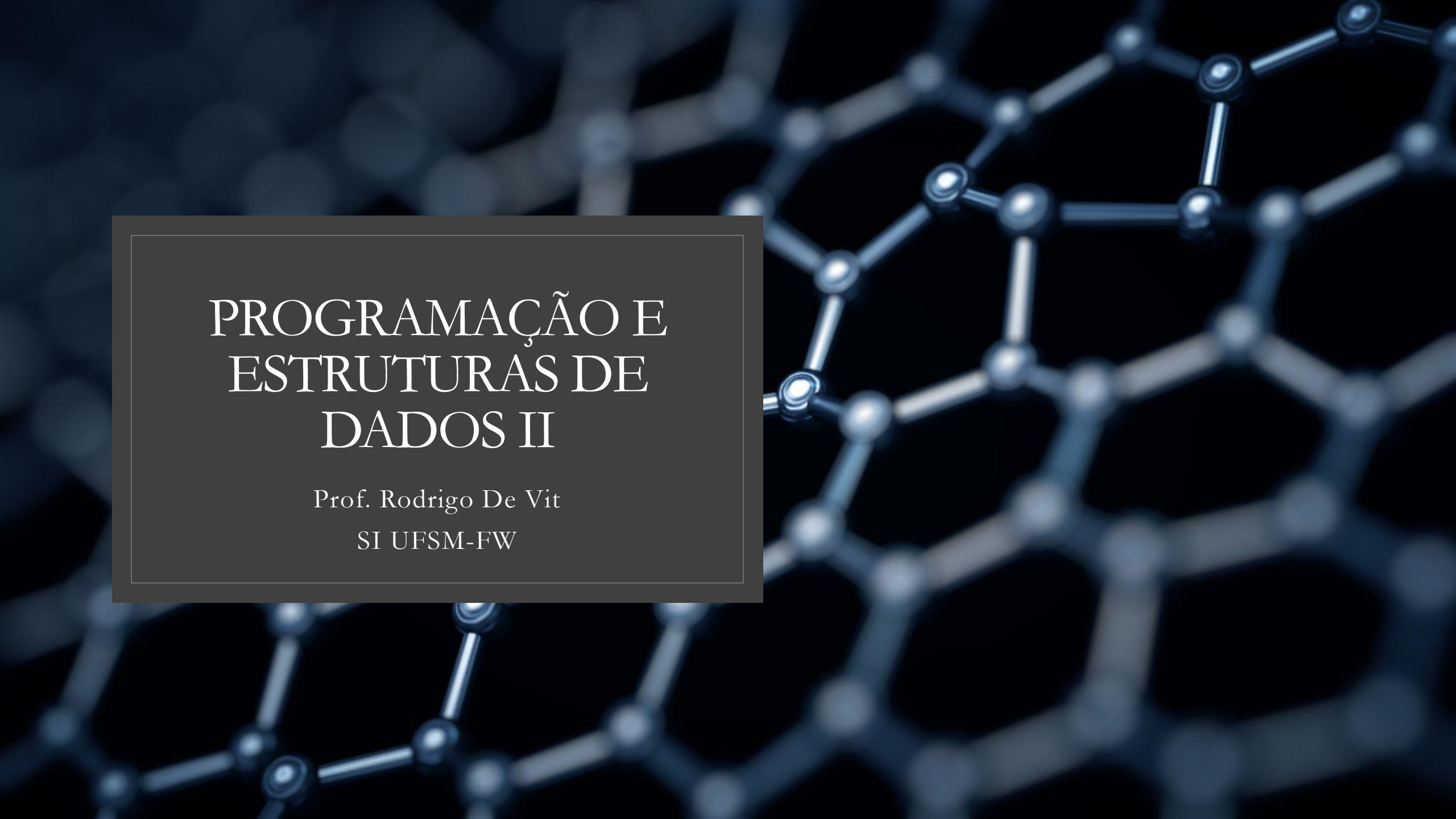


PROGRAMAÇÃO E ESTRUTURAS DE DADOS II

Prof. Rodrigo De Vit

SI UFSM-FW

Busca Binária - Revisão

- **No caso dos elementos do vetor estarem em ordem,** podemos aplicar um algoritmo mais eficiente para realizarmos a busca. Trata-se do algoritmo de ***busca binária*****. A ideia do algoritmo é testar o elemento que buscamos com o valor do elemento armazenado no meio do vetor.** Se o elemento que buscamos **for menor que o elemento do meio,** sabemos que, se o elemento estiver presente no vetor, **ele estará na primeira parte do vetor; se for maior, estará na segunda parte do vetor; se for igual, achamos o elemento no vetor.** Se concluirmos que o elemento está numa das partes do vetor, **repetimos o procedimento considerando apenas a parte que restou: comparamos o elemento que buscamos com o elemento armazenado no meio dessa parte. Este procedimento é continuamente repetido, subdividindo a parte de interesse, até encontrarmos o elemento ou chegarmos a uma parte do vetor com tamanho zero.**

```
int busca_bin (int n, int* vet, int elem)
{
   /* no inicio consideramos todo o vetor */
   int ini = 0;
   int fim = n-1;
   int meio;
   /* enquanto a parte restante for maior que zero */
   while (ini <= fim) {
   meio = (ini + fim) / 2;
   if (elem < vet[meio])
      fim = meio - 1; /* ajusta posicão final */
   else if (elem > vet[meio])
      ini = meio + 1; /* ajusta posicão inicial */
   else
      return meio; /* elemento encontrado */
}
   /* não encontrou: restou parte de tamanho zero */
   return -1;
}
```

bsearch() e qsort(), de <stdlib.h> - Revisão


```
/* bsearch example */
#include <stdio.h>      /* printf */
#include <stdlib.h>     /* qsort, bsearch, NULL */

int compareints (const void * a, const void * b)
{
  return ( *(int*)a - *(int*)b );
}

int values[] = { 50, 20, 60, 40, 10, 30 };
```

```
int main ()
{
  int * pItem;
  int key = 40;
  qsort (values, 6, sizeof (int), compareints);
  pItem = (int*) bsearch (&key, values, 6, sizeof (int), compareints);
  if (pItem!=NULL)
    printf ("%d is in the array.\n",*pItem);
  else
    printf ("%d is not in the array.\n",key);
  return 0;
}
```

Árvore binária de busca - Revisão

- O **algoritmo de busca binária** apresenta bom desempenho computacional e **deve ser usado quando temos os dados ordenados armazenados num vetor**. No entanto, se precisarmos **inserir e remover elementos da estrutura, e ao mesmo tempo dar suporte a eficientes funções de busca, a estrutura de vetor (e, consequentemente, o uso do algoritmo de busca binária) não se torna adequada.** Para **inserirmos um novo elemento num vetor ordenado, temos que rearrumar os elementos no vetor, para abrir espaço para a inserção do novo elemento.** Situação análoga ocorre quando removemos um elemento do vetor. **Precisamos portanto de uma estrutura dinâmica que dê suporte a operações de busca.**
- Deve-se verificar a relação entre o número de nós de uma árvore binária e sua altura. **A cada nível, o número (potencial) de nós vai dobrando**.
- Assim, dizemos que uma árvore binária de altura *h* pode ter no máximo *O(2h)* nós, ou, pelo outro lado, que uma árvore binária com *n* nós pode ter uma altura mínima de *O(log n)*. **Essa relação entre o número de nós e a altura mínima da árvore é importante porque se as condições forem favoráveis, podemos alcançar qualquer um dos *n* nós de uma árvore binária a partir da raiz em, no máximo, *O(log n)* passos.** Se tivéssemos os *n* nós **em uma lista linear, o número máximo de passos seria *O(n)*, e, para os valores de *n* encontrados na prática, *log n* é muito menor do que *n*.**

Árvore binária de busca - Revisão

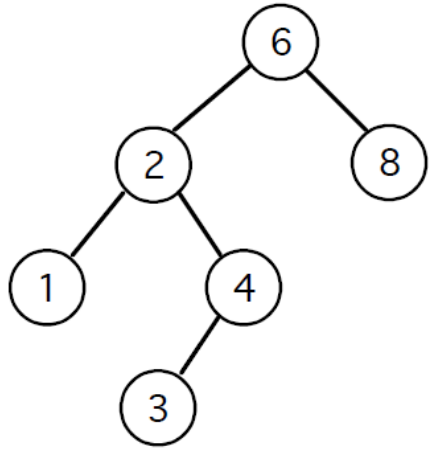


Figura 16.1: Exemplo de árvore binária de busca.

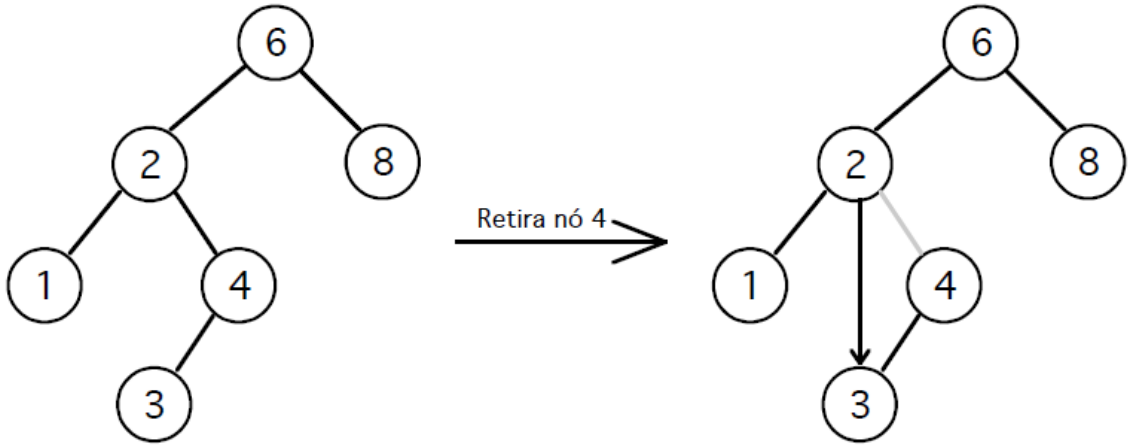


Figura 16.2: Retirada de um elemento com um único filho.

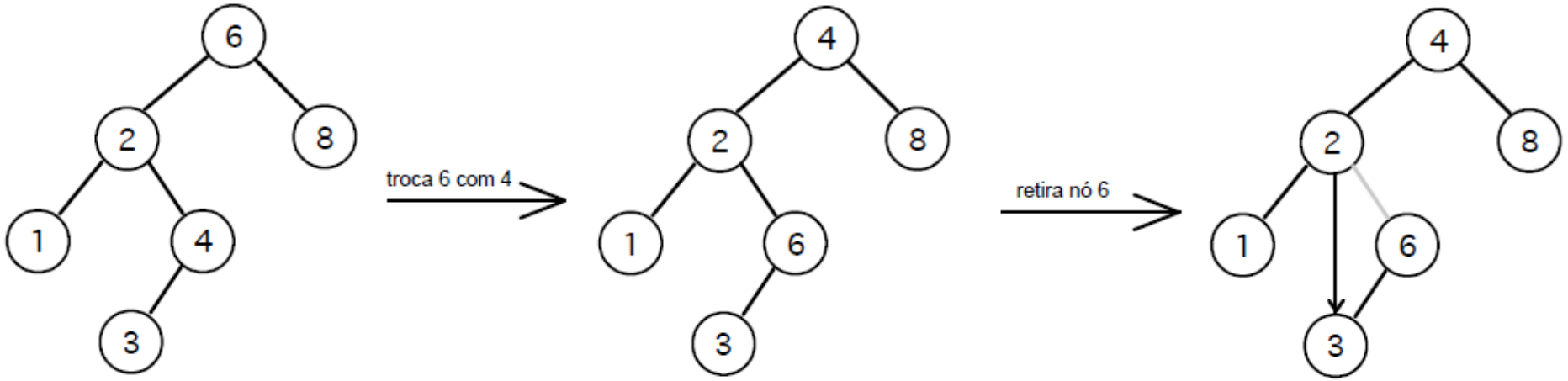
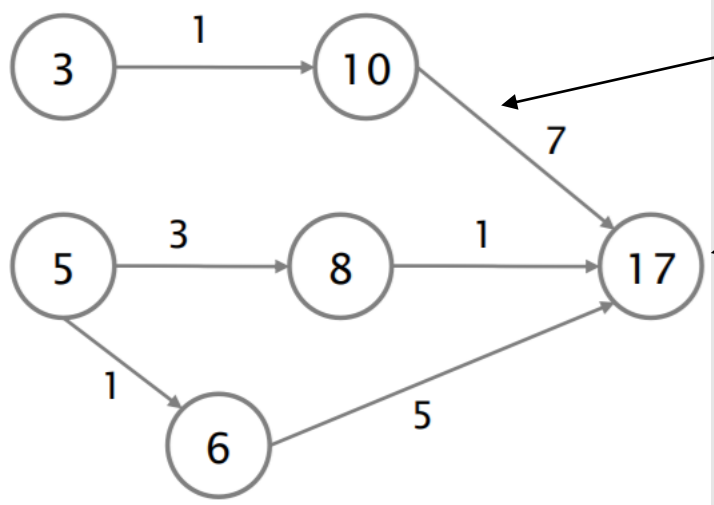


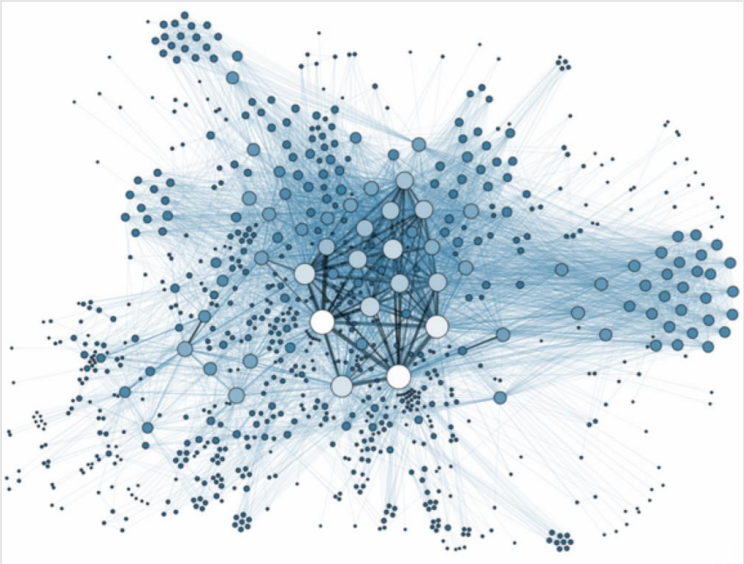
Figura 16.3: Exemplo da operação para retirar o elemento com informação igual a 6.

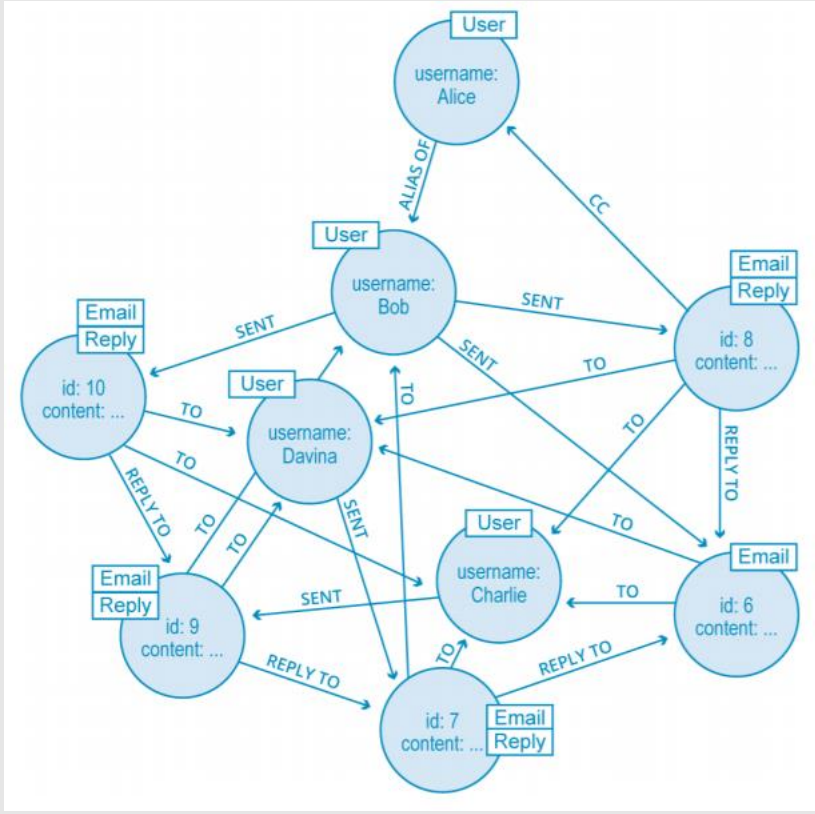
Grafos



Aresta

Vértice





Aplicações de Grafos

grafo	vértices	arestas
Cronograma	tarefas	restrições de preferência
Malha viária	interseções de ruas	ruas
Rede de água (telefônica,...)	Edificações (telefones,...)	Canos (cabos,...)
Redes de computadores	computadores	linhas
Software	funções	chamadas de função
Web	páginas Web	links
Redes Sociais	pessoas	relacionamentos
...		

Grafo não dirigido

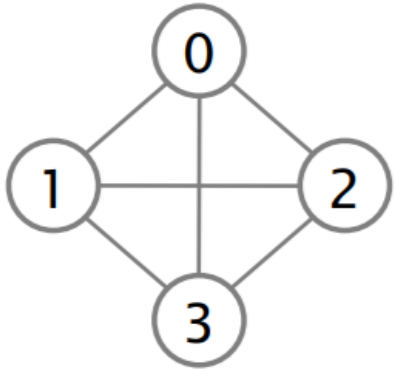
Um *grafo não dirigido* é um par G = (V,E), onde

V é um conjunto de *nós* ou *vértices* e

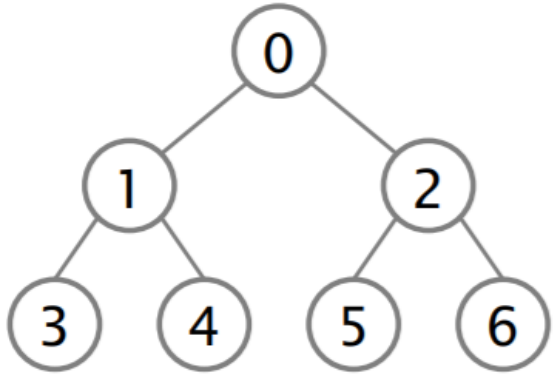
E é um conjunto de *arestas*

uma *aresta* é um conjunto de 2 vértices

Exemplos



vértices: V = {0,1,2,3}
arestas: E = {{0,1},{0,2},{0,3},{1,2},{1,3},{2,3}}



vértices : V = {0,1,2,3,4,5,6}
arestas: E = {{0,1},{0,2},{1,3},{1,4},{2,5},{2,6}}

Grafo dirigido (orientado, ou Digrafo)

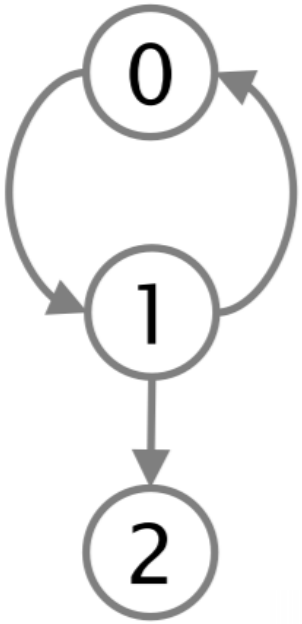
Um *grafo dirigido* é um par G = (V,E), onde
V é um conjunto de *n nós* ou *vértices* e
E é um conjunto de *m arcos*

um *arco* é um par ordenado de vértices

Exemplo

vértices: V = {0,1,2}

arcos: E = {(0,1), (1,0), (1,2)}

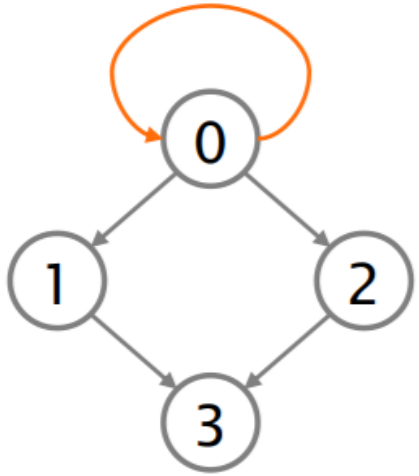


Grafo dirigido (orientado, ou Digrafo)

Exemplo - Digrafo com auto-arco

vértices: V = {0,1,2,3}

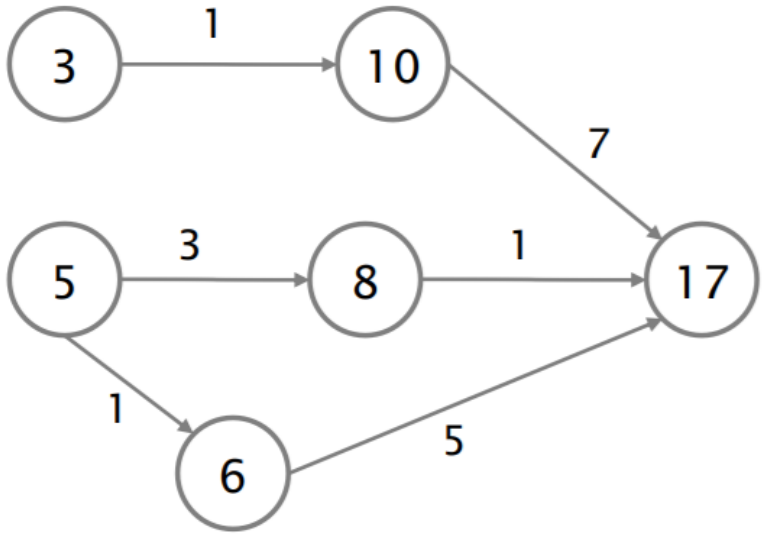
arcos: E = {(0,0), (0,1), (0,2), (1,3), (2,3)}



Grafo ponderado

Um *grafo ponderado* é uma tripla G = (V,E,p), onde
V é um conjunto de *n nós* ou *vértices* e
E é um conjunto de *m arcos*
p é uma função que atribui a cada arco um *peso*

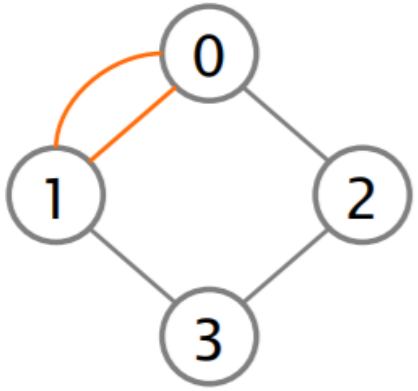
Exemplo



Multigrafo

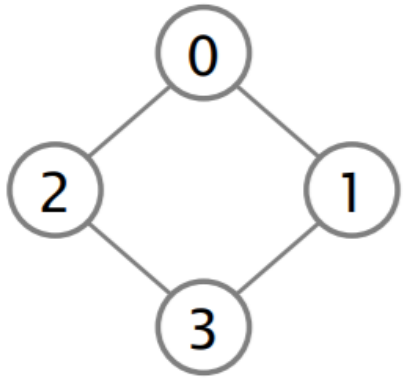
Um *multigrafo* é um grafo onde
dois nós podem estar conectados por mais de uma aresta

Exemplo

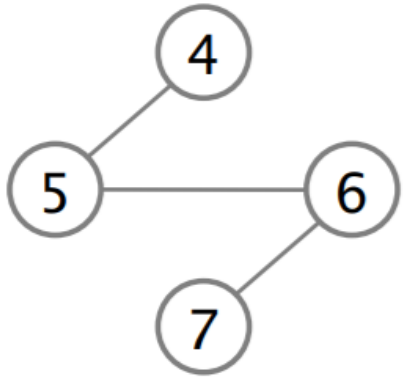


Vértices adjacentes

Vértices conectados por arestas

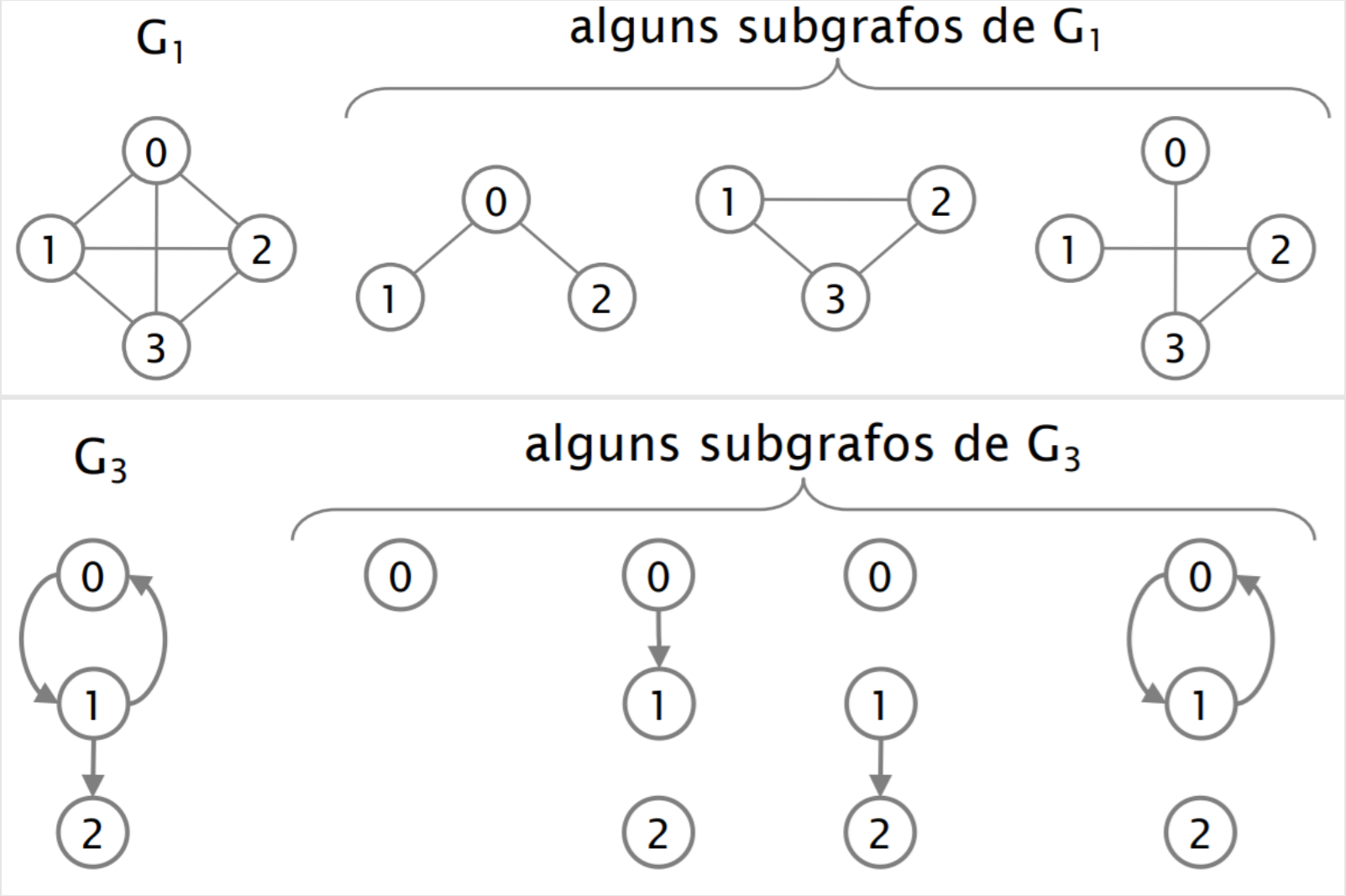


0 e 1
0 e 2
1 e 3
2 e 3



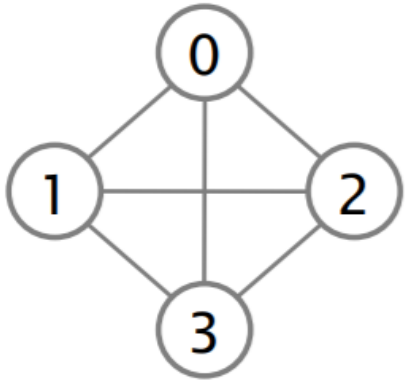
4 e 5
5 e 6
6 e 7

Subgrafo



Grafo completo

Um grafo não direcionado é *completo* sse cada vértice está conectado a cada um dos outros vértices por uma aresta



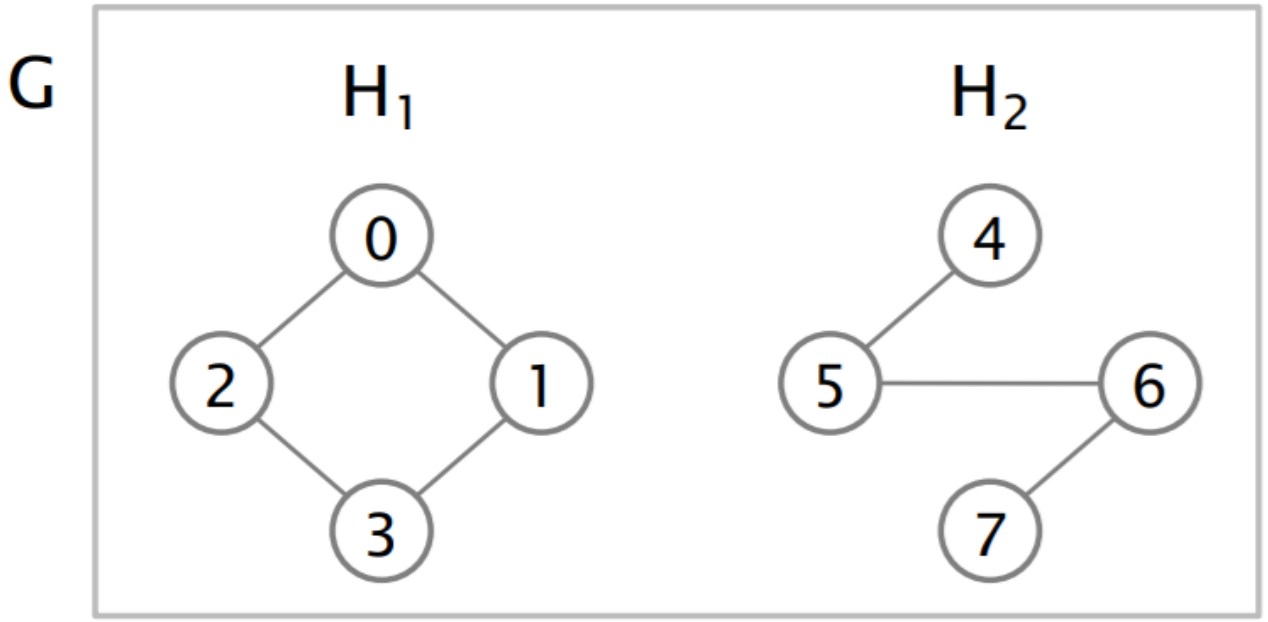
Quantas arestas há em um grafo completo de n vértices?

n (n-1)/2

Grafo conectado

Um grafo não direcionado é *conectado* ou *conexo* sse existe um caminho entre quaisquer dois vértices

Componente conexa de um grafo



Grau

Um vértice possui *grau n* sse
há exatamente *n* arestas incidentes ao vértice

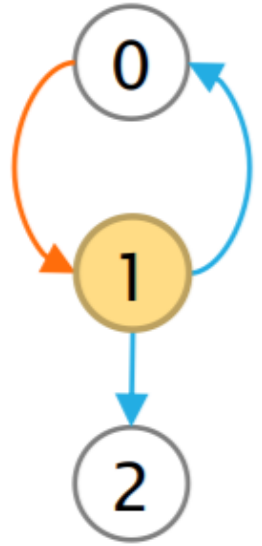
Exemplo:

grau do vértice 1: 3

grau de entrada do vértice 1: 1

grau de saída do vértice 1: 2

G_3



Caminhos

Caminho

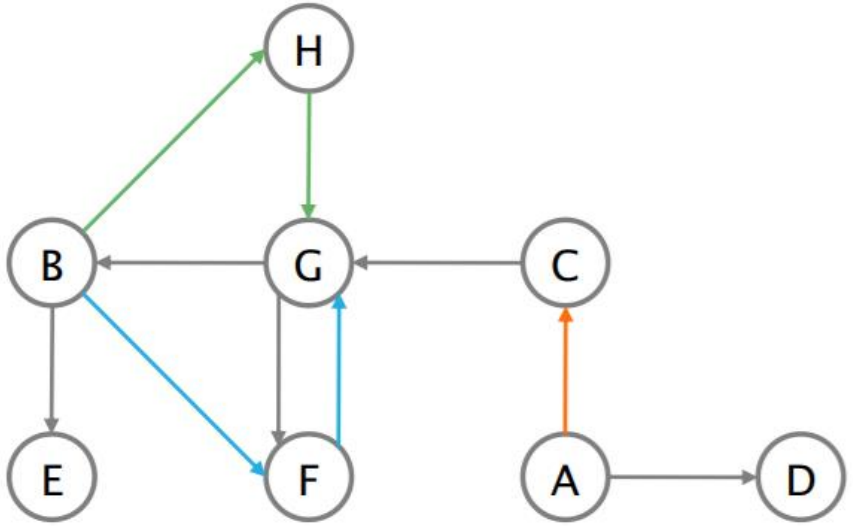
de comprimento 1 entre A e C

de comprimento 2 entre B e G, passando por H

de comprimento 2 entre B e G, passando por F

de comprimento 3 de A a F

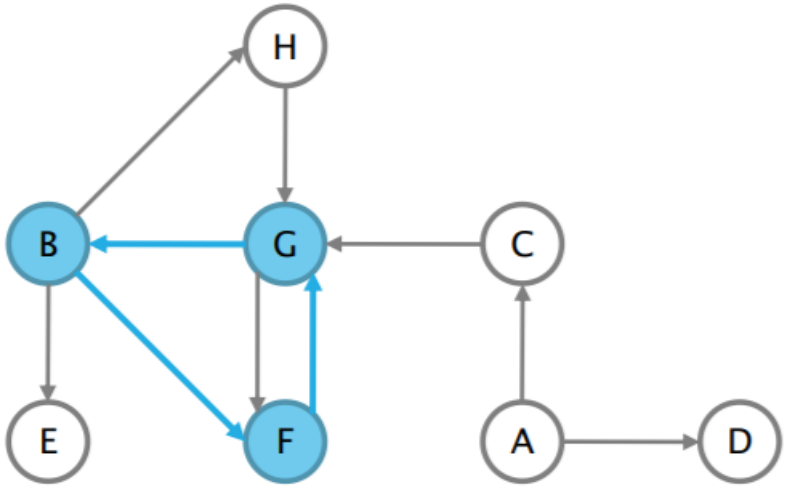
Ciclos



Ciclos

Um *ciclo* é um caminho de um nó para ele mesmo

exemplo: B-F-G-B



Grafo cíclico

contém um ou mais ciclos

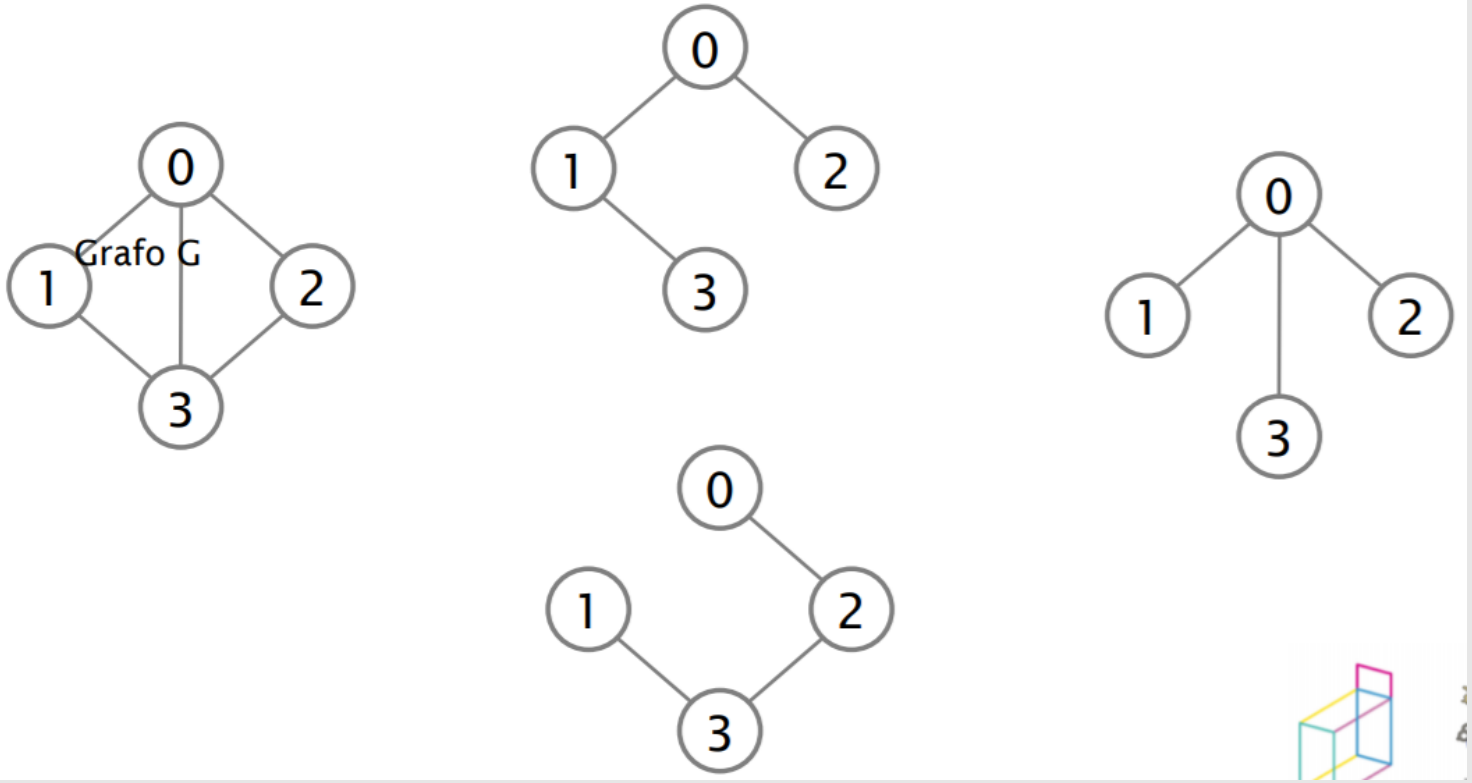
Grafo acíclico

não contém ciclos

em grafos direcionados e não direcionados

Árvore Geradora

subgrafo acíclico contendo todos os vértice com caminhos entre quaisquer 2 vértices

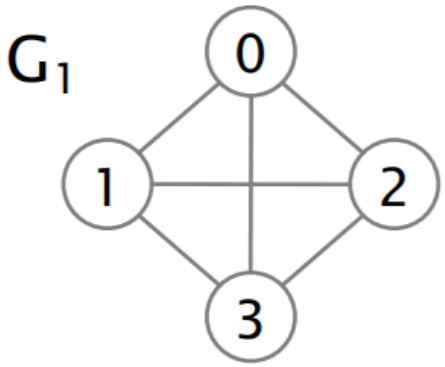


Uma árvore geradora é simplesmente um conjunto de arestas do grafo que gera uma árvore.

De um modo geral, obtém-se uma árvore geradora removendo arestas até eliminar os ciclos, mas mantendo a conexidade. É fácil ver que se um grafo tem n nós, então uma árvore geradora do grafo terá exatamente n — 1 arestas. Observe-se ainda que um grafo pode admitir diferentes árvores geradoras, conforme a escolha de arestas a eliminar.

Representações de grafo - Matriz de adjacências

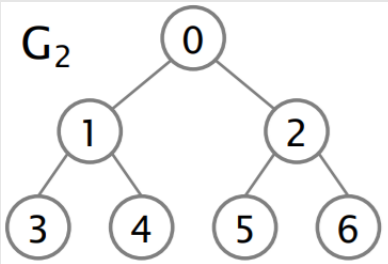
$$mat[i][j] = \begin{cases} 1, \text{ se houver uma aresta do nó i para o nó j} \\ 0, \text{ caso contrário} \end{cases}$$



	0	1	2	3
0	0	1	1	1
1	1	0	1	1
2	1	1	0	1
3	1	1	1	0

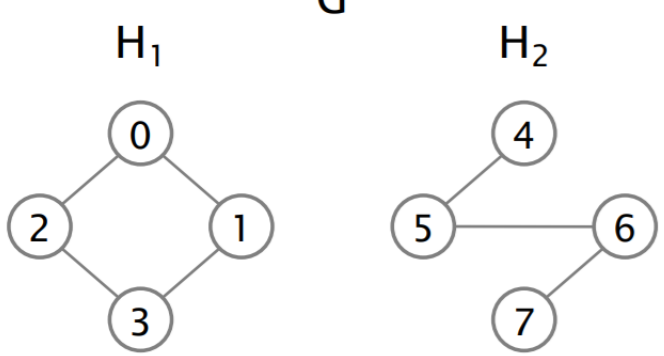
matrizes simétricas para grafos não direcionados

Representações de grafo - Matriz de adjacências



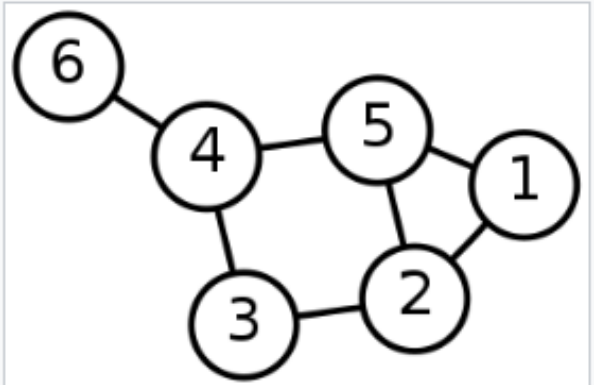
	0	1	2	3	4	5	6
0	0	1	1	0	0	0	0
1	1	0	0	1	1	0	0
2	1	0	0	0	0	1	1
3	0	1	0	0	0	0	0
4	0	1	0	0	0	0	0
5	0	0	1	0	0	0	0
6	0	0	1	0	0	0	0

G



	0	1	2	3	4	5	6	7
0	0	1	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0
2	1	0	0	1	0	0	0	0
3	0	1	1	0	0	0	0	0
4	0	0	0	0	0	1	0	0
5	0	0	0	0	1	0	1	0
6	0	0	0	0	0	1	0	1
7	0	0	0	0	0	0	1	0

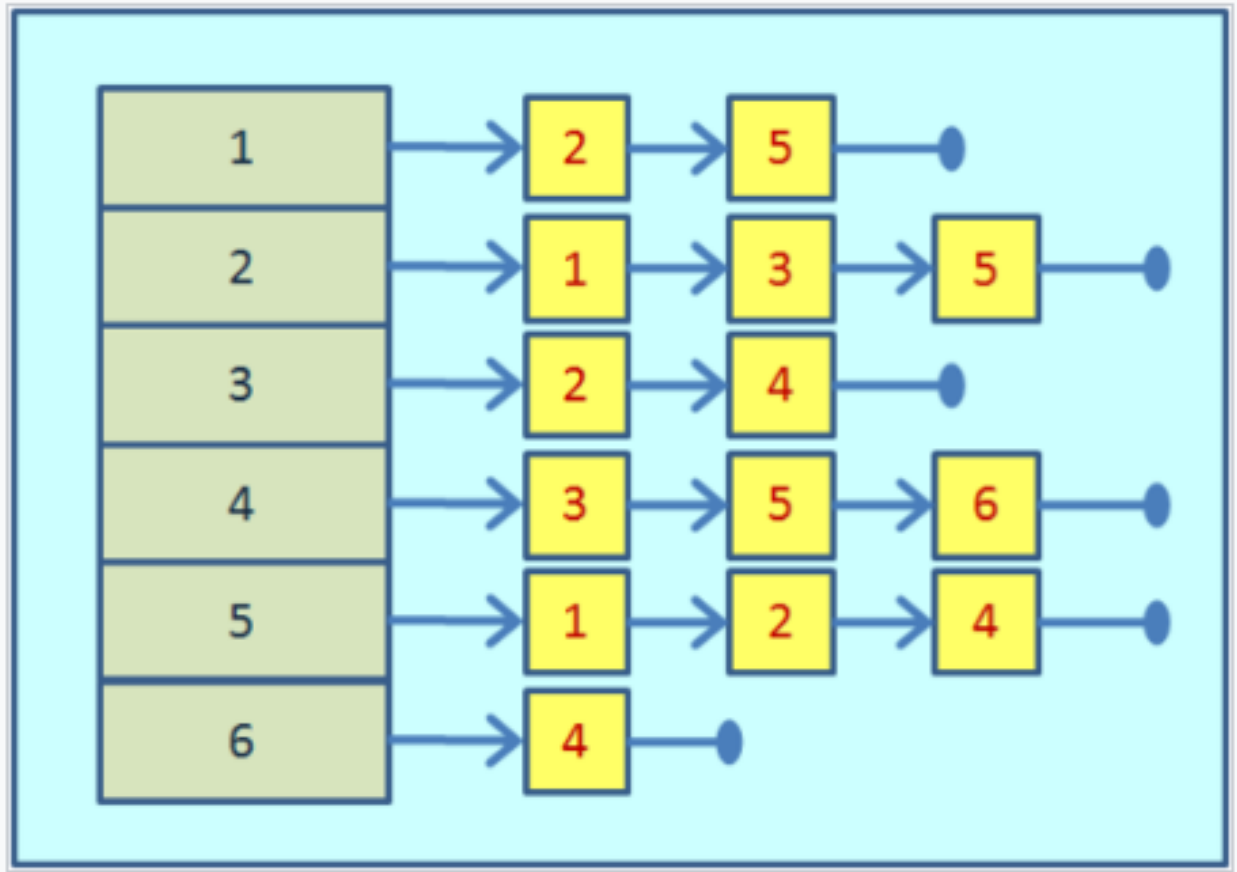
Representações de grafo - Listas de adjacências



Um grafo não dirigido com 6 vértices e 7 arestas.

O grafo da figura acima tem essa representação de lista de adjacência:

1	adjacente a	2,5
2	adjacente a	1,3,5
3	adjacente a	2,4
4	adjacente a	3,5,6
5	adjacente a	1,2,4
6	adjacente a	4



Lista de adjacências do grafo acima como encontrada em Cormen el al..

Códigos – Matriz e Listas de adjacências

- MatrizDeAdjacências.c

- ListaDeAdjacências.c
  - `void GRAPHremoveArc( Graph G, vertex v, vertex w); // implementar`
  - `void GRAPHshow( Graph G); // implementar`
  - `void GRAPHshowAll( Graph G); // implementar`

- Implementar o algoritmo de 8.2: “proc profr (v:nó)_”, versão recursiva.