UNIVERSIDADE FEDERAL DO MARANHÃO - UFMA

Estruturas de Dados II

Grafos

Portela

Bacharelado em Ciência da Computação

Grafos

- Aula 9
  - Representação de grafos
    - Matriz de Adjacência
    - Listas de Adjacência
  - Busca em Largura - BSF
  - Busca em Profundidade - DFS

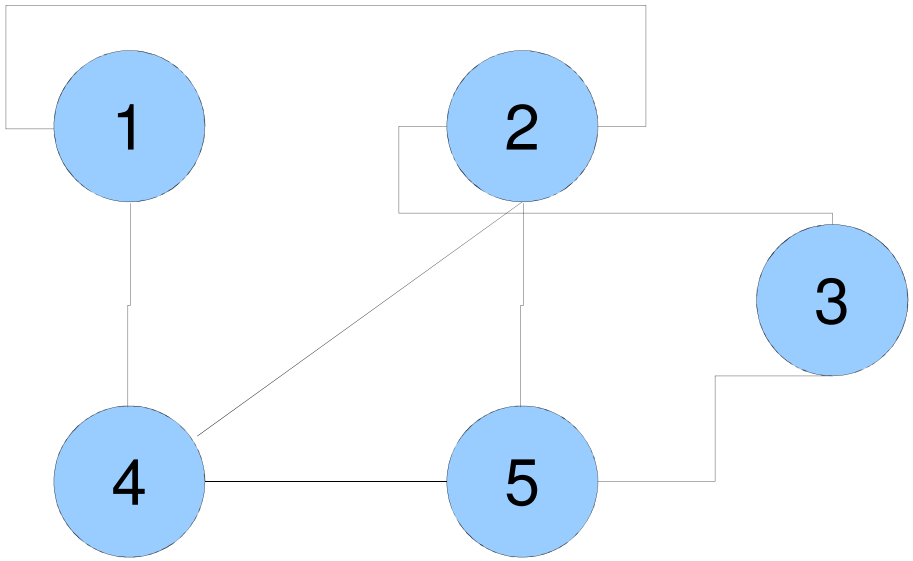
Grafos

- É uma estrutura de dados G, definida como G={V, E}, onde:
  - V = conjunto de nós / vértices
    - Servem para modelar elementos de um problema
  - E = conjunto de arcos / arestas
    - Cada vértice liga um par de arestas
    - Podem ser ponderadas ou não
    - Podem ser direcionadas ou não

Representação de Grafos

- Matriz de Adjacência
  - Grafos não-ponderados
    - Uma matriz booleana com |V|x|V| elementos indica se há uma aresta entre qualquer par u, v (**u e v são nós do grafo)**
  - Grafos ponderados
    - A matriz armazena o peso de cada aresta
- Listas de Adjacência
    - Para cada nó do grafo, há um vetor indicando seus vizinhos

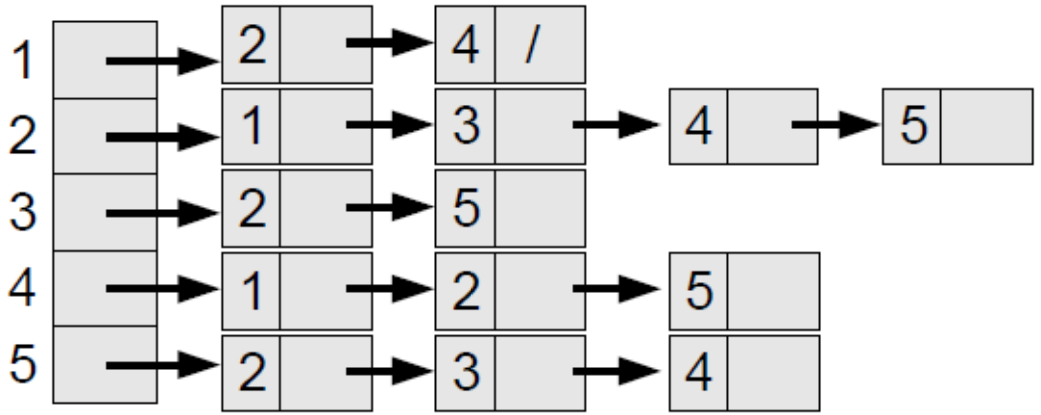
Exemplo de Grafo



Matriz de Adjacência

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

Listas de Adjacência



Matriz de Adjacência

```
function createGraph(N)
      local g = {}
      for i=1,N do
         g[i] = {}
         for j=1,N do
            g[i][j] = false
         end
      end
      return g
   end
function createEdge(G, u, v)
      G[u][v] = true
   end
   function hasEdge(G, u, v)
      return G[u][v]
end
```

Listas de Adjacência

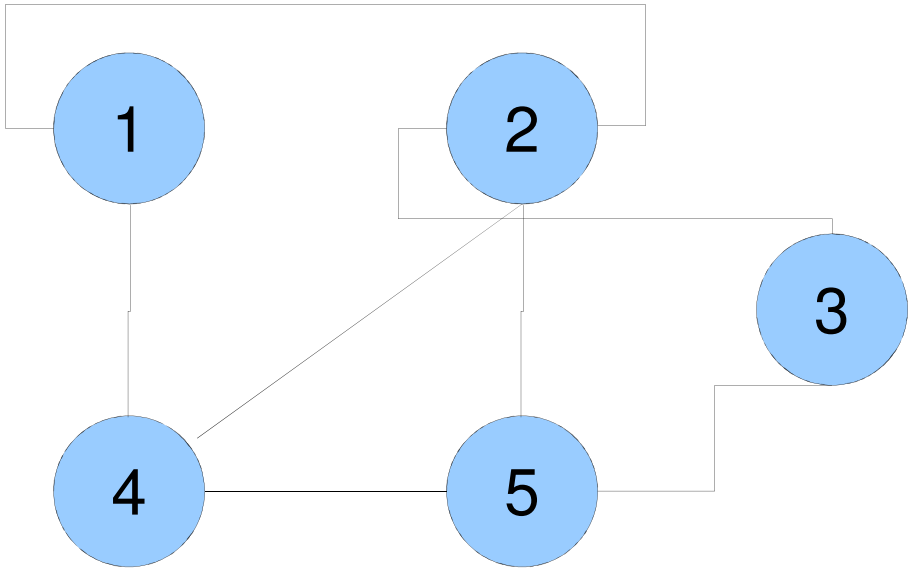
```
function createGraph(N)
      local g = {dg={}, adj={}}
      for i=1,N do g.dg[i]=0; g.adj[i]={} end
      return g
  end
function createEdge(G, u, v)
      G.dg[u] = G.dg[u]+1
      G.adj[u][ G.dg[u] ] = v
      G.dg[v] = G.dg[v]+1
      G.adj[v][ G.dg[v] ] = u
  end
function hasEdge(G, u, v)
      for i=1,G.dg[u] do
         if G.adj[u][i]==v then return true end
      end
      return false
  end
```

Busca em Largura

- Percorre todos os vértices de um grafo G a partir de um vértice de origem ***s até descobrir cada vértice acessível a partir de s***
- Visita primeiro os vértices mais próximos de ***s***
- Em inglês: BFS (*breadth first search*)

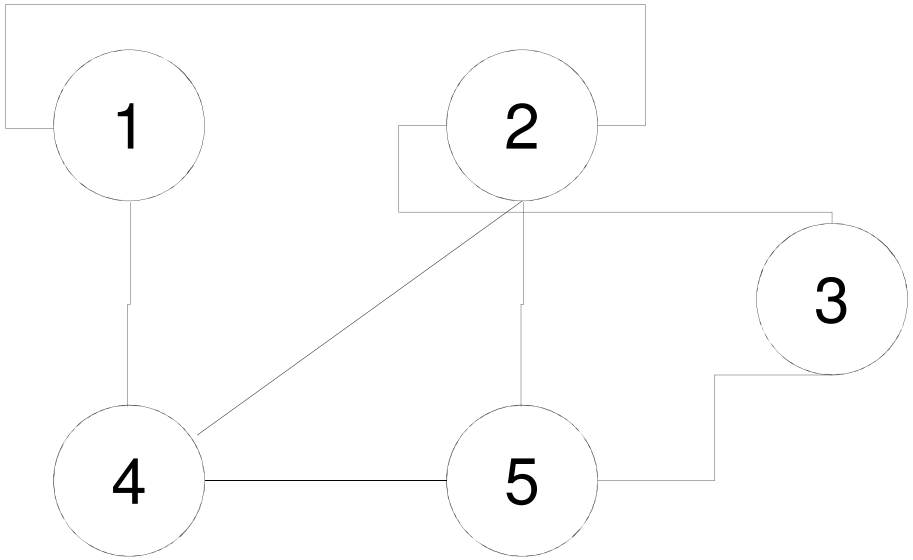
Busca em Largura

- Inicia pintando todos de branco (não-visitados)



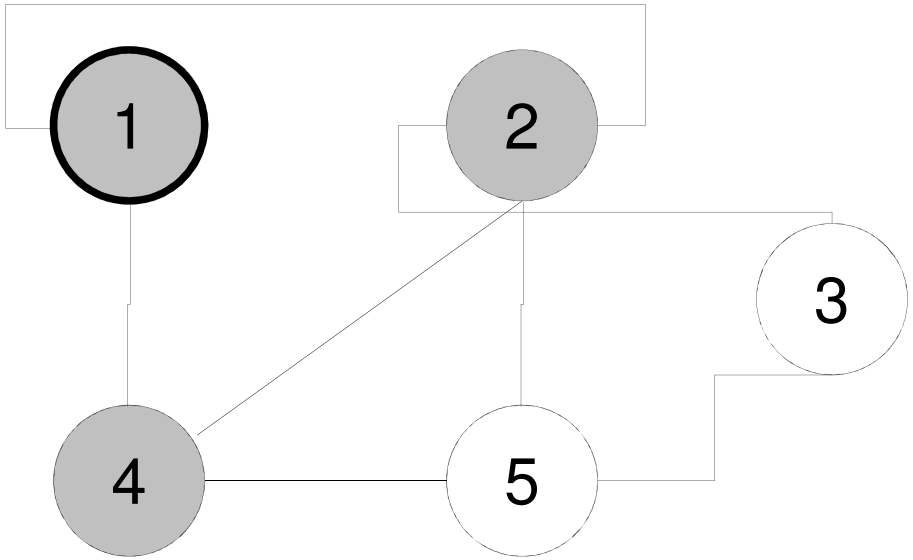
Busca em Largura

- Suponha o vértice inicial ***s = 1***



Busca em Largura

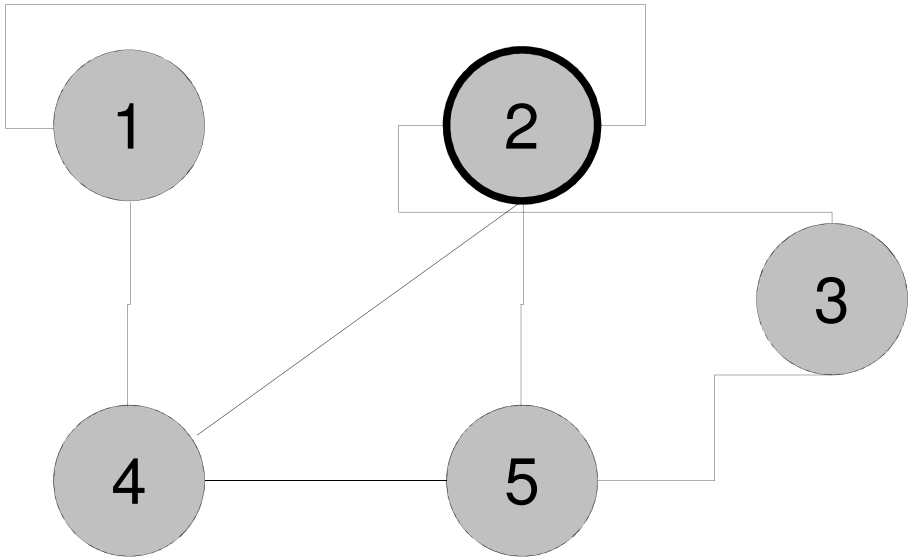
- BFS(1)



- Fila: 2 | 4

Busca em Largura

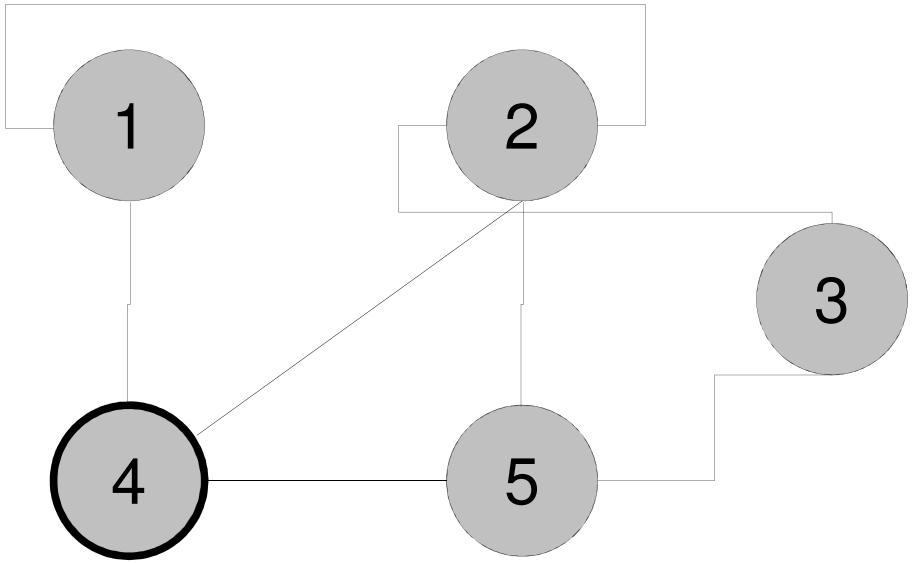
- BFS(2)



- Fila: 4 | 3 | 5

Busca em Largura

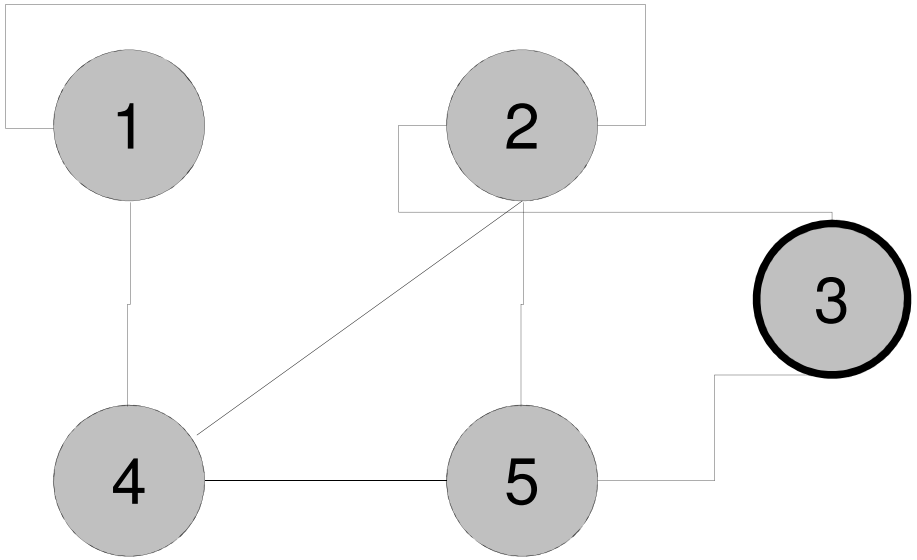
- BFS(4)



- Fila: 3 | 5

Busca em Largura

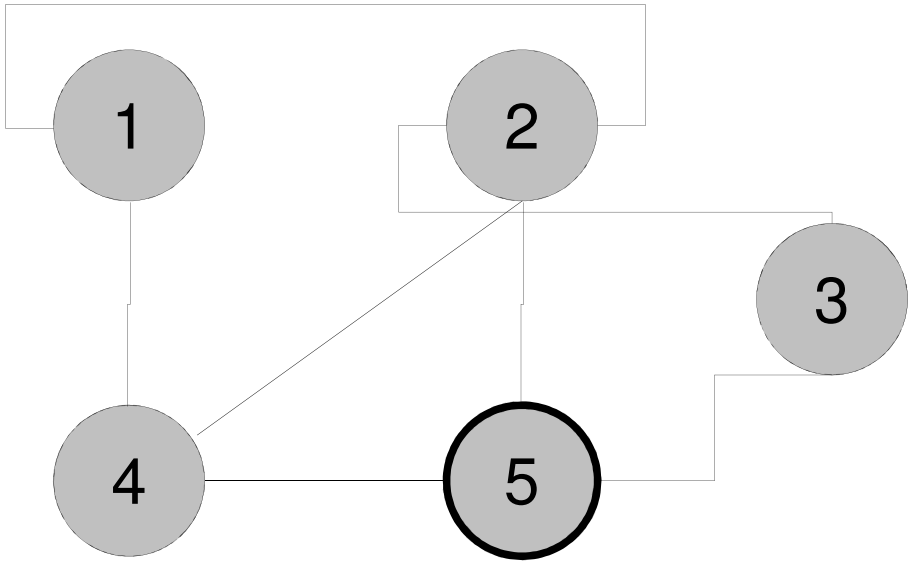
- BFS(3)



- Fila: 5

Busca em Largura

- BFS(5)



- Fila: /

Busca em Largura

```
function BFS(G, s)
      local WHITE, GRAY = 1,2
      local color = {}
      for i=1,#G.dg do color[i]=WHITE end
      Q = createQueue()
      insertQueue(Q, s)
      color[s] = GRAY
      while not emptyQueue(Q) do
         u = removeQueue(Q)
         for i=1,G.dg[u] do
            v = G.adj[u][i]
            if color[v]==WHITE then
               insertQueue(v)
               color[v] = GRAY
            end
         end
      end
   end
```

Busca em Largura

A idéia da busca em largura é bastante simples: os vértices do grafo são visitados nível a nível, ou seja, todos os vértices a uma distância k do vértice inicial são visitados antes de qualquer vértice a uma distância $k+1$ do inicial.

Busca em Profundidade

- Começa num nó raiz (selecionando algum nó como sendo o raiz, no caso de um grafo) e explora tanto quanto possível cada um dos seus ramos, antes de retroceder (backtracking).

- Em inglês: DFS (*depth first search*)

Busca em Profundidade

- A estratégia é buscar o mais profundo no grafo sempre que possível.

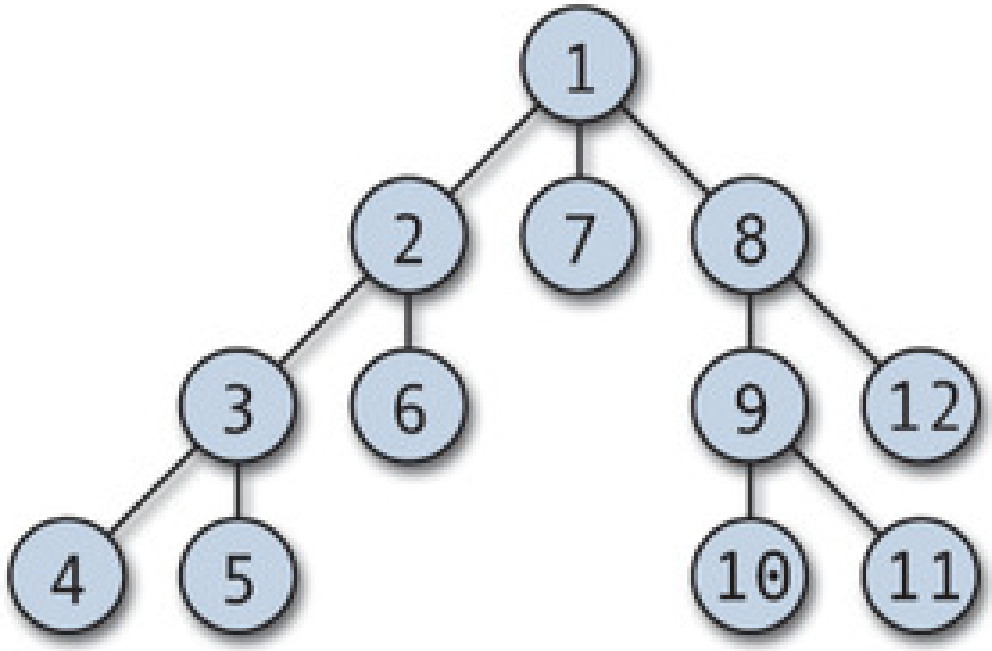
- As arestas são exploradas a partir do vértice v mais recentemente descoberto que ainda possui arestas não exploradas saindo dele.

Busca em Profundidade

- Quando todas as arestas adjacentes a v tiverem sido exploradas a busca anda para trás para explorar vértices que saem do vértice do qual v foi descoberto.

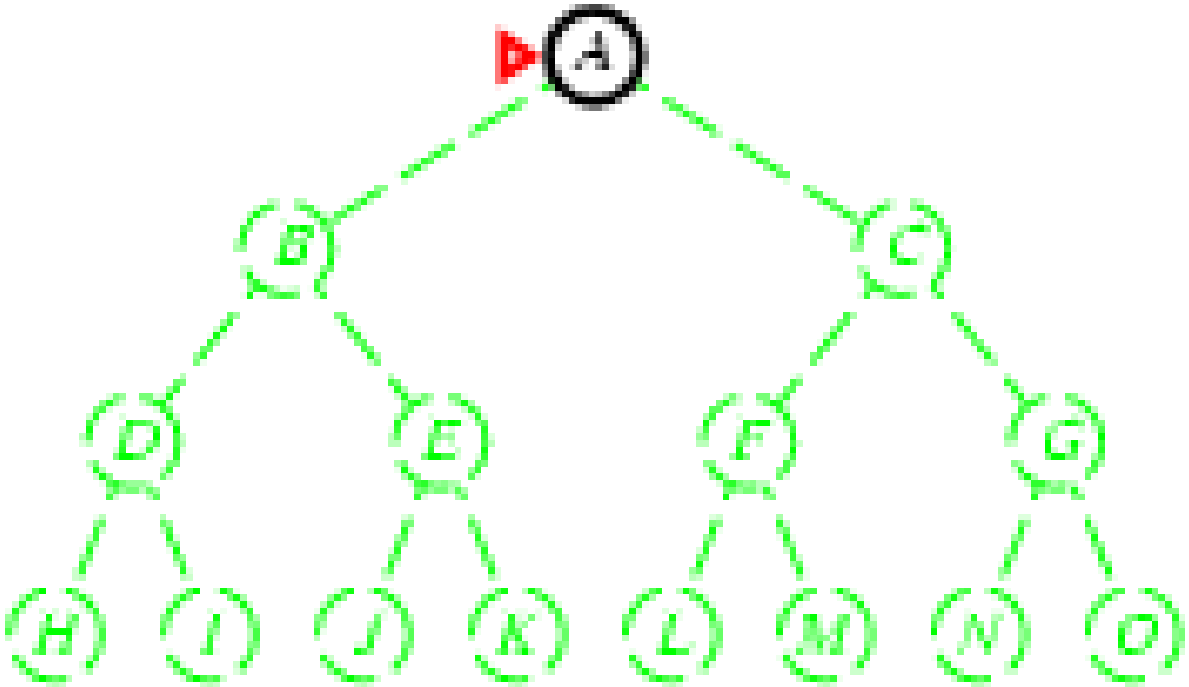
- O algoritmo é a base para muitos outros algoritmos importantes, tais como verificação de grafos acíclicos, ordenação topológica e componentes fortemente conectados.

Busca em Profundidade

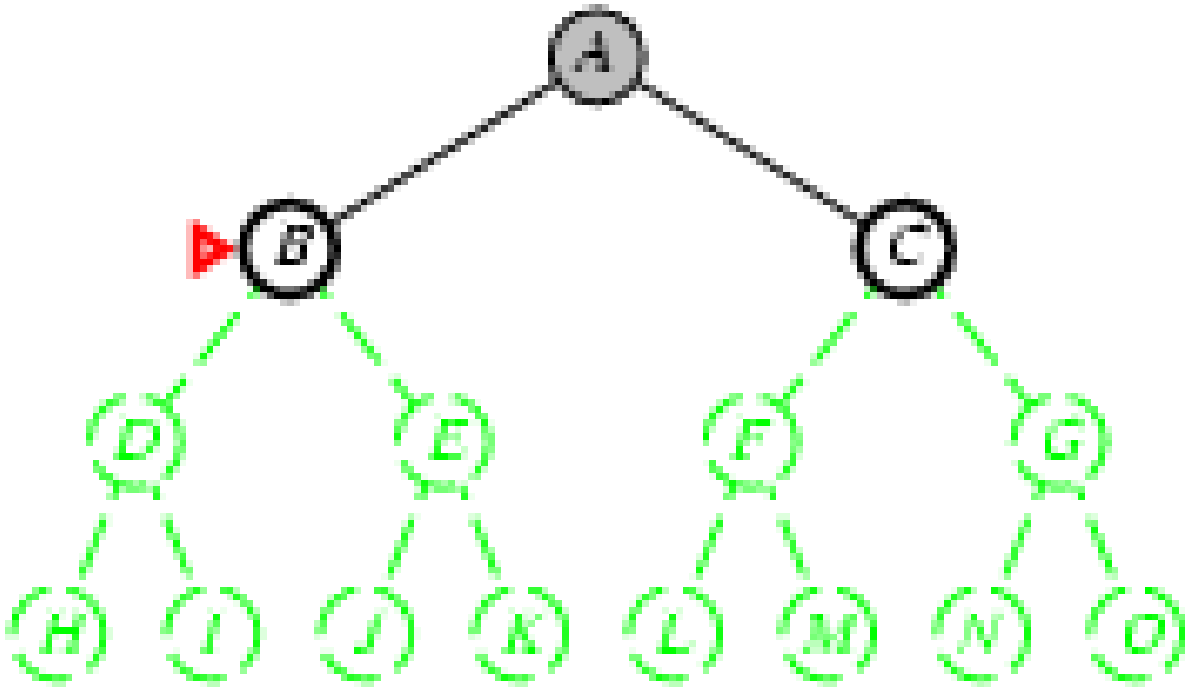


Ordem dos vértices explorados na busca em profundidade

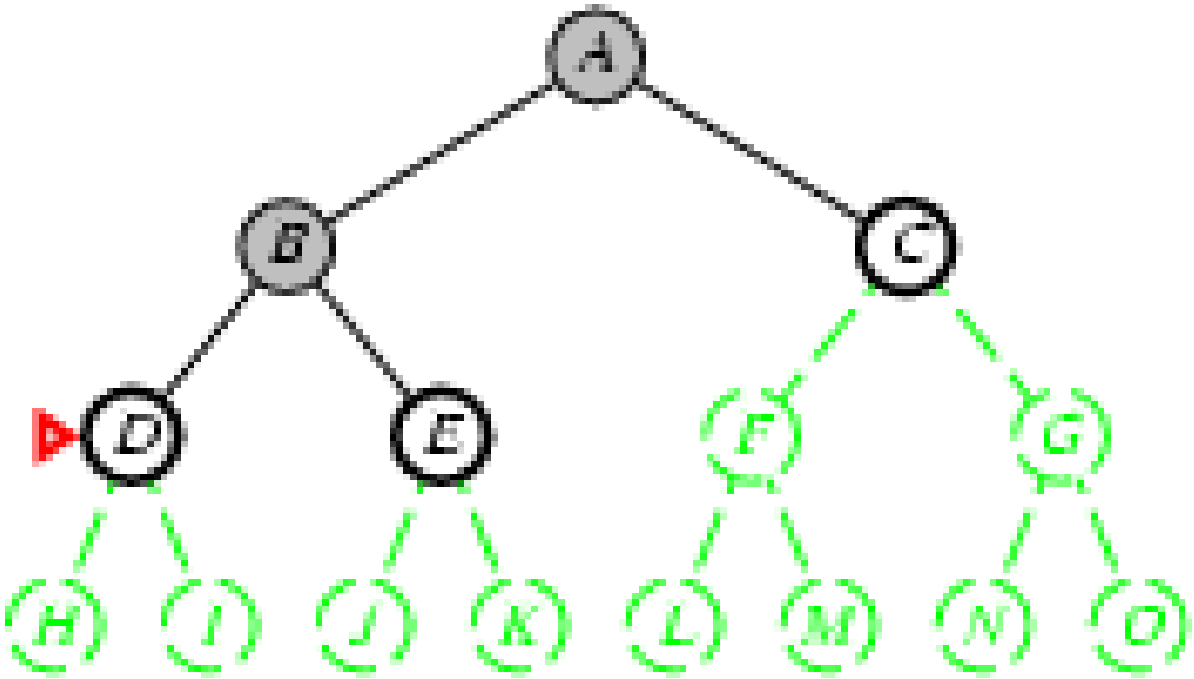
Busca em Profundidade



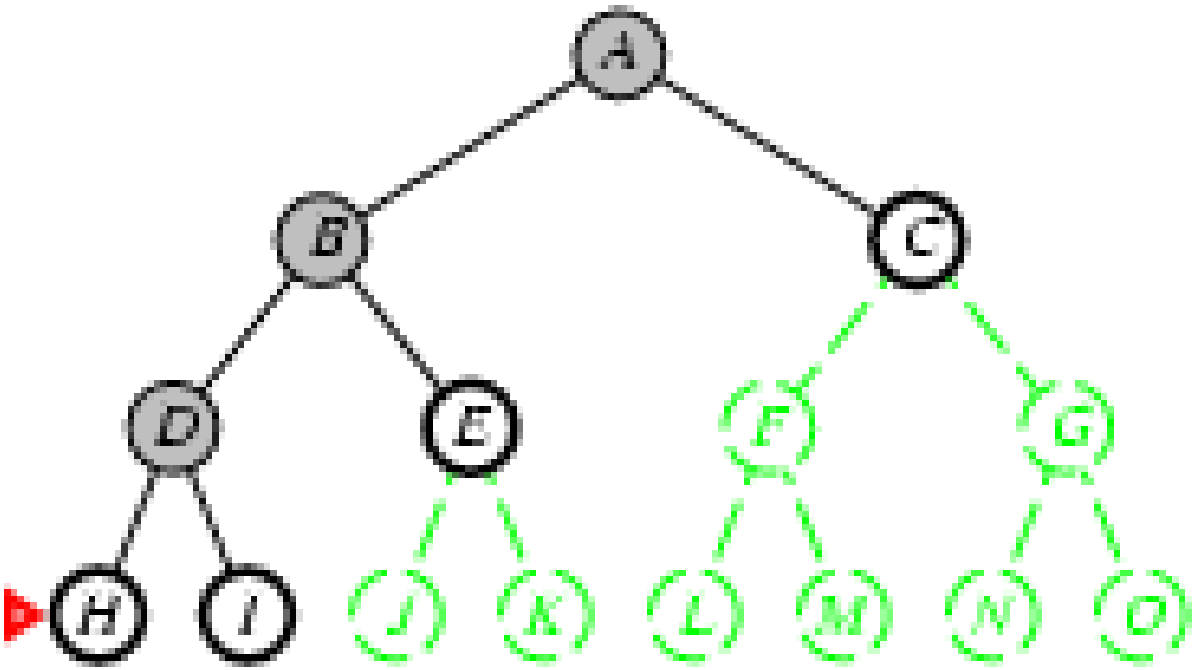
Busca em Profundidade



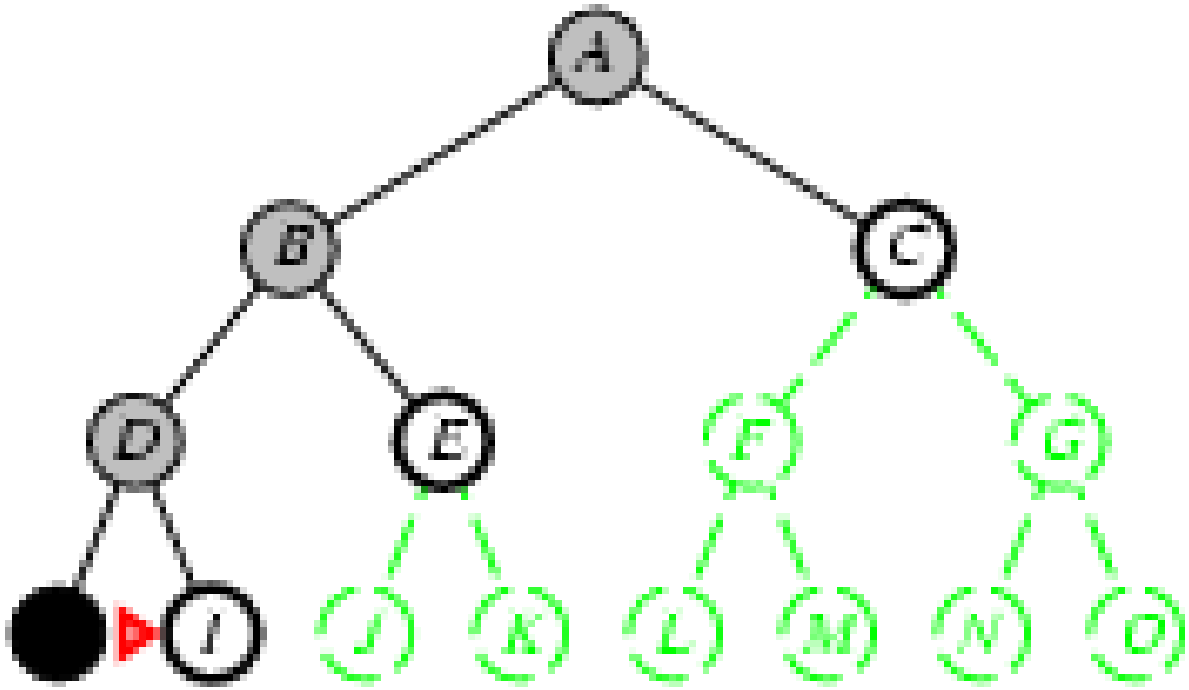
Busca em Profundidade



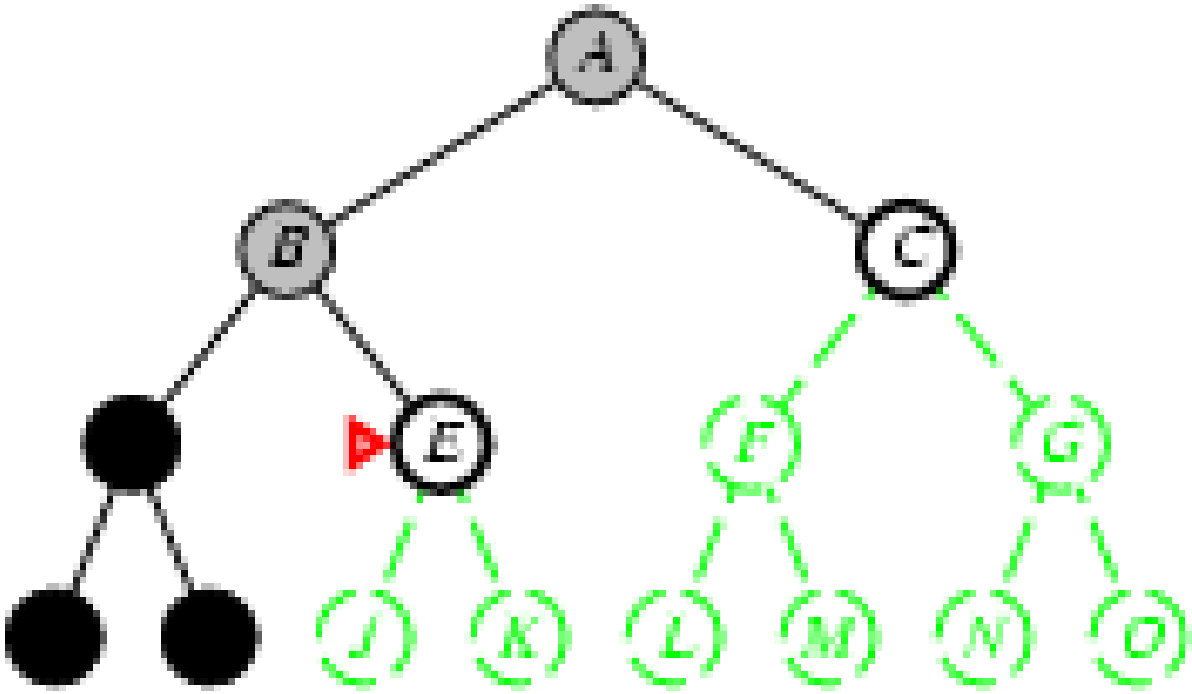
Busca em Profundidade



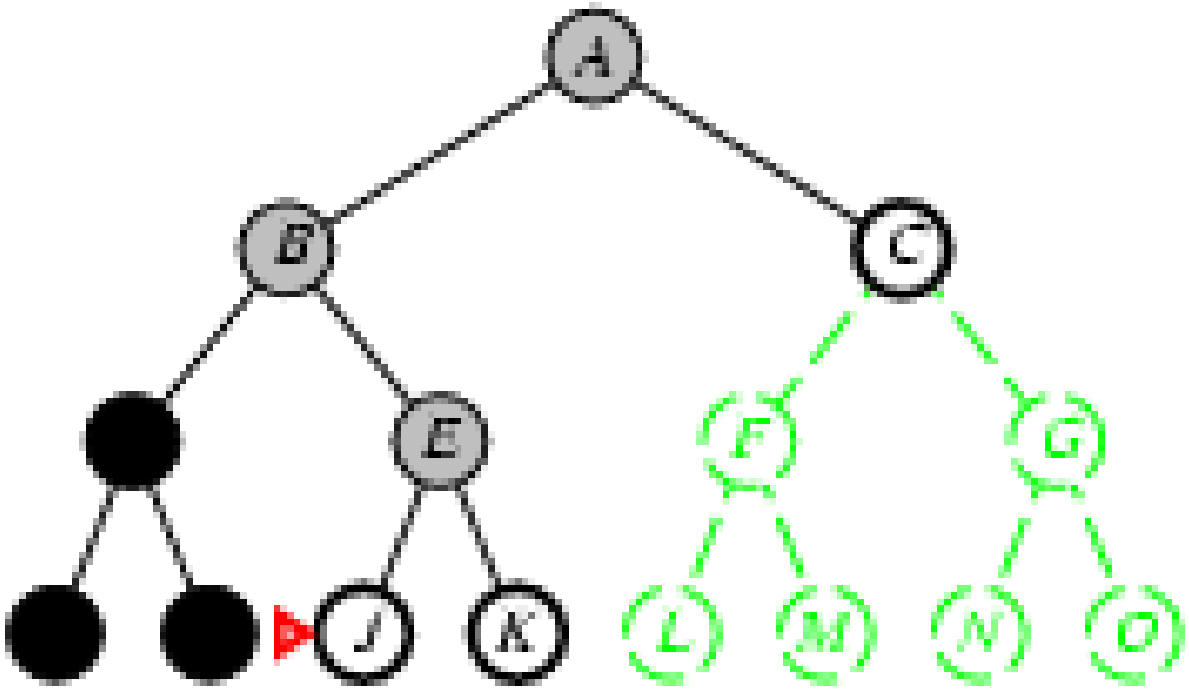
Busca em Profundidade



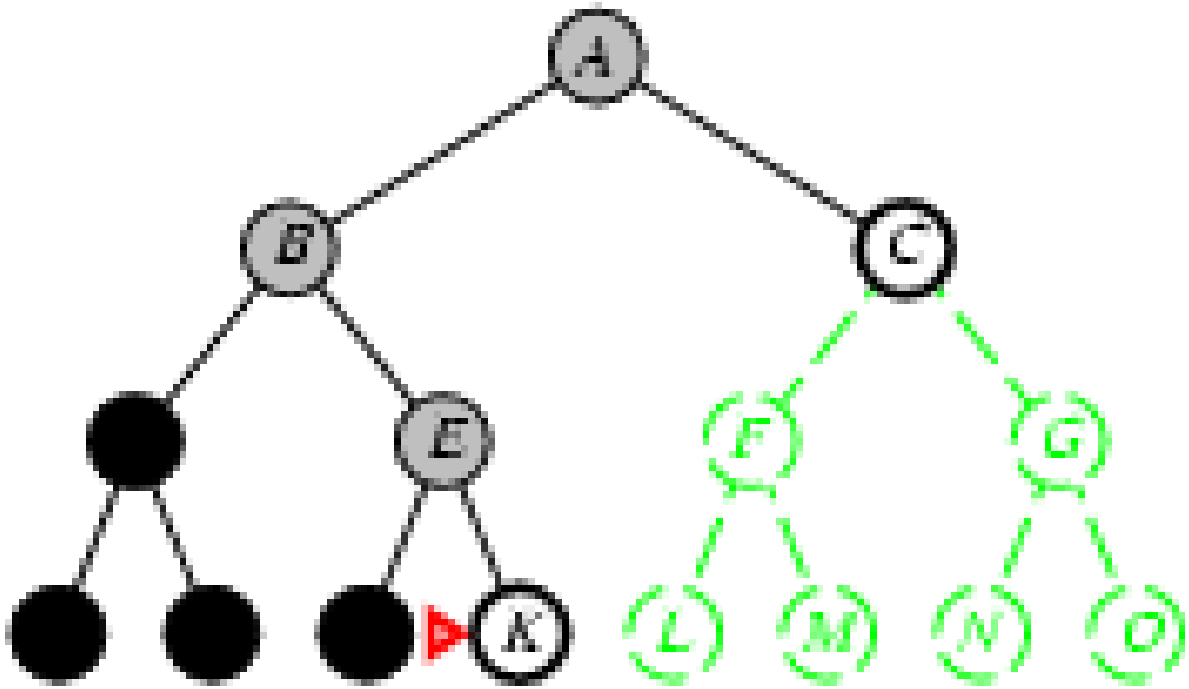
Busca em Profundidade



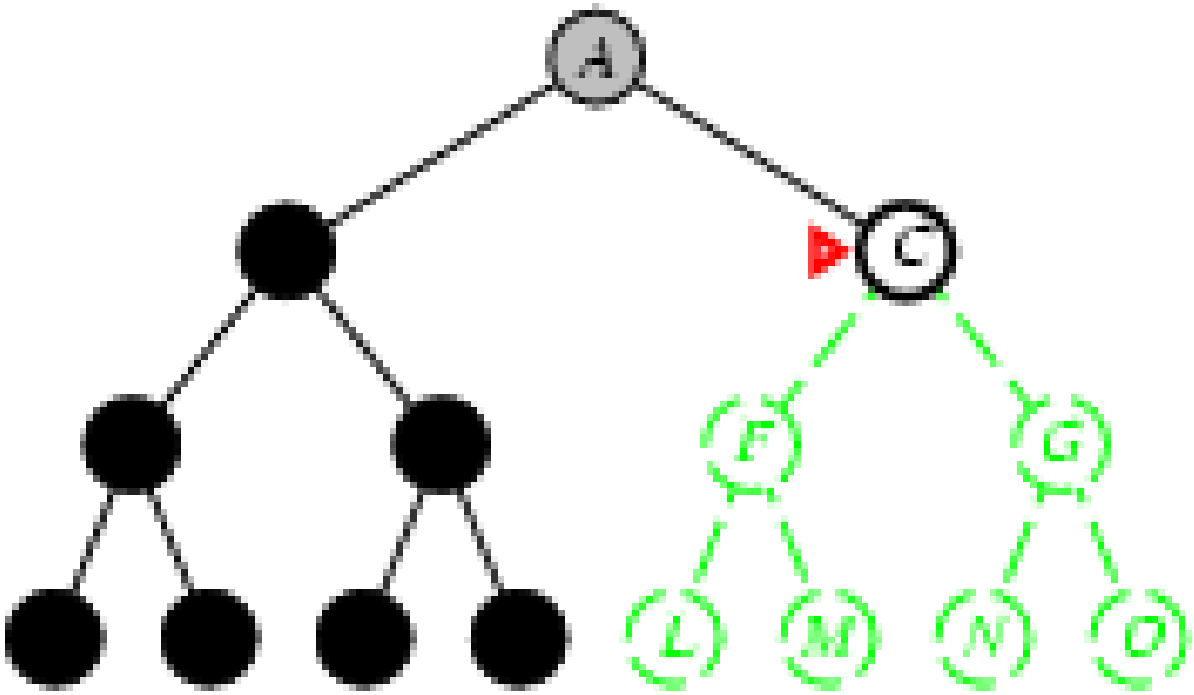
Busca em Profundidade



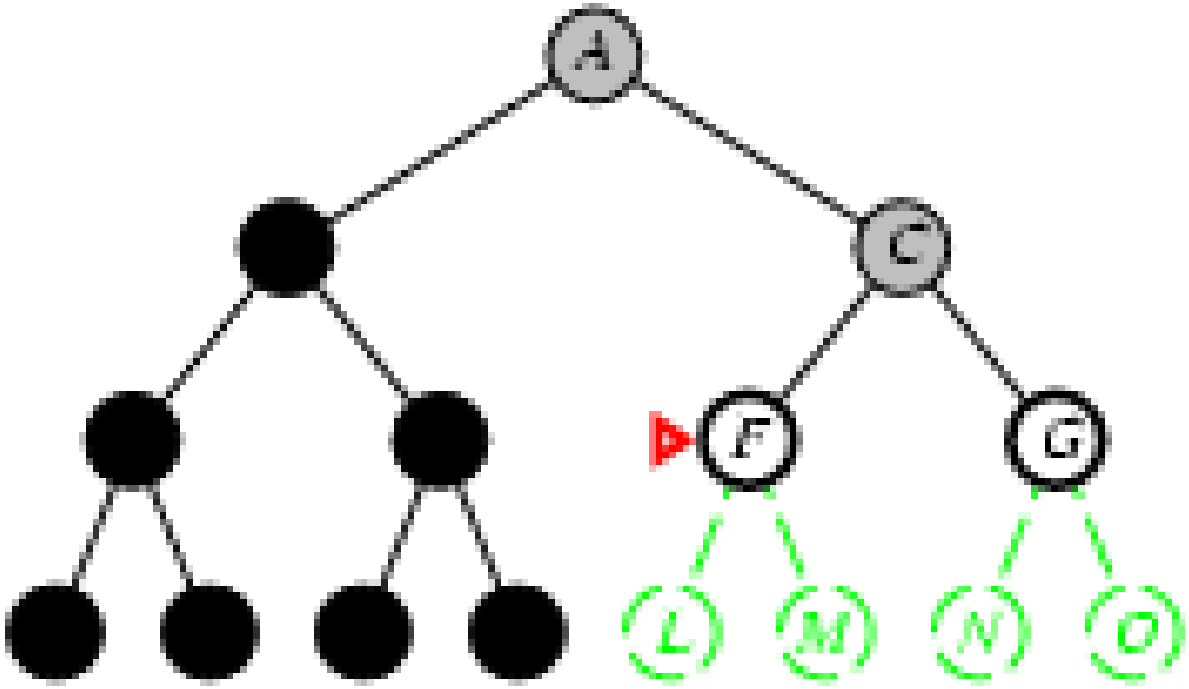
Busca em Profundidade



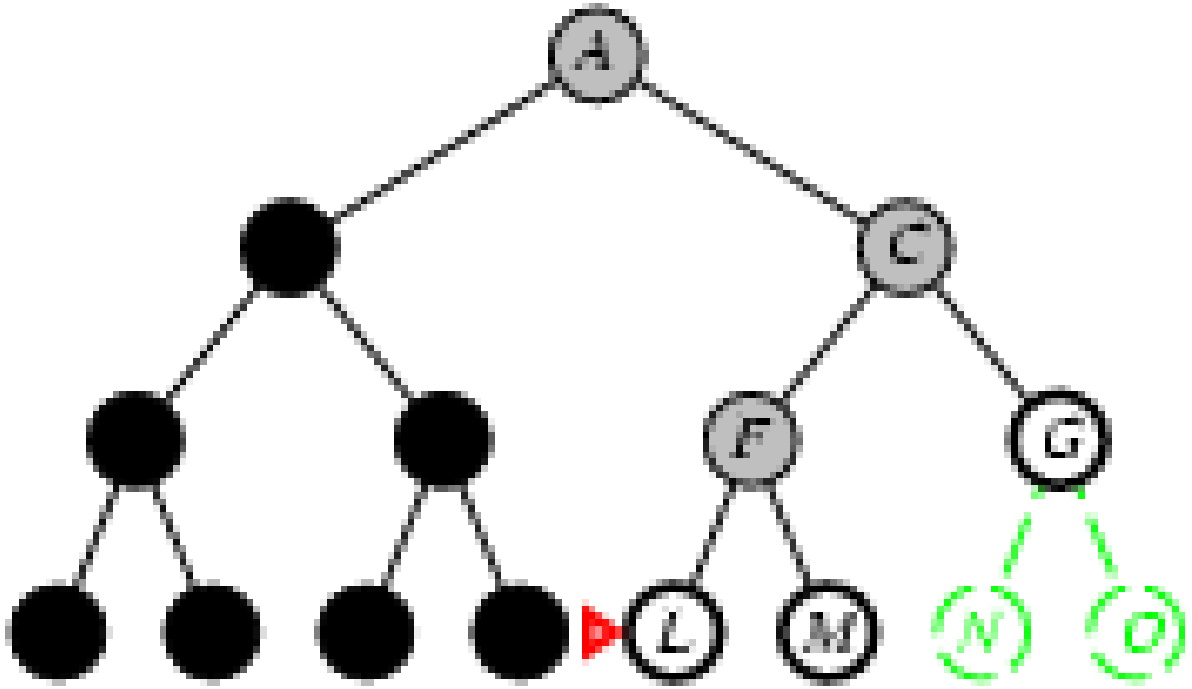
Busca em Profundidade



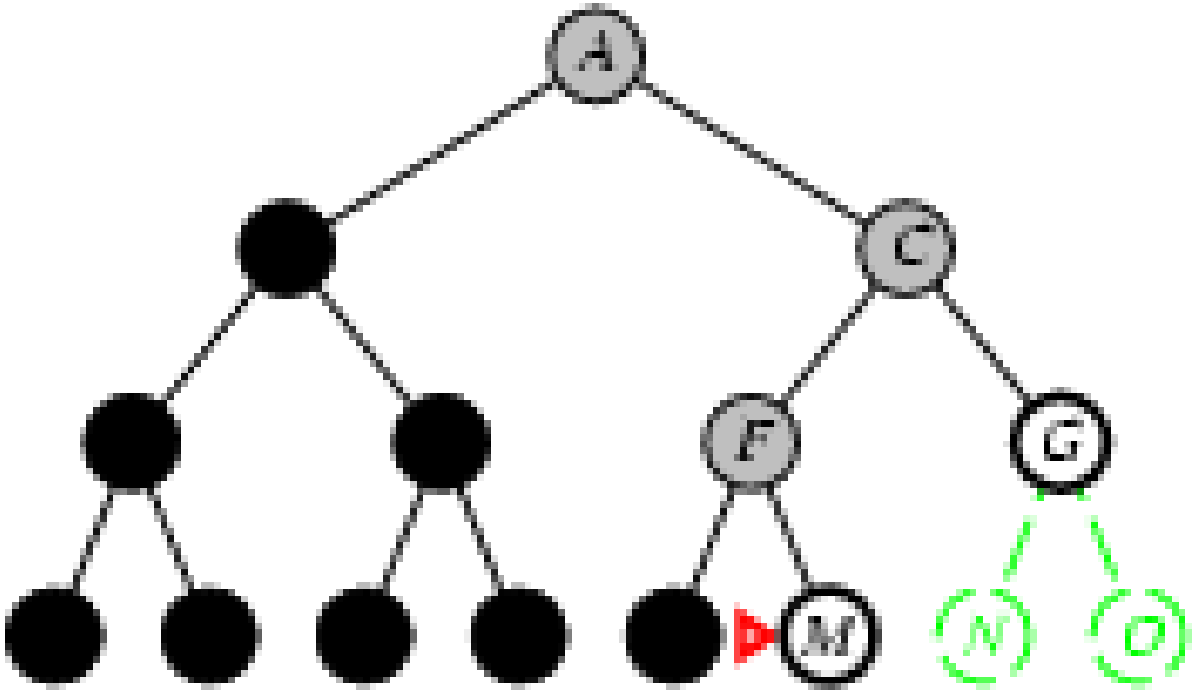
Busca em Profundidade



Busca em Profundidade



Busca em Profundidade



Busca em Profundidade

```
Algoritmo DFS(G: grafo)
  para cada  vértice u de G faça
    cor[u]← branco
    pred[u]=←-1
  fim para
  para cada vertice u de G faça
    se (cor[u] = branco) entao
      visita(u)
    fim se
  fim para
```

```
Visita(u: vertice)
  cor[u] ← cinza
  tempo ← tempo +1
  d[u]← tempo
  para cada v adjacente a u faça
    se cor[v] = branco entao
        pred[v]←u
        visita(v)
    fim se
  cor[u]←preto
  tempo←tempo+1
  t[u]← tempo
fim Visita
```

Busca em Profundidade

A ideia básica da busca em profundidade é buscar "mais a fundo" no grafo quando possível. Assim, a partir de um vértice *v*, as arestas ainda não exploradas o são e, ao final, a busca retorna ao vértice *w (*essa volta é também chamada de backtracking), que levou ao descobrimento de *v* pela aresta (*w; v*) e explora suas arestas ainda não visitadas. Assim a busca continua até que todos os vértices sejam descobertos.