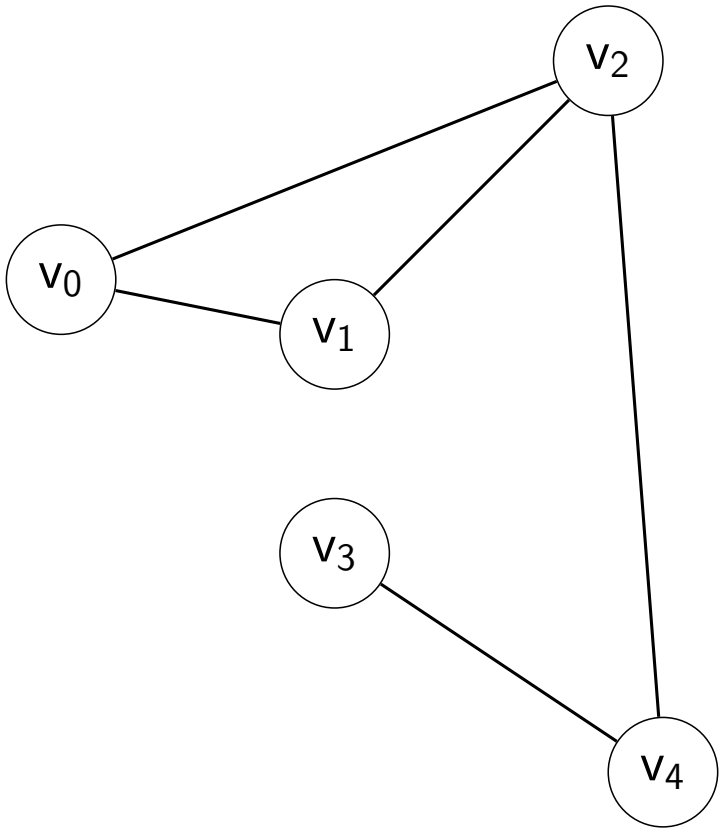
Algoritmos e Estruturas de Dados II
Aula 05 – Grafos: Listas de Adjacências

Prof. Luciano A. Digiampietri
digiampietri@usp.br
@digiampietri

Grafos – Relembrando

- São representados como um conjunto de nós (vértices) conectados par a par por linhas (arestas)



Grafos - Relembrando

Podem ser representados utilizando:

- Matrizes de Adjacências
- **Listas de Adjacências**

Listas de Adjacências

Definiremos as estruturas de dados para representar um grafo, bem como algoritmos para:

Listas de Adjacências

Definiremos as estruturas de dados para representar um grafo, bem como algoritmos para:

- Inicializar um Grafo
- Imprimir um Grafo
- Liberar a Memória de um Grafo
- Inserir uma Aresta
- Remover uma Aresta
- Verificar se uma Aresta Existe
- Retornar o Número de Vértices
- Retornar o Número de Arestas
- Verifica se um Vértice Possui Vizinhos
- Retornar o Grau de um Vértice

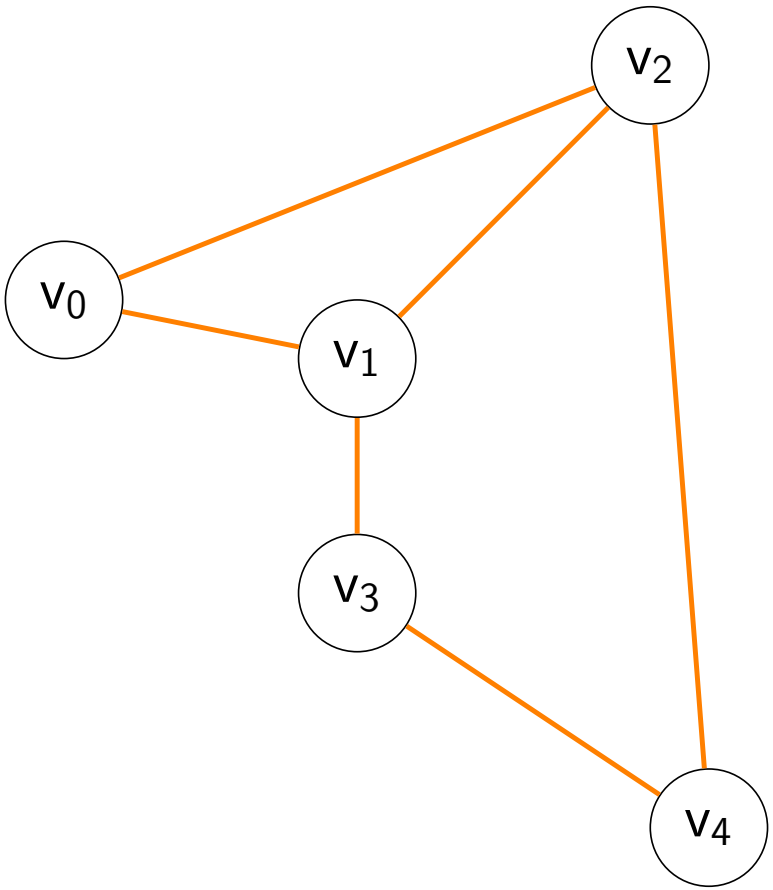
Grafos – Listas de Adjacências

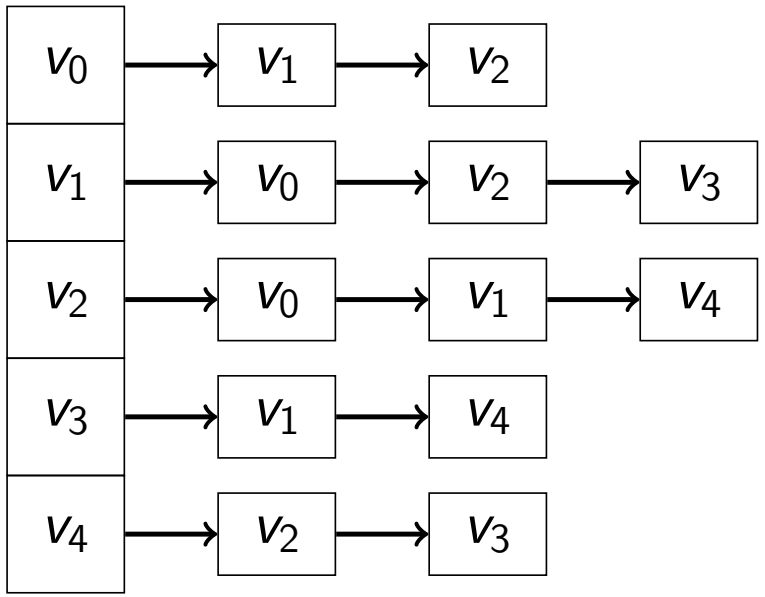
- A representação usando listas de adjacências de um grafo com n vértices consiste de um arranjo de n listas ligadas, uma para cada vértice no grafo.

Grafos – Listas de Adjacências

- A representação usando listas de adjacências de um grafo com n vértices consiste de um arranjo de n listas ligadas, uma para cada vértice no grafo.
  - Para cada vértice u, a lista contém todos os vizinhos de u
  - Ou seja, todos os vértices v_i para os quais existe uma aresta (u, v_i)

Grafos – Listas de Adjacências





Representação

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

#define true 1
#define false 0

typedef int bool;
```

```
typedef struct aux{
  int vertice;
  struct aux* prox;
} ElemLista, *PONT;

typedef struct {
  int numVertices;
  int numArestas;
  ElemLista** A;
} Grafo;
```

Representação

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

#define true 1
#define false 0

typedef int bool;
```

```
typedef struct aux{
  int vertice;
  struct aux* prox;
} ElemLista, *PONT;

typedef struct {
  int numVertices;
  int numArestas;
  ElemLista** A;
} Grafo;
```

Representação

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

#define true 1
#define false 0

typedef int bool;
```

```
typedef struct aux{
  int vertice;
  struct aux* prox;
} ElemLista, *PONT;

typedef struct {
  int numVertices;
  int numArestas;
  ElemLista** A;
} Grafo;
```

Representação

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

#define true 1
#define false 0

typedef int bool;
```

```
typedef struct aux{
   int vertice;
   struct aux* prox;
} ElemLista, *PONT;

typedef struct {
   int numVertices;
   int numArestas;
   ElemLista** A;
} Grafo;
```

Representação

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

#define true 1
#define false 0

typedef int bool;
```

```
typedef struct aux{
   int vertice;
   struct aux* prox;
} ElemLista, *PONT;

typedef struct {
   int numVertices;
   int numArestas;
   ElemLista** A;
} Grafo;
```

Representação

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

#define true 1
#define false 0

typedef int bool;
```

```
typedef struct aux{
  int vertice;
  struct aux* prox;
} ElemLista, *PONT;

typedef struct {
  int numVertices;
  int numArestas;
  ElemLista** A;
} Grafo;
```

Inicialização

```
bool inicializaGrafo(Grafo* g, int vertices){



}
```

Inicialização

```
bool inicializaGrafo(Grafo* g, int vertices){
  if (g==NULL || vertices<1) return false;




}
```

Inicialização

```
bool inicializaGrafo(Grafo* g, int vertices){
  if (g==NULL || vertices<1) return false;
  g->numVertices = vertices;
  g->numArestas = 0;




}
```

Inicialização

```
bool inicializaGrafo(Grafo* g, int vertices){
  if (g==NULL || vertices<1) return false;
  g->numVertices = vertices;
  g->numArestas = 0;
  g->A=(ElemLista**)malloc(sizeof(ElemLista*)*vertices);




}
```

Inicialização

```
bool inicializaGrafo(Grafo* g, int vertices){
  if (g==NULL || vertices<1) return false;
  g->numVertices = vertices;
  g->numArestas = 0;
  g->A=(ElemLista**)malloc(sizeof(ElemLista*)*vertices);
  int x;
  for (x=0; x<vertices; x++){

  }

}
```

Inicialização

```
bool inicializaGrafo(Grafo* g, int vertices){
  if (g==NULL || vertices<1) return false;
  g->numVertices = vertices;
  g->numArestas = 0;
  g->A=(ElemLista**)malloc(sizeof(ElemLista*)*vertices);
  int x;
  for (x=0; x<vertices; x++){
    g->A[x] = NULL;
  }

}
```

Inicialização

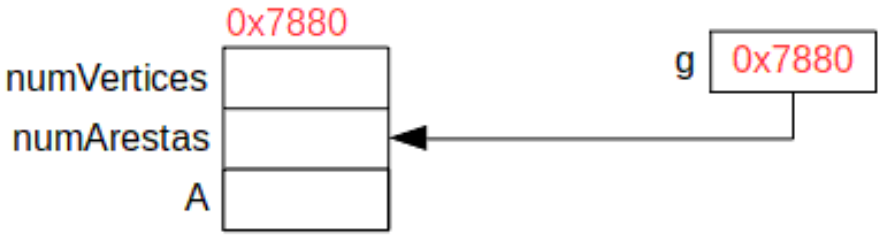
```
bool inicializaGrafo(Grafo* g, int vertices){
  if (g==NULL || vertices<1) return false;
  g->numVertices = vertices;
  g->numArestas = 0;
  g->A=(ElemLista**)malloc(sizeof(ElemLista*)*vertices);
  int x;
  for (x=0; x<vertices; x++){
    g->A[x] = NULL;
  }
  return true;
}
```

Inicialização

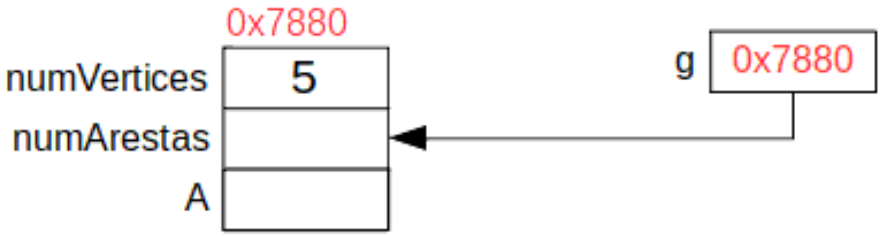
```
bool inicializaGrafo(Grafo* g, int vertices){
  if (g==NULL || vertices<1) return false;
  g->numVertices = vertices;
  g->numArestas = 0;
  g->A=(ElemLista**)malloc(sizeof(ElemLista*)*vertices);
  int x;
  for (x=0; x<vertices; x++){
    g->A[x] = NULL;
  }
  return true;
}
```

$O(V)$

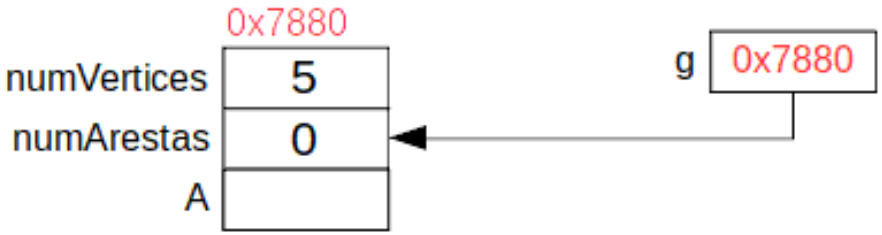
Inicialização



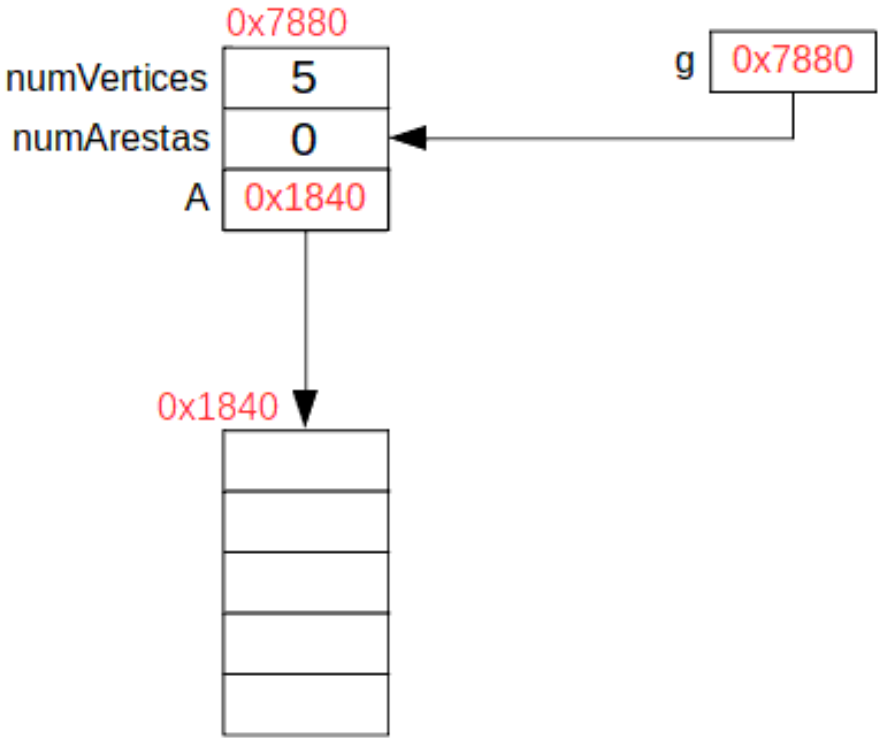
Inicialização



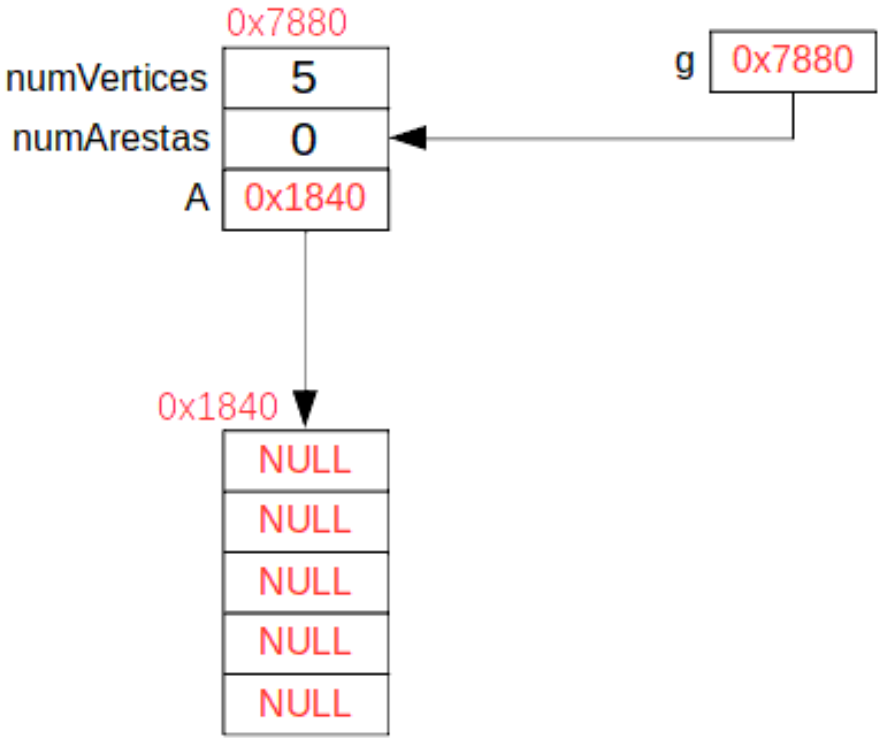
Inicialização



Inicialização



Inicialização



Imprimindo um Grafo

```
void exibeGrafo(Grafo* g){
```

Imprimindo um Grafo

```
void exibeGrafo(Grafo* g){
  if (!g) return;
```

Imprimindo um Grafo

```
void exibeGrafo(Grafo* g){
  if (!g) return;
  printf("\nImprimindo grafo
          (vertices: %i; arestas: %i).\n",
          g->numVertices, g->numArestas);
```

Imprimindo um Grafo

```
void exibeGrafo(Grafo* g){
  if (!g) return;
  printf("\nImprimindo grafo
          (vertices: %i; arestas: %i).\n",
          g->numVertices, g->numArestas);
  ElemLista* atual;
```

Imprimindo um Grafo

```
void exibeGrafo(Grafo* g){
  if (!g) return;
  printf("\nImprimindo grafo
          (vertices: %i; arestas: %i).\n",
          g->numVertices, g->numArestas);
  ElemLista* atual;
  int x;
  for (x=0;x<g->numVertices;x++){









  }
```

Imprimindo um Grafo

```
void exibeGrafo(Grafo* g){
  if (!g) return;
  printf("\nImprimindo grafo
          (vertices: %i; arestas: %i).\n",
          g->numVertices, g->numArestas);
  ElemLista* atual;
  int x;
  for (x=0;x<g->numVertices;x++){
    printf("[%2i]",x);
    atual = g->A[x];









  }
```

Imprimindo um Grafo

```
void exibeGrafo(Grafo* g){
  if (!g) return;
  printf("\nImprimindo grafo
          (vertices: %i; arestas: %i).\n",
          g->numVertices, g->numArestas);
  ElemLista* atual;
  int x;
  for (x=0;x<g->numVertices;x++){
    printf("[%2i]",x);
    atual = g->A[x];
    while (atual){



    }

  }
```

Imprimindo um Grafo

```
void exibeGrafo(Grafo* g){
  if (!g) return;
  printf("\nImprimindo grafo
          (vertices: %i; arestas: %i).\n",
          g->numVertices, g->numArestas);
  ElemLista* atual;
  int x;
  for (x=0;x<g->numVertices;x++){
    printf("[%2i]",x);
    atual = g->A[x];
    while (atual){
      printf(" ->%3i ",atual->vertice);
      atual = atual->prox;
    }

  }
```

Imprimindo um Grafo

```
void exibeGrafo(Grafo* g){
  if (!g) return;
  printf("\nImprimindo grafo
          (vertices: %i; arestas: %i).\n",
          g->numVertices, g->numArestas);
  ElemLista* atual;
  int x;
  for (x=0;x<g->numVertices;x++){
    printf("[%2i]",x);
    atual = g->A[x];
    while (atual){
      printf(" ->%3i ",atual->vertice);
      atual = atual->prox;
    }
    printf("\n");
  }
```

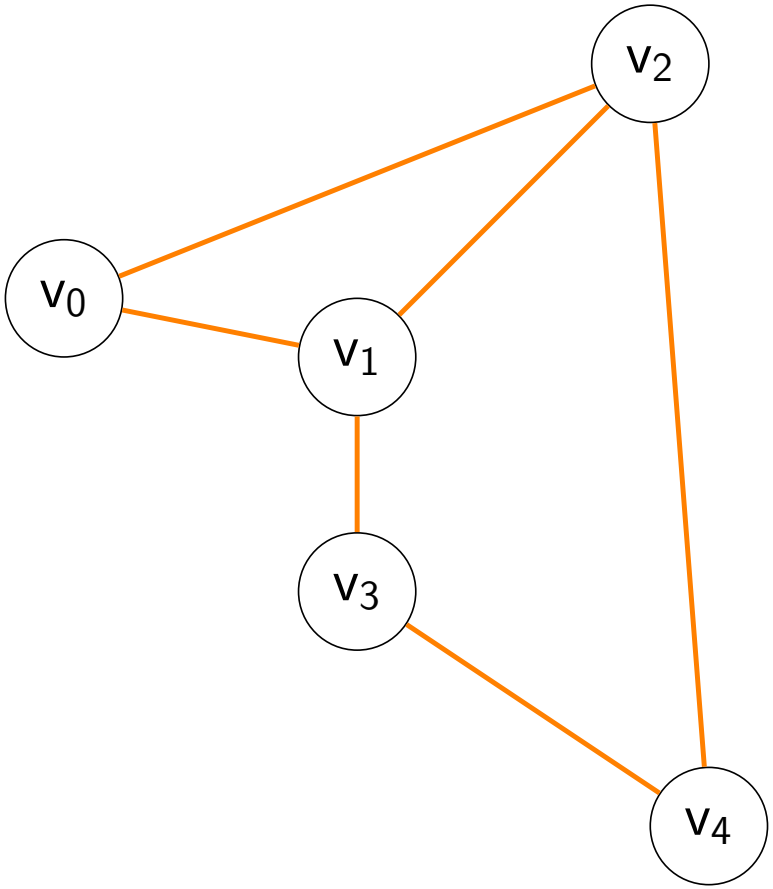
Imprimindo um Grafo

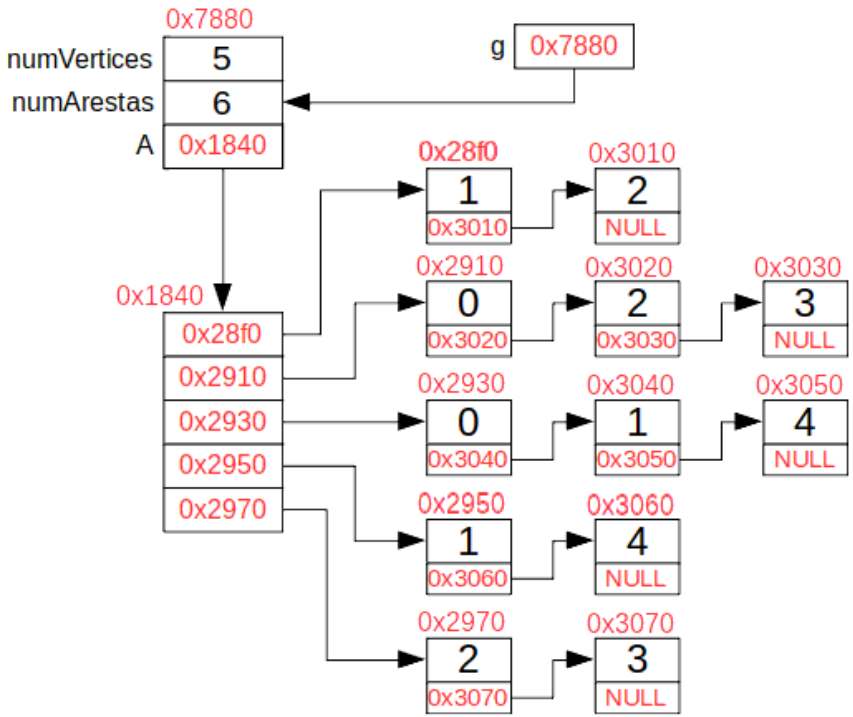
```
void exibeGrafo(Grafo* g){
  if (!g) return;
  printf("\nImprimindo grafo
          (vertices: %i; arestas: %i).\n",
          g->numVertices, g->numArestas);
  ElemLista* atual;
  int x;
  for (x=0;x<g->numVertices;x++){
    printf("[%2i]",x);
    atual = g->A[x];
    while (atual){
      printf(" ->%3i ",atual->vertice);
      atual = atual->prox;
    }
    printf("\n");
  }
```

$$O(V) + O(E)$$

Imprimindo um Grafo

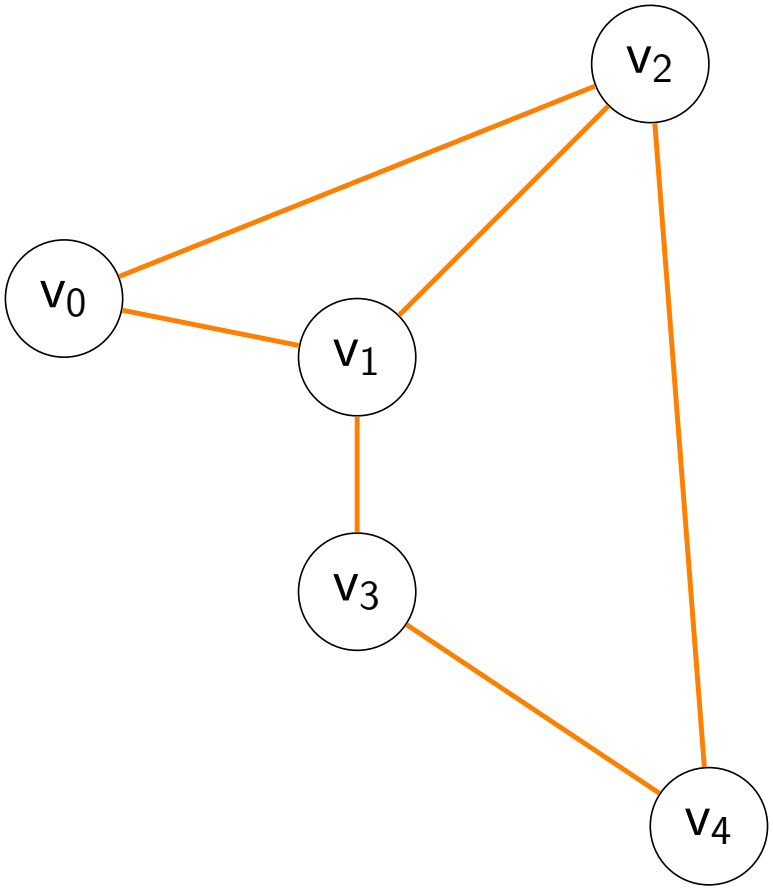
Grafo não ponderado e não dirigido:





Imprimindo um Grafo

Grafo não ponderado e não dirigido:




```
Imprimindo grafo (vertices: 5; arestas: 6).
[ 0] ->  1  ->  2
[ 1] ->  0  ->  2  ->  3
[ 2] ->  0  ->  1  ->  4
[ 3] ->  1  ->  4
[ 4] ->  2  ->  3
```

Liberando a Memória de um Grafo

```
bool liberaGrafo(Grafo* g){

}
```

Liberando a Memória de um Grafo

```
bool liberaGrafo(Grafo* g){
  if (g==NULL) return false;




}
```

Liberando a Memória de um Grafo

```
bool liberaGrafo(Grafo* g){
  if (g==NULL) return false;
  ElemLista *atual, *apagar;




}
```

Liberando a Memória de um Grafo

```
bool liberaGrafo(Grafo* g){
  if (g==NULL) return false;
  ElemLista *atual, *apagar;
  int x;
  for (x=0; x<g->numVertices; x++){




  }




}
```

Liberando a Memória de um Grafo

```
bool liberaGrafo(Grafo* g){
  if (g==NULL) return false;
  ElemLista *atual, *apagar;
  int x;
  for (x=0; x<g->numVertices; x++){
    atual = g->A[x];









  }









}
```

Liberando a Memória de um Grafo

```
bool liberaGrafo(Grafo* g){
  if (g==NULL) return false;
  ElemLista *atual, *apagar;
  int x;
  for (x=0; x<g->numVertices; x++){
    atual = g->A[x];
    while (atual){



    }
  }



}
```

Liberando a Memória de um Grafo

```
bool liberaGrafo(Grafo* g){
  if (g==NULL) return false;
  ElemLista *atual, *apagar;
  int x;
  for (x=0; x<g->numVertices; x++){
    atual = g->A[x];
    while (atual){
      apagar = atual;
      atual = atual->prox;
      free(apagar);
    }
  }




}
```

Liberando a Memória de um Grafo

```
bool liberaGrafo(Grafo* g){
  if (g==NULL) return false;
  ElemLista *atual, *apagar;
  int x;
  for (x=0; x<g->numVertices; x++){
    atual = g->A[x];
    while (atual){
      apagar = atual;
      atual = atual->prox;
      free(apagar);
    }
  }
  free(g->A);



}
```

Liberando a Memória de um Grafo

```
bool liberaGrafo(Grafo* g){
  if (g==NULL) return false;
  ElemLista *atual, *apagar;
  int x;
  for (x=0; x<g->numVertices; x++){
    atual = g->A[x];
    while (atual){
      apagar = atual;
      atual = atual->prox;
      free(apagar);
    }
  }
  free(g->A);
  g->numVertices = 0;
  g->numArestas = 0;
  g->A = NULL;

}
```

Liberando a Memória de um Grafo

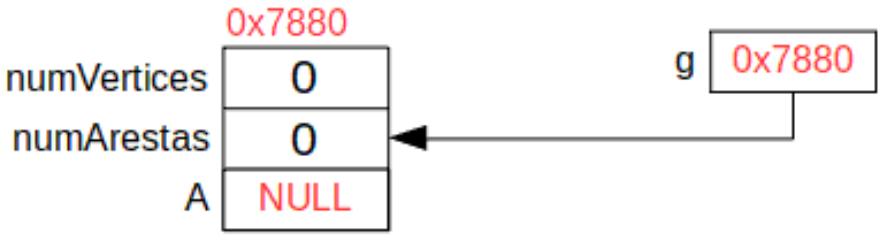
```
bool liberaGrafo(Grafo* g){
  if (g==NULL) return false;
  ElemLista *atual, *apagar;
  int x;
  for (x=0; x<g->numVertices; x++){
    atual = g->A[x];
    while (atual){
      apagar = atual;
      atual = atual->prox;
      free(apagar);
    }
  }
  free(g->A);
  g->numVertices = 0;
  g->numArestas = 0;
  g->A = NULL;
  return true;
}
```

Liberando a Memória de um Grafo

```
bool liberaGrafo(Grafo* g){
  if (g==NULL) return false;
  ElemLista *atual, *apagar;
  int x;
  for (x=0; x<g->numVertices; x++){
    atual = g->A[x];
    while (atual){
      apagar = atual;
      atual = atual->prox;
      free(apagar);
    }
  }
  free(g->A);
  g->numVertices = 0;
  g->numArestas = 0;
  g->A = NULL;
  return true;
}
```

$$O(V) + O(E)$$

Liberando a Memória de um Grafo



Inserindo uma Aresta

```
bool insereAresta(Grafo* g, int v1, int v2){




}
```

Inserindo uma Aresta

```
bool insereAresta(Grafo* g, int v1, int v2){
  if (!g || v1 < 0 || v2 < 0 ||
      v1 >= g->numVertices || v2 >= g->numVertices ||
      v1 == v2) return false;









}
```

Inserindo uma Aresta

```
bool insereAresta(Grafo* g, int v1, int v2){
  if (!g || v1 < 0 || v2 < 0 ||
      v1 >= g->numVertices || v2 >= g->numVertices ||
      v1 == v2) return false;
  if (insereArestaAux(g, v1, v2)){


  }

}
```

Inserindo uma Aresta

```
bool insereAresta(Grafo* g, int v1, int v2){
  if (!g || v1 < 0 || v2 < 0 ||
      v1 >= g->numVertices || v2 >= g->numVertices ||
      v1 == v2) return false;
  if (insereArestaAux(g, v1, v2)){
    insereArestaAux(g, v2, v1);

  }

}
```

Inserindo uma Aresta

```
bool insereAresta(Grafo* g, int v1, int v2){
  if (!g || v1 < 0 || v2 < 0 ||
      v1 >= g->numVertices || v2 >= g->numVertices ||
      v1 == v2) return false;
  if (insereArestaAux(g, v1, v2)){
    insereArestaAux(g, v2, v1);
    g->numArestas++;
  }

}
```

Inserindo uma Aresta

```
bool insereAresta(Grafo* g, int v1, int v2){
  if (!g || v1 < 0 || v2 < 0 ||
      v1 >= g->numVertices || v2 >= g->numVertices ||
      v1 == v2) return false;
  if (insereArestaAux(g, v1, v2)){
    insereArestaAux(g, v2, v1);
    g->numArestas++;
  }
  return true;
}
```

Inserindo uma Aresta

```
bool insereAresta(Grafo* g, int v1, int v2){
  if (!g || v1 < 0 || v2 < 0 ||
      v1 >= g->numVertices || v2 >= g->numVertices ||
      v1 == v2) return false;
  if (insereArestaAux(g, v1, v2)){
    insereArestaAux(g, v2, v1);
    g->numArestas++;
  }
  return true;
}
```

$$O(1)+O(insereArestaAux)$$

Inserindo uma Aresta

```
bool insereArestaAux(Grafo* g, int v1, int v2){

}
```

Inserindo uma Aresta

```
bool insereArestaAux(Grafo* g, int v1, int v2){
  ElemLista *novo, *ant = NULL;
  ElemLista* atual = g->A[v1];




}
```

Inserindo uma Aresta

```
bool insereArestaAux(Grafo* g, int v1, int v2){
  ElemLista *novo, *ant = NULL;
  ElemLista* atual = g->A[v1];
  while (atual && atual->vertice<v2){


  }









}
```

Inserindo uma Aresta

```
bool insereArestaAux(Grafo* g, int v1, int v2){
  ElemLista *novo, *ant = NULL;
  ElemLista* atual = g->A[v1];
  while (atual && atual->vertice<v2){
    ant = atual;
    atual = atual->prox;
  }




}
```

Inserindo uma Aresta

```
bool insereArestaAux(Grafo* g, int v1, int v2){
  ElemLista *novo, *ant = NULL;
  ElemLista* atual = g->A[v1];
  while (atual && atual->vertice<v2){
    ant = atual;
    atual = atual->prox;
  }
  if (atual && atual->vertice==v2) return false;









}
```

Inserindo uma Aresta

```
bool insereArestaAux(Grafo* g, int v1, int v2){
  ElemLista *novo, *ant = NULL;
  ElemLista* atual = g->A[v1];
  while (atual && atual->vertice<v2){
    ant = atual;
    atual = atual->prox;
  }
  if (atual && atual->vertice==v2) return false;
  novo = (ElemLista*)malloc(sizeof(ElemLista));
  novo->vertice = v2;
  novo->prox = atual;



}
```

Inserindo uma Aresta

```
bool insereArestaAux(Grafo* g, int v1, int v2){
  ElemLista *novo, *ant = NULL;
  ElemLista* atual = g->A[v1];
  while (atual && atual->vertice<v2){
    ant = atual;
    atual = atual->prox;
  }
  if (atual && atual->vertice==v2) return false;
  novo = (ElemLista*)malloc(sizeof(ElemLista));
  novo->vertice = v2;
  novo->prox = atual;
  if (ant) ant->prox = novo;


}
```

Inserindo uma Aresta

```
bool insereArestaAux(Grafo* g, int v1, int v2){
  ElemLista *novo, *ant = NULL;
  ElemLista* atual = g->A[v1];
  while (atual && atual->vertice<v2){
    ant = atual;
    atual = atual->prox;
  }
  if (atual && atual->vertice==v2) return false;
  novo = (ElemLista*)malloc(sizeof(ElemLista));
  novo->vertice = v2;
  novo->prox = atual;
  if (ant) ant->prox = novo;
  else g->A[v1] = novo;

}
```

Inserindo uma Aresta

```
bool insereArestaAux(Grafo* g, int v1, int v2){
  ElemLista *novo, *ant = NULL;
  ElemLista* atual = g->A[v1];
  while (atual && atual->vertice<v2){
    ant = atual;
    atual = atual->prox;
  }
  if (atual && atual->vertice==v2) return false;
  novo = (ElemLista*)malloc(sizeof(ElemLista));
  novo->vertice = v2;
  novo->prox = atual;
  if (ant) ant->prox = novo;
  else g->A[v1] = novo;
  return true;
}
```

Inserindo uma Aresta

```
bool insereArestaAux(Grafo* g, int v1, int v2){
  ElemLista *novo, *ant = NULL;
  ElemLista* atual = g->A[v1];
  while (atual && atual->vertice<v2){
    ant = atual;
    atual = atual->prox;
  }
  if (atual && atual->vertice==v2) return false;
  novo = (ElemLista*)malloc(sizeof(ElemLista));
  novo->vertice = v2;
  novo->prox = atual;
  if (ant) ant->prox = novo;
  else g->A[v1] = novo;
  return true;
}
```

$O(adj(v1))$

Removendo uma Aresta

```
bool removeAresta(Grafo* g, int v1, int v2){




}
```

Removendo uma Aresta

```
bool removeAresta(Grafo* g, int v1, int v2){
  if (!g || v1 < 0 || v2 < 0 ||
      v1 >= g->numVertices || v2 >= g->numVertices) return false;









}
```

Removendo uma Aresta

```
bool removeAresta(Grafo* g, int v1, int v2){
  if (!g || v1 < 0 || v2 < 0 ||
      v1 >= g->numVertices || v2 >= g->numVertices) return false;
  if(removeArestaAux(g, v1, v2)){
    removeArestaAux(g, v2, v1);
    g->numArestas--;
    return true;
  }

}
```

Removendo uma Aresta

```
bool removeAresta(Grafo* g, int v1, int v2){
  if (!g || v1 < 0 || v2 < 0 ||
      v1 >= g->numVertices || v2 >= g->numVertices) return false;
  if(removeArestaAux(g, v1, v2)){
    removeArestaAux(g, v2, v1);
    g->numArestas--;
    return true;
  }
  return false;
}
```

Removendo uma Aresta

```
bool removeAresta(Grafo* g, int v1, int v2){
  if (!g || v1 < 0 || v2 < 0 ||
      v1 >= g->numVertices || v2 >= g->numVertices) return false;
  if(removeArestaAux(g, v1, v2)){
    removeArestaAux(g, v2, v1);
    g->numArestas--;
    return true;
  }
  return false;
}
```

$$O(1) + O(removeArestaAux)$$

Removendo uma Aresta

```
bool removeArestaAux(Grafo* g, int v1, int v2){




}
```

Removendo uma Aresta

```
bool removeArestaAux(Grafo* g, int v1, int v2){
  ElemLista* ant = NULL;
  ElemLista* atual = g->A[v1];




}
```

Removendo uma Aresta

```
bool removeArestaAux(Grafo* g, int v1, int v2){
  ElemLista* ant = NULL;
  ElemLista* atual = g->A[v1];
  while (atual && atual->vertice<v2){


  }









}
```

Removendo uma Aresta

```
bool removeArestaAux(Grafo* g, int v1, int v2){
  ElemLista* ant = NULL;
  ElemLista* atual = g->A[v1];
  while (atual && atual->vertice<v2){
    ant = atual;
    atual = atual->prox;
  }









}
```

Removendo uma Aresta

```
bool removeArestaAux(Grafo* g, int v1, int v2){
  ElemLista* ant = NULL;
  ElemLista* atual = g->A[v1];
  while (atual && atual->vertice<v2){
    ant = atual;
    atual = atual->prox;
  }
  if (atual && atual->vertice==v2){









  }

}
```

Removendo uma Aresta

```
bool removeArestaAux(Grafo* g, int v1, int v2){
  ElemLista* ant = NULL;
  ElemLista* atual = g->A[v1];
  while (atual && atual->vertice<v2){
    ant = atual;
    atual = atual->prox;
  }
  if (atual && atual->vertice==v2){
    if (ant) ant->prox = atual->prox;



  }

}
```

Removendo uma Aresta

```
bool removeArestaAux(Grafo* g, int v1, int v2){
  ElemLista* ant = NULL;
  ElemLista* atual = g->A[v1];
  while (atual && atual->vertice<v2){
    ant = atual;
    atual = atual->prox;
  }
  if (atual && atual->vertice==v2){
    if (ant) ant->prox = atual->prox;
    else g->A[v1] = atual->prox;



  }

}
```

Removendo uma Aresta

```
bool removeArestaAux(Grafo* g, int v1, int v2){
  ElemLista* ant = NULL;
  ElemLista* atual = g->A[v1];
  while (atual && atual->vertice<v2){
    ant = atual;
    atual = atual->prox;
  }
  if (atual && atual->vertice==v2){
    if (ant) ant->prox = atual->prox;
    else g->A[v1] = atual->prox;
    free(atual);

  }

}
```

Removendo uma Aresta

```
bool removeArestaAux(Grafo* g, int v1, int v2){
  ElemLista* ant = NULL;
  ElemLista* atual = g->A[v1];
  while (atual && atual->vertice<v2){
    ant = atual;
    atual = atual->prox;
  }
  if (atual && atual->vertice==v2){
    if (ant) ant->prox = atual->prox;
    else g->A[v1] = atual->prox;
    free(atual);
    return true;
  }

}
```

Removendo uma Aresta

```
bool removeArestaAux(Grafo* g, int v1, int v2){
  ElemLista* ant = NULL;
  ElemLista* atual = g->A[v1];
  while (atual && atual->vertice<v2){
    ant = atual;
    atual = atual->prox;
  }
  if (atual && atual->vertice==v2){
    if (ant) ant->prox = atual->prox;
    else g->A[v1] = atual->prox;
    free(atual);
    return true;
  }
  return false;
}
```

Removendo uma Aresta

```
bool removeArestaAux(Grafo* g, int v1, int v2){
  ElemLista* ant = NULL;
  ElemLista* atual = g->A[v1];
  while (atual && atual->vertice<v2){
    ant = atual;
    atual = atual->prox;
  }
  if (atual && atual->vertice==v2){
    if (ant) ant->prox = atual->prox;
    else g->A[v1] = atual->prox;
    free(atual);
    return true;
  }
  return false;
}
```

$O(adj(v1))$

Verificando a Existência de uma Aresta

```
bool arestaExiste(Grafo* g, int v1, int v2){




}
```

Verificando a Existência de uma Aresta

```
bool arestaExiste(Grafo* g, int v1, int v2){
  if (!g || v1 < 0 || v2 < 0 ||
      v1 >= g->numVertices || v2 >= g->numVertices)
      return false;









}
```

Verificando a Existência de uma Aresta

```
bool arestaExiste(Grafo* g, int v1, int v2){
  if (!g || v1 < 0 || v2 < 0 ||
      v1 >= g->numVertices || v2 >= g->numVertices)
      return false;
  ElemLista* atual = g->A[v1];




}
```

Verificando a Existência de uma Aresta

```
bool arestaExiste(Grafo* g, int v1, int v2){
  if (!g || v1 < 0 || v2 < 0 ||
      v1 >= g->numVertices || v2 >= g->numVertices)
      return false;
  ElemLista* atual = g->A[v1];
  while (atual && atual->vertice < v2)



}
```

Verificando a Existência de uma Aresta

```
bool arestaExiste(Grafo* g, int v1, int v2){
  if (!g || v1 < 0 || v2 < 0 ||
      v1 >= g->numVertices || v2 >= g->numVertices)
      return false;
  ElemLista* atual = g->A[v1];
  while (atual && atual->vertice < v2)
    atual = atual->prox;


}
```

Verificando a Existência de uma Aresta

```
bool arestaExiste(Grafo* g, int v1, int v2){
  if (!g || v1 < 0 || v2 < 0 ||
      v1 >= g->numVertices || v2 >= g->numVertices)
      return false;
  ElemLista* atual = g->A[v1];
  while (atual && atual->vertice < v2)
    atual = atual->prox;
  if (atual && atual->vertice == v2) return true;

}
```

Verificando a Existência de uma Aresta

```
bool arestaExiste(Grafo* g, int v1, int v2){
  if (!g || v1 < 0 || v2 < 0 ||
      v1 >= g->numVertices || v2 >= g->numVertices)
      return false;
  ElemLista* atual = g->A[v1];
  while (atual && atual->vertice < v2)
    atual = atual->prox;
  if (atual && atual->vertice == v2) return true;
  return false;
}
```

Verificando a Existência de uma Aresta

```
bool arestaExiste(Grafo* g, int v1, int v2){
  if (!g || v1 < 0 || v2 < 0 ||
      v1 >= g->numVertices || v2 >= g->numVertices)
      return false;
  ElemLista* atual = g->A[v1];
  while (atual && atual->vertice < v2)
    atual = atual->prox;
  if (atual && atual->vertice == v2) return true;
  return false;
}
```

$O(adj(v1))$

Número de Vértices

```
int numeroDeVertices(Grafo* g){


}
```

Número de Vértices

```
int numeroDeVertices(Grafo* g){
  if (g!=NULL) return g->numVertices;

}
```

Número de Vértices

```
int numeroDeVertices(Grafo* g){
  if (g!=NULL) return g->numVertices;
  else return -1;
}
```

Número de Vértices

```
int numeroDeVertices(Grafo* g){
  if (g!=NULL) return g->numVertices;
  else return -1;
}
```

O(1)

Número de Aresta

```
int numeroDeArestas(Grafo* g){


}
```

Número de Aresta

```
int numeroDeArestas(Grafo* g){
  if (g!=NULL) return g->numArestas;

}
```

Número de Aresta

```
int numeroDeArestas(Grafo* g){
  if (g!=NULL) return g->numArestas;
  else return -1;
}
```

Número de Aresta

```
int numeroDeArestas(Grafo* g){
  if (g!=NULL) return g->numArestas;
  else return -1;
}

int numeroDeArestas2(Grafo* g){









}
```

Número de Aresta

```
int numeroDeArestas(Grafo* g){
  if (g!=NULL) return g->numArestas;
  else return -1;
}

int numeroDeArestas2(Grafo* g){
  if (g==NULL) return -1;









}
```

Número de Aresta

```
int numeroDeArestas(Grafo* g){
  if (g!=NULL) return g->numArestas;
  else return -1;
}

int numeroDeArestas2(Grafo* g){
  if (g==NULL) return -1;
  ElemLista *atual;
  int x, arestas = 0;









}
```

Número de Aresta

```
int numeroDeArestas(Grafo* g){
  if (g!=NULL) return g->numArestas;
  else return -1;
}

int numeroDeArestas2(Grafo* g){
  if (g==NULL) return -1;
  ElemLista *atual;
  int x, arestas = 0;
  for (x=0; x<g->numVertices; x++){









  }

}
```

Número de Aresta

```
int numeroDeArestas(Grafo* g){
  if (g!=NULL) return g->numArestas;
  else return -1;
}

int numeroDeArestas2(Grafo* g){
  if (g==NULL) return -1;
  ElemLista *atual;
  int x, arestas = 0;
  for (x=0; x<g->numVertices; x++){
    atual = g->A[x];









  }

}
```

Número de Aresta

```
int numeroDeArestas(Grafo* g){
  if (g!=NULL) return g->numArestas;
  else return -1;
}

int numeroDeArestas2(Grafo* g){
  if (g==NULL) return -1;
  ElemLista *atual;
  int x, arestas = 0;
  for (x=0; x<g->numVertices; x++){
    atual = g->A[x];
    while (atual){


    }
  }

}
```

Número de Aresta

```
int numeroDeArestas(Grafo* g){
  if (g!=NULL) return g->numArestas;
  else return -1;
}

int numeroDeArestas2(Grafo* g){
  if (g==NULL) return -1;
  ElemLista *atual;
  int x, arestas = 0;
  for (x=0; x<g->numVertices; x++){
    atual = g->A[x];
    while (atual){
      arestas++;
      atual = atual->prox;
    }
  }

}
```

Número de Aresta

```
int numeroDeArestas(Grafo* g){
  if (g!=NULL) return g->numArestas;
  else return -1;
}

int numeroDeArestas2(Grafo* g){
  if (g==NULL) return -1;
  ElemLista *atual;
  int x, arestas = 0;
  for (x=0; x<g->numVertices; x++){
    atual = g->A[x];
    while (atual){
      arestas++;
      atual = atual->prox;
    }
  }
  return arestas/2;
}
```

Número de Aresta

```
int numeroDeArestas(Grafo* g){
  if (g!=NULL) return g->numArestas;
  else return -1;
}
```

$O(1)$

```
int numeroDeArestas2(Grafo* g){
  if (g==NULL) return -1;
  ElemLista *atual;
  int x, arestas = 0;
  for (x=0; x<g->numVertices; x++){
    atual = g->A[x];
    while (atual){
      arestas++;
      atual = atual->prox;
    }
  }
  return arestas/2;
}
```

$O(V) + O(E)$

Verificar se um Vértice Possui Vizinhos

```
bool possuiVizinhos(Grafo* g, int v){


}
```

Verificar se um Vértice Possui Vizinhos

```
bool possuiVizinhos(Grafo* g, int v){
  if (!g || v < 0 || v >= g->numVertices || !(g->A[v]))
      return false;

}
```

Verificar se um Vértice Possui Vizinhos

```
bool possuiVizinhos(Grafo* g, int v){
  if (!g || v < 0 || v >= g->numVertices || !(g->A[v]))
      return false;
  return true;
}
```

Verificar se um Vértice Possui Vizinhos

```
bool possuiVizinhos(Grafo* g, int v){
  if (!g || v < 0 || v >= g->numVertices || !(g->A[v]))
      return false;
  return true;
}
```

O(1)

Grau de um Vértice

```
int retornaGrauDoVertice(Grafo* g, int v){




}
```

Grau de um Vértice

```
int retornaGrauDoVertice(Grafo* g, int v){
  if (!g || v < 0 || v >= g->numVertices) return -1;









}
```

Grau de um Vértice

```
int retornaGrauDoVertice(Grafo* g, int v){
  if (!g || v < 0 || v >= g->numVertices) return -1;
  int grau = 0;
  ElemLista* atual = g->A[v];




}
```

Grau de um Vértice

```
int retornaGrauDoVertice(Grafo* g, int v){
  if (!g || v < 0 || v >= g->numVertices) return -1;
  int grau = 0;
  ElemLista* atual = g->A[v];
  while (atual){


  }

}
```

Grau de um Vértice

```
int retornaGrauDoVertice(Grafo* g, int v){
  if (!g || v < 0 || v >= g->numVertices) return -1;
  int grau = 0;
  ElemLista* atual = g->A[v];
  while (atual){
    grau++;
    atual = atual->prox;
  }

}
```

Grau de um Vértice

```
int retornaGrauDoVertice(Grafo* g, int v){
  if (!g || v < 0 || v >= g->numVertices) return -1;
  int grau = 0;
  ElemLista* atual = g->A[v];
  while (atual){
    grau++;
    atual = atual->prox;
  }
  return grau;
}
```

Grau de um Vértice

```
int retornaGrauDoVertice(Grafo* g, int v){
  if (!g || v < 0 || v >= g->numVertices) return -1;
  int grau = 0;
  ElemLista* atual = g->A[v];
  while (atual){
    grau++;
    atual = atual->prox;
  }
  return grau;
}
```

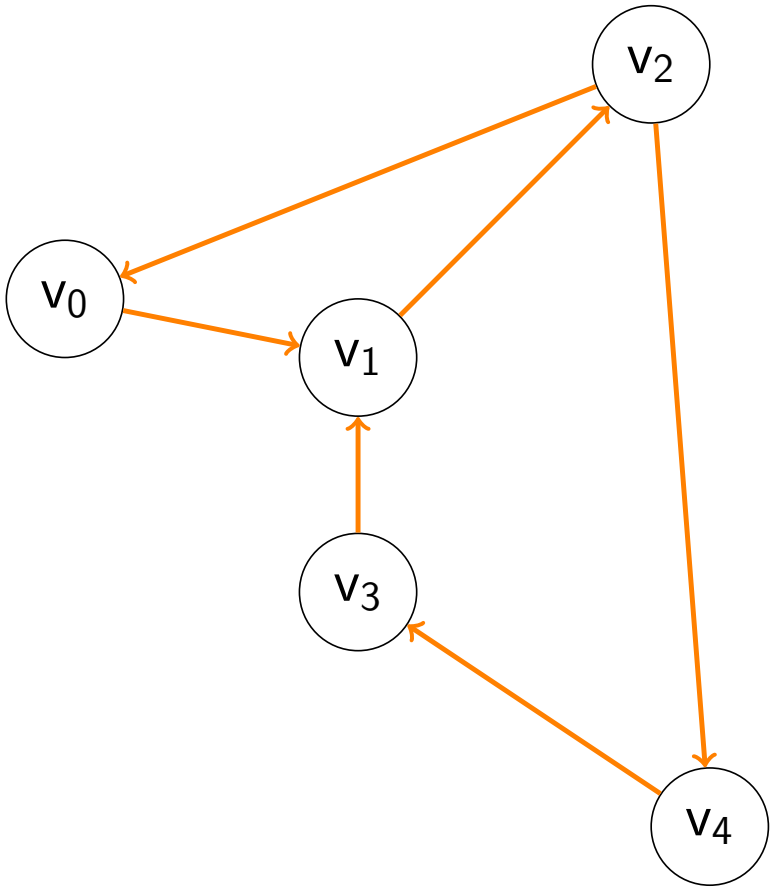
$O(adj(v))$

Grafos – Listas de Adjacências

- E se o grafo for dirigido?

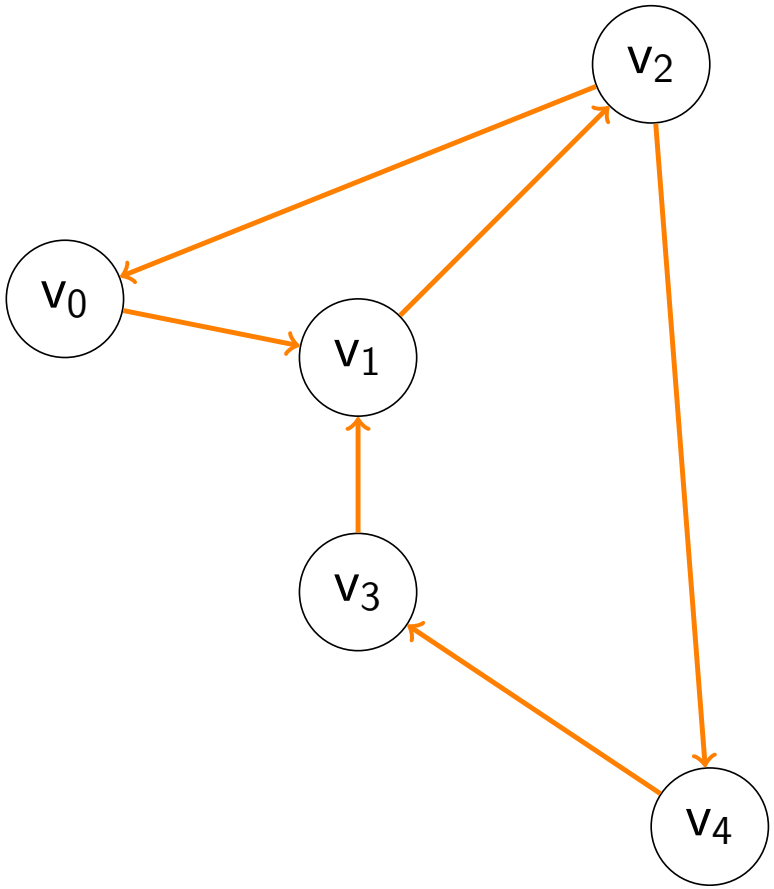
Grafos – Listas de Adjacências

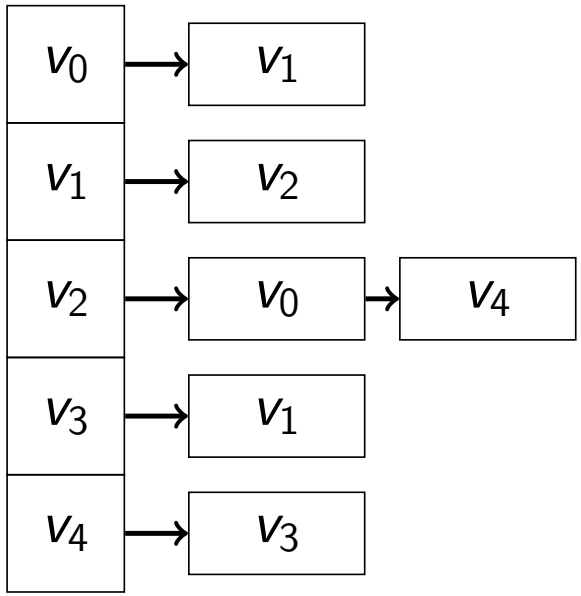
- E se o grafo for dirigido?



Grafos – Listas de Adjacências

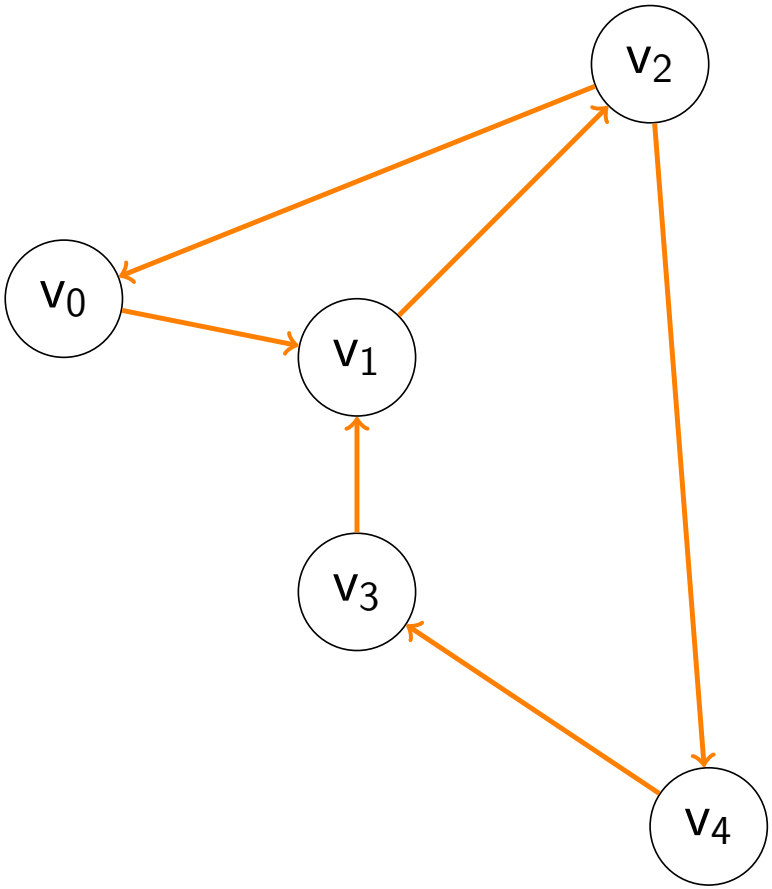
- E se o grafo for dirigido?

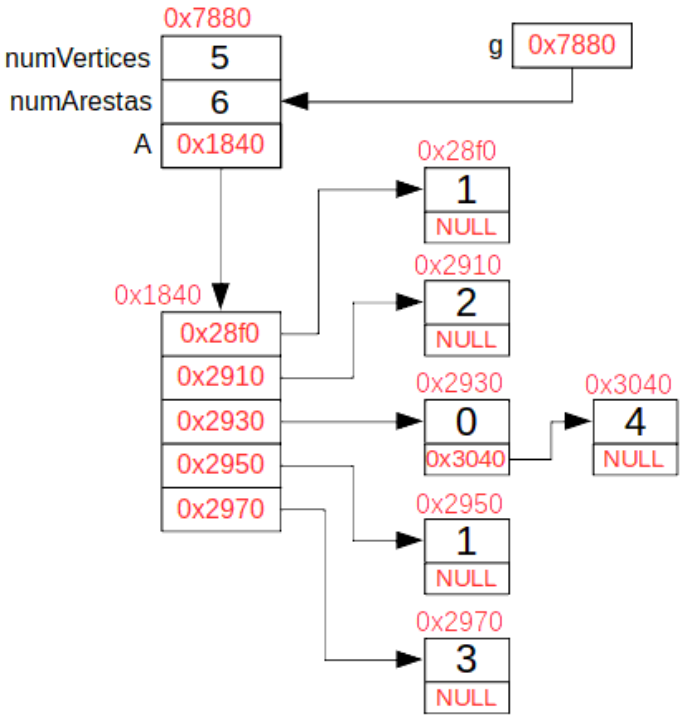




Grafos – Listas de Adjacências

- E se o grafo for dirigido?





Listas de Adjacências - Grafo Dirigido

A estrutura de dados será a mesma, haverá pequenas mudanças em algumas funções:

Listas de Adjacências - Grafo Dirigido

A estrutura de dados será a mesma, haverá pequenas mudanças em algumas funções:

- Inicializar um Grafo
- Imprimir um Grafo
- Liberar a Memória de um Grafo
- Inserir uma Aresta
- Remover uma Aresta
- Verificar se uma Aresta Existe
- Retornar o Número de Vértices
- Retornar o Número de Arestas*
- Verificar se um Vértice Possui Vizinhos
- Retornar o Grau de um Vértice

Inserindo uma Aresta - Digrafo

```
bool insereAresta(Grafo* g, int v1, int v2){
```

Inserindo uma Aresta - Digrafo

```
bool insereAresta(Grafo* g, int v1, int v2){
  if (!g || v1 < 0 || v2 < 0 ||
      v1 >= g->numVertices || v2 >= g->numVertices
      || v1 == v2) return false;
  ElemLista *novo, *ant = NULL;
  ElemLista* atual = g->A[v1];
  while (atual && atual->vertice<v2){
    ant = atual;
    atual = atual->prox;
  }
  if (atual && atual->vertice==v2) return true;
  novo = (ElemLista*)malloc(sizeof(ElemLista));
  novo->vertice = v2;
  novo->prox = atual;
  if (ant) ant->prox = novo;
  else g->A[v1] = novo;
  g->numArestas++;
  return true;
}
```

$O(adj(v1))$

Removendo uma Aresta - Digrafo

```
bool removeAresta(Grafo* g, int v1, int v2){
```

```
}
```

Removendo uma Aresta - Digrafo

```
bool removeAresta(Grafo* g, int v1, int v2){
  if (!g || v1 < 0 || v2 < 0 ||
      v1 >= g->numVertices || v2 >= g->numVertices) return false;
  ElemLista* ant = NULL;
  ElemLista* atual = g->A[v1];
  while (atual && atual->vertice<v2){
    ant = atual;
    atual = atual->prox;
  }
  if (atual && atual->vertice==v2){
    if (ant) ant->prox = atual->prox;
    else g->A[v1] = atual->prox;
    free(atual);
    g->numArestas--;
    return true;
  }
  return false;
}
```

$O(adj(v1))$

Número de Aresta - Digrafo

```
int numeroDeArestas(Grafo* g){
  if (g!=NULL) return g->numArestas;
  else return -1;
}

int numeroDeArestas2(Grafo* g){




}
```

O(1)

Número de Aresta - Digrafo

```
int numeroDeArestas(Grafo* g){
  if (g!=NULL) return g->numArestas;
  else return -1;
}
```

$O(1)$

```
int numeroDeArestas2(Grafo* g){
  if (g==NULL) return -1;
  ElemLista *atual;
  int x, arestas = 0;
  for (x=0; x<g->numVertices; x++){
    atual = g->A[x];
    while (atual){
      arestas++;
      atual = atual->prox;
    }
  }
  return arestas/2;
}
```

$O(V) + O(E)$

Grau de um Vértice - Digrafo

```
int retornaGrauDoVertice(Grafo* g, int v){



}
```

Grau de um Vértice - Digrafo

```
int retornaGrauDoVertice(Grafo* g, int v){
  if (!g || v < 0 || v >= g->numVertices) return -1;
  int x, grau = 0;
  for (x=0; x<g->numVertices; x++){
    ElemLista* atual = g->A[x];
    while (atual){
      if (x == v) grau++;
      if (atual->vertice == v) grau++;
      atual = atual->prox;
    }
  }
  return grau;
}
```

$$O(V) + O(E)$$

Grafos – Listas de Adjacências

E se o grafo for ponderado?

Grafos – Listas de Adjacências

E se o grafo for ponderado?

Trataremos isso na próxima aula!

Algoritmos e Estruturas de Dados II
Aula 05 – Grafos: Listas de Adjacências

Prof. Luciano A. Digiampietri
digiampietri@usp.br
@digiampietri