INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
CEARÁ
CAMPUS DE CANINDÉ

Aula 08 – Estruturas de Dados

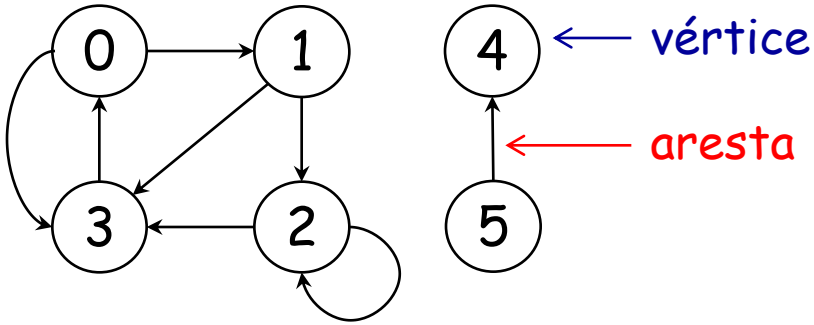
- Assunto: **Grafos**.
- Objetivos:

  - **Apresentar os conceitos elementares de Grafos e sua aplicação no cotidiano;**
  - **Aplicação de algoritmos em situações do dia-a-dia;**
  - **Elaborar programas usando Grafos.**
- Roteiro:

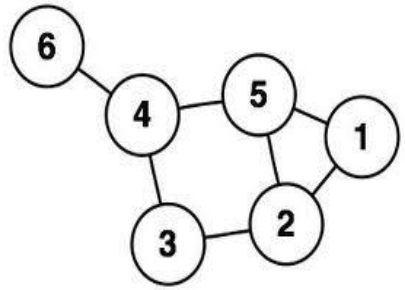
  1. Introdução.
  2. Definições e Conceitos.
  3. Implementação de Grafos com Ponteiros
  4. Operando Grafos.
  5. Exercícios.

Definição Básica

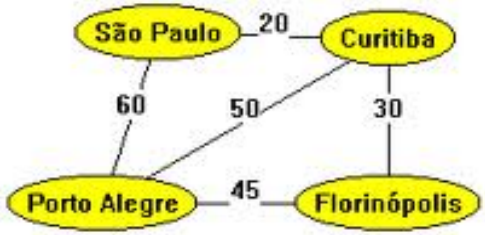
Um **grafo** é constituído de um conjunto "não" vazio de objetos denominados de vértices e um conjunto de arestas conectando pares de vértices.



Grafo direcionado



Grafo não direcionado

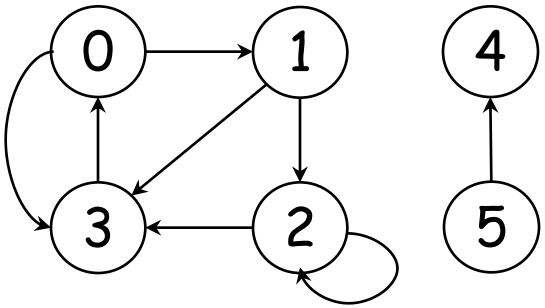


Grafo rotulado

Grafo Direcionado (ou Dígrafo)

Um **grafo direcionado** *G* é um par (*V*, *A*), em que *V* é um conjunto finito de vértices e *A* é um conjunto de arestas com uma relação binária em *V*.

Usando o grafo direcionado ilustrado na figura:



Em grafos direcionados podem existir arestas de um vértice para ele mesmo, chamados de ***self-loops***.

Tem-se:

- conjunto de vértices:

$V = \{0, 1, 2, 3, 4, 5\}$

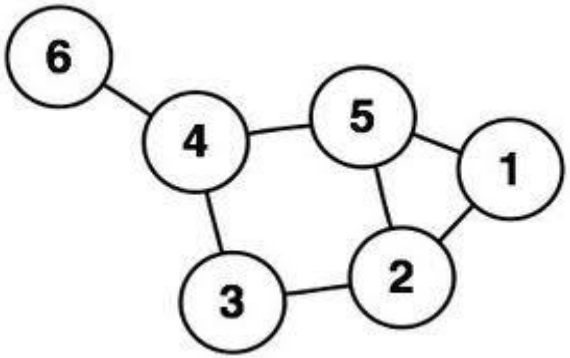
- conjunto de arestas:

$A = \{(0, 1), (0, 3), (1, 2), (1, 3), (2, 2), (2, 3), (3, 0), (5, 4)\}$

Grafo Não Direcionado

Um **grafo não direcionado** *G* é um par (*V*, *A*), em que o conjunto de arestas *A* é constituído de pares de vértices não ordenados. As arestas (*u*, *v*) e (*v*, *u*) são consideradas como única. Em um grafo não direcionado *self-loops* "não" são permitidos.

Usando o grafo não direcionado ilustrado na figura:



Tem-se:

- conjunto de vértices:

$V = \{1, 2, 3, 4, 5, 6\}$

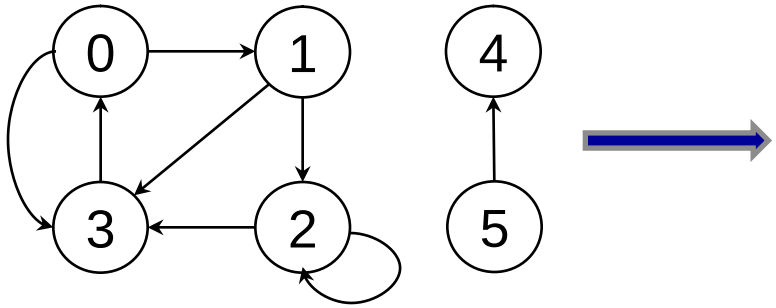
- conjunto de arestas:

$A = \{(1, 2), (1, 5),$
$(2, 1), (2, 3), (2, 5),$
$(3, 2), (3, 4),$
$(4, 3), (4, 5), (4, 6),$
$(5, 1), (5, 2), (5, 4)$
$(6, 4)\}$

Adjacência

Em um grafo direcionado, a aresta (u, v) sai do vértice u e entra no vértice v. Por exemplo, na figura abaixo os arcos que saem do vértice 2 são (2, 2) e (2, 3), e os arcos que incidem sobre o vértice 2 são (1, 2) e (2, 2).

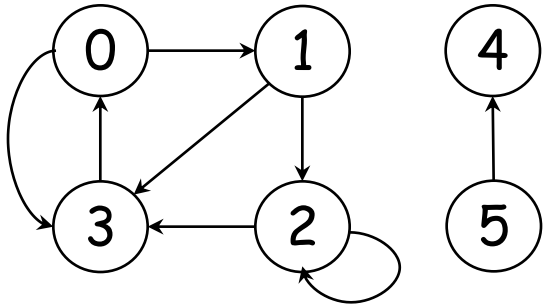
Se (u, v) é uma aresta no grafo $G = (V, A)$, o vértice v é **adjacente** ao vértice u. Quando o grafo é não direcionado, a relação de adjacência é simétrica.



Relação de adjacências:
0 é adjacente de 3
1 é adjacente de 0
2 é adjacente de 1 e 2
3 é adjacente de 0, 1 e 2
4 é adjacente de 5

Grau de um Vértice (1/2)

Em um grafo direcionado, o grau de um vértice corresponde ao número de arestas que saem do vértice (***out-degree***) somado ao número de arestas que chegam ao vértice (***in-degree***). Para o grafo a seguir:



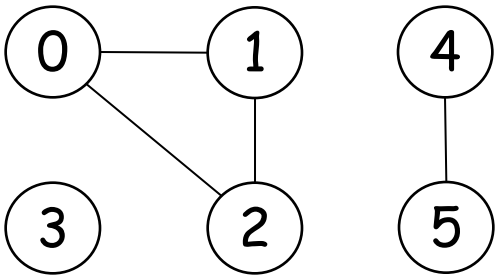
Tem-se:

Vértice	out-degree	in-degree	Grau do Vértice
0	1, 3 = 2	3 = 1	2 + 1 = 3
1	2, 3 = 2	0 = 1	2 + 1 = 3
2	2, 3 = 2	1, 2 = 2	2 + 2 = 4
3	0 = 1	0, 1, 2 = 3	1 + 3 = 4
4	nenhum	5 = 1	0 + 1 = 1
5	4 = 1	nenhum	1 + 0 = 1

Grau de um Vértice (2/2)

Em um grafo não direcionado, o grau de um vértice corresponde ao número de arestas que incidem nele.

Para o grafo a seguir:



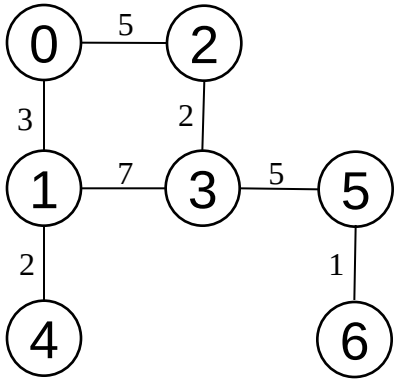
Tem-se:

Vértice	arestas	Grau do Vértice
0	(0, 1), (0, 2)	2
1	(1, 0), (1, 2)	2
2	(2, 0), (2, 1)	2
3	nenhuma	0 (isolado ou não conectado)
4	(4, 5)	1
5	(5, 4)	1

Tipo Abstrato de Dados Grafo (1/7)

Estrutura de dados mais um conjunto de operações associados.
Implementação por meio de "Listas de Adjacência" usando apontadores:

Grafo



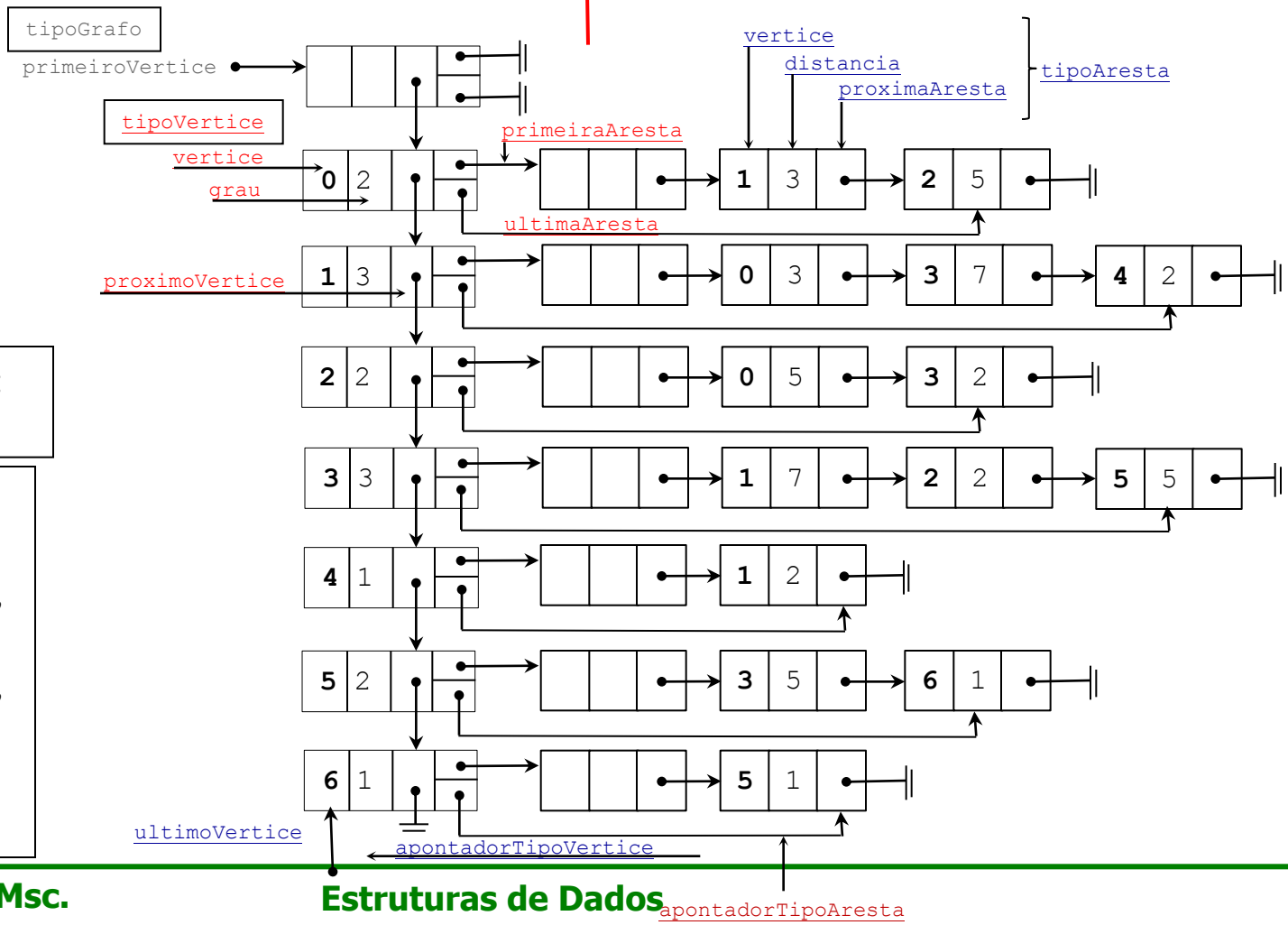
- conjunto de vértices:

$V = \{0, 1, 2, 3, 4, 5, 6\}$

- conjunto de arestas:

$A = \{(0, 1), (0, 2),$
$(1, 0), (1, 3), (1, 4),$
$(2, 0), (2, 3),$
$(3, 1), (3, 2), (3, 5),$
$(4, 1),$
$(5, 3), (5, 6),$
$(6, 5)\}$

Lista de vértices ⟷ Lista de adjacências (arestas)



Tipo Abstrato de Dados Grafo (2/7)

Estrutura de dados (**tipoGrafo.h**):

```
typedef struct tipoVertice *apontadorTipoVertice;
typedef struct tipoAresta *apontadorTipoAresta;

struct tipoVertice {
  int vertice;
  int grau;
  int visitado;
  apontadorTipoVertice proximoVertice;
  apontadorTipoAresta primeiraAresta;
  apontadorTipoAresta ultimaAresta;
};

struct tipoAresta {
  int vertice;
  int distancia;
  apontadorTipoAresta proximaAresta;
};

struct tipoGrafo {
  apontadorTipoVertice primeiroVertice;
  apontadorTipoVertice ultimoVertice;
};
```

Tipo Abstrato de Dados Grafo (3/7)

Conjunto de operação (**operGrafo.h**): (1/5)

```
// Cria a célula cabeça da lista de vértices:
// 1. cria uma célula do tipoVertice colocando o endereço no campo primeiroVertice do Grafo
// 2. incializa os campos do tipoVertice: vertice, grau, proximoVertice, primeiraAresta e
//    ultimaAresta com valores nulos
// 3. o apontador ultimoVertice recebe o endereço da célula cabeça
void fazGrafoVazio(struct tipoGrafo *grafo) {
  grafo->primeiroVertice = (apontadorTipoVertice)malloc(sizeof(tipoVertice));
  grafo->primeiroVertice->vertice = -1;
  grafo->primeiroVertice->grau = -1;
  grafo->primeiroVertice->proximoVertice = NULL;
  grafo->primeiroVertice->primeiraAresta = NULL;
  grafo->primeiroVertice->ultimaAresta = NULL;
  grafo->ultimoVertice = grafo->primeiroVertice;
}

// verifica se o Grafo está vazio (sem nenhum vértice)
int grafoVazio(struct tipoGrafo *grafo) {
  return(grafo->primeiroVertice == grafo->ultimoVertice);
}

// retorna o endereço do "vertice" (NULL se o vértice não for localizado)
apontadorTipoVertice localizaVertice(int vertice, struct tipoGrafo *grafo) {
  apontadorTipoVertice p = grafo->primeiroVertice->proximoVertice;
  while ((p != NULL) && (p->vertice != vertice)) {
    p = p->proximoVertice;
  }
  return(p);
}
```

Tipo Abstrato de Dados Grafo (4/7)

Conjunto de operação (**operGrafo.h**): (2/5)

```
// Cria a célula cabeça da lista de adjacências do vértice:
// 1. cria uma célula do tipoAresta colocando o endereço no campo primeiroAresta do vértice
// 2. incializa os campos do tipoAresta: vertice, distancia e proximaAresta com valores nulos
// 3. o apontador ultimoAresta recebe o endereço da célula cabeça
void fazListaAdjacenciasVazia(apontadorTipoVertice vertice) {
  vertice->primeiraAresta = (apontadorTipoAresta)malloc(sizeof(tipoAresta));
  vertice->primeiraAresta->vertice = -1;
  vertice->primeiraAresta->distancia = -1;
  vertice->primeiraAresta->proximaAresta = NULL;
  vertice->ultimaAresta = vertice->primeiraAresta;
}

// verifica se a lista de adjacências do vértice está vazia
int listaAdjacenciasVazia(apontadorTipoVertice vertice) {
  return(vertice->primeiraAresta == vertice->ultimaAresta);
}

// retorna o endereço da "aresta" na lista de adjacências do vértice (NULL se a aresta
// não for localizada)
apontadorTipoAresta localizaAresta(apontadorTipoVertice vertice, int aresta) {
  apontadorTipoAresta p = vertice->primeiraAresta->proximaAresta;
  while ((p != NULL) && (p->vertice != aresta)) {
    p = p->proximaAresta;
  }

  return(p);
}
```

Tipo Abstrato de Dados Grafo (5/7)

Conjunto de operação (**operGrafo.h**): (3/5)

```
// Insere um novo vértice no final da lista de vértices:
// 1. cria uma nova célula colocando-a no final da lista (campo ultimoVertice)
// 2. atualiza o apontador ultimoVertice
// 3. armazena o valor do vértice no campo correspondente
// 4. inicializa o grau do vértice com o valor zero
// 5. faz a nova célula através do campo proximoVertice apontar para o vazio
// 6. faz a lista de adjacências do novo vértice vazia
void insereVertice(int vertice, struct tipoGrafo *grafo) {
  grafo->ultimoVertice->proximoVertice = (apontadorTipoVertice)malloc(sizeof(tipoVertice));
  grafo->ultimoVertice = grafo->ultimoVertice->proximoVertice;
  grafo->ultimoVertice->vertice = vertice;
  grafo->ultimoVertice->grau = 0;
  grafo->ultimoVertice->proximoVertice = NULL;
  fazListaAdjacenciasVazia(grafo->ultimoVertice);
}

// Insere uma nova aresta no final da lista de adjacências do vértice de origem:
// 1. cria uma nova célula colocando-a no final da lista de adjacências (campo ultimaAresta)
// 2. atualiza o apontador ultimaAresta
// 3. armazena o valor do vértice destino da aresta no campo correspondente
// 4. armazena a distância da aresta
// 5. faz a nova célula através do campo proximoAresta apontar para o vazio
// 6. atualiza o grau do vértice de origem
void insereAresta(apontadorTipoVertice verticeOrigem, int verticeDestino, int distancia) {
  verticeOrigem->ultimaAresta->proximaAresta = (apontadorTipoAresta)malloc(sizeof(tipoAresta));
  verticeOrigem->ultimaAresta = verticeOrigem->ultimaAresta->proximaAresta;
  verticeOrigem->ultimaAresta->vertice = verticeDestino;
  verticeOrigem->ultimaAresta->distancia = distancia;
  verticeOrigem->ultimaAresta->proximaAresta = NULL;

  verticeOrigem->grau = verticeOrigem->grau + 1;
}
```

Tipo Abstrato de Dados Grafo (6/7)

Conjunto de operação (**operGrafo.h**): (4/5)

No processo para remover uma célula em listas encadeadas com apontadores é realizado usando como referência o endereço da célula "anterior" àquela que efetivamente será removida.

```
// retorna o endereço da célula "anterior" a aresta do vértice que será removida
apontadorTipoAresta localizaArestaAnterior(apontadorTipoVertice vertice, int aresta) {
  apontadorTipoAresta pAnt = vertice->primeiraAresta;
  apontadorTipoAresta p = vertice->primeiraAresta->proximaAresta;
  while ((p != NULL) && (p->vertice != aresta)) {
    pAnt = p;
    p = p->proximaAresta;
  }

  return(pAnt);
}

// remove a "próxima" aresta de "p" da lista de adjacência de "pVertice":
// 1. "q" recebe o endereço da aresta que será devolvido ao sistema de gerenciamento de memória
// 2. a aresta anterior aponta para a próxima aresta àquela que será removida
// 3. verifica se a aresta que será removida não é a última aresta da lista de adjacências
// 4. atuliza o grau do vértice
// 5. libera (free) o endereço da aresta removida
void removeAresta(apontadorTipoAresta p, apontadorTipoVertice pVertice) {
  apontadorTipoAresta q = p->proximaAresta;

  p->proximaAresta = q->proximaAresta;
//  se retirando a última aresta da lista de adjacencias
  if (p->proximaAresta == NULL)
     pVertice->ultimaAresta = p;

  Vertice->grau = pVertice->grau - 1;

  free(q); // libera a memória ocupada pela aresta
}
```

Tipo Abstrato de Dados Grafo (7/7)

Conjunto de operação (**operGrafo.h**): (5/5)

```
// retorna o endereço da célula "anterior" ao vértice que será removido
apontadorTipoVertice localizaVerticeAnterior(int vertice, struct tipoGrafo *grafo) {
  apontadorTipoVertice pAnt = grafo->primeiroVertice;
  apontadorTipoVertice p = grafo->primeiroVertice->proximoVertice;
  while ((p != NULL) && (p->vertice != vertice)) {
    pAnt = p;
    p = p->proximoVertice;
  }

  return(pAnt);
}

// remove o "próximo" vértice de "p" da lista de vértices do Grafo:
void removeVertice(apontadorTipoVertice p, struct tipoGrafo *grafo) {
  apontadorTipoVertice verticeDestino, verticeOrigem = p->proximoVertice;

// processo para remover "todas" as arestas do vértice
  apontadorTipoAresta arestaDois, arestaUm = verticeOrigem->primeiraAresta;
  while (!listaAdjacenciasVazia(verticeOrigem)) {
    verticeDestino = localizaVertice(arestaUm->proximaAresta->vertice, grafo);
    arestaDois = localizaArestaAnterior(verticeDestino, verticeOrigem->vertice);
    removeAresta(arestaUm, verticeOrigem);
    removeAresta(arestaDois, verticeDestino);
  }
  free(verticeOrigem->primeiraAresta); // remove a célula cabeça da lista de adjacências

  p->proximoVertice = verticeOrigem->proximoVertice;
//  verifice se o vértice removido é o último da lista de vértices
  if (p->proximoVertice == NULL)
     grafo->ultimoVertice = p;

  free(verticeOrigem); // libera a memória ocupada pela aresta
}
```

Caminhamento em Grafos

Problema:
dado um vértice *v* de um grafo *G*=(*V*, *A*),

Objetivo:
deseja-se caminhar entre "todos" os vértices *G* que são "alcançáveis" a partir de *v*.

Formas principais de caminhamento:
1) Busca em Profundidade
   (DFS- *Depth-First Search*)

2) Busca em Largura
   (BFS- *Breadth-First Search*)

DFS- Busca em Profundidade (1/3)

Buscar o vértice mais "profundo" sempre que possível. Na busca em profundidade, as arestas são exploradas a partir do vértice mais recentemente visitado.

Algoritmo:

1. visitar *v* e marcá-lo como "visitado";
2. percorrer "todas" as arestas *v*:
   2.1 fazer busca em profundidade no vértice da aresta;
   2.2 posicionar na próxima aresta de *v*.

Nota:

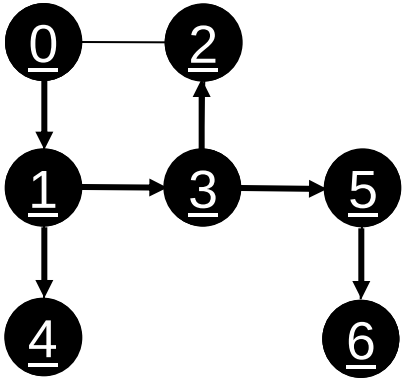
O algoritmo DFS é recurssivo, ou seja, quando todas as arestas adjacentes tiverem sido exploradas, a busca anda para trás (*backtrack*) para explorar as outras arestas do vértice de origem.

DFS- Busca em Profundidade (2/3)

```
// percorre a lista de vértice do grafo alterando o status do campo "visitado" do vértice
// para falso (valor inteiro 0)
void buscaEmProfundidade(apontadorTipoVertice p, struct tipoGrafo *grafo) {
  apontadorTipoVertice aux = grafo->primeiroVertice->proximoVertice;
  while (aux != NULL) {
    aux->visitado = 0;
    aux = aux->proximoVertice;
  }

  DFS(p, grafo);
}
// DFS- Busca em Profundidade
void DFS(apontadorTipoVertice p, struct tipoGrafo *grafo) {
  apontadorTipoVertice aux;

  printf("%d ", p->vertice);
  p->visitado = 1; // marca o vértice como "visitado"

// percorre "todas" as arestas do vértice apontado por "p"
  apontadorTipoAresta aresta = p->primeiraAresta->proximaAresta;
  while (aresta != NULL) {
// se o vértice ainda não foi visitado aplicar a busca em profundidade no vértice da aresta
    aux = localizaVertice(aresta->vertice, grafo);
    if (aux->visitado == 0)
      DFS(aux, grafo); // aplica a DFS no vértice da aresta

    aresta = aresta->proximaAresta;
  }
}
```

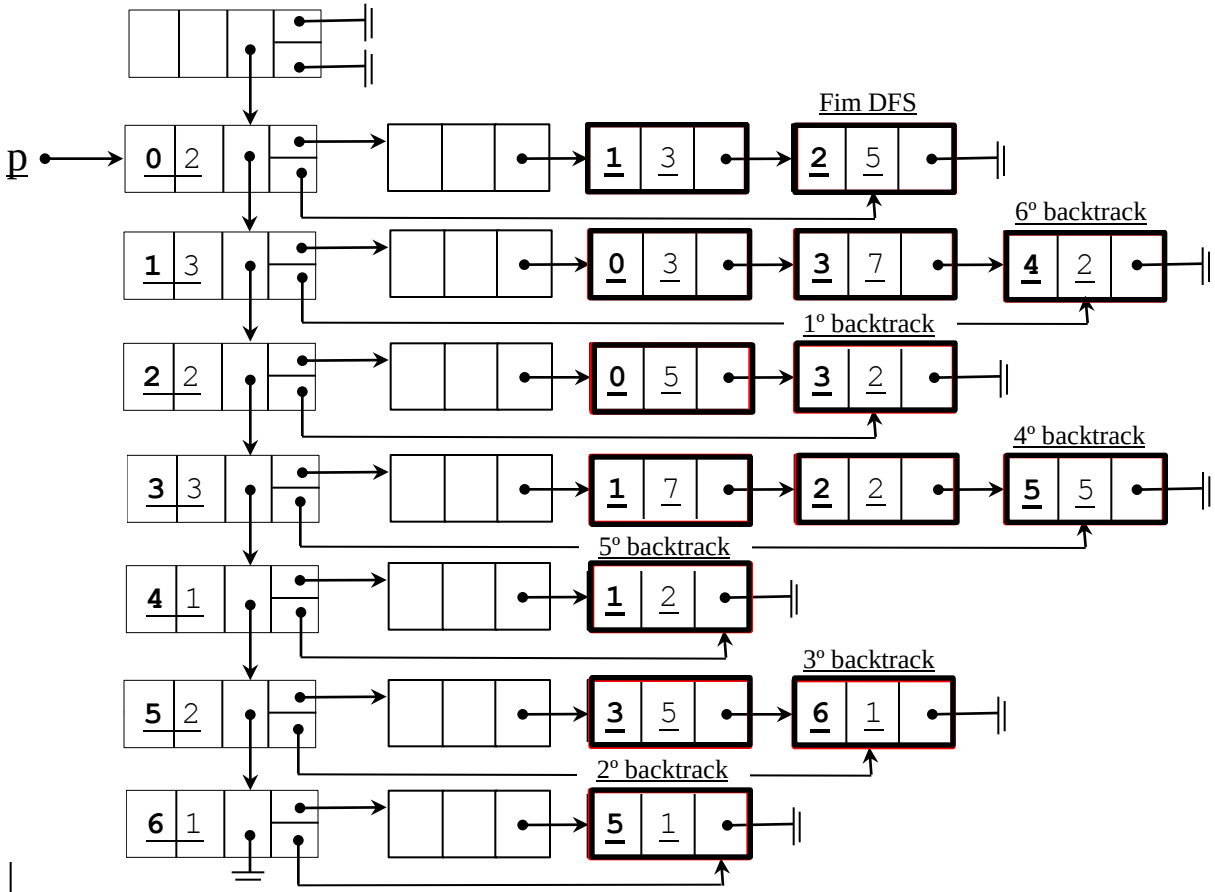
DFS- Busca em Profundidade (3/3)

Grafo



DFS a partir do vértice 0:

0 1 3 2 5 6 4



Legenda da visitação:

Branco: vértice não visitado

Cinza: visitação em andamento

Preto: visitação concluída