Estruturas de Dados

Árvores B+ e Grafos

Universidade Estadual Vale do Acaraú – UVA

Paulo Regis Menezes Sousa

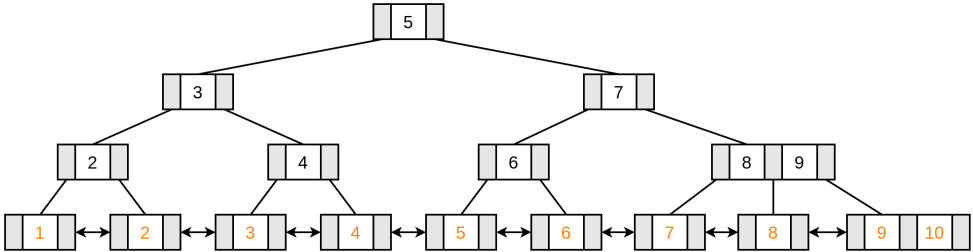
paulo_regis@uvanet.br

Árvores B+

Grafos

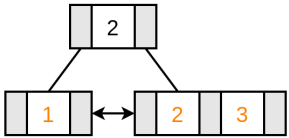
- Aplicações
- Representações
- Implementação

- A versão mais conhecida da árvore B é a árvore B+. O que distingue a árvore B+ da árvore B são dois aspectos principais:
  - todos os nós folha estão ligados entre si em uma lista duplamente ligada.
  - os dados são armazenados apenas nos nós folha. Os nós internos apenas possuem chaves e agem como roteadores para o nó folha correto.



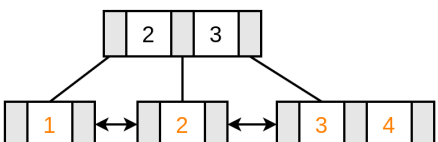
- Consideremos uma árvore de ordem 3.
- Inserimos as chaves 1 e 2 no nó raiz em ordem crescente

- Ao inserir a chave 3 excedemos a capacidade do nó raiz.
  - *Assim como em uma árvore B, precisamos realizar uma operação de divisão. No entanto, ao contrário da árvore B, devemos copiar a chave do nó acima no novo nó folha à direita.*

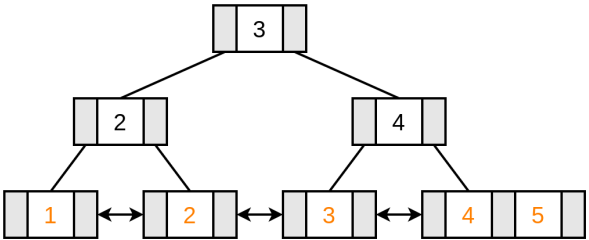


  - Isso garante que temos um par chave/valor para a chave 2 nos nós folha.

- Adicionamos a chave 4 ao nó folha mais à direita. Como está cheio, precisamos realizar outra operação de divisão e copiar a chave 3 para o nó raiz:

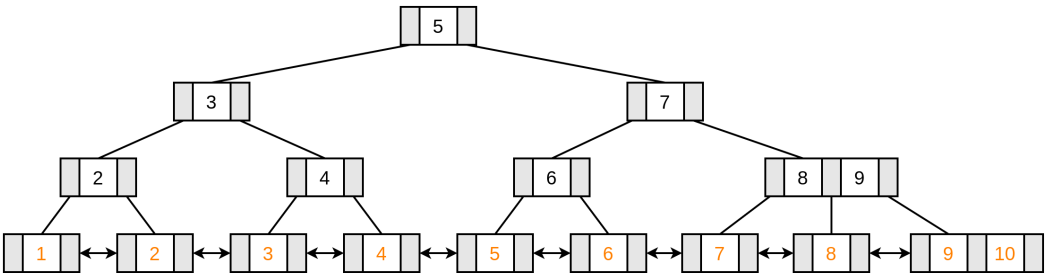


- Adicionamos a chave 5 ao nó folha mais à direita. Para manter a ordem, dividiremos o nó folha e copiaremos 4. Isso irá estourar o nó raiz, teremos que realizar outra operação de divisão do nó raiz em dois nós e promovendo 3 em um novo nó raiz:

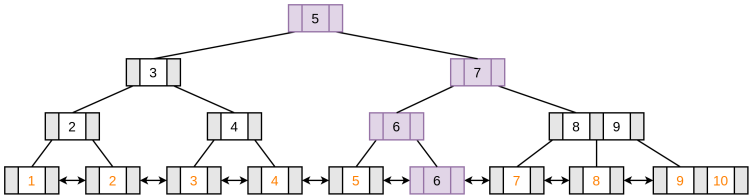


- Observe a diferença entre dividir um nó folha e dividir um nó interno.
- Quando dividimos o nó interno na segunda operação de divisão, não copiamos a chave 3.

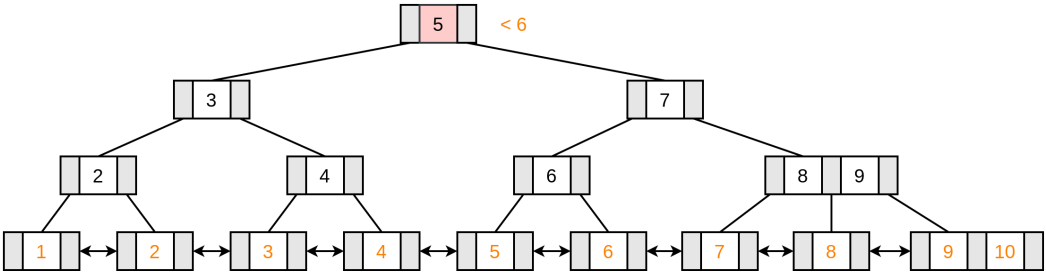
- Da mesma forma, vamos adicionando as chaves de 6 a 10, cada vez dividindo e copiando quando necessário até chegarmos à nossa árvore final:

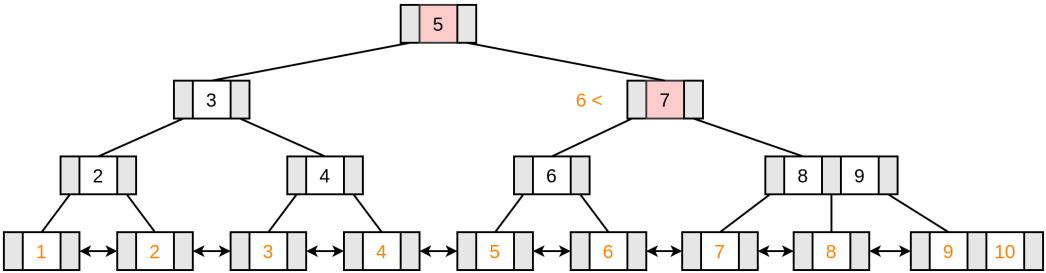


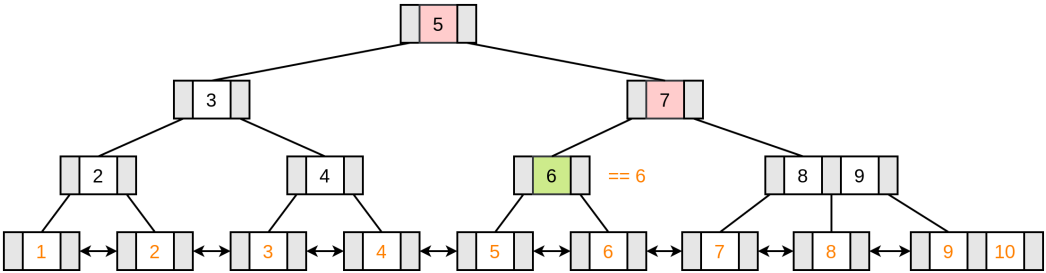
- Pesquisar uma chave específica em uma árvore B+ é semelhante a pesquisar uma chave em uma árvore B normal.

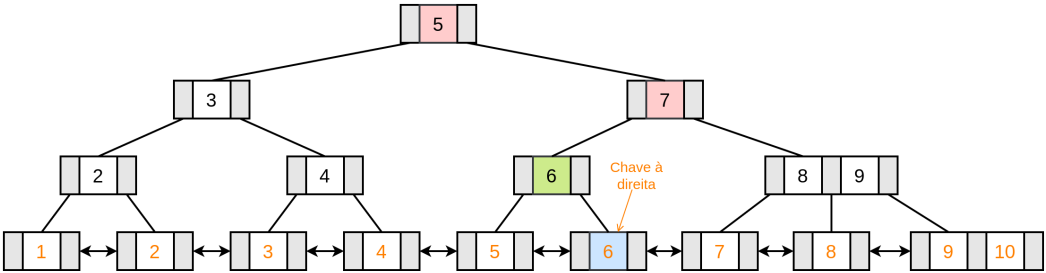


- Os nós sombreados nos mostram o caminho que tomamos para encontrar valor 6.
- Pesquisar em uma árvore B+ significa que devemos descer até um nó folha para obter os dados. Ao contrário das árvores B, onde podemos encontrar os dados em qualquer nível.

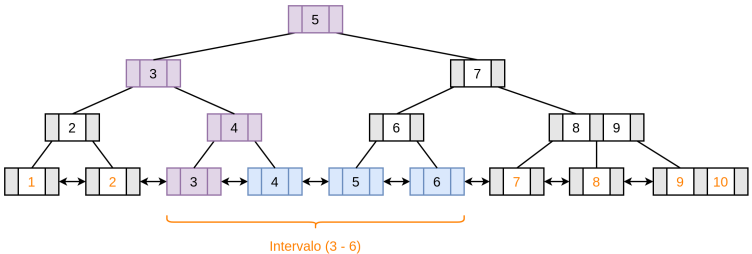

5
< 6
3
7
2
4
6
8
9
1
2
3
4
5
6
7
8
9
10


5
3
6 <
7
2
4
6
8
9
1
2
3
4
5
6
7
8
9
10

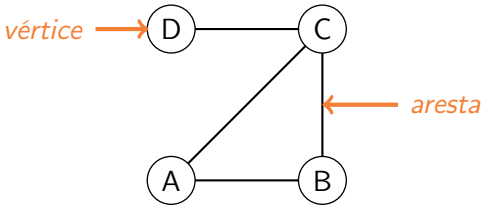

5
3
7
2
4
6
== 6
8
9
1
2
3
4
5
6
7
8
9
10


5
3
7
2
4
6
8
9
Chave à direita
1
2
3
4
5
6
7
8
9
10

- Além das consultas de correspondência exata de chave, as árvores B+ suportam consultas de intervalo.
- Isso se dá pelo fato de que os nós de folha da árvore B+ estão todos vinculados.
- Para realizar uma consulta de intervalo, tudo o que precisamos fazer é:
  - encontrar a correspondência exata da chave mais baixa
  - a partir daí, seguir a lista ligada até chegar ao nó folha com a chave mais alta



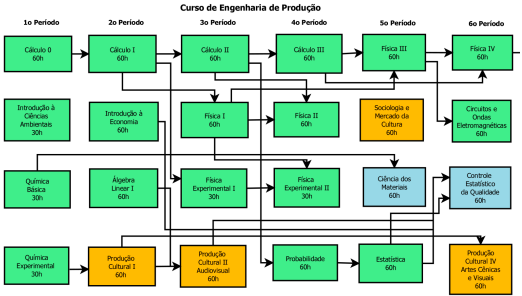
- Grafos são estruturas de dados formadas por um conjunto de vértices e um conjunto de arestas.



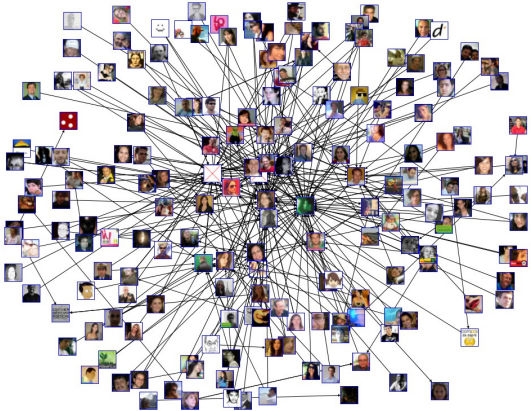
Exemplo de grafo

- Associando-se significados aos vértices e às arestas, o grafo passa a constituir um modelo de uma situação ou informação real.

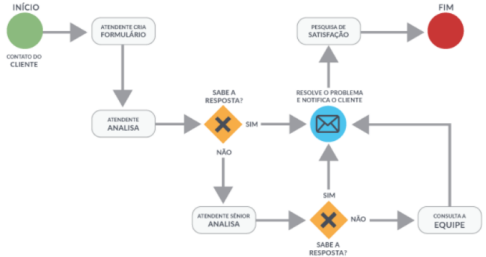
Rotas de vôos



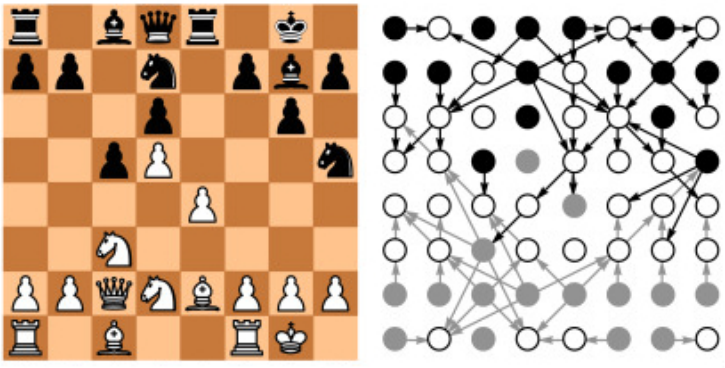
Grade curricular



Mapa de redes sociais



Processo/tarefas

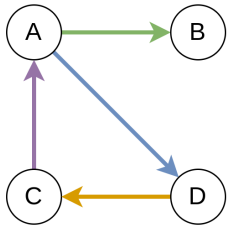


Os vértices são as casas de um tabuleiro de xadrez. Há um arco de x para y se uma peça do jogo pode ir de x a y em um só movimento.

- **Matriz de adjacência** é uma forma de representação de grafos simples, econômica e adequada para muitos problemas que envolvem apenas a estrutura do grafo.
- A matriz de adjacência $A(n \times n)$ de um grafo G com n vértices, é uma matriz onde cada elemento a_{ij} representa a presença ou ausência de ligação entre o vértice i e o vértice j.

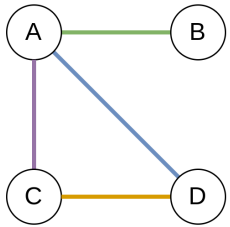
- Grafo orientado

	A	B	C	D
A	0	1	0	1
B	0	0	0	0
C	1	0	0	0
D	0	0	1	0

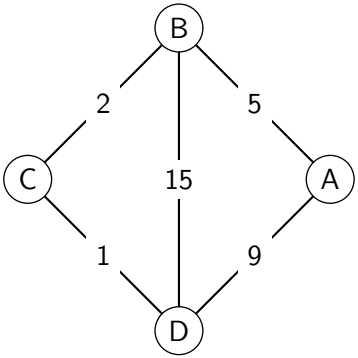


- Grafo não orientado

	A	B	C	D
A	0	1	1	1
B	1	0	0	0
C	1	0	0	1
D	1	0	1	0



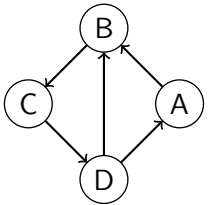
- Valores associados às linhas podem ser representados por uma extensão simples da Matriz de Adjacência.



	A	B	C	D
A	0	5	0	9
B	5	0	2	15
C	0	2	0	1
D	9	15	1	0

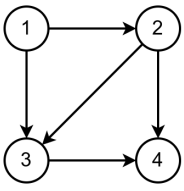
- **Matriz binária**: ocupa pouco espaço, especialmente para grafos grandes
- **Manipulação simples**: recursos para manipular matrizes existem em qualquer linguagem de programação
- Fácil determinar se $(v_i, v_j) \in G(E)$
- Fácil determinar vértices adjacentes a um determinado vértice v
- Quando o grafo é não orientado, a MA é simétrica (mais econômica)
- Inserção de novas arestas é fácil
- Inserção de novos vértices é muito difícil

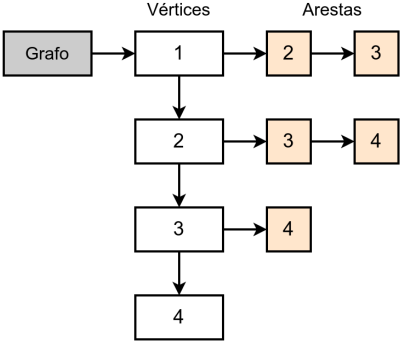
- É uma matriz $B(n \times m)$, sendo n o número de vértices, m o número de arestas e:
  - $b_{ij} = -1$ se o vértice i é a origem da aresta j
  - $b_{ij} = 1$ se o vértice i é o término da aresta j
  - $b_{ij} = 0$ se aresta $(i, j) \notin G(E)$
- Para grafos não orientados, $b_{ij} = 1$ se a aresta j é incidente ao vértice i.



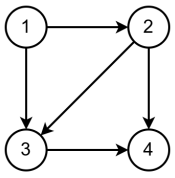
	a1	a2	a3	a4	a5
A	-1	0	0	1	0
B	1	-1	0	0	1
C	0	1	-1	0	0
D	0	0	1	-1	-1

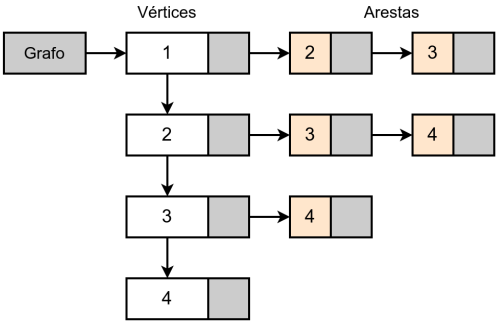
- Para cada vértice v é representada a lista de vértices u tais que $(v, u) \in G(E)$
- Possíveis formas de armazenamento: vetores, vetores + listas encadeadas, listas encadeadas
- Melhor forma de representação: **listas encadeadas**
  - Uso racional do espaço
  - Flexibilidade



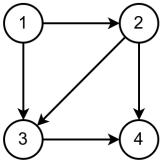


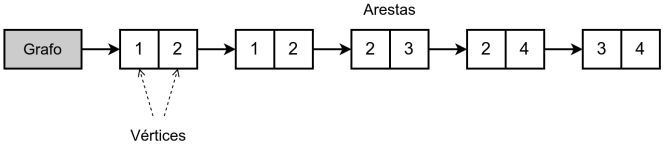
- Nós podem ser estendidos para representar outras informações



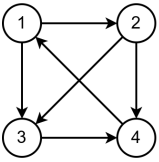


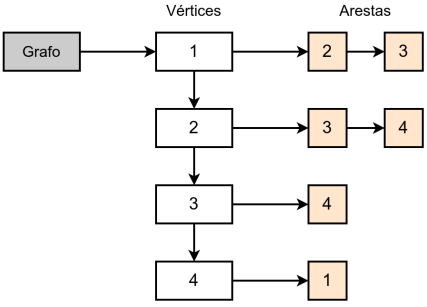
- Lista de incidência





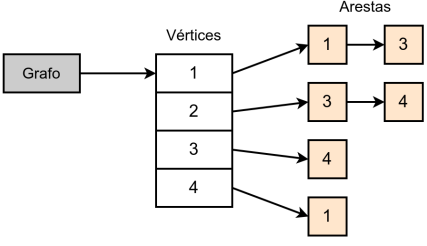
Resumo das formas mais comuns de representação





Grafo	1	2	3	4
1	0	1	1	0
2	0	0	1	1
3	0	0	0	1
4	1	0	0	0

Arestas



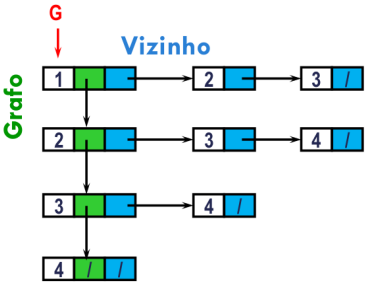
Arestas

Grafo	a	b	c	d	e	f
1	-1	-1	0	0	0	1
2	1	0	-1	-1	0	0
3	0	1	1	0	-1	0
4	0	0	0	1	1	-1

Vértices

- Implementação de grafos usando lista de adjacência
  - São flexíveis para acomodar inserções e remoções, ao contrário das matrizes de **adjacência** e **incidência**
  - Facilitam a identificação dos vértices do grafo, ao contrário das listas de incidência

```c
typedef struct vizinho {
    int id_vizinho;
    struct vizinho *prox;
} Vizinho;

typedef struct grafo {
    int id_vertice;
    Vizinho *prim_vizinho;
    struct grafo *prox;
} Grafo;
```



- Criação da estrutura

```
1  Grafo *criar_grafo() {
2      Grafo *g = malloc(sizeof(Grafo));
3      if (g) {
4          g->id_vertice = 0;
5          g->prox = NULL;
6      }
7      return g;
8  }
```

- Impressão do grafo

```
1  void imprimir(Grafo *g){
2      while(g != NULL){
3          printf("Vértice %d\n", g->id_vertice);
4          printf("Vizinhos: ");
5
6          Vizinho *v = g->prim_vizinho;
7          while(v != NULL){
8              printf("%d ", v->id_vizinho);
9              v = v->prox;
10         }
11         printf("\n\n");
12         g = g->prox;
13     }
14 }
```

- Desalocar grafo

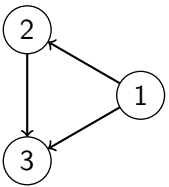
```
1  void liberar(Grafo *g){
2      Grafo *temp = NULL;
3      while(g != NULL){
4          liberar_vizinhos(g->prim_vizinho);
5
6          temp = g;
7          g = g->prox;
8          free(temp);
9      }
10 }
11
12 void liberar_vizinhos(Vizinho *v){
13     while(v != NULL){
14         Vizinho *temp = v;
15         v = v->prox;
16         free(temp);
17     }
18 }
```

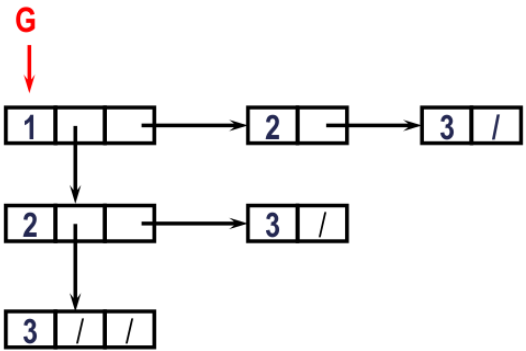
- Busca por um vértice v_1
  1. Basta percorrer a lista de vértices até encontrar v_1
- Busca por uma aresta (v_1, v_2)
  1. Percorrer a lista de vértices até encontrar v_1
  2. Depois percorrer a lista de vizinhos de v_1 até encontrar v_2

```
1  Grafo *buscar_vertice(Grafo *g, int id){
2
3      while((g != NULL) && (g->id_vertice != id)) {
4          g = g->prox;
5      }
6
7      return g;
8  }
```

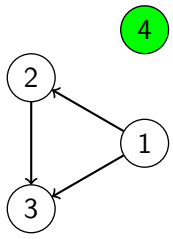
```
1  Vizinho* buscar_aresta(Grafo *g, int id_v1, int id_v2){
2      Grafo *v1 = buscar_vertice(g, id_v1);
3      Grafo *v2 = buscar_vertice(g, id_v2);
4      Vizinho *viz = NULL;
5
6      //checa se ambos os vértices existem
7      if((v1 != NULL) && (v2 != NULL)) {
8          //percorre a lista de vizinhos de v1 procurando por v2
9          viz = v1->prim_vizinho;
10
11         while ((viz != NULL) && (viz->id_vizinho != id_v2)) {
12             viz = viz->prox;
13         }
14     }
15
16     return viz;
17 }
```

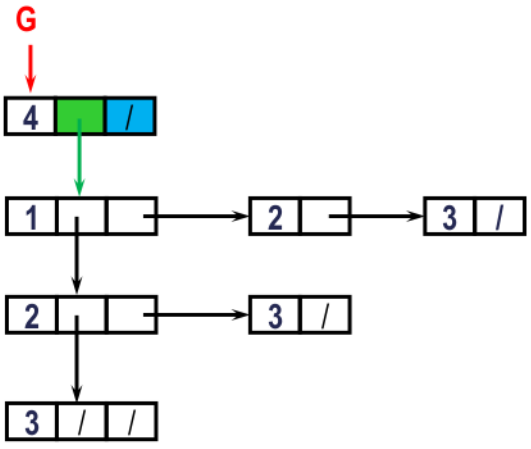
- Insere o vértice na lista encadeada de vértices, como **primeiro vértice da lista**
- **Exemplo**: inserir vértice 4





- Insere o vértice na lista encadeada de vértices, como **primeiro vértice da lista**
- **Exemplo**: inserir vértice 4

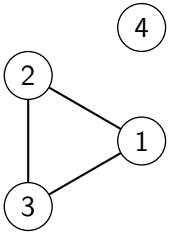


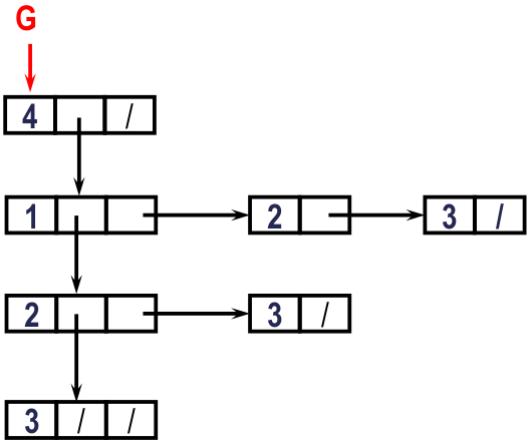


```
1  Grafo *inserir_vertice(Grafo *g, int id){
2      Grafo *p = buscar_vertice(g, id);
3
4      if (p == NULL){
5          p = criar_grafo();
6          p->id_vertice = id;
7          p->prox = g;
8          p->prim_vizinho = NULL;
9          g = p;
10     }
11
12     return g;
13 }
```

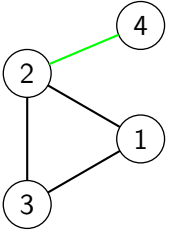
- Grafo não orientado
  - **Inserção de aresta** $\{v_1, v_2\}$: inserir v_2 na lista de vizinhos de v_1, e v_1 na lista de vizinhos de v_2 (ou seja, inserir as arestas (v_1, v_2) e (v_2, v_1))
- Grafo orientado
  - **Inserção de aresta** (v_1, v_2): inserir v_2 na lista de vizinhos de v_1
- Em ambos os casos, verificar se a aresta já existe antes de realizar a inserção

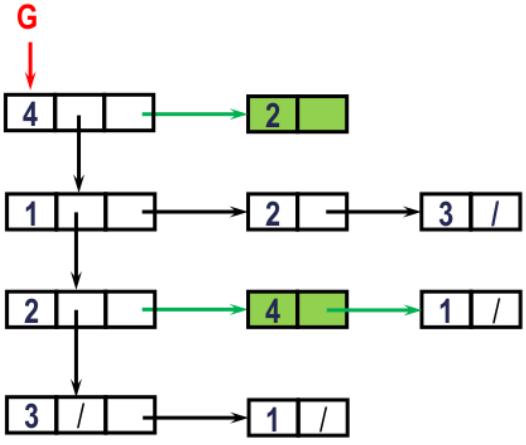
- **Exemplo**: inserir aresta $\{2, 4\}$





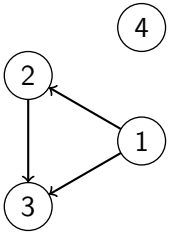
- **Exemplo**: inserir aresta $\{2, 4\}$

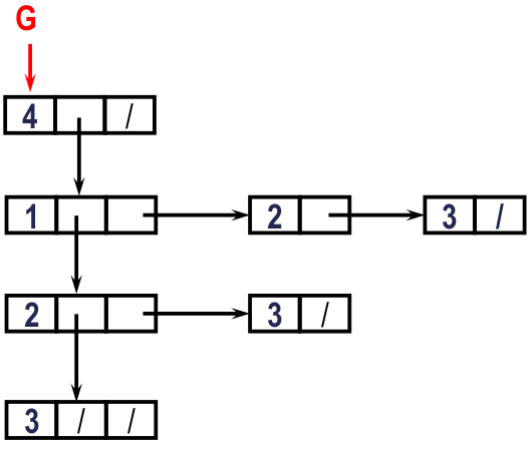




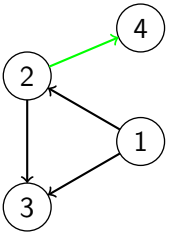
```c
void inserir_um_sentido(Grafo *g, int id_v1, int id_v2){
    Grafo *v1 = buscar_vertice(g, id_v1);
    Vizinho *novo_viz = (Vizinho *) malloc(sizeof(Vizinho));

    novo_viz->id_vizinho = id_v2;
    novo_viz->prox = v1->prim_vizinho;

    v1->prim_vizinho = novo_viz;
}

void inserir_aresta(Grafo *g, int id_v1, int id_v2){
    Vizinho *viz = buscar_aresta(g, id_v1, id_v2);

    if(viz == NULL) {
        inserir_um_sentido(g, id_v1, id_v2);
        inserir_um_sentido(g, id_v2, id_v1);
    }
}
```

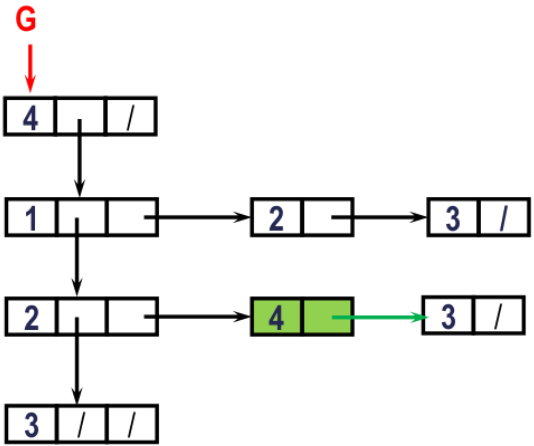
- **Exemplo**: inserir aresta $(2, 4)$





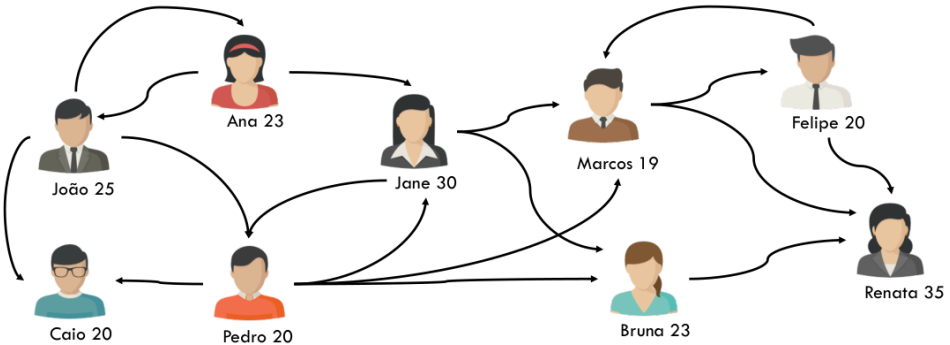
- **Exemplo**: inserir aresta $(2, 4)$





```c
void inserir_aresta_digrafo(Grafo *g, int id_v1, int id_v2) {
    Vizinho *viz = buscar_aresta(g, id_v1, id_v2);

    if (viz == NULL) {
        inserir_um_sentido(g, id_v1, id_v2);
    }
}
```

Considere o grafo a seguir, que representa seguidores no Instagram. Cada pessoa tem nome e idade (nome é o id do vértice). Uma aresta (v_1, v_2) significa que v_1 segue v_2 no Instagram.



Implementar funções em C para responder às seguintes questões:

Exercício 1

Quantas pessoas uma determinada pessoa segue?

```
int contar_seguidos(Grafo *g, char *nome);
```

Exercício 2

Quem são os seguidores de uma determinada pessoa? A função retorna quantidade de seguidores e, caso a **flag** imprime seja **True**, também deve imprimir os nomes dos seguidores.

```
int contar_seguidores(Grafo *vertice, char *nome, int imprime);
```

Exercício 3

Quem é a pessoa mais popular (tem mais seguidores)?

```
Grafo *get_mais_popular(Grafo *g);
```

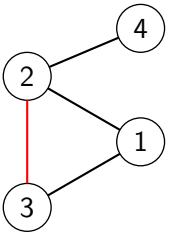
Exercício 4

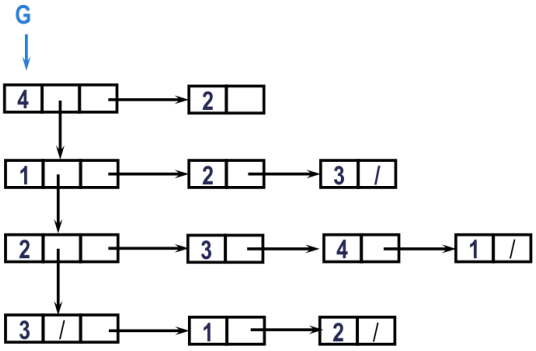
Quais são as pessoas que só seguem pessoas mais velhas do que ela própria? A função retorna quantidade de pessoas e, caso a **flag** imprime seja **True**, também deve imprimir os nomes das pessoas.

```
int contar_segue_mais_velho(Grafo *g, int imprime);
```

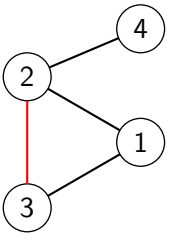
- **Grafo não orientado**
  - Exclusão de aresta v_1, v_2: excluir v_2 da lista de vizinhos de v_1, e v_1 da lista de vizinhos de v_2 (ou seja, excluir as arestas (v_1, v_2) e (v_2, v_1)).
- **Grafo orientado**
  - Exclusão de aresta (v_1, v_2): excluir v_2 da lista de vizinhos de v_1.

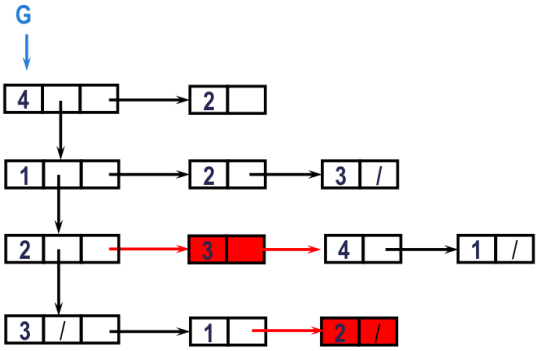
- **Exemplo**: excluir aresta $\{2, 3\}$



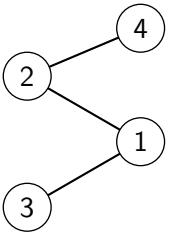


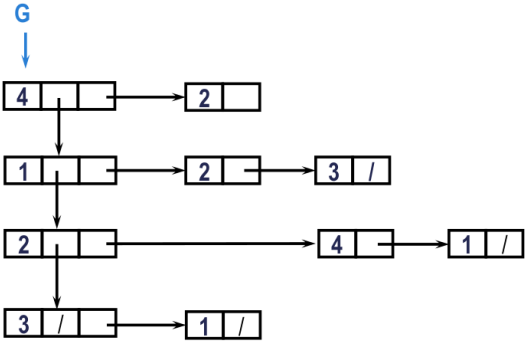
- **Exemplo**: excluir aresta $\{2, 3\}$





- **Exemplo**: excluir aresta $\{2, 3\}$

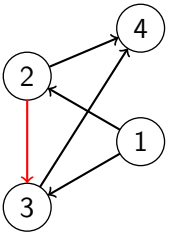


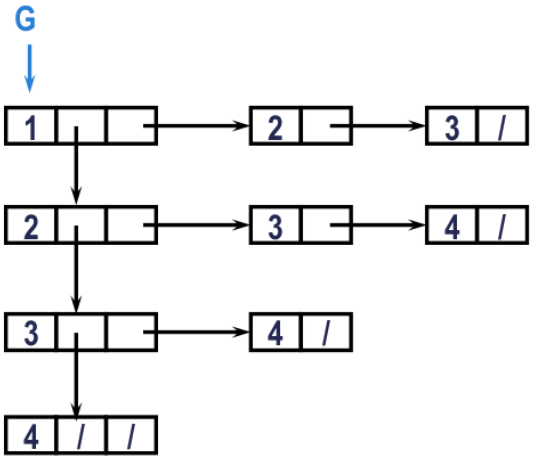


```c
void remover_um_sentido(Grafo *g, int id_v1, int id_v2) {
    Grafo *p = buscar_vertice(g, id_v1);
    Vizinho *ant, *atual;

    if(p != NULL) {
        ant = NULL;
        atual = p->prim_vizinho;
        while ((atual != NULL) && (atual->id_vizinho != id_v2)) {
            ant = atual;
            atual = atual->prox;
        }
        if (ant == NULL) // v2 era o primeiro nó da lista
            p->prim_vizinho = atual->prox;
        else
            ant->prox = atual->prox;
        free(atual);
    }
}
```

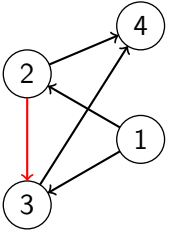
```
1  void remover_aresta(Grafo *g ,int id_v1, int id_v2){
2      Vizinho* v = buscar_aresta(g,v1,v2);
3
4      if (v != NULL) {
5          remover_um_sentido(g, id_v1, id_v2);
6          remover_um_sentido(g, id_v2, id_v1);
7      }
8  }
```

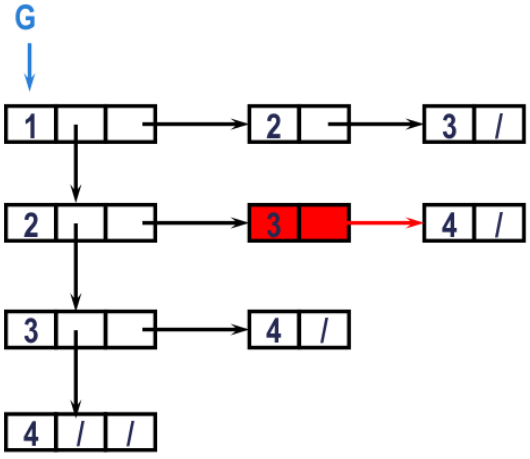
- **Exemplo**: excluir aresta $(2, 3)$



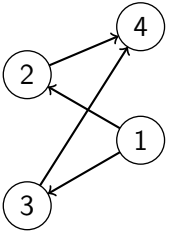


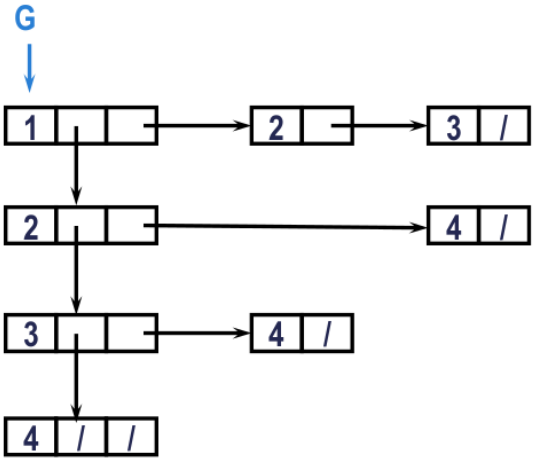
- **Exemplo**: excluir aresta $(2, 3)$





- **Exemplo**: excluir aresta $(2, 3)$

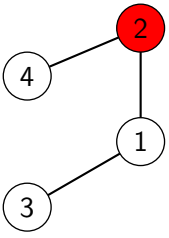


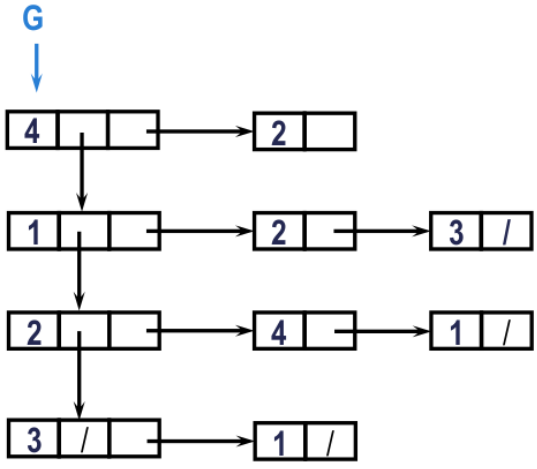


```
1  void remover_aresta_digrafo(Grafo *g ,int id_v1, int id_v2){
2      Vizinho *v = buscar_aresta(g, id_v1, id_v2);
3
4      if (v != NULL) {
5          remover_um_sentido(g, id_v1, id_v2);
6      }
7  }
```

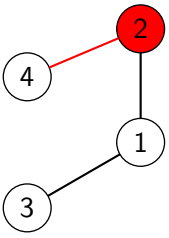
1. Exclui lista de vizinhos
2. Exclui vértice
3. Exclui todos os vizinhos que tinham este vértice como extremidade
4. Libera memória

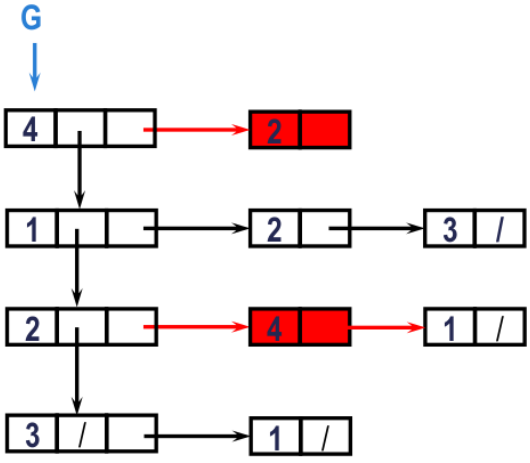
- **Exemplo**: excluir vértice 2



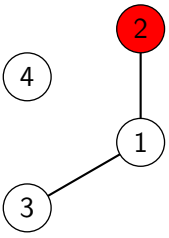


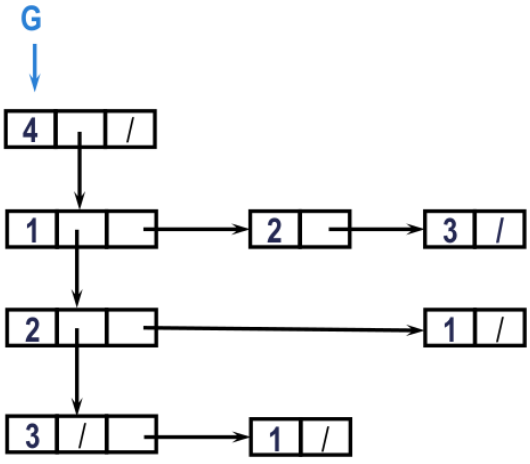
- Excluir todos os vizinhos do vértice 2
- Remove vizinho 4



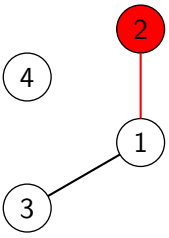


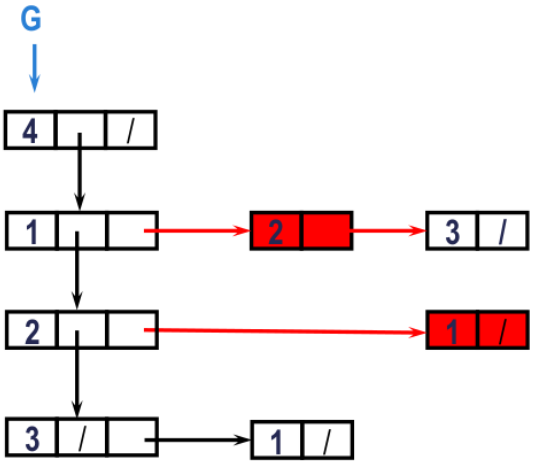
- Excluir todos os vizinhos do vértice 2
- Remove vizinho 4



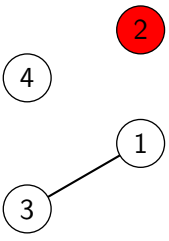


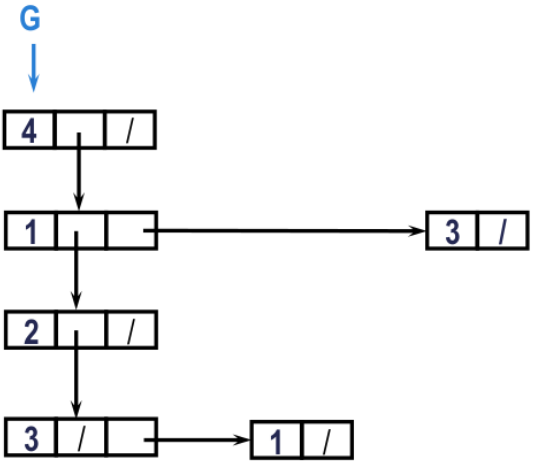
- Excluir todos os vizinhos do vértice 2
- Remove vizinho 1



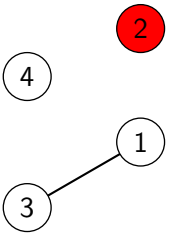


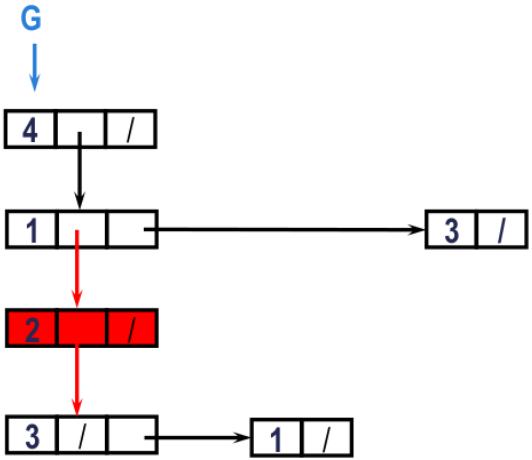
- Excluir todos os vizinhos do vértice 2
- Remove vizinho 1



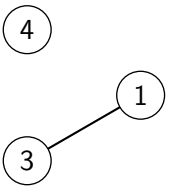


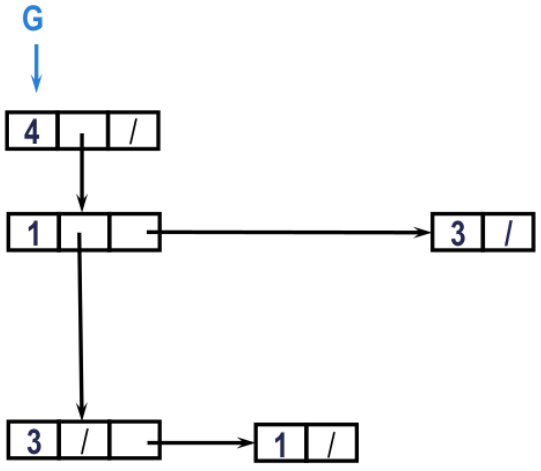
- Excluir vértice 2





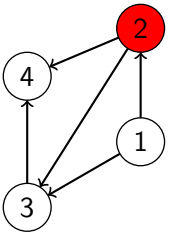
- Excluir vértice 2

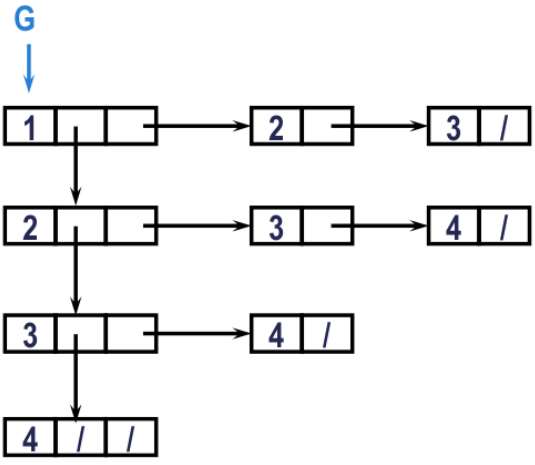




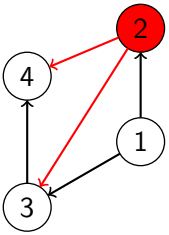
```
1  Grafo *remover_vertice(Grafo *g, int id) {
2      Grafo *p = g;
3      Grafo *ant = NULL;
4      while((p != NULL) && (p->id_vertice != id)){
5          ant = p;
6          p = p->prox;
7      }
8      if (p != NULL) {
9          while (p->prim_vizinho != NULL)
10             remover_aresta(g, v, p->prim_vizinho->id_vizinho);
11         if (ant == NULL)
12             g = g->prox;
13         else
14             ant->prox = p->prox;
15         free(p);
16     }
17     return g;
18 }
```

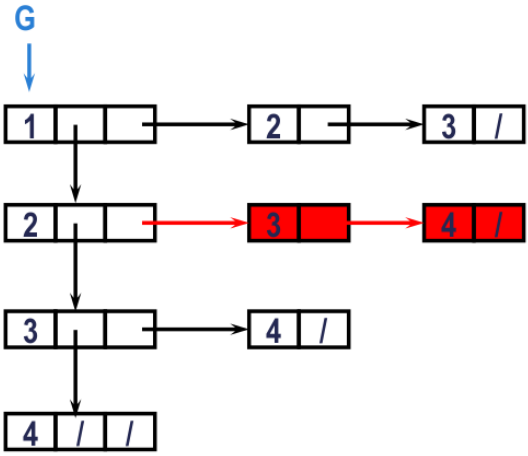
- **Exemplo**: excluir vértice 2



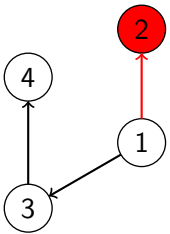


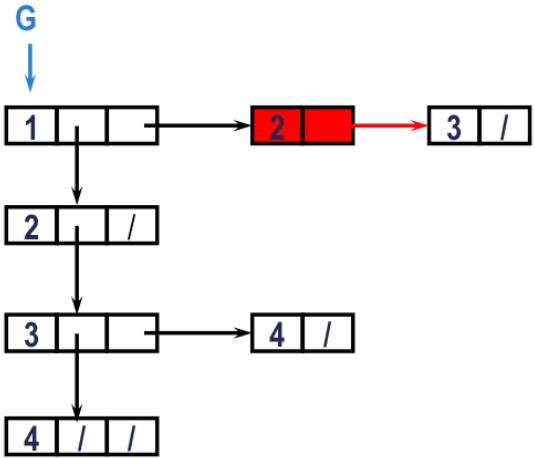
- Excluir todos os vizinhos do vértice 2



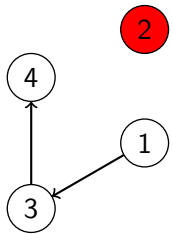


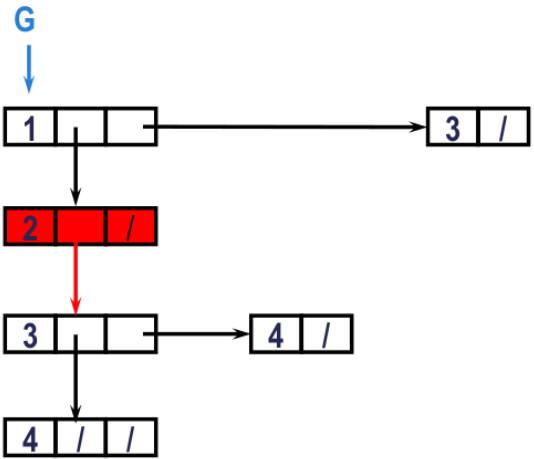
- Retira 2 da lista de vizinhos dos outros nós



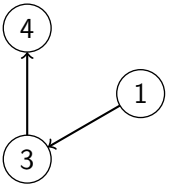


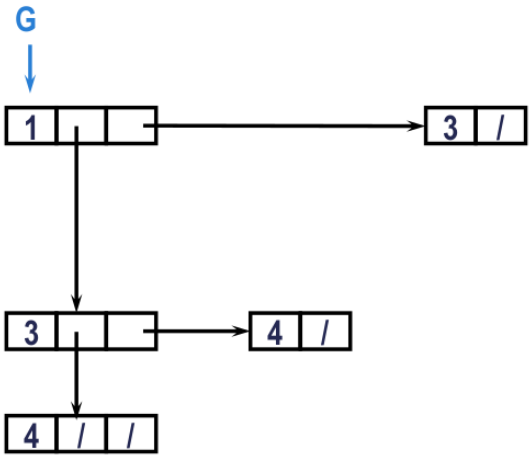
- Exclui o vértice 2





- Exclui o vértice 2





Exercício 5

Escreva uma função em C para exclusão de vértice em grafo orientado

```
Grafo *remover_vertice_digrafo(Grafo *g, int id);
```

Exercício 6

Crie uma implementação genérica de grafo orientado usando listas encadeadas com funções para:

a) Criar um grafo

b) Adicionar um vértice

c) Adicionar uma aresta

d) Imprimir o grafo