* atual){
  if (atual == NULL) {
    fprintf(stderr, "atual == NULL\n");
    return VERTICE_INVALIDO;
  }
  return(atual->prox);
}
```




```c
typedef struct str_aresta {
  int vdest;
  Peso peso;
  struct str_aresta* prox;
} Aresta;
```

```c
typedef struct {
  Aresta** listaAdj;
  int numVertices;
  int numArestas;
} Grafo;
```

Protótipo da função na implementação por matriz de adjacência:

```c
int proxListaAdj(int v, Grafo* grafo, int atual);
```

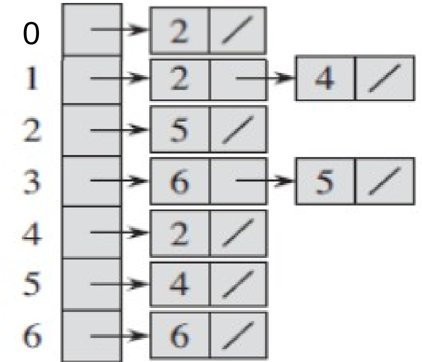
Próximo da lista de adjacência

```
/*
   Aresta* proxListaAdj(int v, Grafo* grafo, Aresta* atual):
   Retorna o proximo vertice adjacente a v, partindo do vertice "atual" adjacente a v
   ou NULL se a lista de adjacencia tiver terminado sem um novo proximo.
*/
Aresta* proxListaAdj(int v, Grafo* grafo, Aresta* atual){
  if (atual == NULL) {
    fprintf(stderr, "atual == NULL\n");
    return VERTICE_INVALIDO;
  }
  return(atual->prox);
}
```

INCONSISTÊNCIA NA INTERFACE!!!

Protótipo da função na implementação por matriz de adjacência:

```
int proxListaAdj(int v, Grafo* grafo, int atual);
```




```
typedef struct str_aresta {
  int vdest;
  Peso peso;
  struct str_aresta* prox;
} Aresta;
```

```
typedef struct {
  Aresta** listaAdj;
  int numVertices;
  int numArestas;
} Grafo;
```

Vamos definir um tipo Apontador

- Apontador é algo (um tipo coerente com a implementação por matriz ou lista de adjacência) que pode ser usado para identificar um vértice adjacente
  - Matriz de adjacência:
  - Lista de adjacência:

EACH USP
Profa. Ariane Machado Lima

Vamos definir um tipo Apontador

- Apontador é algo (um tipo coerente com a implementação por matriz ou lista de adjacência) que pode ser usado para identificar um vértice adjacente
  - Matriz de adjacência: int
  - Lista de adjacência:

Vamos definir um tipo Apontador

- Apontador é algo (um tipo coerente com a implementação por matriz ou lista de adjacência) que pode ser usado para identificar um vértice adjacente
  - Matriz de adjacência: int
  - Lista de adjacência: Aresta*

Vamos definir um tipo Apontador

- Apontador é algo (um tipo coerente com a implementação por matriz ou lista de adjacência) que pode ser usado para identificar um vértice adjacente
  - Matriz de adjacência: int
  - Lista de adjacência: Aresta*

```
typedef int Apontador;
```

← No arquivo grafo_matrizadj.h

```
typedef Aresta* Apontador;
```

← No arquivo grafo_listaadj.h

O que muda na implementação por matriz de adjacência?

```
#define MAXNUMVERTICES 100
#define AN -1                 /* aresta nula, ou seja, valor que representa ausencia de aresta */
#define VERTICE_INVALIDO -1   /* numero de vertice invalido ou ausente */

#include <stdbool.h>   /* variaveis bool assumem valores "true" ou "false" */

typedef int Peso;
typedef struct  {
  Peso mat[MAXNUMVERTICES + 1][MAXNUMVERTICES + 1];
  int numVertices;
  int numArestas;
} Grafo;
typedef int Apontador;

/*
   Apontador proxListaAdj(int v, Grafo* grafo, Apontador atual):
   Trata-se de um iterador sobre a lista de adjacência do vertice v.
   Retorna o proximo vertice adjacente a v, partindo do vertice "atual" adjacente a v
   ou VERTICE_INVALIDO se a lista de adjacencia tiver terminado sem um novo proximo.
*/
Apontador proxListaAdj(int v, Grafo* grafo, Apontador atual);
```

O que muda na implementação por lista de adjacência?

```c
#include <stdbool.h>    /* variaveis bool assumem valores "true" ou "false" */

#define VERTICE_INVALIDO NULL  /* numero de vertice invalido ou ausente */
#define AN -1                  /* aresta nula */

typedef int Peso;

/*
  tipo estruturado taresta:
      vertice destino, peso, ponteiro p/ prox. aresta
*/
typedef struct str_aresta {
  int vdest;
  Peso peso;
  struct str_aresta* prox;
} Aresta;

typedef Aresta* Apontador;

/*
  tipo estruturado grafo:
      vetor de listas de adjacencia (cada posicao contem o ponteiro
                  para o inicio da lista de adjacencia do vertice)
      numero de vertices
*/
typedef struct {
  Apontador* listaAdj;
  int numVertices;
  int numArestas;
} Grafo;
```

O que muda na implementação por lista de adjacência?

```
/*
  inicializaGrafo(TipoGrafo* grafo, int nv): Cria um grafo com n vertices.
  Aloca espaco para o vetor de apontadores de listas de adjacencias e,
  para cada vertice, inicializa o apontador de sua lista de adjacencia.
  Retorna true se inicializou com sucesso e false caso contrario.
  Vertices vao de 1 a nv.
*/
bool inicializaGrafo(Grafo *grafo, int nv) {
  int i;

  if (nv <= 0) {
    fprintf(stderr, "ERRO na chamada de inicializaGrafo: Numero de vertices deve ser positivo.\n");
    return false;
  }

  grafo->numVertices = nv;
  if (!(grafo->listaAdj = (Apontador*) calloc(nv , sizeof(Apontador)))){
    fprintf(stderr, "ERRO: Falha na alocacao de memoria na funcao inicializaGrafo\n");
    return false;
  }
  grafo->numArestas = 0;
  //calloc ja inicializa com zeros.... nao precisa inicializar grafo->listaAdj[i]
  return true;
}
```

Próximo da lista de adjacência

```
/*
   Apontador proxListaAdj(int v, Grafo* grafo, Apontador atual):
   Retorna o proximo vertice adjacente a v, partindo do vertice "atual" adjacente a v
   ou NULL se a lista de adjacencia tiver terminado sem um novo proximo.
*/
Apontador proxListaAdj(int v, Grafo* grafo, Apontador atual){
  if (atual == NULL) {
    fprintf(stderr, "atual == NULL\n");
    return VERTICE_INVALIDO;
  }
  return(atual->prox);
}
```

Complexidades
(considerando um grafo de v vértices e a arestas)

	Matriz de adj.	Lista de adj.
inicializaGrafo	$O(v^2)$	$O(v)$
imprimeGrafo	$O(v^2)$	
insereAresta	$O(1)$	
existeAresta	$O(1)$	
removeAresta	$O(1)$	
listaAdjVazia	$O(v)$	$O(1)$
proxListaAdj	$O(v)$	
liberaGrafo	$O(1)$	

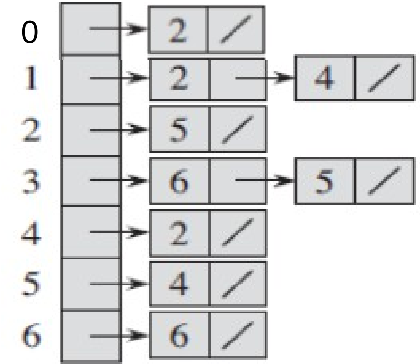
Complexidades
(considerando um grafo de v vértices e a arestas)

	Matriz de adj.	Lista de adj.
inicializaGrafo	$O(v^2)$	$O(v)$
imprimeGrafo	$O(v^2)$	
insereAresta	$O(1)$	
existeAresta	$O(1)$	
removeAresta	$O(1)$	
listaAdjVazia	$O(v)$	$O(1)$
proxListaAdj	$O(v)$	$O(1)$
liberaGrafo	$O(1)$	

Primeiro da lista de adjacência

Na matriz era só chamar:

proxListaAdj(v, grafo, -1);

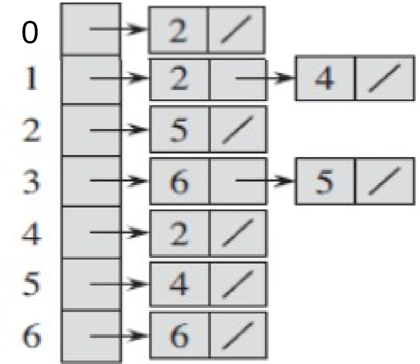



```c
typedef struct str_aresta {
  int vdest;
  Peso peso;
  struct str_aresta* prox;
} Aresta;
```

```c
typedef struct {
  Apontador* listaAdj;
  int numVertices;
  int numArestas;
} Grafo;
```

Primeiro da lista de adjacência

```
/*
  primeiroListaAdj(v, Grafo): retorna o endereco do primeiro vertice
  adjacente a v.
*/
Apontador primeiroListaAdj(int v, Grafo *grafo) {
  return(grafo->listaAdj[v]);
}
```

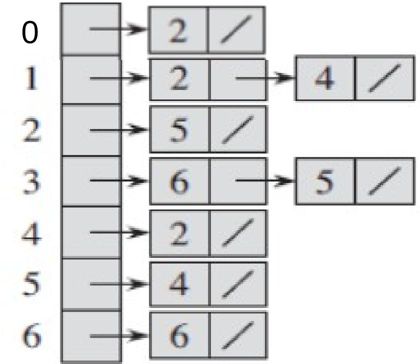



```
typedef struct str_aresta {
  int vdest;
  Peso peso;
  struct str_aresta* prox;
} Aresta;
```

```
typedef struct {
  Apontador* listaAdj;
  int numVertices;
  int numArestas;
} Grafo;
```

Primeiro da lista de adjacência

```
/*
  primeiroListaAdj(v, Grafo): retorna o endereco do primeiro vertice
  adjacente a v.
*/
Apontador primeiroListaAdj(int v, Grafo *grafo) {
  return(grafo->listaAdj[v]);
}
```




```
typedef struct str_aresta {
  int vdest;
  Peso peso;
  struct str_aresta* prox;
} Aresta;
```

```
typedef struct {
  Apontador* listaAdj;
  int numVertices;
  int numArestas;
} Grafo;
```

E na matriz de adjacência? Como seria essa função? Implemente!

Existência de aresta

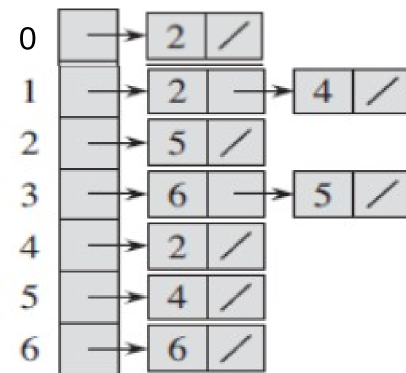
```
/*
  bool existeAresta(int v1, int v2, Grafo *grafo):
  Retorna true se existe a aresta (v1, v2) no grafo e false caso contrário
*/
bool existeAresta(int v1, int v2, Grafo *grafo)
{
```

Essa é para vocês fazerem agora !!

```
typedef struct str_aresta {
  int vdest;
  Peso peso;
  struct str_aresta* prox;
} Aresta;
```

```
typedef struct {
  Apontador* listaAdj;
  int numVertices;
  int numArestas;
} Grafo;
```

0
1
2
3
4
5
6

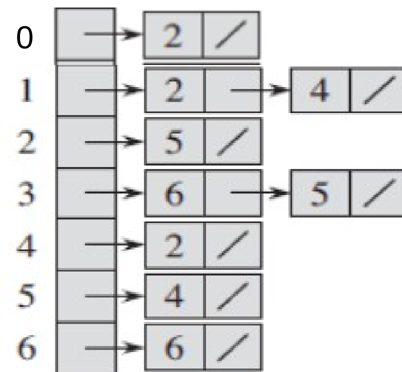


Existência de aresta

```
/*
  bool existeAresta(int v1, int v2, Grafo *grafo):
  Retorna true se existe a aresta (v1, v2) no grafo e false caso contrário
*/
bool existeAresta(int v1, int v2, Grafo *grafo)
{
  Apontador q;

  if (! (verificaValidadeVertice(v1, grafo) && verificaValidadeVertice(v2, grafo)))
      return false;

  q = grafo->listaAdj[v1];
  while ((q != NULL) && (q->vdest != v2))
    q = q->prox;
  if (q != NULL) return true;
  return false;
}
```

```
typedef struct str_aresta {
  int vdest;
  Peso peso;
  struct str_aresta* prox;
} Aresta;
```

```
typedef struct {
  Apontador* listaAdj;
  int numVertices;
  int numArestas;
} Grafo;
```



Complexidades (considerando um grafo de v vértices e a arestas)

	Matriz de adj.	Lista de adj.
inicializaGrafo	$O(v^2)$	$O(v)$
imprimeGrafo	$O(v^2)$	
insereAresta	$O(1)$	
existeAresta	$O(1)$	
removeAresta	$O(1)$	
listaAdjVazia	$O(v)$	$O(1)$
proxListaAdj	$O(v)$	
liberaGrafo	$O(1)$	

Complexidades (considerando um grafo de v vértices e a arestas)

	Matriz de adj.	Lista de adj.
inicializaGrafo	$O(v^2)$	$O(v)$
imprimeGrafo	$O(v^2)$	
insereAresta	$O(1)$	
existeAresta	$O(1)$	$O(v)$
removeAresta	$O(1)$	
listaAdjVazia	$O(v)$	$O(1)$
proxListaAdj	$O(v)$	
liberaGrafo	$O(1)$	

Obtenção do peso da aresta

```
/*
  Peso obtemPesoAresta(int v1, int v2, Grafo *grafo):
  Retorna o peso da aresta (v1, v2) no grafo se ela existir e AN caso contrário
*/
Peso obtemPesoAresta(int v1, int v2, Grafo *grafo)
{
```

O(v)

Implemente as demais a seguir:

EACH USP
Profa. Ariane Machado Lima

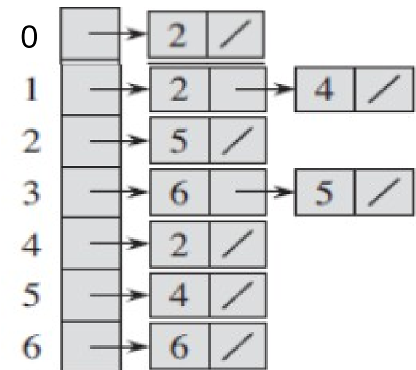
Inserção de aresta

```
/*
  void insereAresta(int v1, int v2, Peso peso, Grafo *grafo):
  Insere a aresta (v1, v2) com peso "peso" no grafo.
  Nao verifica se a aresta ja existia (isso deve ser feito pelo usuario antes, se necessario).
*/
void insereAresta(int v1, int v2, Peso peso, Grafo *grafo){
```

(**não** verifica se ela já existe; sem ordenação dos adjacentes)

```
typedef struct str_aresta {
  int vdest;
  Peso peso;
  struct str_aresta* prox;
} Aresta;
```

```
typedef struct {
  Apontador* listaAdj;
  int numVertices;
  int numArestas;
} Grafo;
```

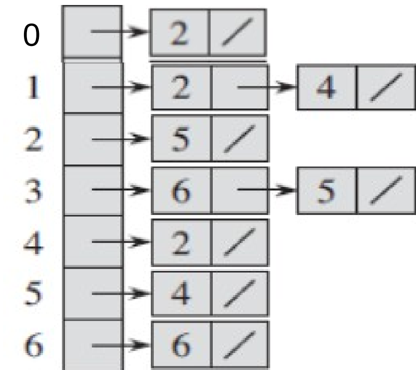


Remoção de aresta

```
/*
  bool removeArestaObtendoPeso(int v1, int v2, Peso* peso, Grafo *grafo);
  Remove a aresta (v1, v2) do grafo.
  Se a aresta existia, coloca o peso dessa aresta em "peso" e retorna true,
  caso contrario retorna false (e "peso" é inalterado).
*/
bool removeArestaObtendoPeso(int v1, int v2, Peso* peso, Grafo *grafo)
{
```

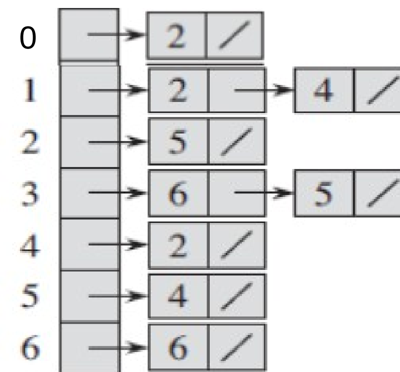
```
typedef struct str_aresta {
  int vdest;
  Peso peso;
  struct str_aresta* prox;
} Aresta;
```

```
typedef struct {
  Apontador* listaAdj;
  int numVertices;
  int numArestas;
} Grafo;
```



EACH USP Profa. Ariane Machado Lima

Liberação do espaço em memória




```c
typedef struct str_aresta {
  int vdest;
  Peso peso;
  struct str_aresta* prox;
} Aresta;
```

```c
typedef struct {
  Apontador* listaAdj;
  int numVertices;
  int numArestas;
} Grafo;
```

Impressão do grafo

```
/*
    void imprimeGrafo(Grafo* grafo):
    Imprime os vertices e arestas do grafo no seguinte formato:
    v1: (adj11, peso11); (adj12, peso12); ...
    v2: (adj21, peso21); (adj22, peso22); ...
    Assuma que cada vértice é um inteiro de até 2 dígitos.
*/
void imprimeGrafo(Grafo *grafo) {
```

Incrementando o Makefile

```
testa_matriz: grafo_matrizadj.o testa_grafo.o
	gcc -o testa_grafo_matriz.exe grafo_matrizadj.o testa_grafo.o

grafo_matrizadj.o: grafo_matrizadj.c grafo_matrizadj.h
	gcc -c grafo_matrizadj.c

clean:
	rm -f *.o *.exe



testa_lista: grafo_listaadj.o testa_grafo.o
	gcc -o testa_grafo_lista.exe grafo_listaadj.o testa_grafo.o


grafo_listaadj.o: grafo_listaadj.c grafo_listaadj.h
	gcc -c grafo_listaadj.c

testa_grafo.o: testa_grafo.c grafo_matrizadj.h  grafo_listaadj.h
	gcc -c testa_grafo.c
```

Exercícios

1) Implementar a estrutura e operações de grafos utilizando lista de adjacências para:

   - grafos direcionados
   - grafos não direcionados

2) Você pode implementar outras operações que julgar relevantes também, como por exemplo obtemPeso

3) Capriche no testa_grafo.c! Inclua a leitura de um grafo (prox slide)

```
/*
  LeGrafo(nomearq, Grafo)
  Le o arquivo nomearq e armazena na estrutura Grafo
  Layout do arquivo:
    A 1a linha deve conter o número de vertices e o numero de arestas do grafo,
    separados por espaço.
    A 2a linha em diante deve conter a informacao de cada aresta, que consiste
    no indice do vertice de origem, indice do vertice de destino e o peso da
    aresta, tambem separados por espacos.
    Observações:
      Os vertices devem ser indexados de 0 a |V|-1
      Os pesos das arestas sao numeros racionais nao negativos.

  Exemplo: O arquivo abaixo contem um grafo com 4 vertices (0,1,2,3) e
  7 arestas.

  4 7
  0 3 6.3
  2 1 5.0
  2 0 9
  1 3 1.7
  0 1 9
  3 1 5.6
  0 2 7.2

  Codigo de saida:
    1: leitura bem sucedida
    0: erro na leitura do arquivo
*/
int leGrafo(char* nomearq, Grafo *grafo) {
```

```c
int leGrafo(char* nomearq, Grafo *grafo) {
  FILE *fp;
  int nvertices, narestas;
  int v1, v2;
  Peso peso;
  int idar;

  fp = fopen(nomearq, "r");
  if (fp==NULL)
     return(0);

  if (fscanf(fp, "%d %d", &nvertices, &narestas)!=2)
    return(0);

  inicializaGrafo(grafo, nvertices);
```

Coloque aqui antes uma msg de erro

Coloque aqui antes uma msg de erro

(...)

Profa. Ariane Machado Lima