# Métodos de Busca

**Solange Rezende**
Departamento de Ciências de Computação
ICMC-USP, São Carlos
solange@icmc.usp.br

Neste tema são descritos: formulação do problema de busca, estratégias de controle e algumas estratégias de busca em espaço de estados

# Busca em IA

- Objetivo: Transformar um problema do mundo real em um problema de busca

  -> **Modelar o problema como espaço de estados**

- Usar uma *estratégia de busca* para encontrar a solução

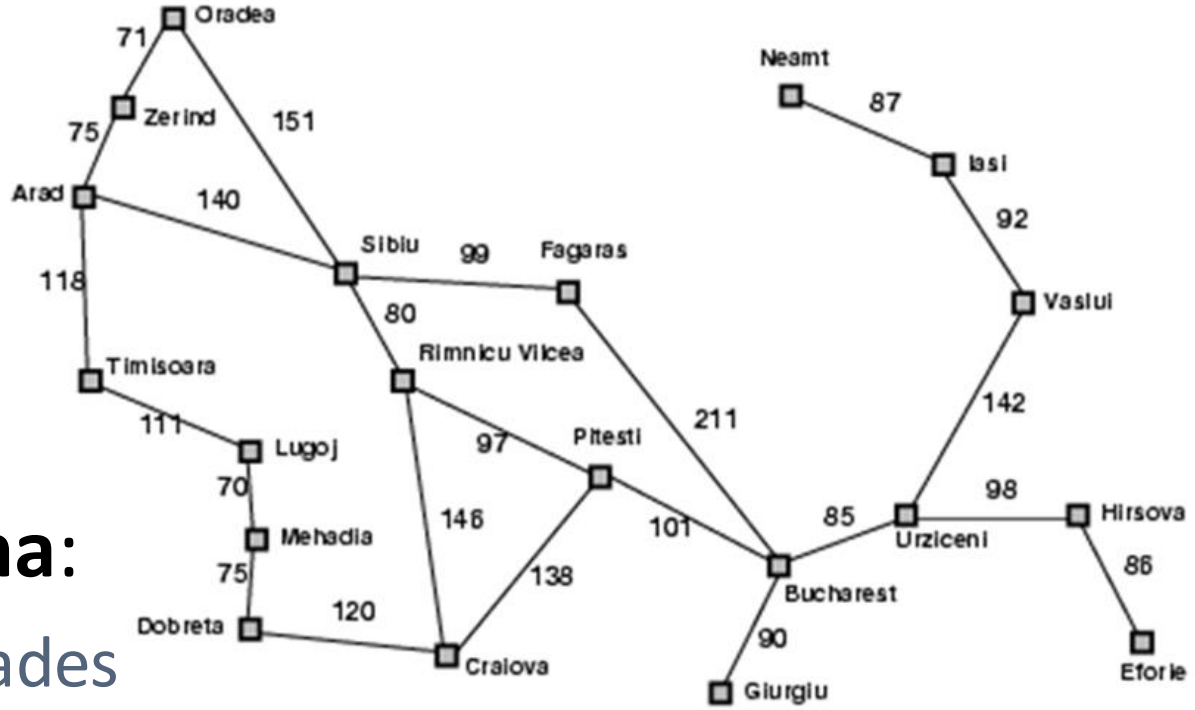
  -> **Busca no espaço de estados**

# Formulação do Problema

## Solução
## Tipos
## Sistemas de produção

# Férias na Romênia

- Você passando férias na Romênia: **Arad**
- Os voos de volta partem amanha de **Bucharest**



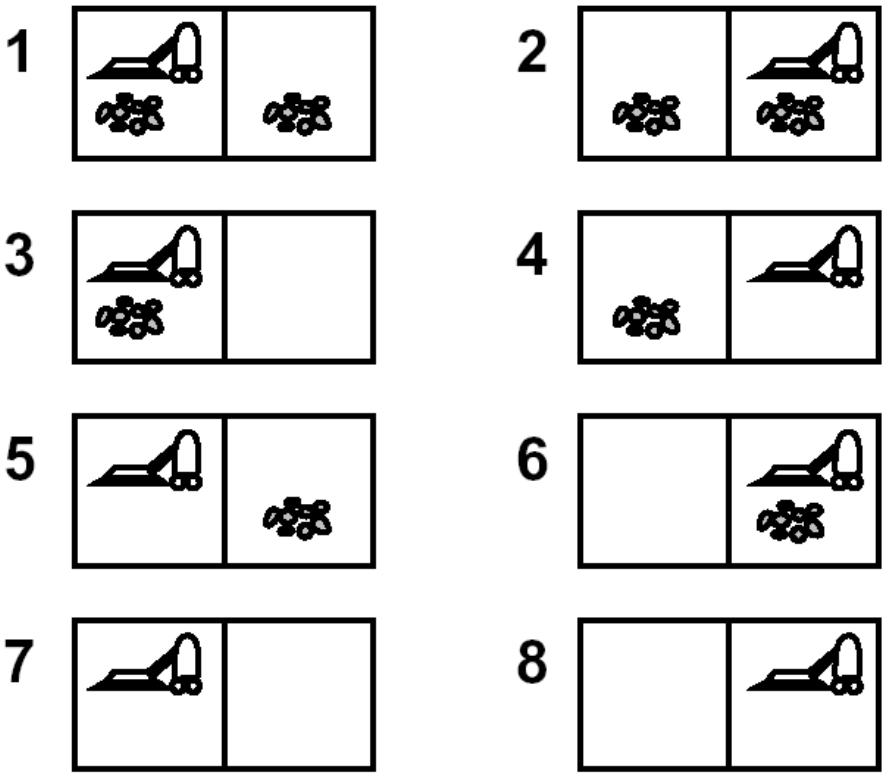

- Formular **meta**:
  - Estar Bucharest
- Formular **problema**:
  - Estado: várias cidades
  - Ações: dirigir entre as cidades
- Encontrar **solução**:
  - Sequencia de cidades, ex: Arad, Sibiu, Fagaras, Bucharest

# Tipos de problemas nas buscas

- **Determinístico**, totalmente observável → problema de **único estado**
  - O agente sabe exatamente em qual estado estará; solução é uma sequência
- **Não observável** → problema **sem sensores**
  - O agente pode não ter ideia de onde está; solução é uma sequência
- **Não-determinístico** e/ou parcialmente observável → **problema de contingência**
  - Os perceptores proveem informação nova sobre o estado corrente
- **Espaço de estados desconhecido** → problema de **exploração**

# Exemplo: Mundo do Aspirador

## Fonte: Russell e Norvig




**Problema de estado único**: começa no #5.
**Solução?** [Direita, Aspira]

**Sem sensores**, começa em {1,2,3,4,5,6,7,8} ex: Direita vai para {2,4,6,8}
**Solução?** [Direita, Aspira, Esquerda, Aspira]

**Não-determinístico**: Aspirar pode sujar um carpete limpo
**Parcialmente observável**: local, sujeira no local corrente.
**Perceptor**: [E, Limpa], i.e., começa no #5 ou #7
**Solução?** [Direita, se sujeira então Aspira]

# Formulação Problema de Estado Único

- Um **problema** é definido por quatro itens:
  1. **estado inicial** e.g., "Arad"
  2. ações ou **função sucessor** S(x) = conjunto de pares ação-estado Ex: S(Arad) = {< Arad--> Zerind, Zerind >, ...}
  3. **teste da meta**, pode ser
     - **explícito**, e.g., x = "em Bucharest"
     - **implícito**, e.g., Checkmate(x)
  4. custo do caminho (se houver)
     - soma das distâncias, número de ações executadas, etc.
     - c(x,a,y) é o custo do passo, assumido ser = 0
- Uma solução é uma sequência de ações que leva do estado inicial a um estado meta

# Selecionando um Espaço de Estados

- O **mundo real** é absurdamente **complexo**: o espaço de estados deve ser restringido para a resolução de problemas
- **Estado** (restrito) = conjunto de estados reais
- **Ação** (restrita) = combinação complexa de ações reais
  - e.g., "Arad → Zerind" representa um conjunto complexo de possíveis rotas, desvios, paradas, etc.
  - Para a **garantia da realização**, qualquer estado real "em Arad" deve levar a algum estado real "em Zerind"
- **Solução** (restrita) = conjunto de caminhos reais que são soluções no mundo real
- Cada ação restrita deve ser "mais fácil" que o problema original

# Busca em Espaço de Estados

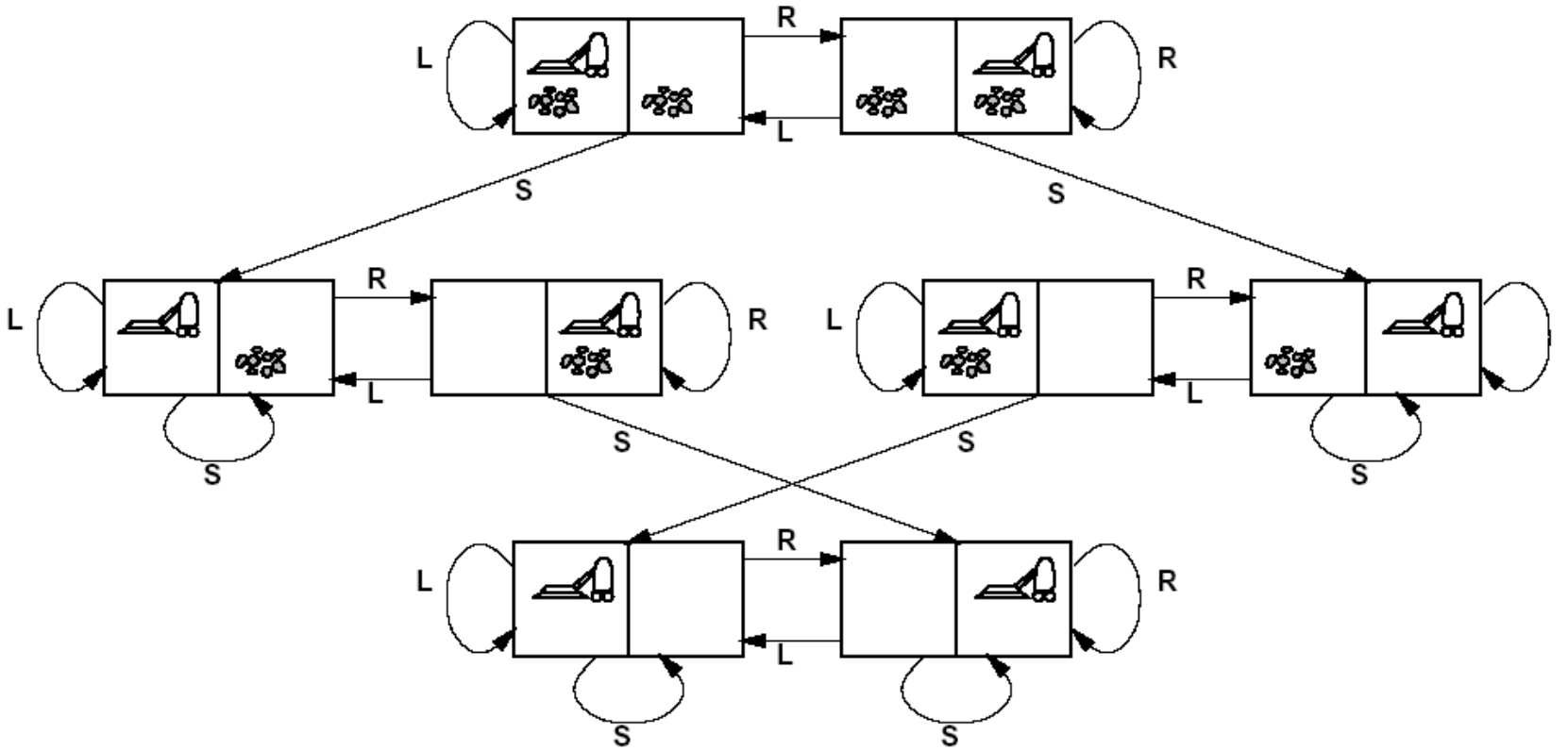
- Um grafo pode ser usado para representar um **espaço de estados** em que:
  - Os **nós** correspondem a situações de um problema
  - As **arestas** correspondem a movimentos permitidos ou ações ou passos da solução
  - Um dado problema é solucionado encontrando-se um **caminho no grafo**

- Um **problema é representado** por:
  - Um **espaço de estados** (um grafo)
  - Um **estado** (nó) **inicial**
  - Uma condição de término ou **critério de parada**; estados (nós) terminais são aqueles que satisfazem a condição de término

- Custos
  - Se não houver custos, há interesse em soluções de **caminho mínimo**
  - No caso em que custos são adicionados aos movimentos normalmente há interesse em soluções de **custo mínimo**
  - O **custo de uma solução** é o custo das arestas ao longo do caminho da solução

# Exemplo: Mundo do Aspirador

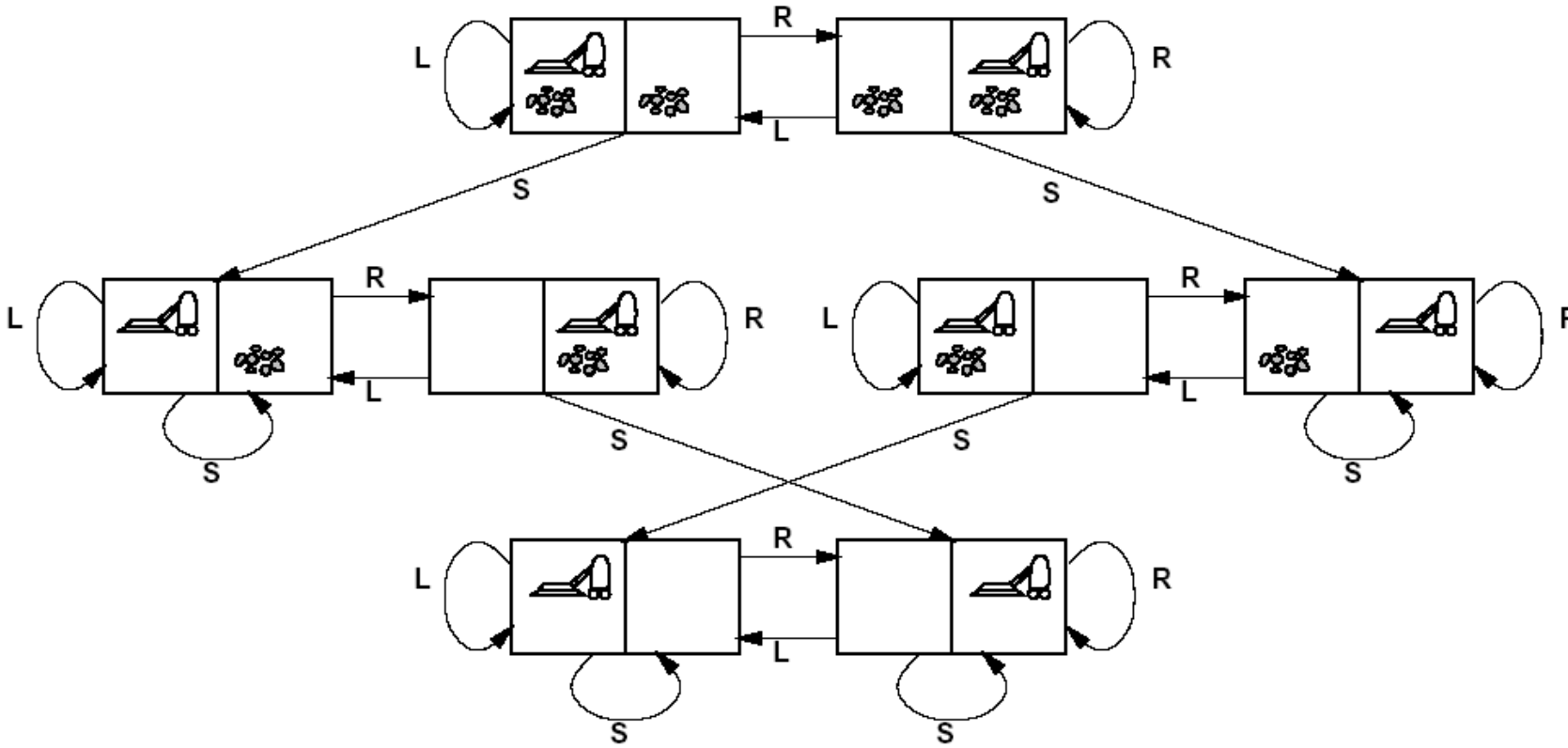
## Grafo do Espaço de Estados do Aspirador




- **Estados?**
- **Operadores?**
- **Estado Final?**
- **Custo do caminho**?

# Exemplo: Mundo do Aspirador

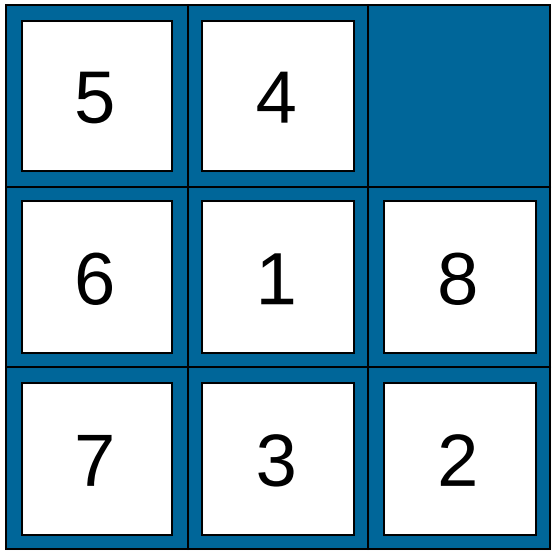
## Grafo do Espaço de Estados do Aspirador



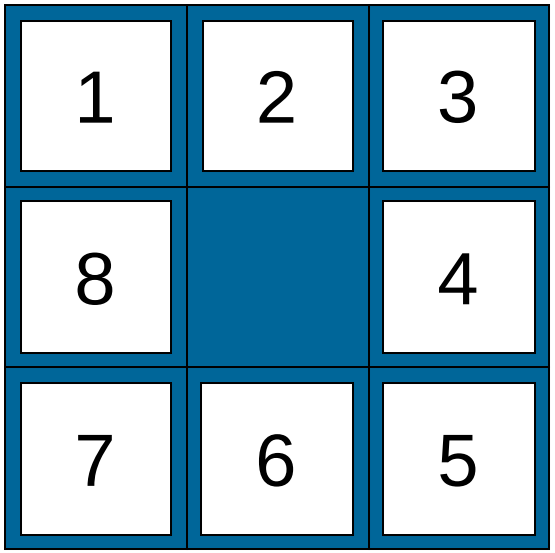

- **Estados**: um dos estados mostrados na figura
- **Operadores**: L (esquerda), R (direita), S (sucção)
- **Estado Final**: nenhuma sujeira em todos os ambientes
- **Custo do caminho**: cada ação tem custo unitário

# Exemplo: Quebra-cabeça de 8 peças

## Fonte: Russell e Norvig




Estado Inicial

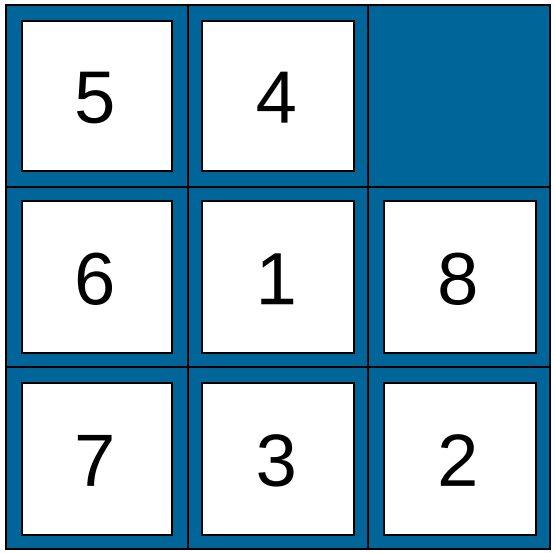



Estado Final

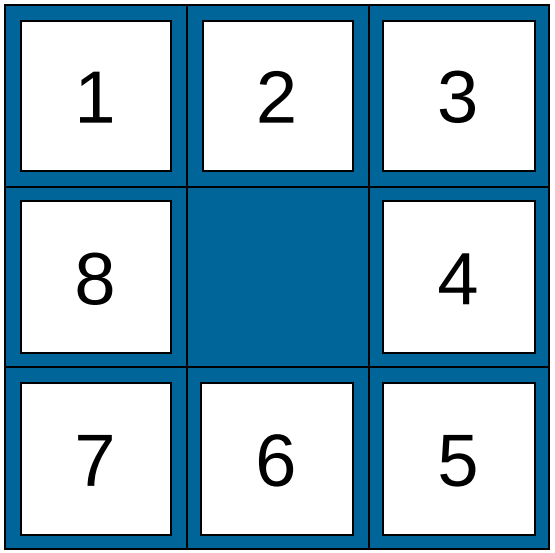
- Estados?
- Operadores?
- Estado Final: = estado fornecido
- Custo do caminho?

# Exemplo: Quebra-cabeça de 8 peças

## Fonte: Russell e Norvig




Estado Inicial

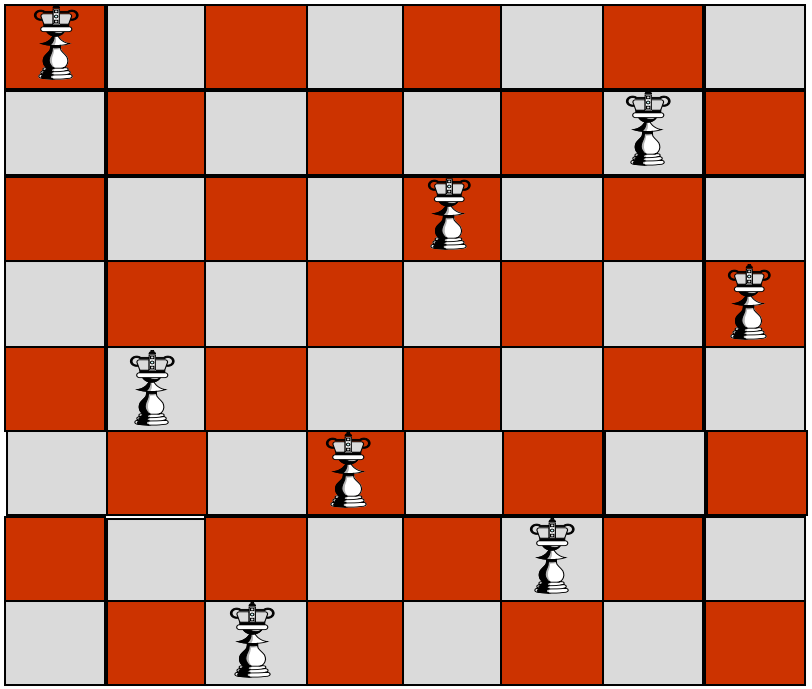



Estado Final

- **Estados**: posições inteiras dos quadrados
- **Operadores**: mover espaço vazio à esquerda, direita, cima, baixo
- **Estado Final:** estado fornecido
- **Custo do caminho**: 1 por movimento

# Exemplo: Problemas das *n* Rainhas

## Fonte: Russell e Norvig

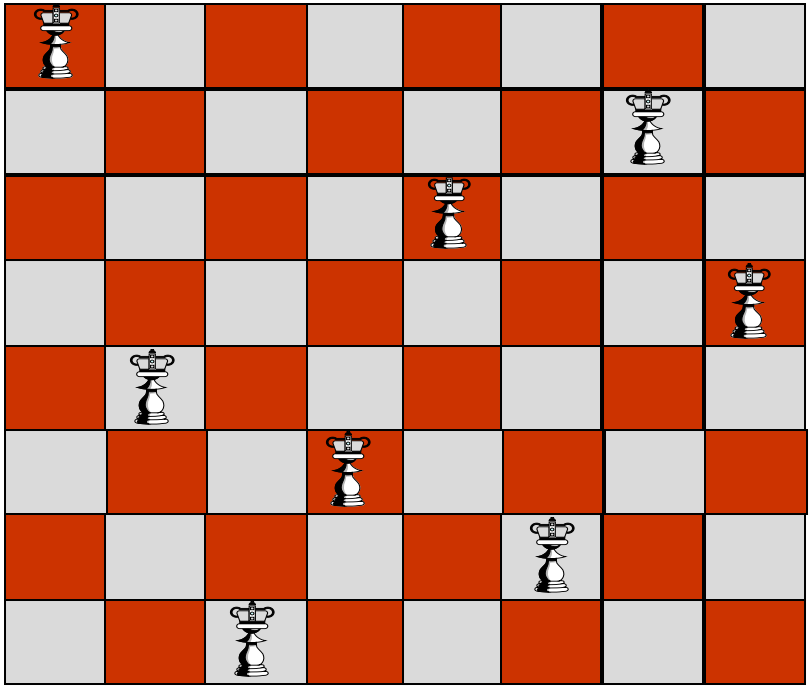


- Colocar ***n* rainhas** em um tabuleiro ***n x n*** com uma única rainha em cada **linha, coluna e diagonal**.
- Para n = 8

- Estados?
- Operadores?
- Estado Final?
- Custo do caminho?

# Exemplo: Problemas das *n* Rainhas

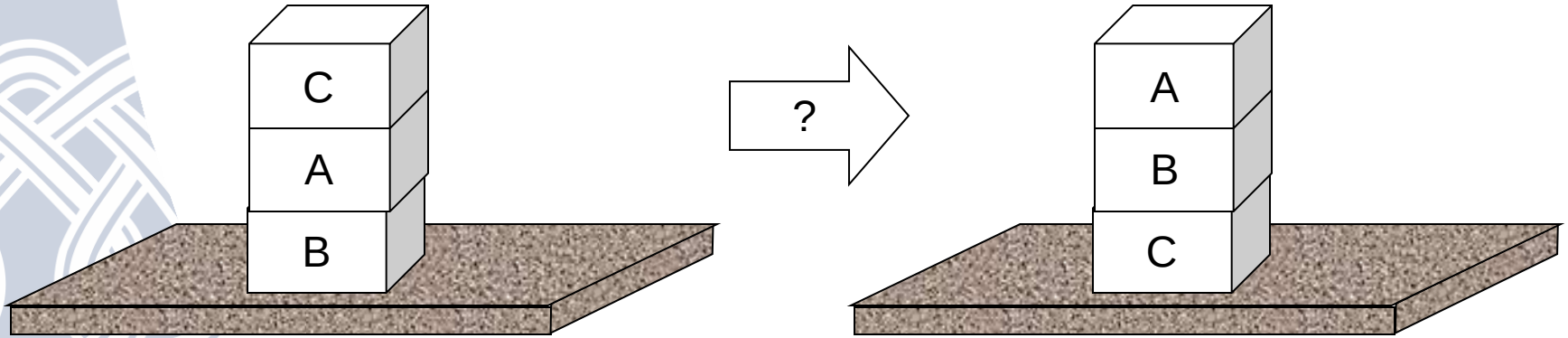
## Fonte: Russell e Norvig



- **Estados**: qualquer arranjo de 0 a 8 rainhas no tabuleiro
- **Operadores**: adicionar uma rainha a qualquer posição
- **Estado Final**: 8 rainhas no tabuleiro, sem ataque
- **Custo do caminho**: zero (apenas o estado final é interessante) mas não confundir com complexidade da busca...

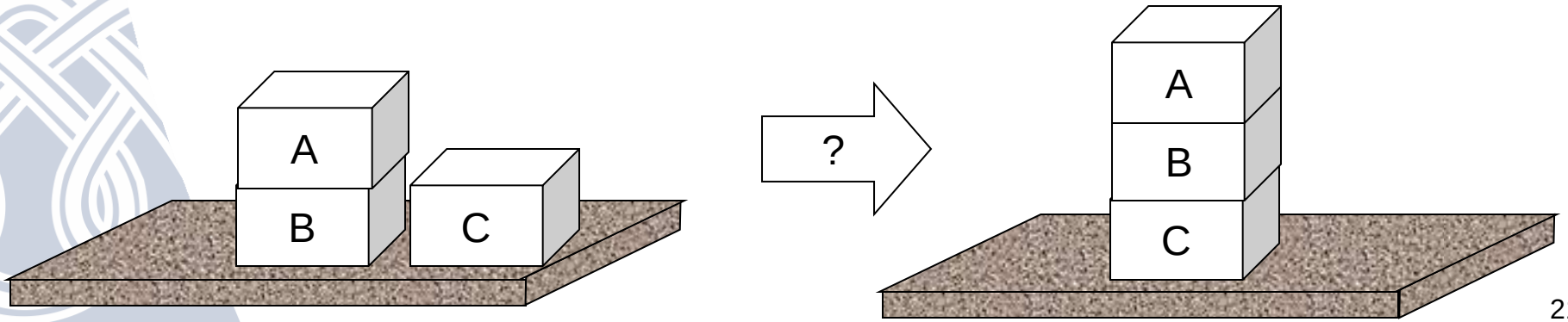
# Exemplo: Pilha de Blocos

- Considere o problema de encontrar um plano (estratégia) para rearranjar uma pilha de blocos como na figura
  - Somente é permitido um movimento por vez
  - Um bloco somente pode ser movido se não há nada em seu topo
  - Um bloco pode ser colocado na mesa ou acima de outro bloco

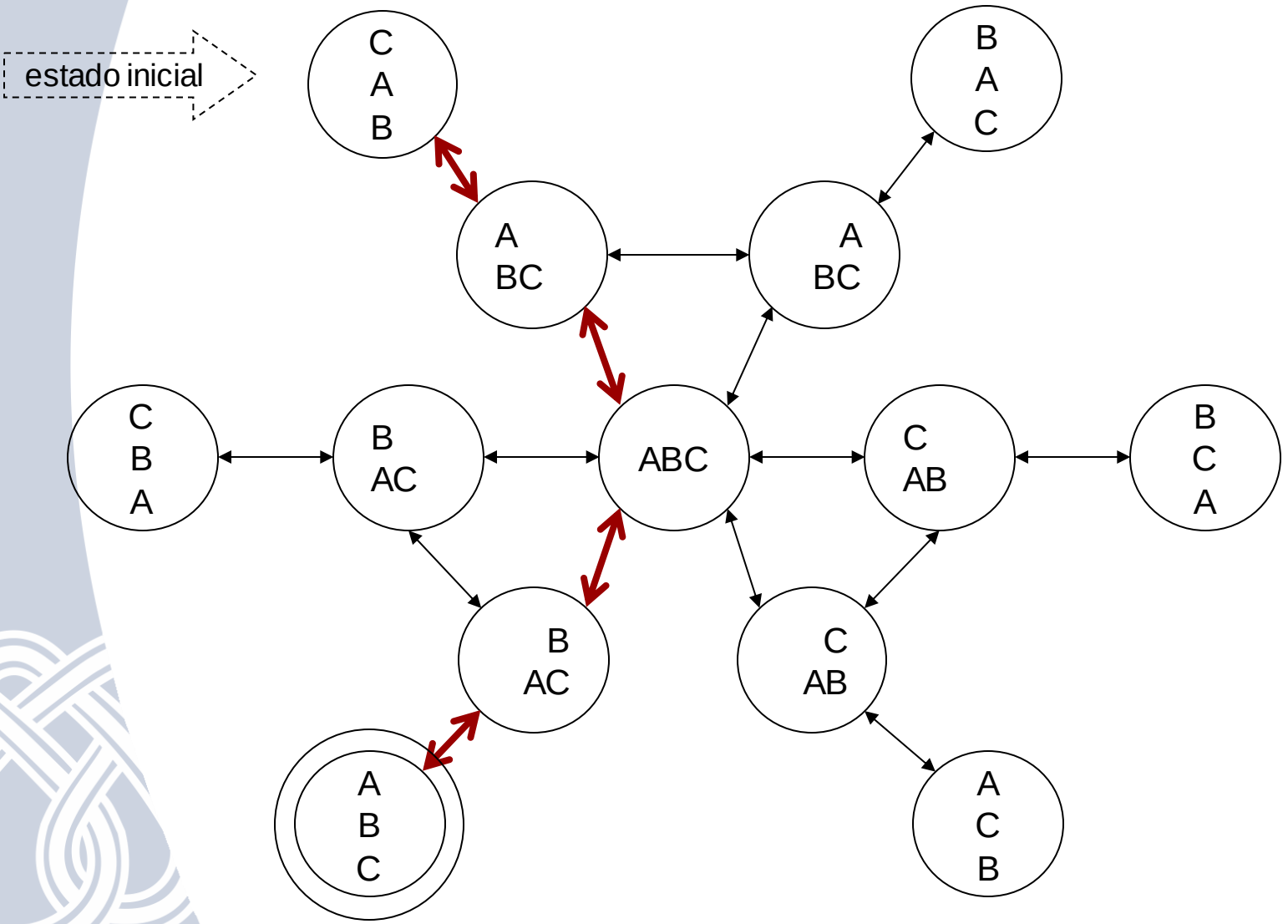


# Exemplo: Pilha de Blocos

- Na situação inicial do problema, há apenas um movimento possível: colocar bloco C na mesa
- Depois que C foi colocado na mesa, há três alternativas
  - Colocar A na mesa ou
  - Colocar A acima de C ou
  - Colocar C acima de A (movimento que não deve ser considerado pois retorna a uma situação anterior do problema)



# Exemplo: Pilha de Blocos



# Sistema de Produção: Definição

- Um **Sistema de Produção** (SP) é composto por três elementos principais:
  - Base de dados global
  - Regras de produção
  - Sistema de controle
- Procedimento PRODUÇÃO
  1. DADOS ← base de dados inicial
  2. até DADOS satisfazer a condição de termino, faça:
     1. selecione alguma regra, R, no conjunto de regras que possa ser aplicada a DADOS
     2. DADOS ← resultado da aplicação de R a DADOS

# Sistema de Controle

- Controle: Passo 2 dentro do loop: maior problema
- Procedimentos de busca:
  - não-informados (busca cega)
  - informados (busca heurística)

# Estratégias de Controle

- Regime de Controle: A forma como é conduzido um processo de busca
- Dois tipos principais:
  - irrevogável
  - **tentativo**
- Dentro do regime de controle existem as **estratégias de controle**.
- Dois tipos de estratégia de controle dentro do regime de controle tentativo:
  - Backtracking
  - Busca em grafos

# Controle

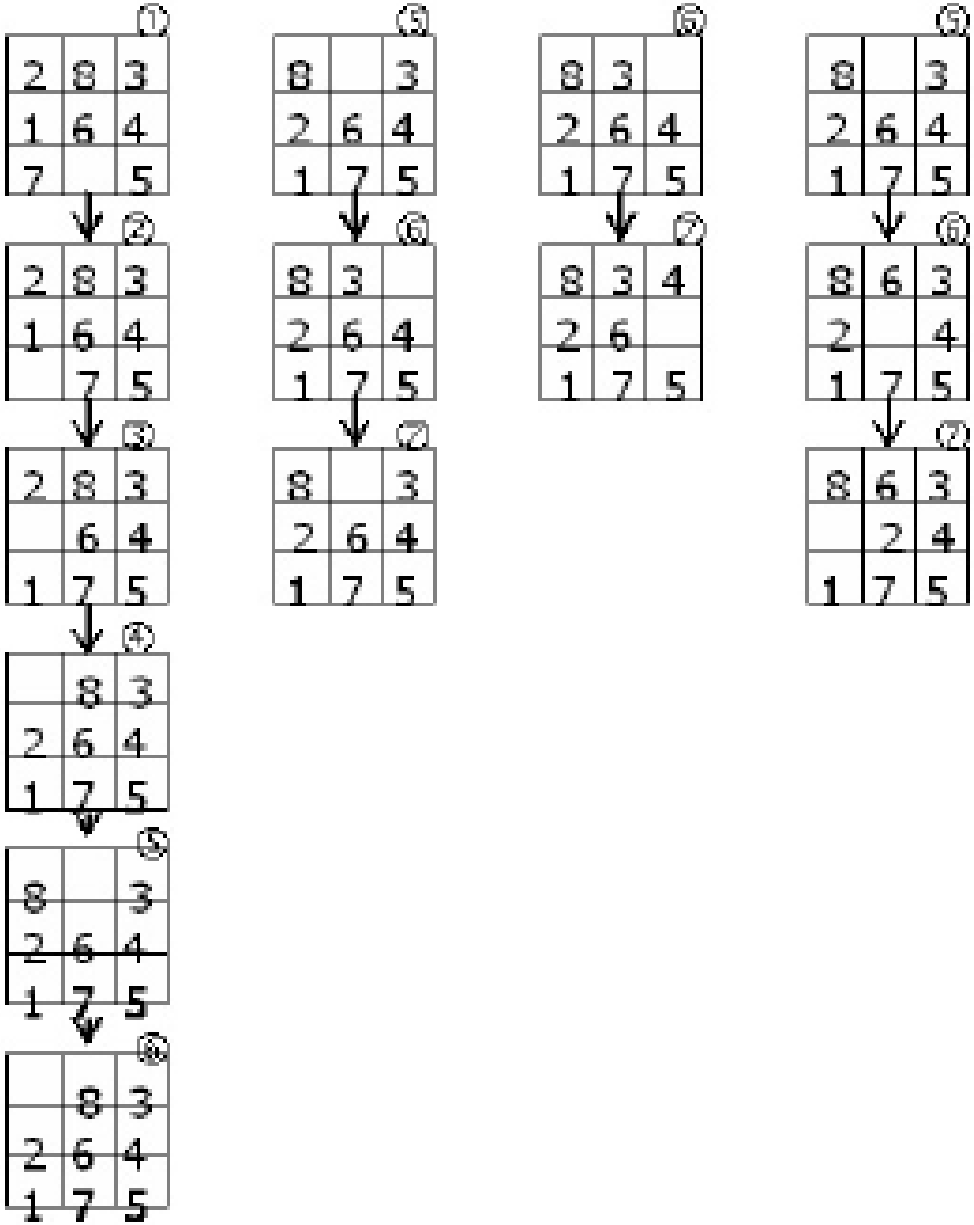
## Backtracking
## Busca em Grafos

# Backtracking

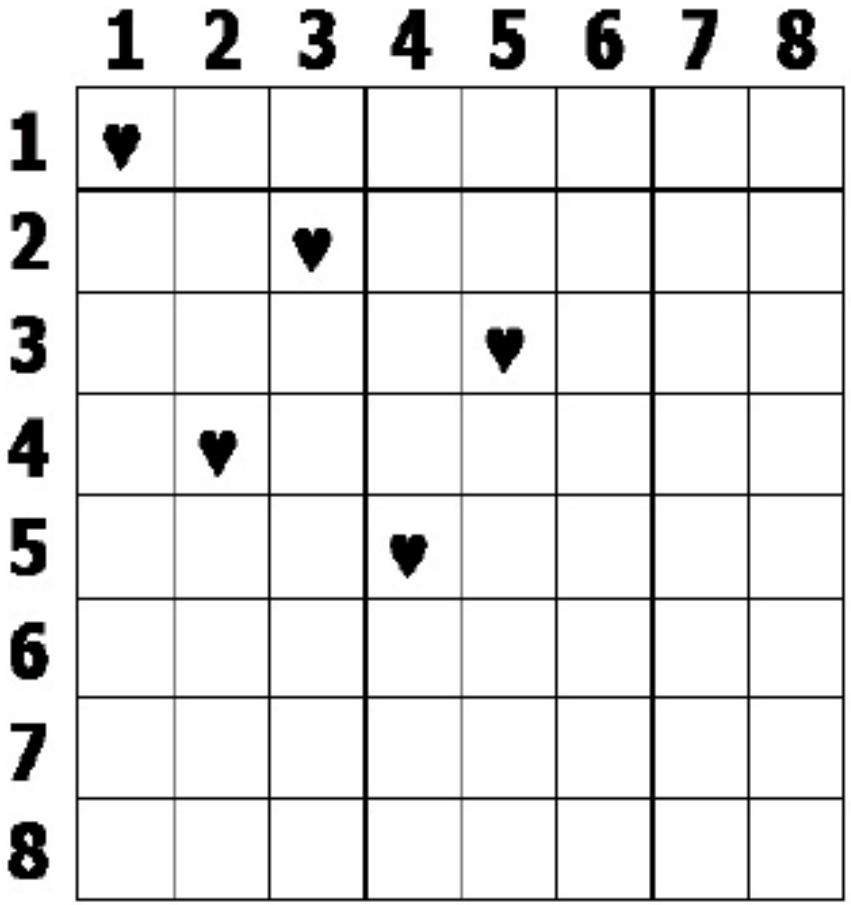
## Procedimento Recursivo
## BACKTRACK (DADOS) [Nilsson, 1982]

1. se TERMO(DADOS), retorne []
2. se DEADEND(DADOS), retorne *FALHA*
3. REGRAS ← REGRASAPL(DADOS)
4. LOOP: se NULL(REGRAS), retorne *FALHA*
5. R ← PRIMEIRA(REGRAS)
6. REGRAS ← CAUDA(REGRAS)
7. RDADOS ←R(DADOS)
8. CAMINHO ← BACKTRACK(RDADOS)
9. se CAMINHO = *FALHA*, vá para LOOP
10. retorne CONC(R,CAMINHO)

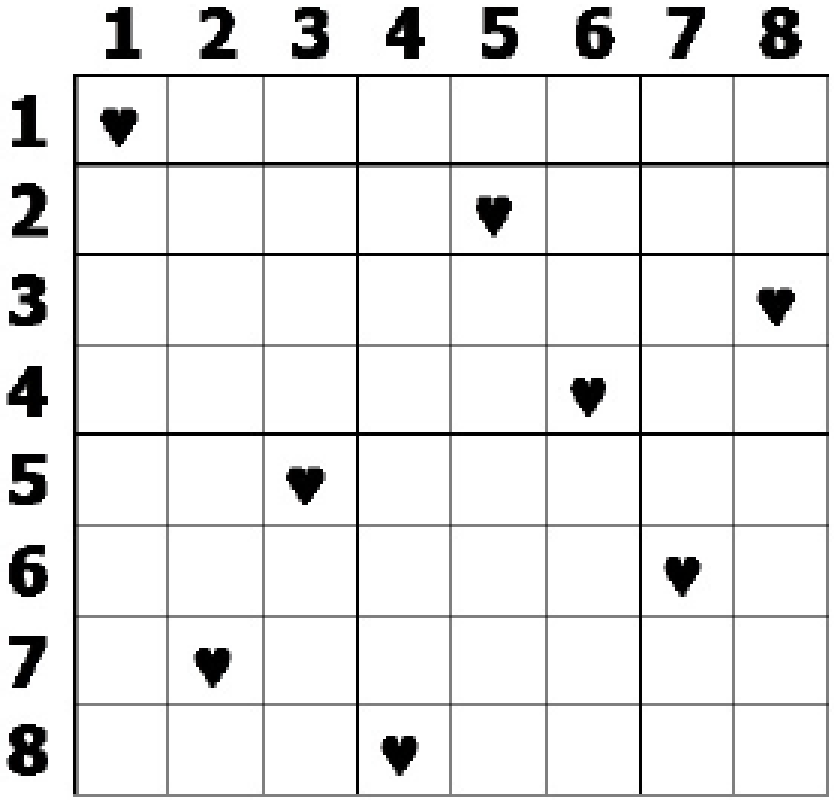
# Quebra-cabeças de 8 peças



# Problemas das 8 rainhas




Uma solução depois de 91 backtrackings



# Busca em grafos

- No **backtracking**: o sistema de controle armazena **somente o caminho correntemente sendo estendido**.
- Um procedimento mais flexível envolve o **armazenamento explícito de todos os caminhos** de tal forma que qualquer um deles pode ser candidato a futura extensão: **BUSCA EM GRAFOS**.

# Busca em grafos: notação

- **Grafo**: conjunto (não necessariamente finito) de nós.
- **Arcos direcionados**: Certos pares de nós são conectados por arcos, e estes arcos são direcionados de um membro do par ao outro (grafo direcionado).
- **Sucessor/pai**: Se um arco é direcionado do nó $n_i$ para o nó ***nj*** , então o nó ***nj*** é o sucessor do nó ***ni*** , e o nó ***ni*** é o pai do nó ***nj*** .
- **Árvore**: caso especial de um grafo no qual cada nó tem, no máximo, um pai.
- **Raiz**: Um nó na árvore que não tem pai é chamado de nó raiz.
- **Folha**: Um nó na árvore que não tem sucessores é chamado de nó folha.

# Busca em grafos: notação

- **Profundidade**: Diz-se que o nó raiz é de profundidade zero. A profundidade de qualquer outro nó na árvore é, por definição, a profundidade de seus pais mais 1.
- **Caminho**: Uma sequência de nós (***n1***, ***n2***, ...,***nk*** ), com cada ***ni*** um sucessor de ***ni-1***, para i = 2, ..., k, é chamada de um caminho de comprimento ***k*** do nó ***n1*** para o nó ***nk*** . Se um caminho existe entre o nó ***ni*** para o nó ***nj*** , então o nó ***nj*** é acessível a partir do nó ***ni*** .
- **Descendente e ancestral**: O nó ***nj*** é então um descendente do nó ***ni*** , e o nó ***ni*** é um ancestral do nó ***nj*** .

# Busca em grafos

**IMPORTANTE**

- Para este modelo, os ***nós*** são rotulados por **bases de dados** e os ***arcos*** são rotulados por **regras**.
- Note que o problema de **encontrar uma sequência de regras** para transformar uma base de dados em outra é **equivalente** ao problema de **encontrar um caminho num grafo.**

# Busca em Grafos: Custo

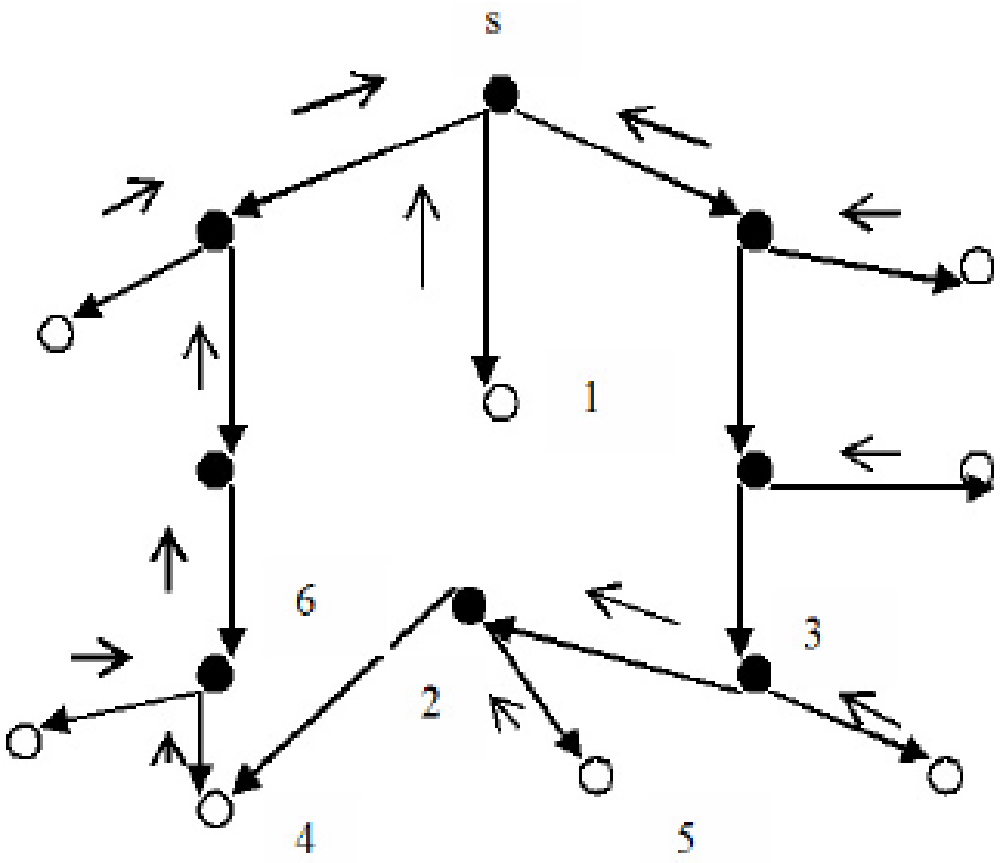
- É conveniente atribuir custos positivos aos arcos, para representar o **custo da aplicação da regra** correspondente.
- Usa-se a notação c(***ni*** , ***nj*** ) para denotar o **custo de um arco** direcionado do nó ***ni*** para o nó ***nj*** .
- O custo de um caminho entre dois nós é a soma dos custos de todos os arcos que conectam os nós no caminho.
- Em alguns problemas, deseja-se encontrar o caminho que tenha **custo mínimo** entre dois nós.

# Busca em grafos

- No tipo **mais simples** de problema, **deseja-se encontrar um caminho** (talvez tendo custo mínimo) entre um **nó** dado ***s***, representando a base de dados inicial e um outro **nó** dado ***t***, representando alguma outra base de dados.
- A situação **mais usual** envolve **encontrar um caminho** entre **o nó *s*** e qualquer membro do **conjunto de nós** {***ti***} que representa as bases de dados que satisfazem a condição de termino. O conjunto {***ti***} é o conjunto meta e cada nó ***t*** em {***ti***} é um nó meta.

1. Crie um grafo de busca, G, consistindo somente do nó inicial, ***s***. Coloque ***s*** numa lista chamada **ABERTOS**.
2. Crie uma lista **FECHADOS** que está inicialmente vazia.
3. LOOP: se ABERTOS está vazia, saia com falha.
4. Selecione o primeiro nó em ABERTOS, remova-o de ABERTOS, coloque-o em FECHADOS. Chame este nó de ***n***.
5. Se ***n*** é um nó meta, saia com sucesso com a solução obtida traçando um caminho entre os ponteiros de ***n*** para ***s*** em G.
6. Expanda o nó ***n***, gerando o conjunto, M, de seus sucessores que não são ancestrais de ***n***. Instale estes membros de M como sucessores de ***n*** em G.
7. Estabeleça um ponteiro para ***n*** a partir daqueles membros de M que ainda não estejam nem em ABERTOS nem em FECHADOS. Adicione estes membros de M a ABERTOS.
8. (Re)ordene a lista ABERTOS, arbitrariamente ou de acordo com mérito heurístico.
9. Vá para LOOP.

- Passo 7: Para cada membro de M que já esteja em ABERTOS ou FECHADOS, decida se direciona ou não seu ponteiro para n. Para cada membro de M já em FECHADOS, decida para cada um de seus descendentes em G se redireciona ou não o seu ponte



# Busca em Espaço de Estados

- **Estratégias Básicas de Busca**
  (Busca não informada ou Cega)
  - As estratégias não-informadas usam **apenas a informação disponível na definição do problema** ou seja, **não utiliza** informações sobre o problema para guiar a busca
  - Estratégia de **busca exaustiva** aplicada até uma solução ser encontrada (ou falhar)
    - Busca em Profundidade (Depth-first)
    - Busca em Largura (Breadth-first)
    - Busca em profundidade limitada
    - Busca por aprofundamento iterativo

# Busca em grafos (não-informados)

- Estratégias de busca: uma estratégia é definida através da ordem da expansão dos nós
- Avaliação:
  - **Completeza**: sempre acha a solução se ela existir?
  - **Complexidade temporal**: número de nós gerados / expandidos
  - **Complexidade espacial:** número máximo de nós na memória
  - **Otimalidade:** sempre acha a solução de custo mínimo?

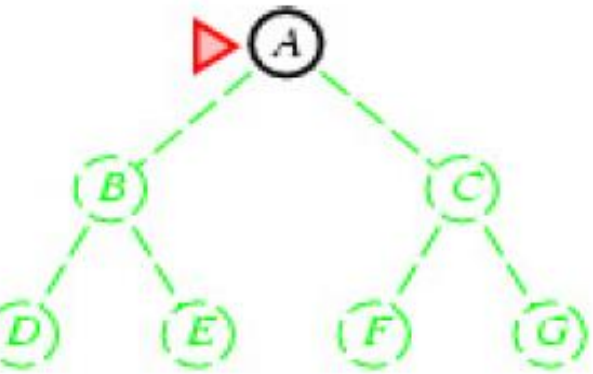
b = fator de ramificação máximo

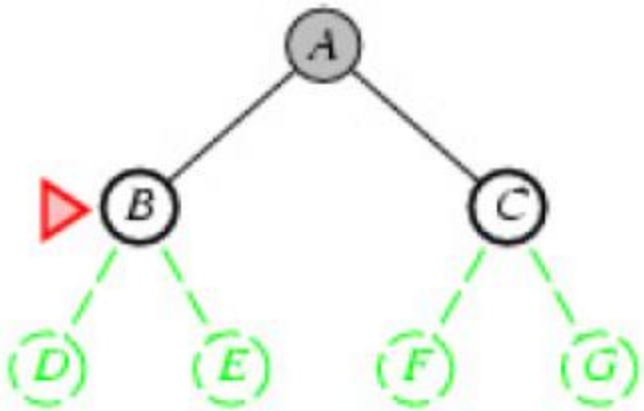
d = profundidade da solução de menor custo

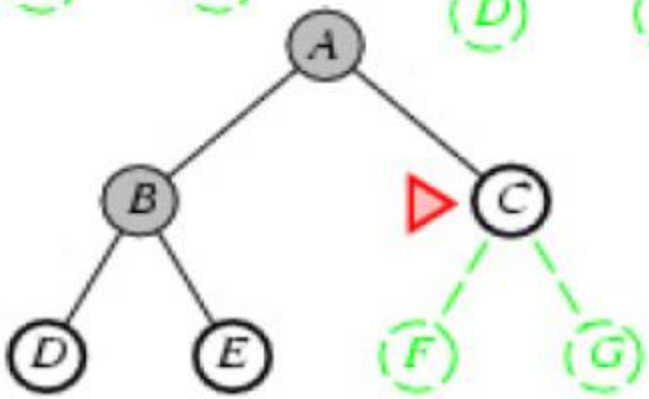
m = profundidade máxima do espaço de estados (pode ser 1)

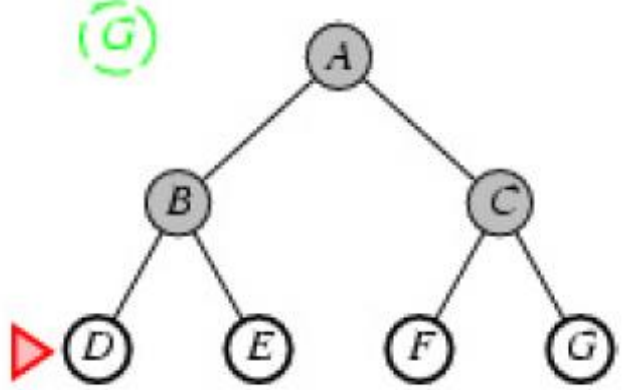
# Busca em Largura (Fonte: Russel e Norvig)

- Expanda o nó de menor profundidade ainda não expandido
- Implementação: ABERTOS é uma lista FIFO, i.e., novos sucessores entram no final



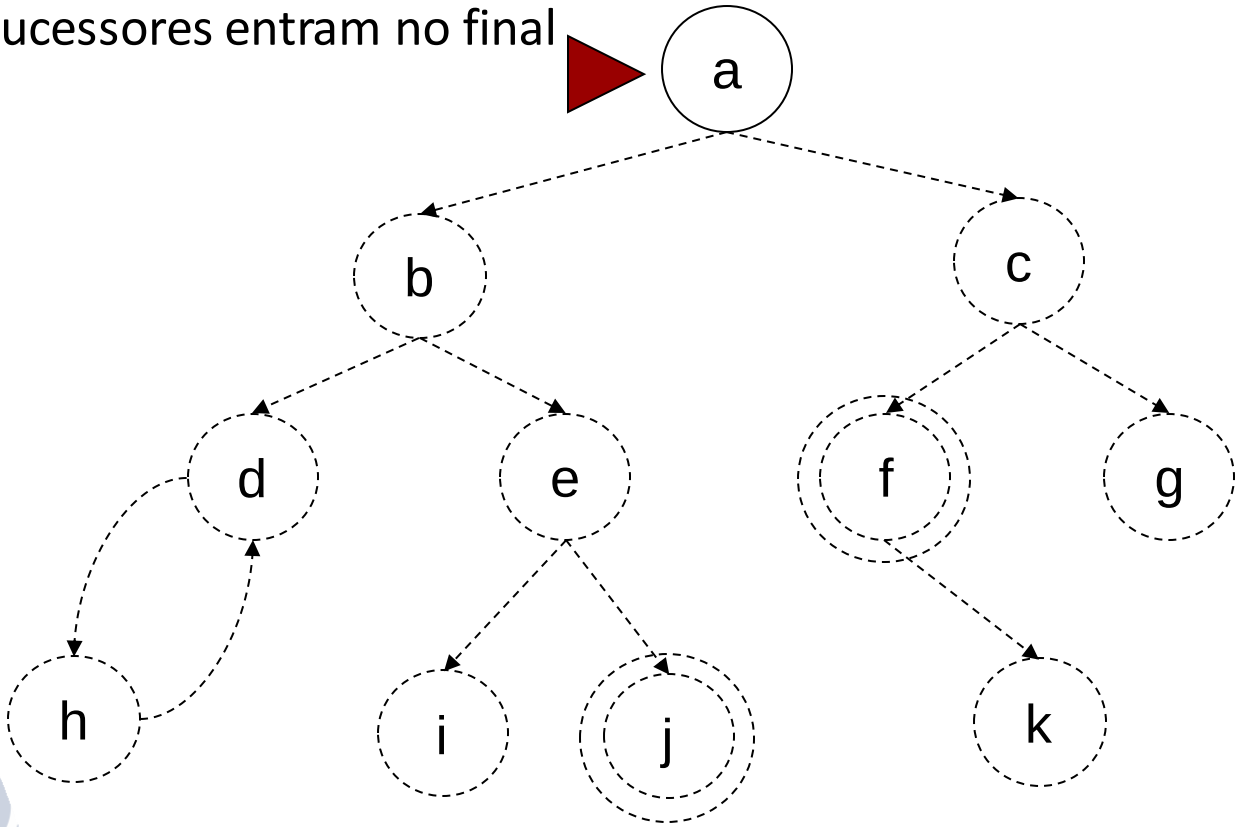



# Busca em Largura

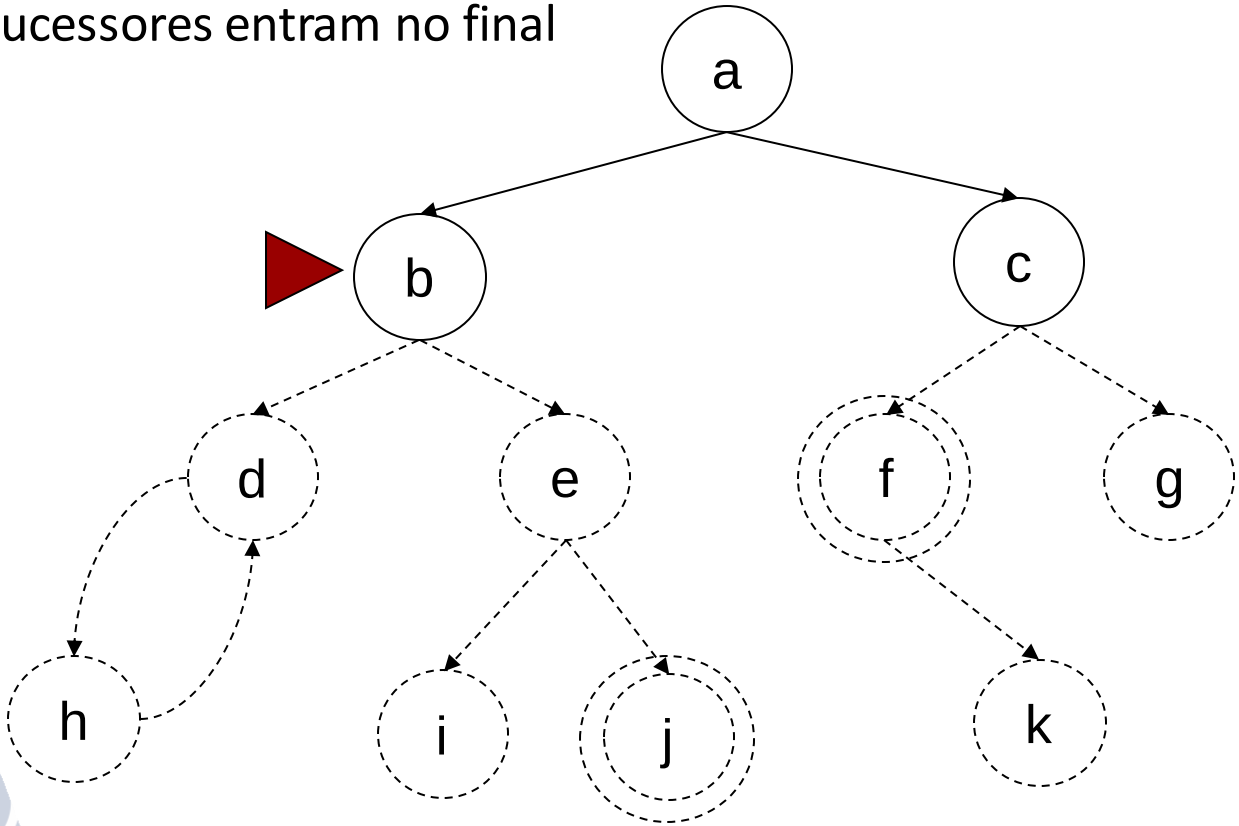
- Expanda o nó de menor profundidade ainda não expandido
- Implementação: ABERTOS é uma lista FIFO, i.e., novos sucessores entram no final




Inserir no final, remover da frente: a

# Busca em Largura

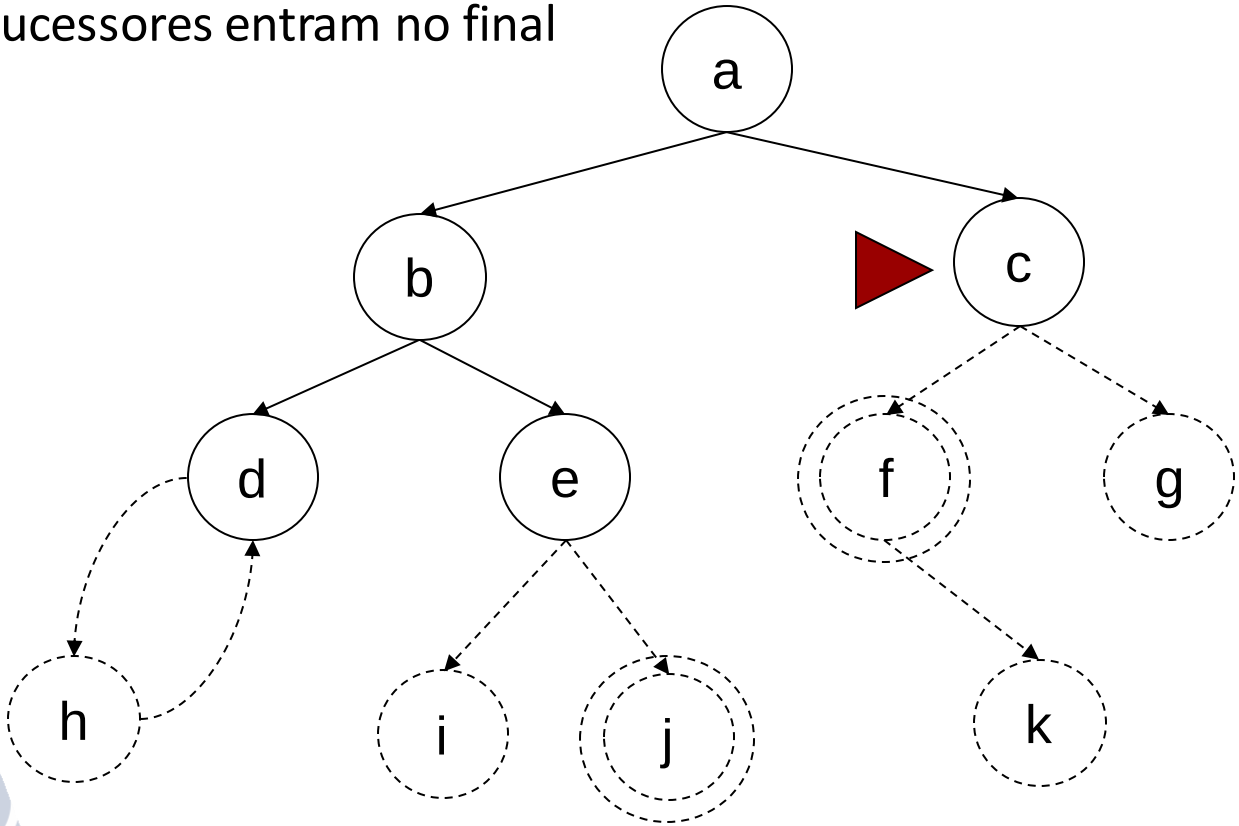
- Expanda o nó de menor profundidade ainda não expandido
- Implementação: ABERTOS é uma lista FIFO, i.e., novos sucessores entram no final




Inserir no final, remover da frente: b, c

# Busca em Largura

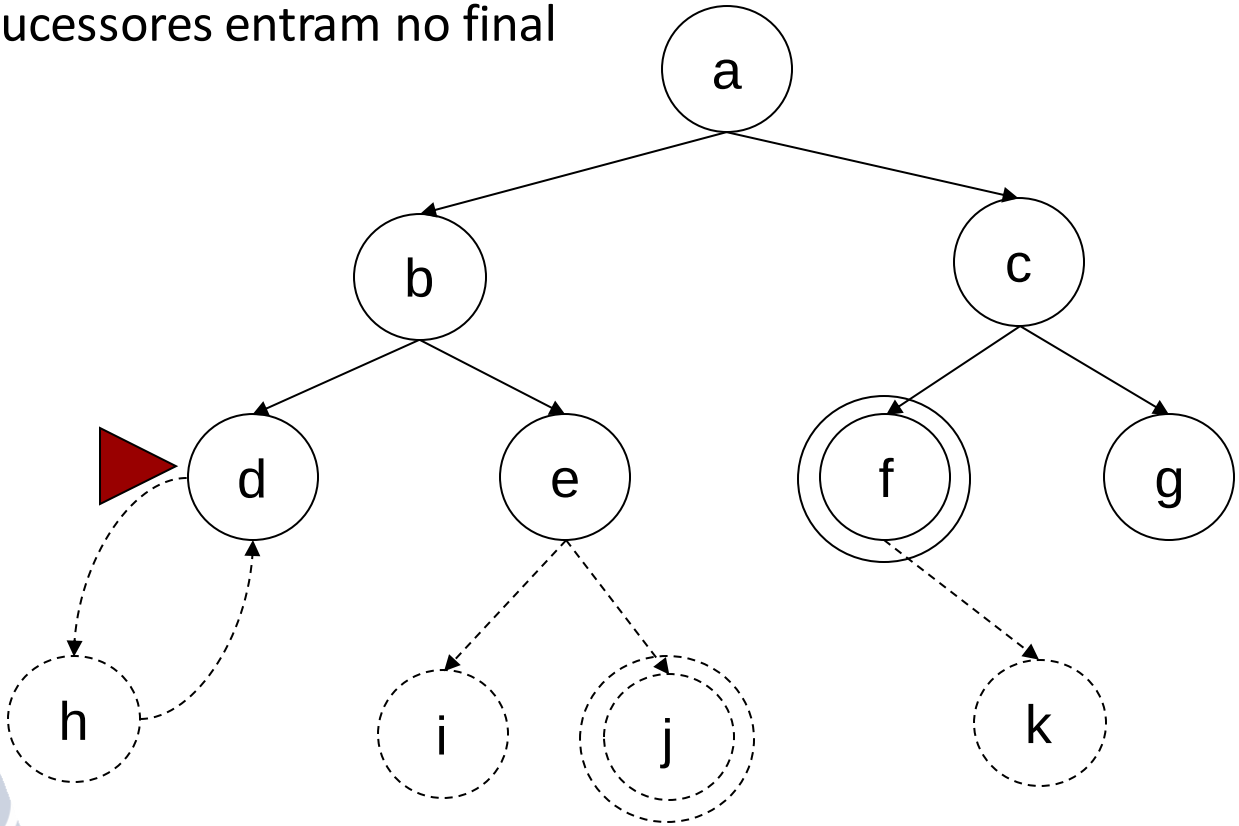
- Expanda o nó de menor profundidade ainda não expandido
- Implementação: ABERTOS é uma lista FIFO, i.e., novos sucessores entram no final




Inserir no final, remover da frente: c, d, e

# Busca em Largura

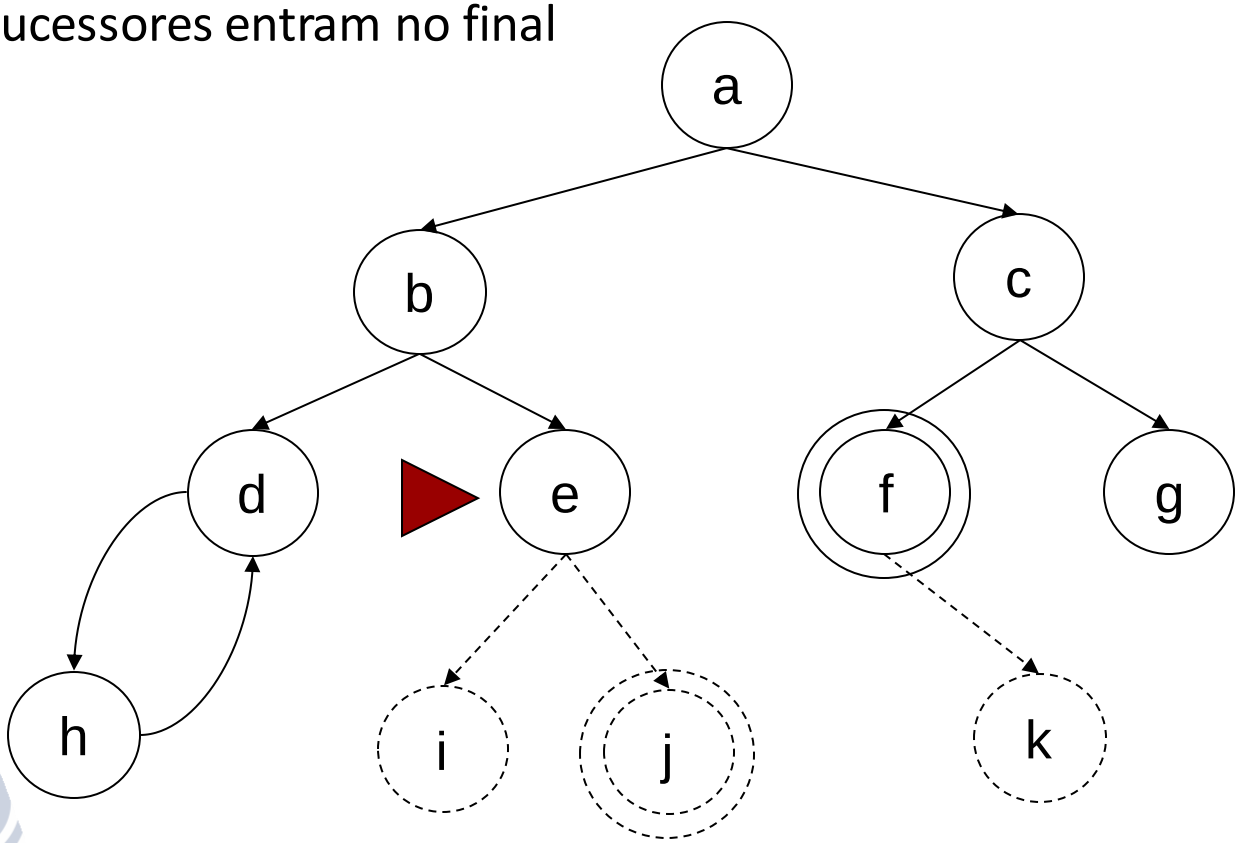
- Expanda o nó de menor profundidade ainda não expandido
- Implementação: ABERTOS é uma lista FIFO, i.e., novos sucessores entram no final




Inserir no final, remover da frente: d, e, f, g

# Busca em Largura

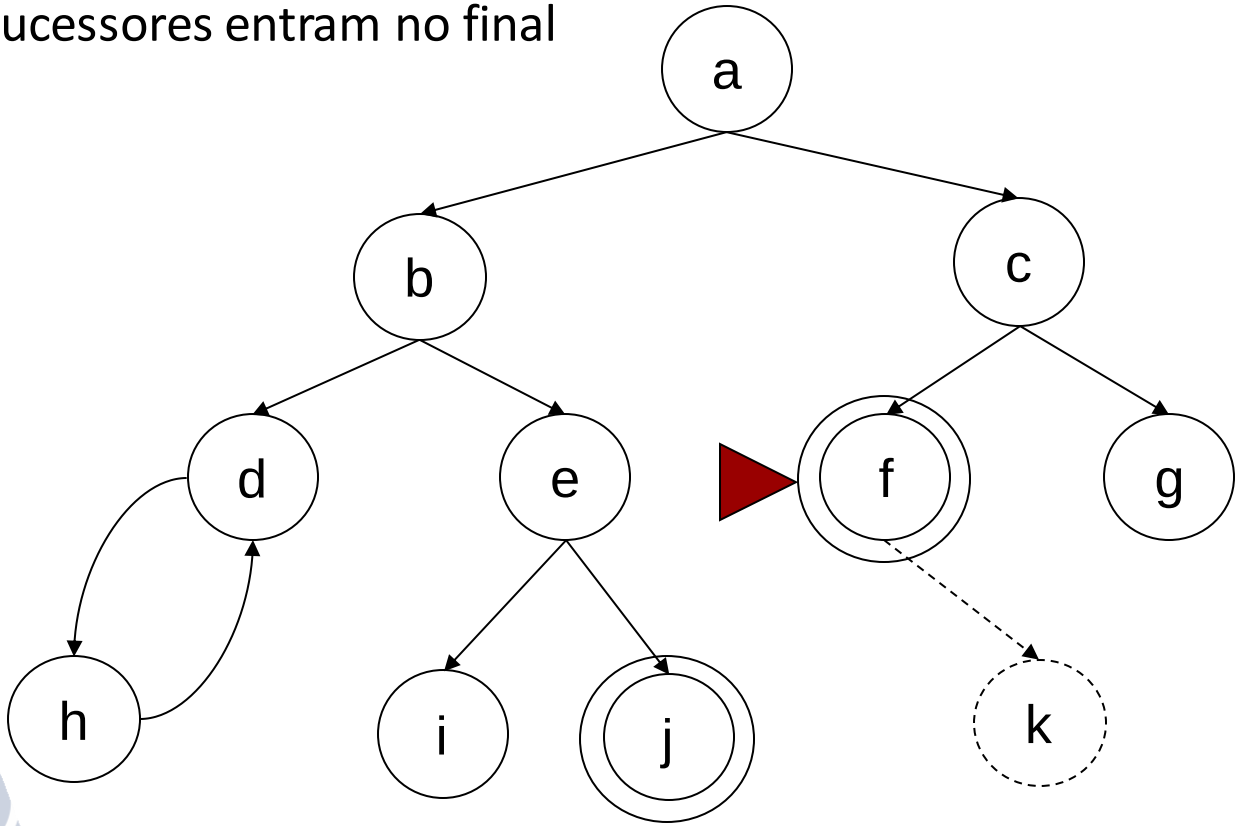
- Expanda o nó de menor profundidade ainda não expandido
- Implementação: ABERTOS é uma lista FIFO, i.e., novos sucessores entram no final




Inserir no final, remover da frente: e, f, g, h

# Busca em Largura

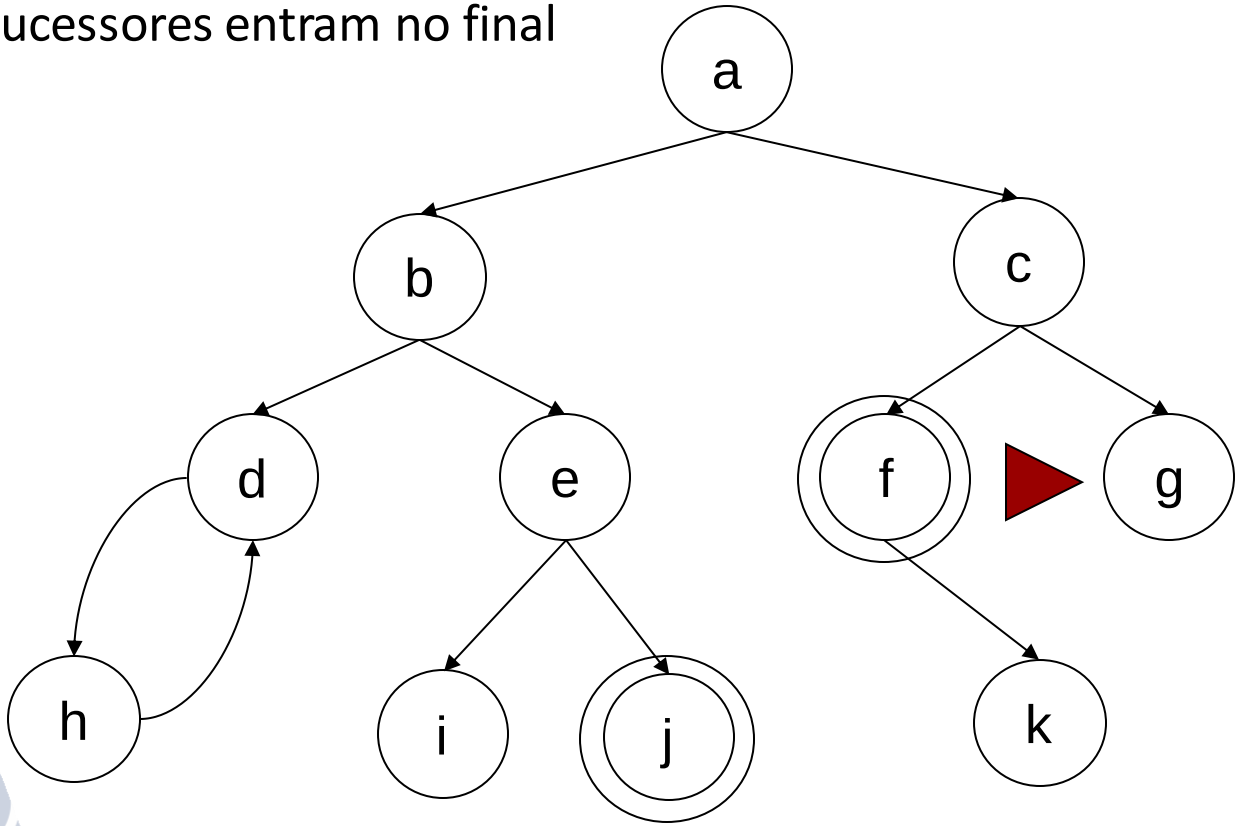
- Expanda o nó de menor profundidade ainda não expandido
- Implementação: ABERTOS é uma lista FIFO, i.e., novos sucessores entram no final




Inserir no final, remover da frente: f, g, h, i, j

# Busca em Largura

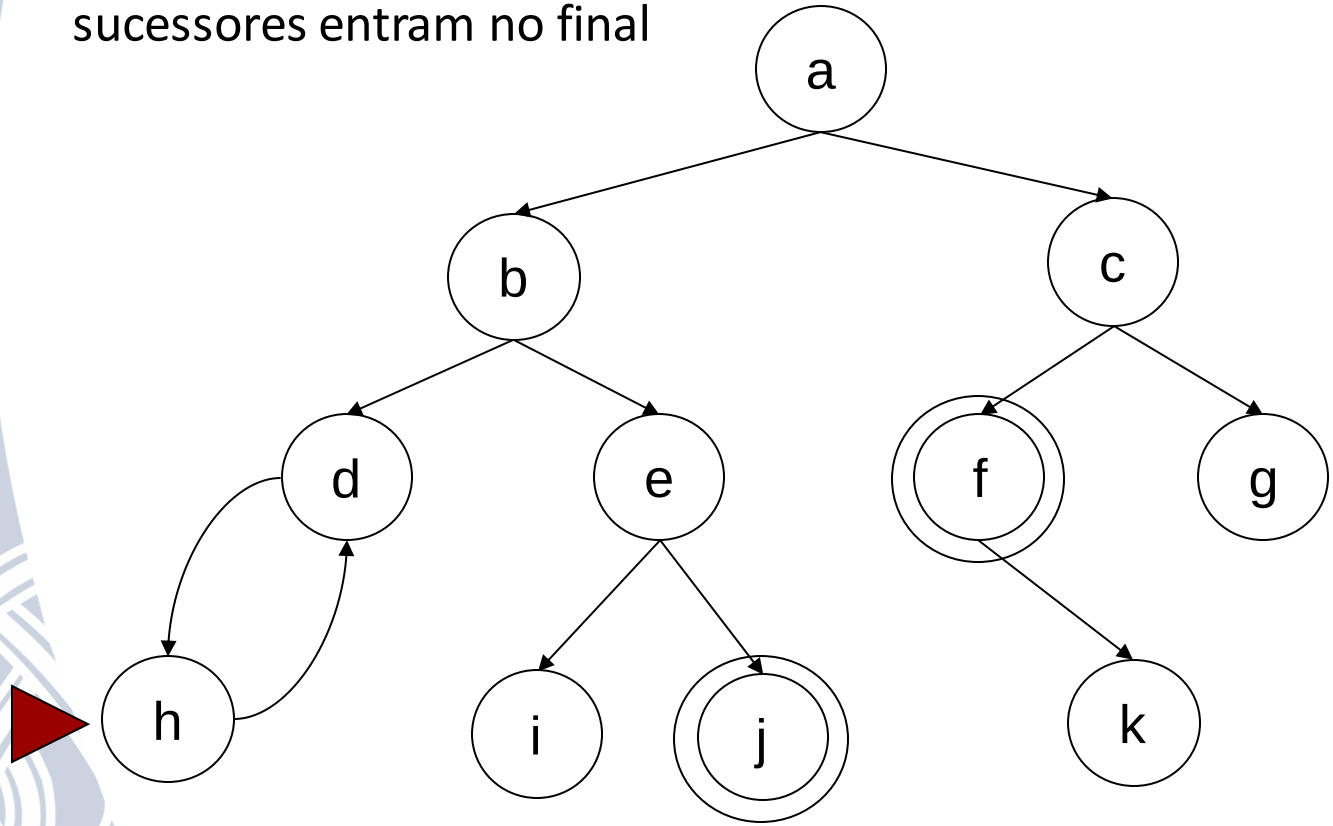
- Expanda o nó de menor profundidade ainda não expandido
- Implementação: ABERTOS é uma lista FIFO, i.e., novos sucessores entram no final




Inserir no final, remover da frente: g, h, i, j, k

# Busca em Largura

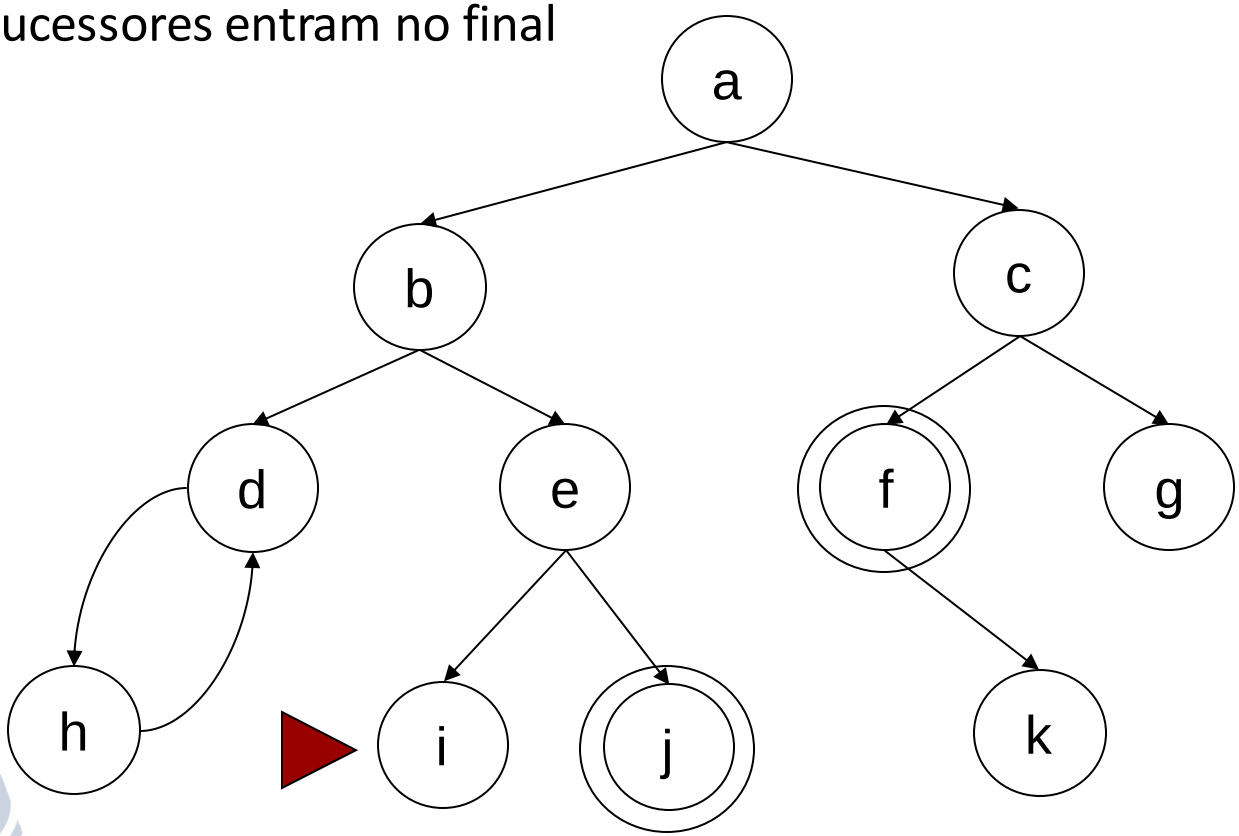
- Expanda o nó de menor profundidade ainda não expandido
- Implementação: ABERTOS é uma lista FIFO, i.e., novos sucessores entram no final




Inserir no final, remover da frente: h, i, j, k

# Busca em Largura

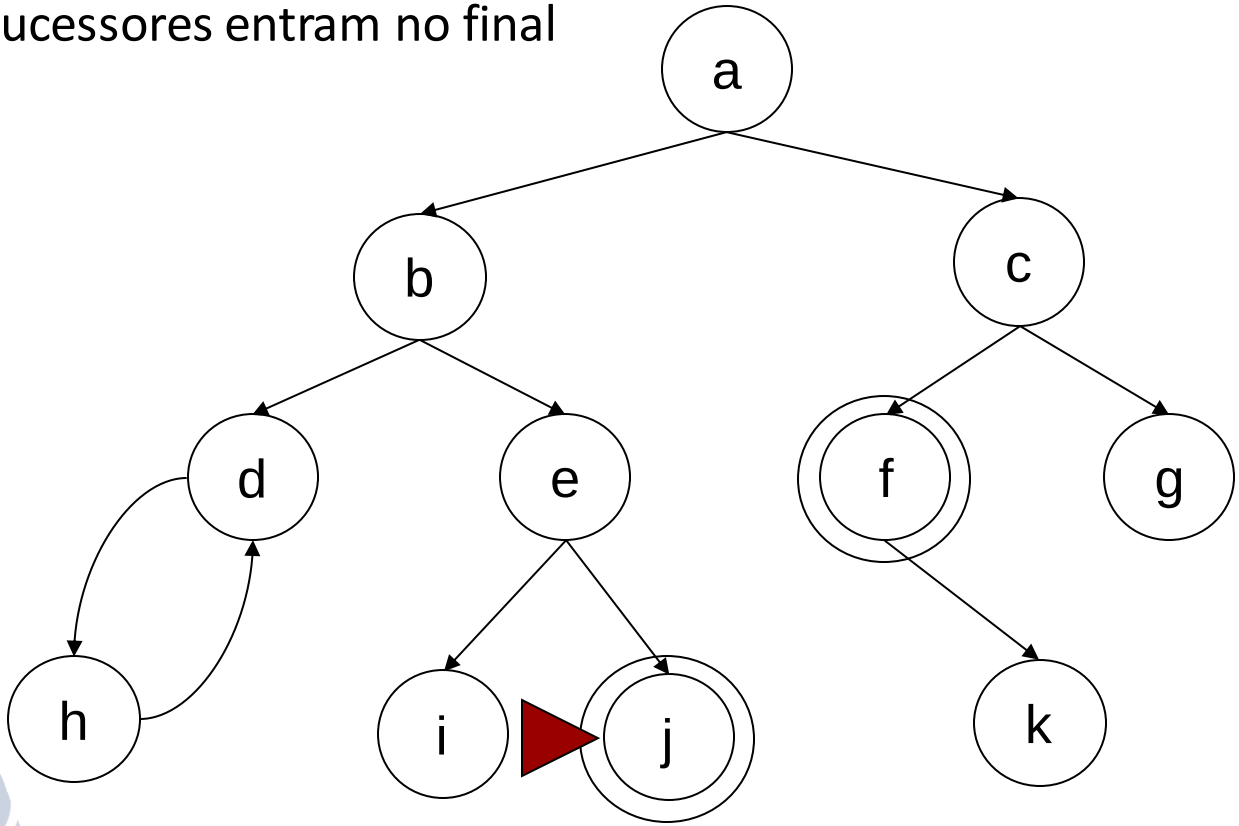
- Expanda o nó de menor profundidade ainda não expandido
- Implementação: ABERTOS é uma lista FIFO, i.e., novos sucessores entram no final




Inserir no final, remover da frente: i, j, k

# Busca em Largura

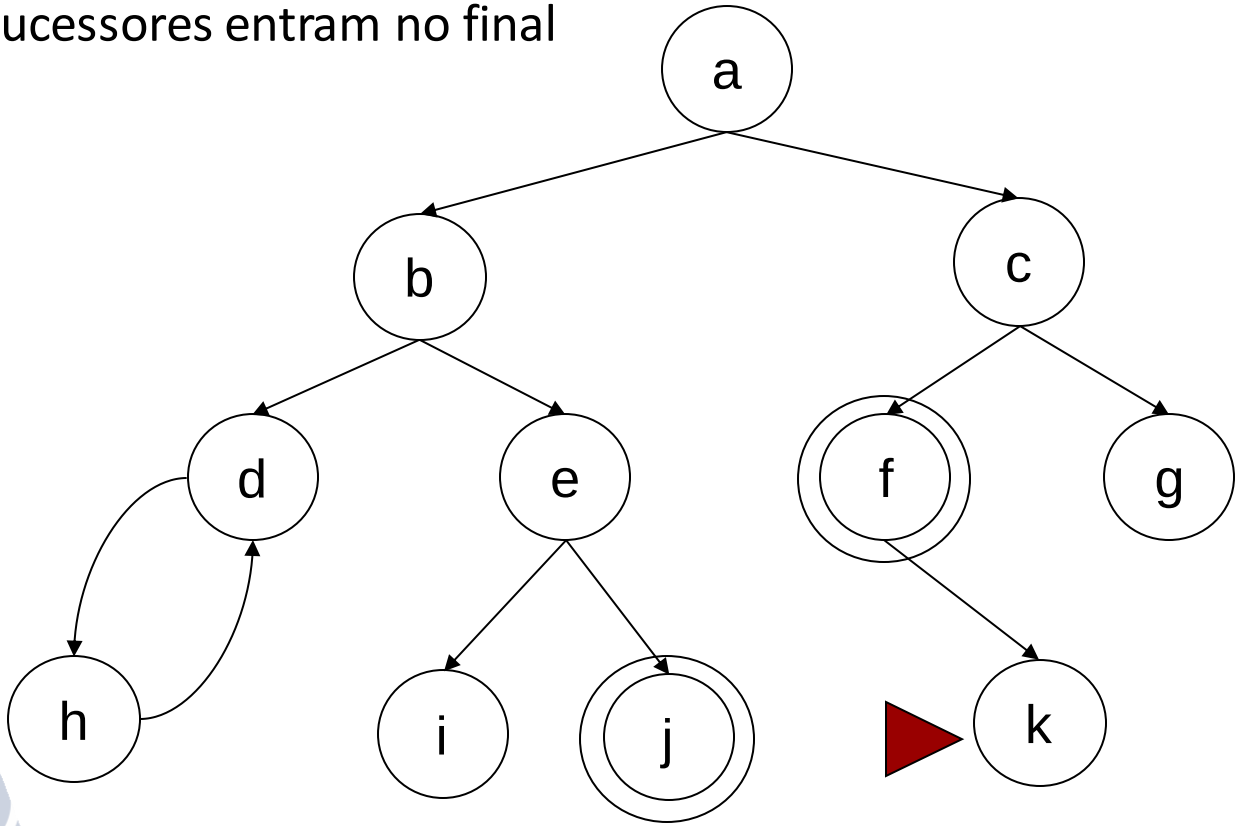
- Expanda o nó de menor profundidade ainda não expandido
- Implementação: ABERTOS é uma lista FIFO, i.e., novos sucessores entram no final




Inserir no final, remover da frente: j, k

# Busca em Largura

- Expanda o nó de menor profundidade ainda não expandido
- Implementação: ABERTOS é uma lista FIFO, i.e., novos sucessores entram no final

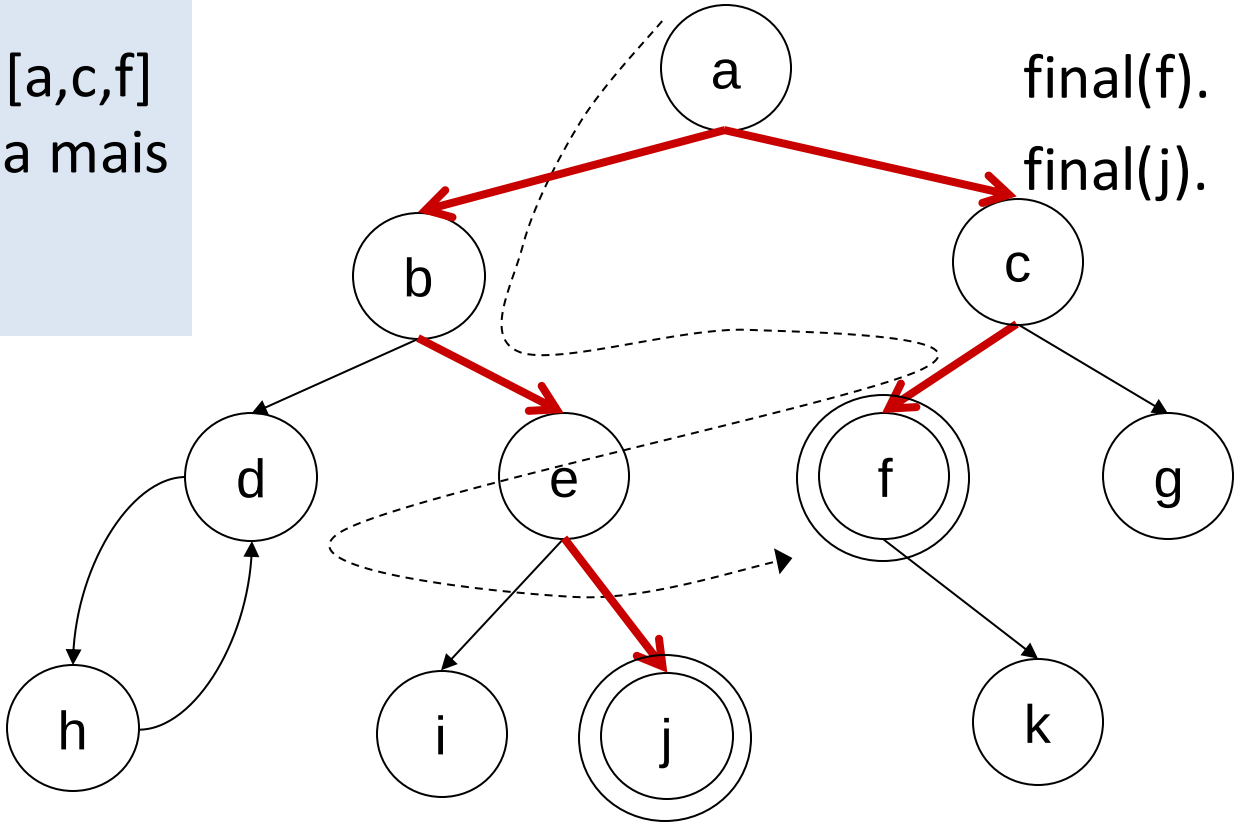



Inserir no final, remover da frente: k

# Busca em Largura

- Estado inicial: a
- Estados finais: j,f
- Nós visitados na ordem: a,b,c,d,e,f
- A solução mais curta [a,c,f] é encontrada antes da mais longa [a,b,e,j]

```
s(a,b).   s(c,f).   s(e,i).
s(a,c).   s(c,g).   s(e,j).
s(b,d).   s(d,h).   s(f,k).
s(b,e).   s(h,d).
```




```
final(f).
final(j).
```

# Algoritmo Busca em Largura

- **Um algoritmo de Busca em Largura pode ser definida da seguinte forma:**

1) Insere na fila F o nó u e marque-o como alcançado
2) Enquanto fila F não vazia faça
   **v** ← elemento da frente da fila
   (retire v da fila)
   **para todo w** que partir de **v**,
   e **w** ainda não foi alcançado
   - marque **w** como alcançado
   - insira **w** na fila **F**

# Propriedades da Busca em Largura

- **Completa**? Sim (se b é finito).
- **Tempo?** $1 + b + b^2 + \ \ldots + b^d + b\ (b^d\ - 1) = O(b^{d\ +\ 1})$.
- **Espaço?** $O(b^{d\ +\ 1})$ (mantém todo nó na memória).
- **Ótima**? Sim (se custo = 1 por passo); não ótima em geral.

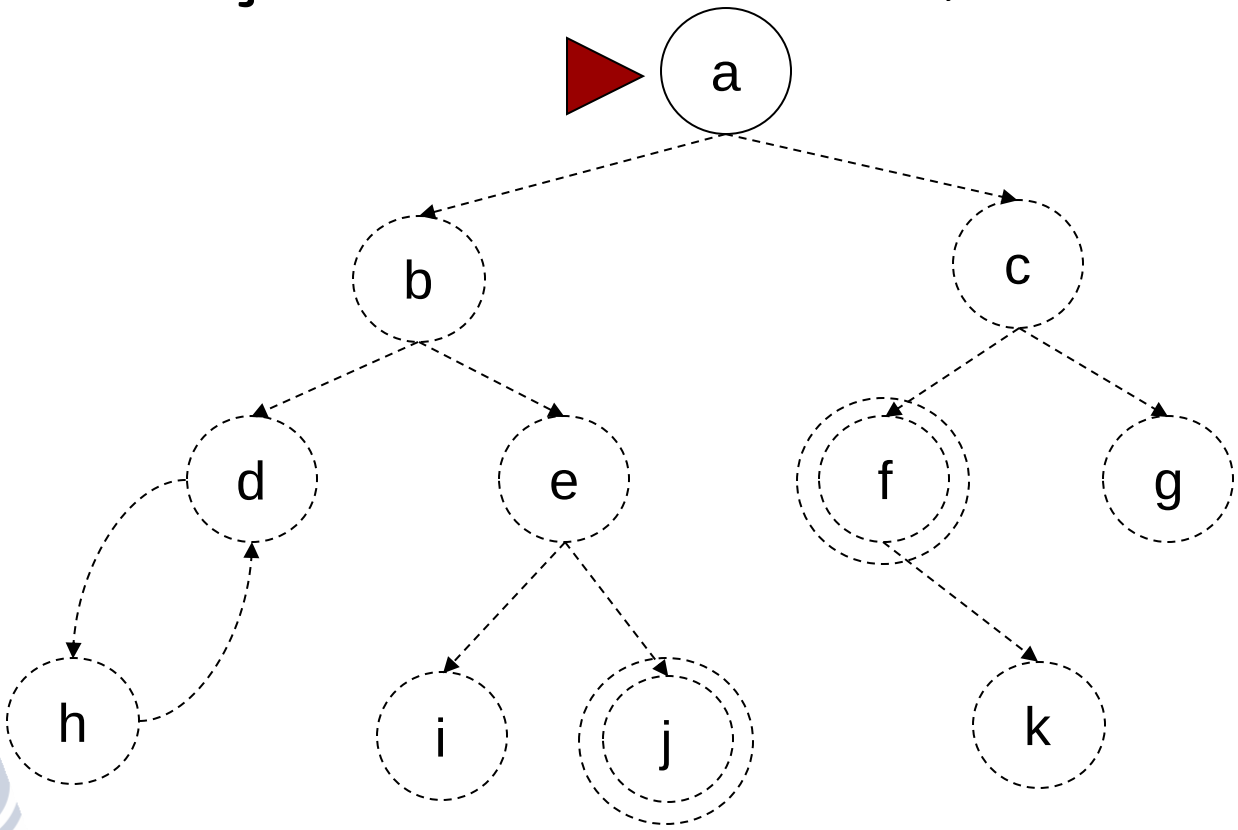
**Espaço é o maior problema:**

- pode-se facilmente gerar nós a 10MB/s, tal que 24h = 860GB.

b = fator de ramificação máximo
d = profundidade da solução de menor custo

# Busca em Profundidade

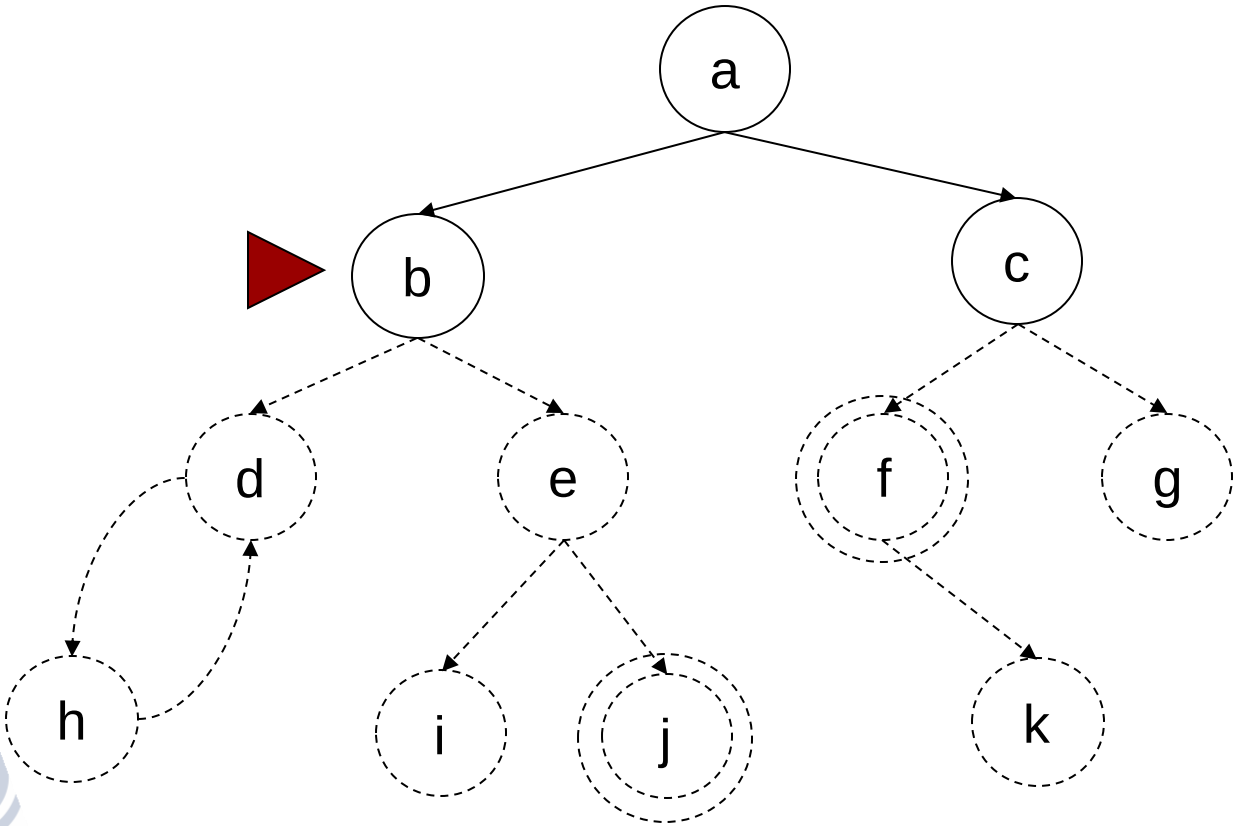
- Expanda o nó de maior profundidade ainda não expandido
- Implementação: ABERTOS é uma lista LIFO, i.e., novos sucessores entram no início




Inserir na frente, remover da frente: a

# Busca em Profundidade

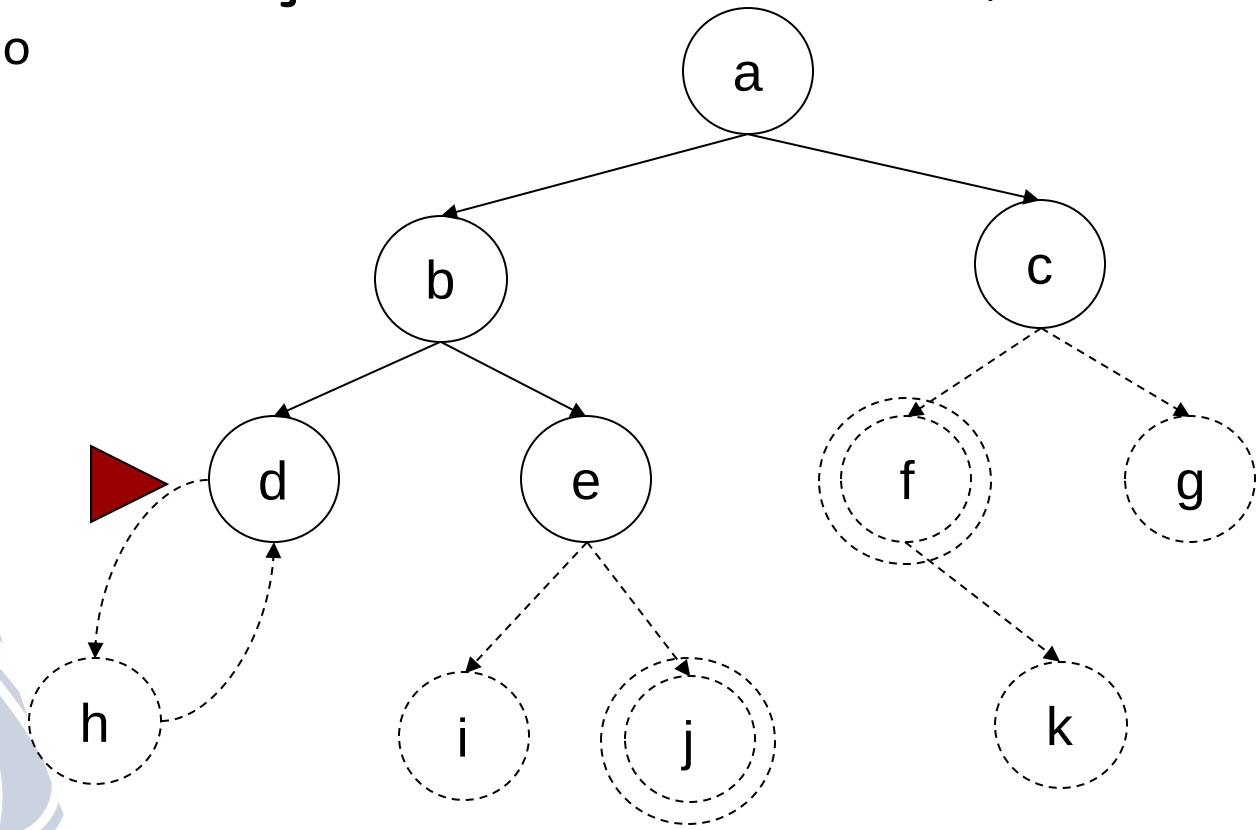
- Expanda o nó de maior profundidade ainda não expandido
- Implementação: ABERTOS é uma lista LIFO, i.e., novos sucessores entram no início




Inserir na frente, remover da frente: b, c

# Busca em Profundidade

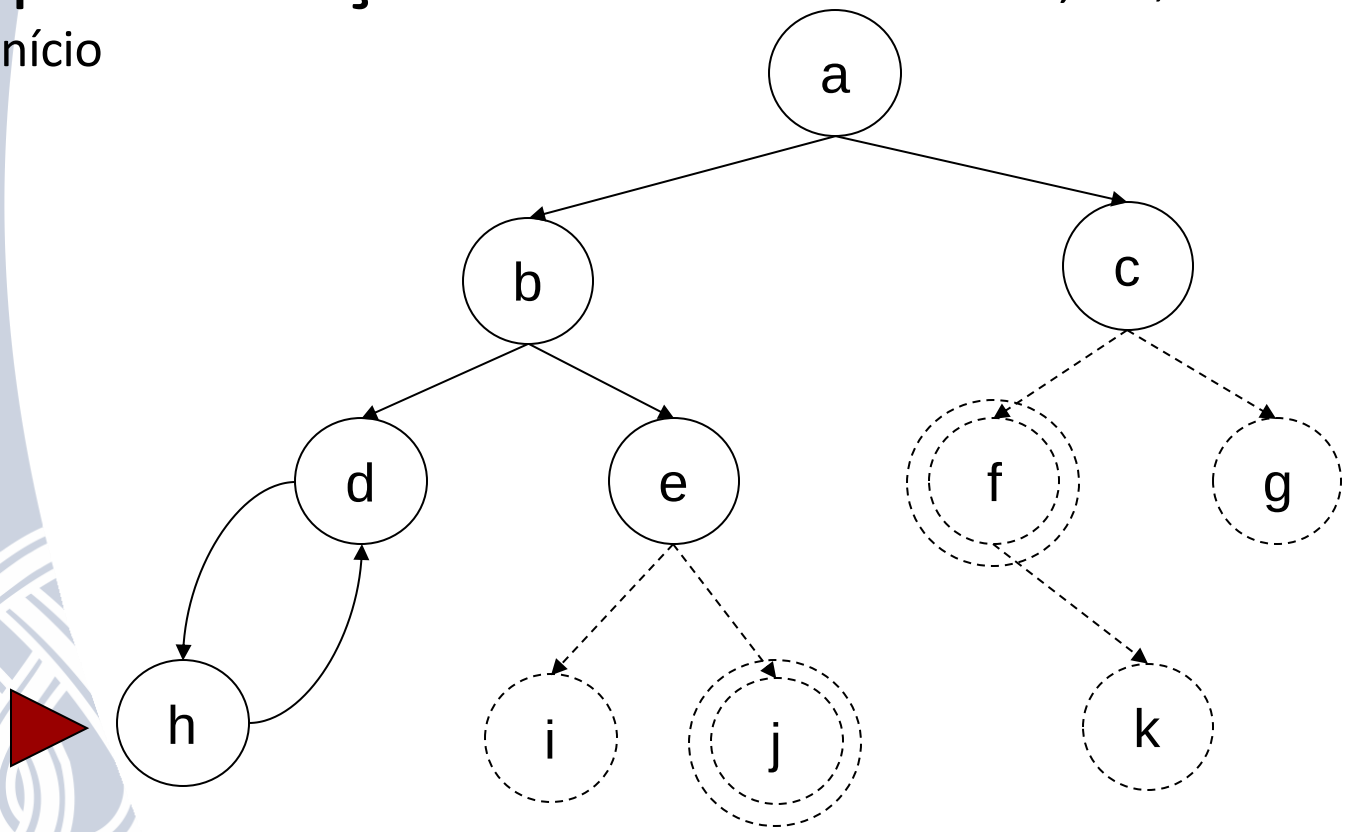
- Expanda o nó de maior profundidade ainda não expandido
- Implementação: ABERTOS é uma lista LIFO, i.e., novos sucessores entram no início




Inserir na frente, remover da frente: d, e, c

# Busca em Profundidade

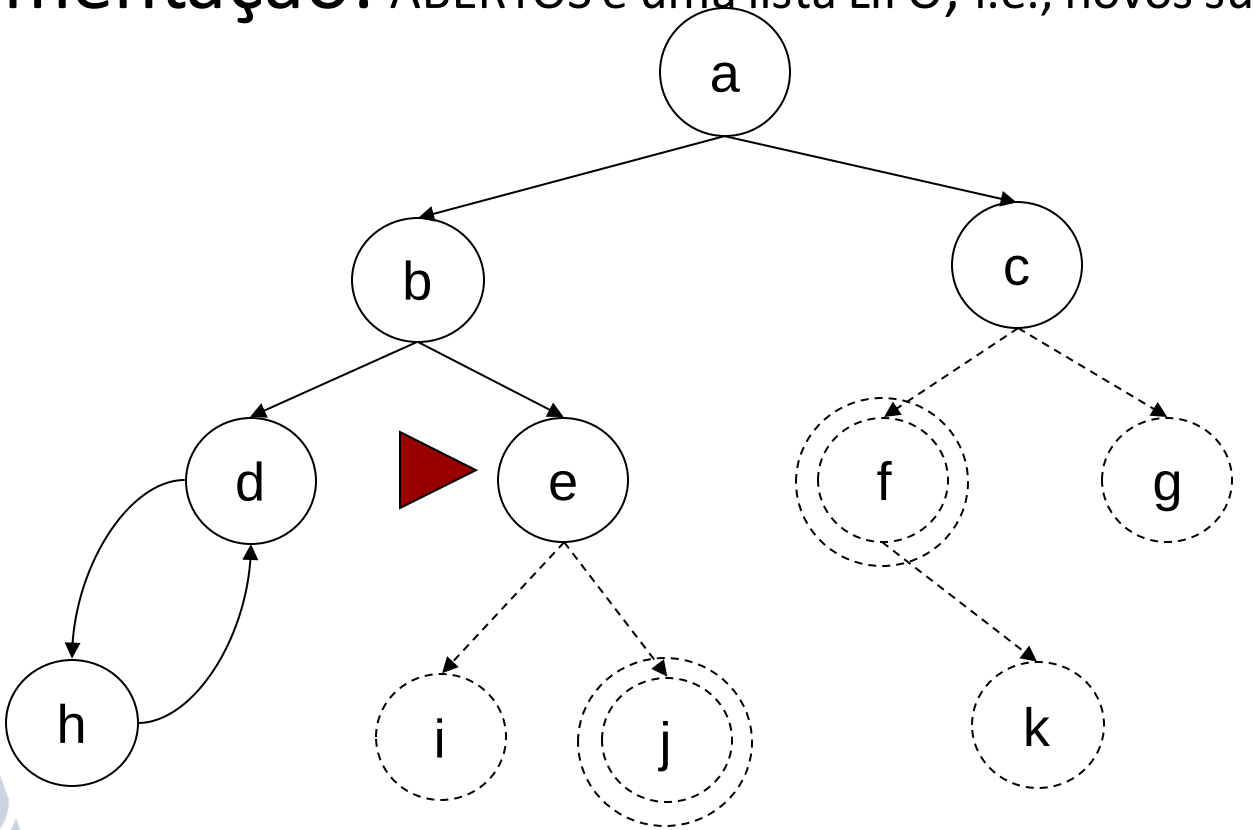
- Expanda o nó de maior profundidade ainda não expandido
- Implementação: ABERTOS é uma lista LIFO, i.e., novos sucessores entram no início




Inserir na frente, remover da frente: h, e, c

# Busca em Profundidade

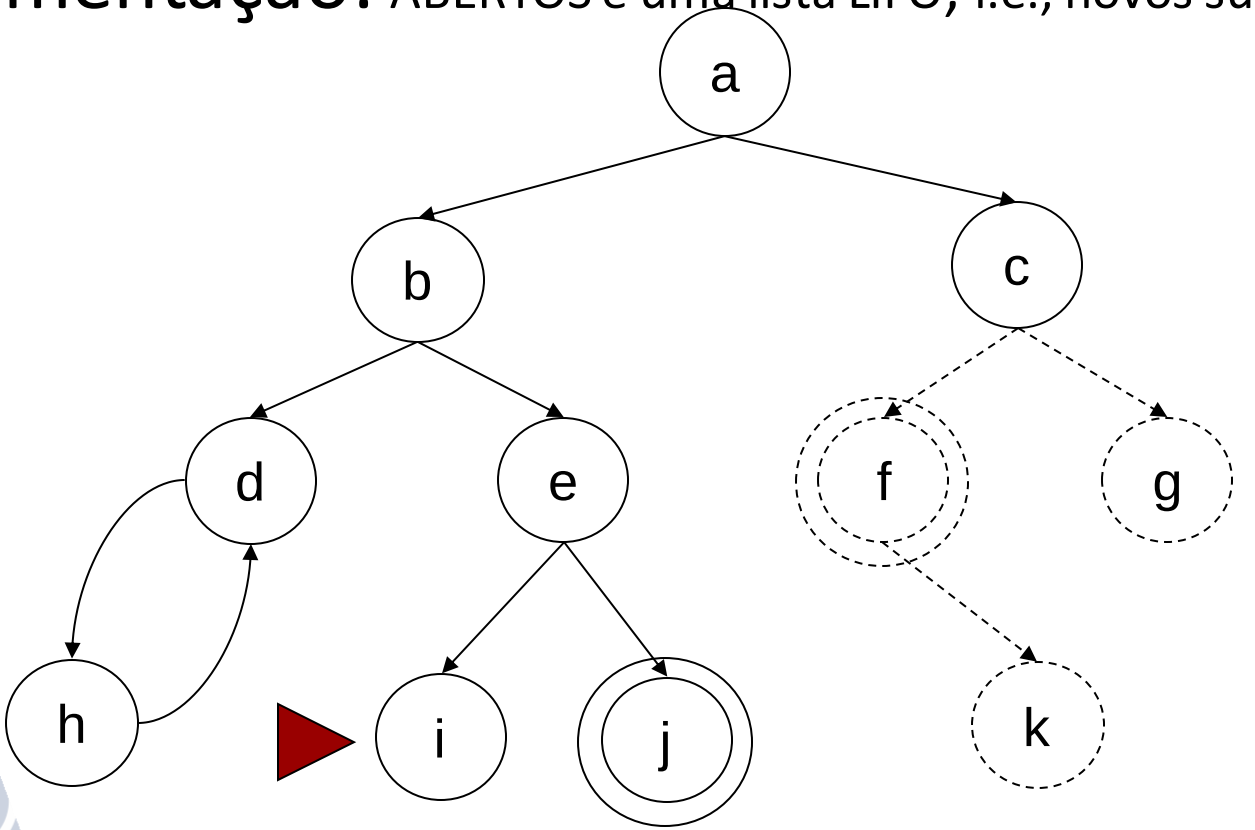
- Expanda o nó de maior profundidade ainda não expandido
- Implementação: ABERTOS é uma lista LIFO, i.e., novos sucessores entram no início




Inserir na frente, remover da frente: e, c

# Busca em Profundidade

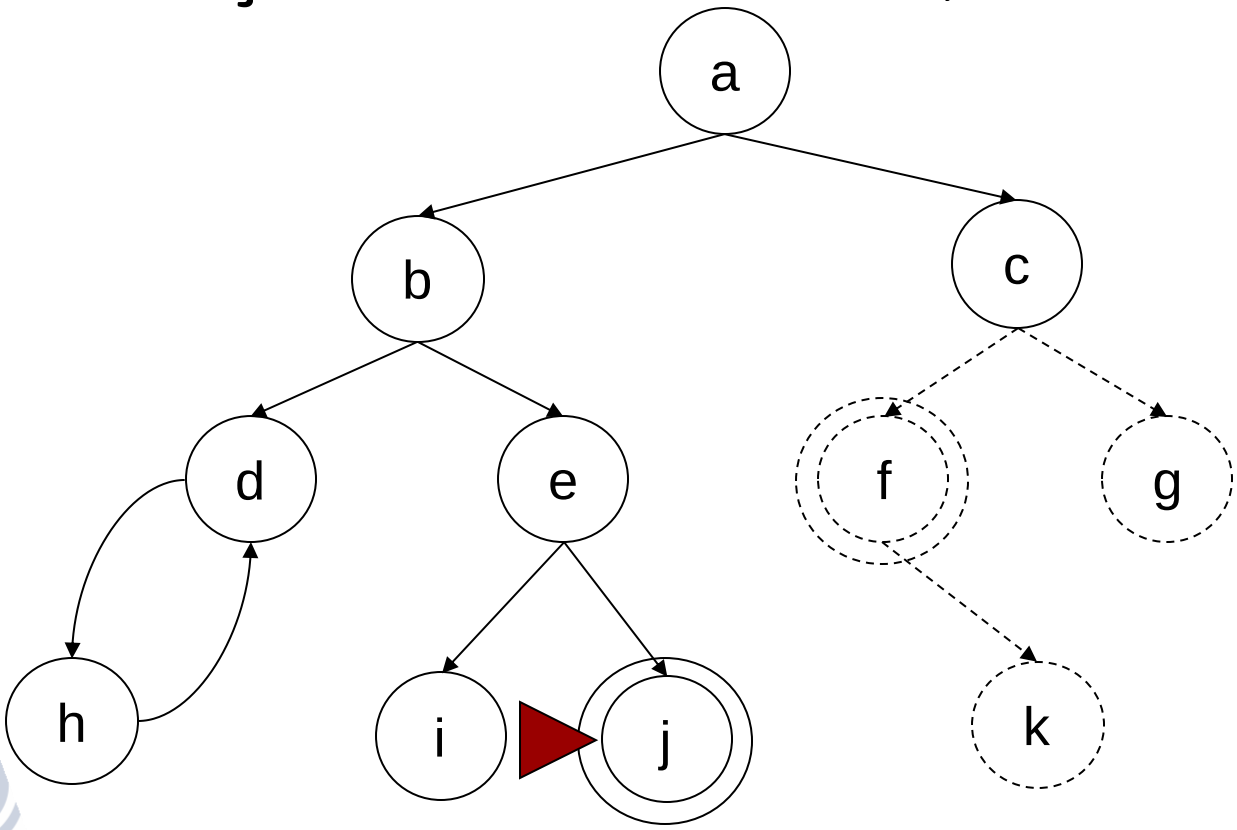
- Expanda o nó de maior profundidade ainda não expandido
- Implementação: ABERTOS é uma lista LIFO, i.e., novos sucessores entram no início




Inserir na frente, remover da frente: i, j, c

# Busca em Profundidade

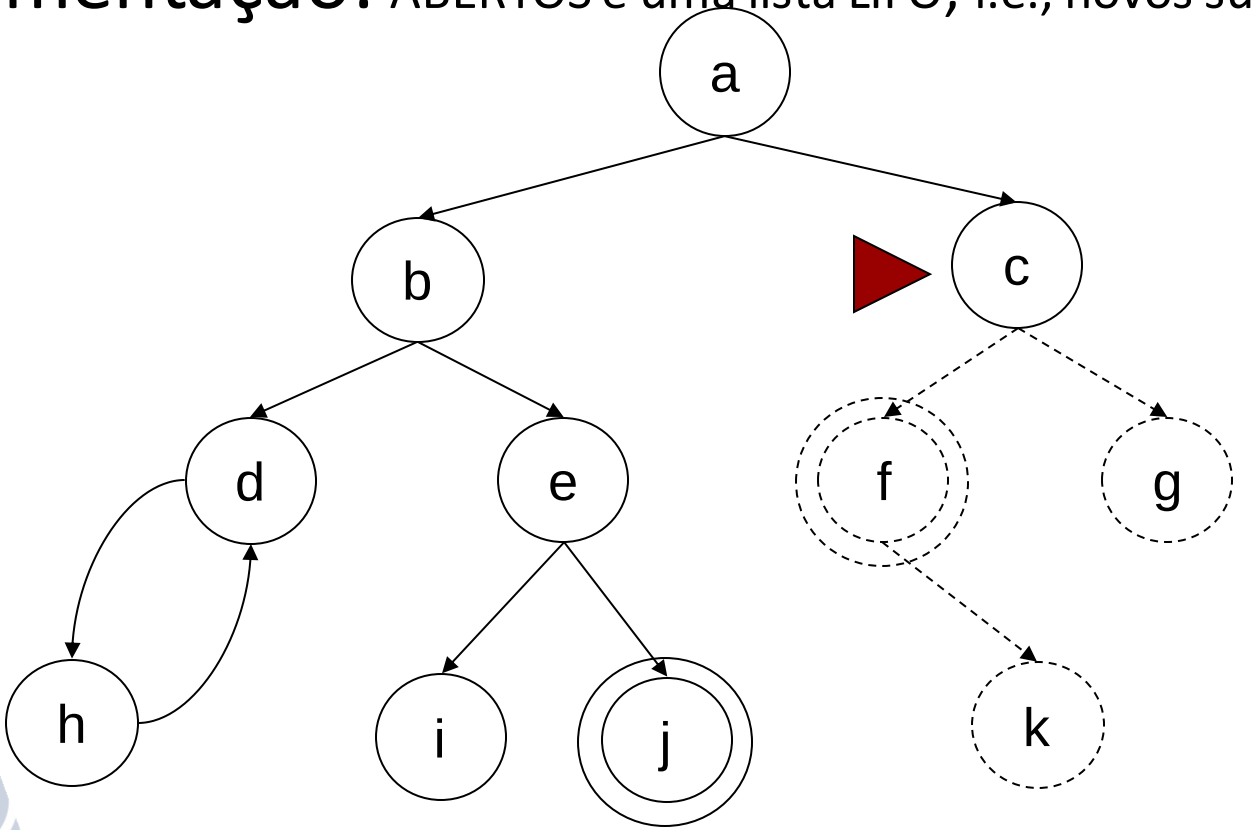
- Expanda o nó de maior profundidade ainda não expandido
- Implementação: ABERTOS é uma lista LIFO, i.e., novos sucessores entram no início




Inserir na frente, remover da frente: j, c

# Busca em Profundidade

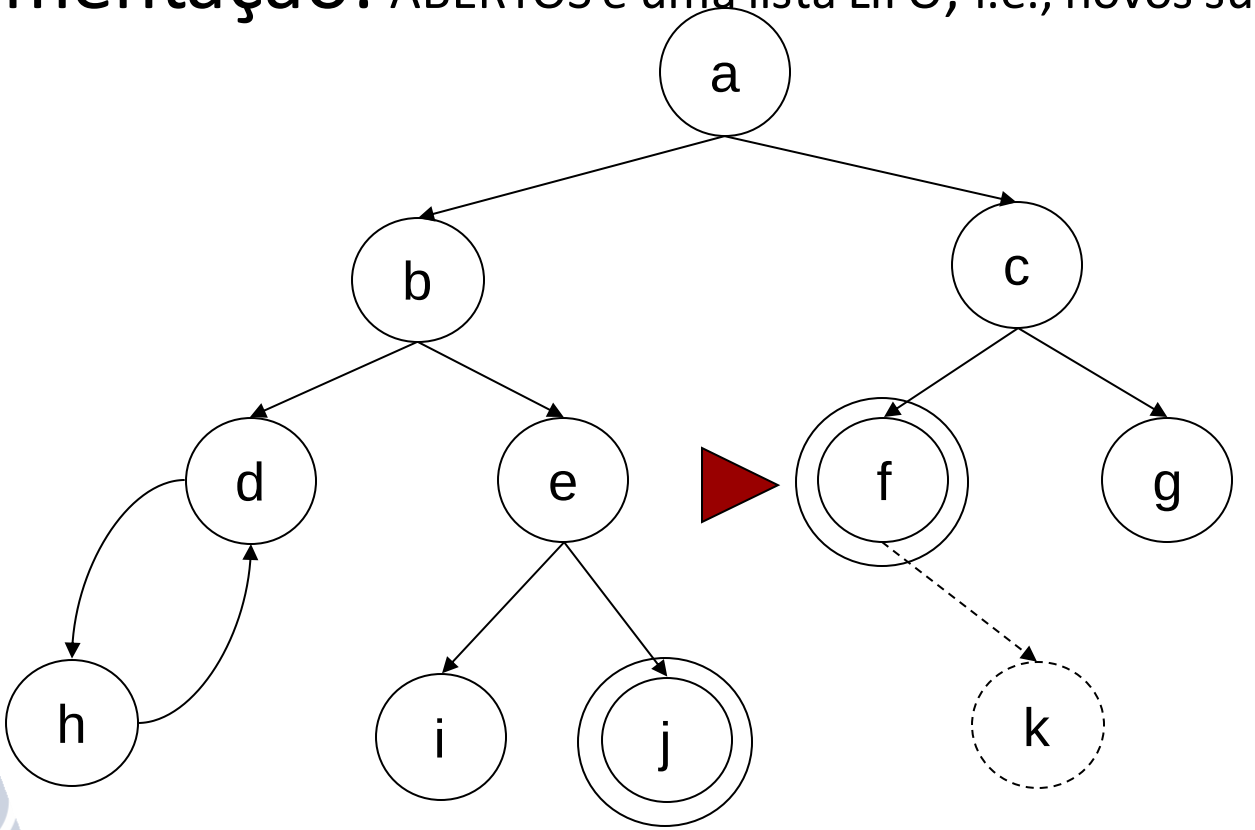
- Expanda o nó de maior profundidade ainda não expandido
- Implementação: ABERTOS é uma lista LIFO, i.e., novos sucessores entram no início




Inserir na frente, remover da frente: c

# Busca em Profundidade

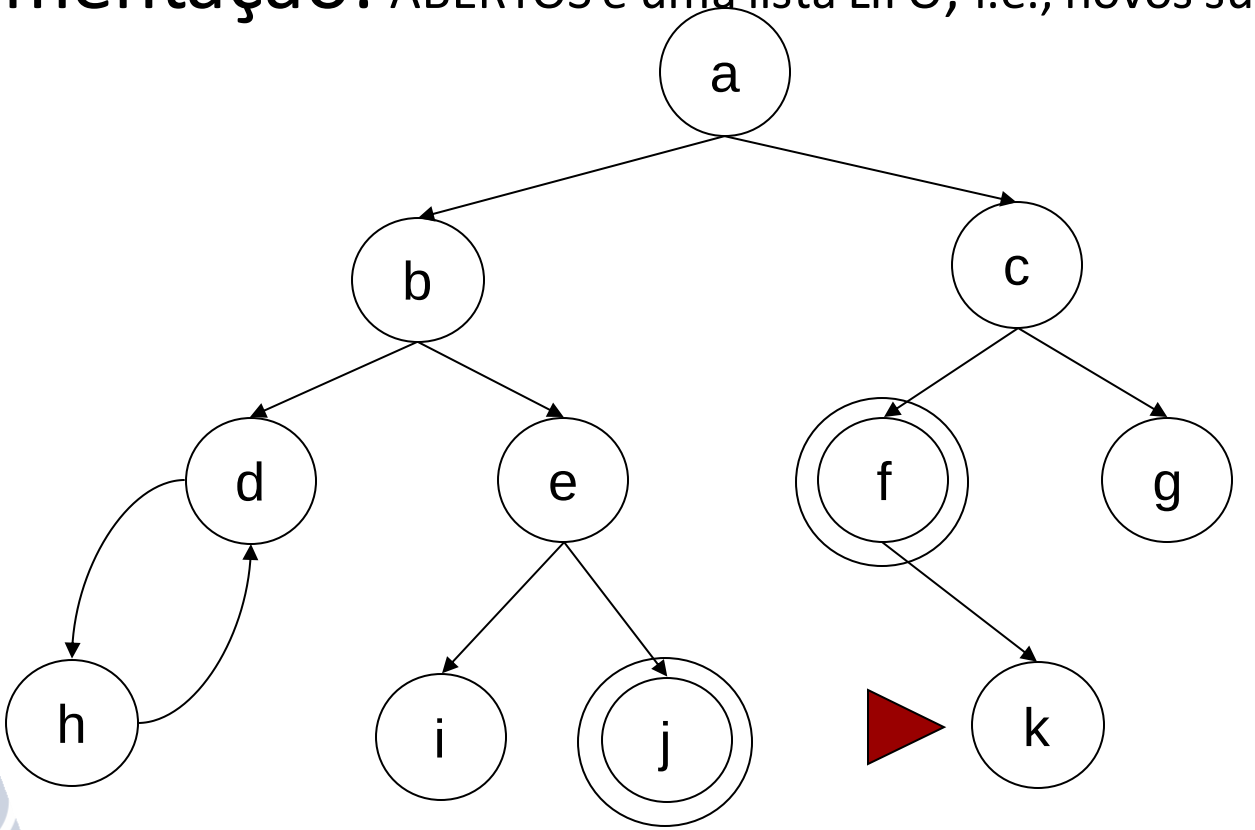
- Expanda o nó de maior profundidade ainda não expandido
- Implementação: ABERTOS é uma lista LIFO, i.e., novos sucessores entram no início




Inserir na frente, remover da frente: f, g

# Busca em Profundidade

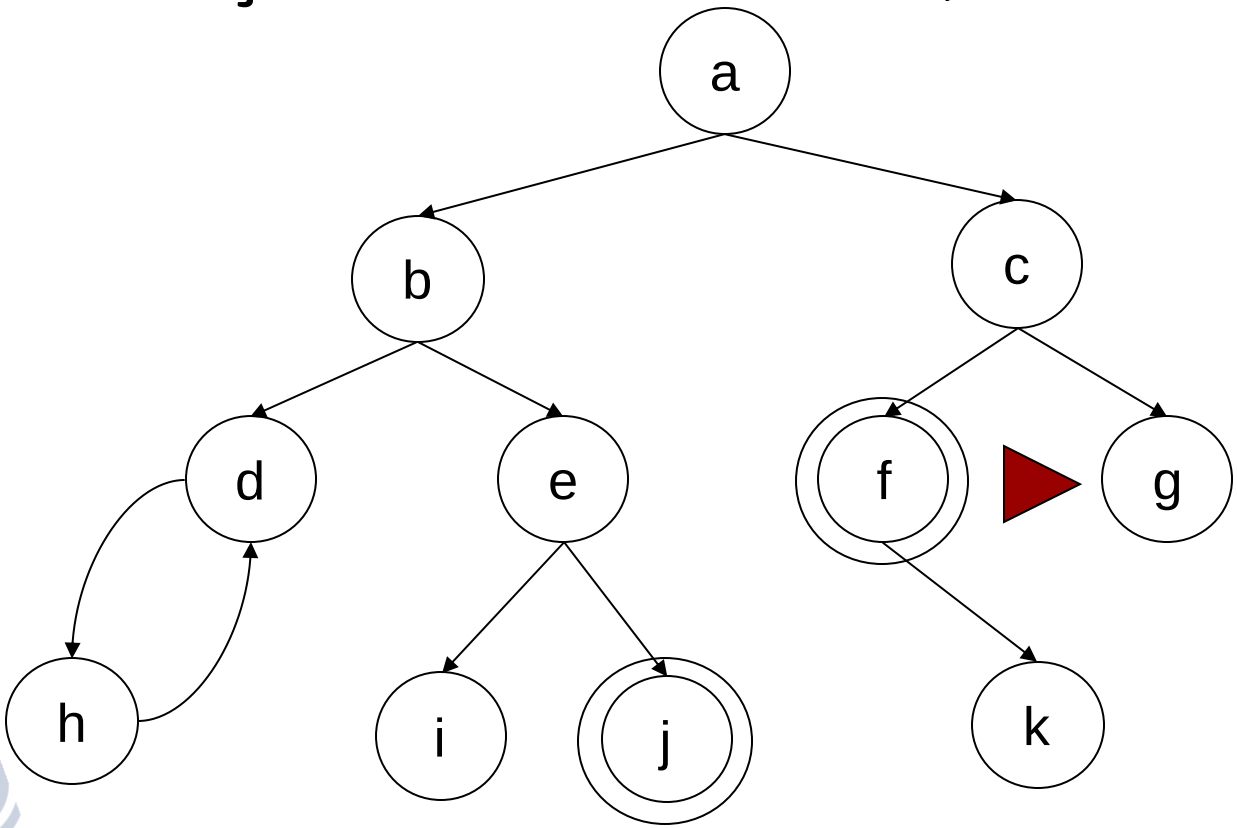
- Expanda o nó de maior profundidade ainda não expandido
- Implementação: ABERTOS é uma lista LIFO, i.e., novos sucessores entram no início




Inserir na frente, remover da frente: k, g

# Busca em Profundidade

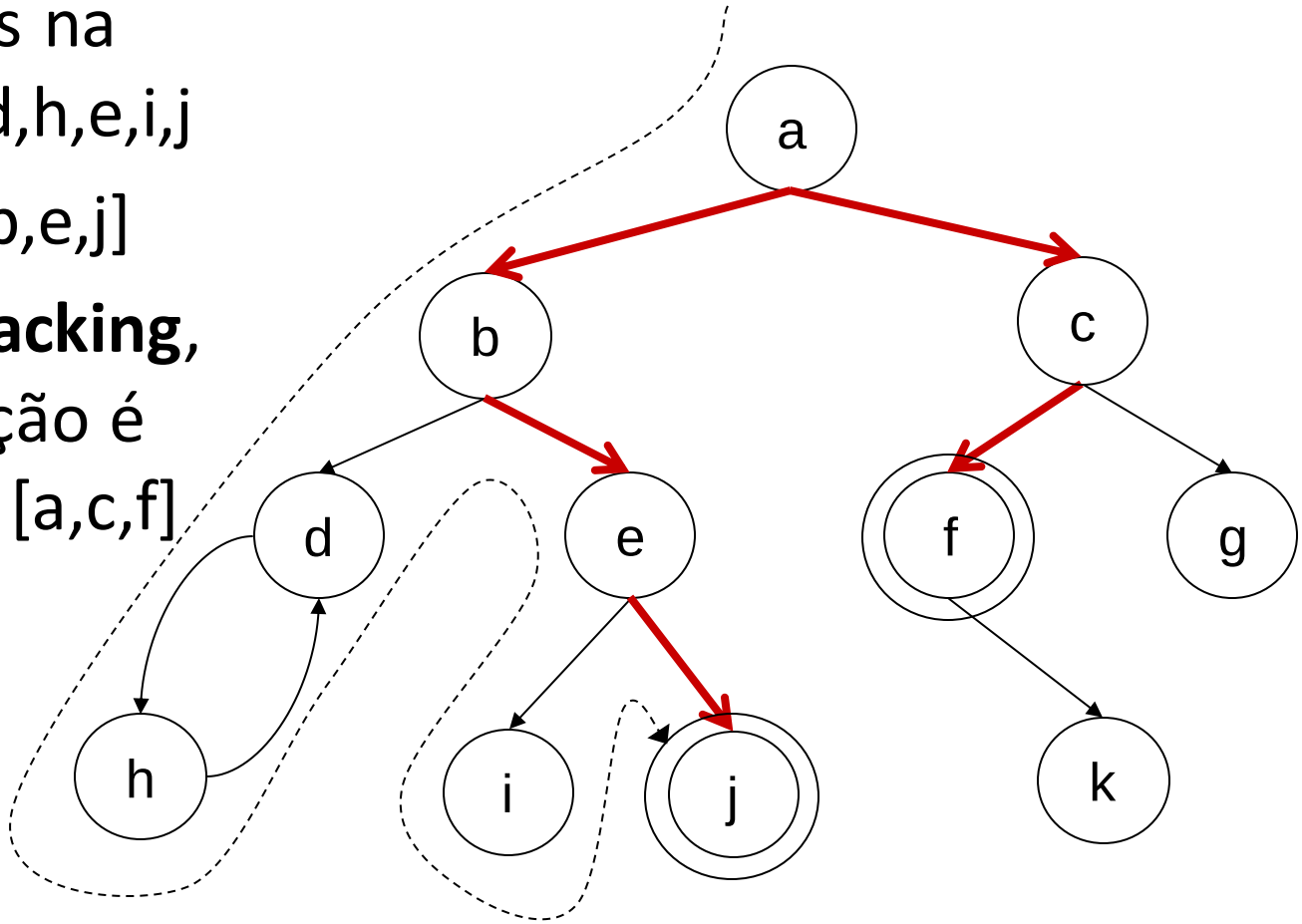
- Expanda o nó de maior profundidade ainda não expandido
- Implementação: ABERTOS é uma lista LIFO, i.e., novos sucessores entram no início




Inserir na frente, remover da frente: g

# Busca em Profundidade

- **Estado inicial**: a
- **Estados finais**: j,f
- Nós visitados na ordem: a,b,d,h,e,i,j
- **Solução**: [a,b,e,j]
- Após **backtracking**, a outra solução é encontrada: [a,c,f]



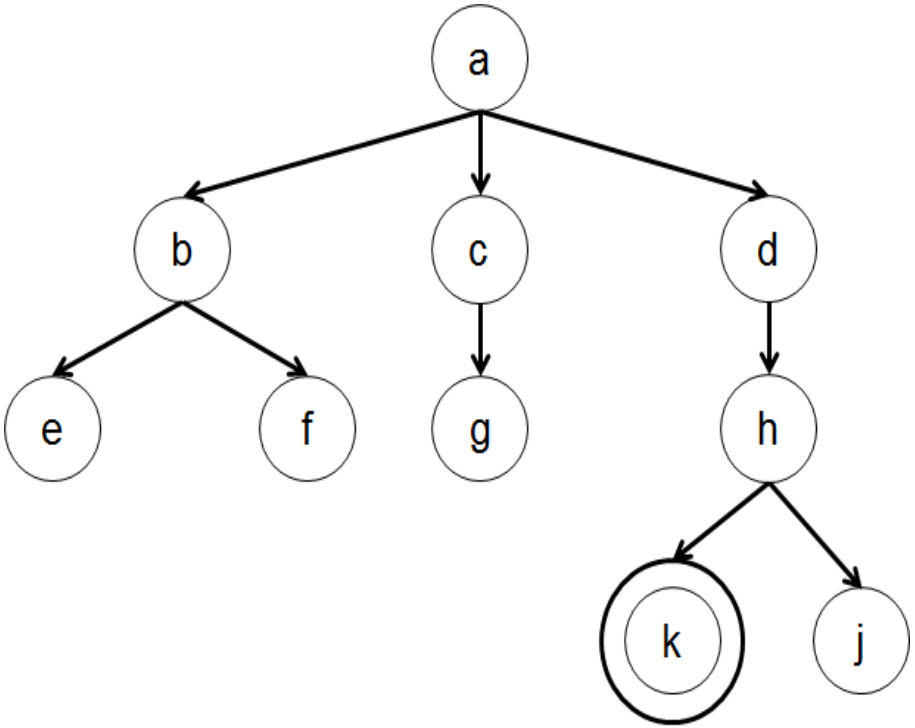
# Algoritmo Busca em Profundidade

- **Um algoritmo de Busca em Profundidade pode ser:**

1) Empilhe um nó **v** origem na pilha **P** e marque-o como alcançado
2) Enquanto a pilha P não vazia faça
   **v** ← elemento do topo da pilha
   //(desempilhe)
   **se existe w** a partir de **v**, // (v,w)
   e **w** ainda não foi alcançado
   - marque **w** como alcançado
   - insira **w** na pilha **P** // empilhe(w)

# Exercício - Busca em Profundidade

- Considere a busca no espaço abaixo, em que **“a”** é o estado inicial e **“k”** é o estado final.




- **1.** Modele o espaço de busca com as regras **pode_ir(X,Y).**

# Propriedades da Busca em Profundidade

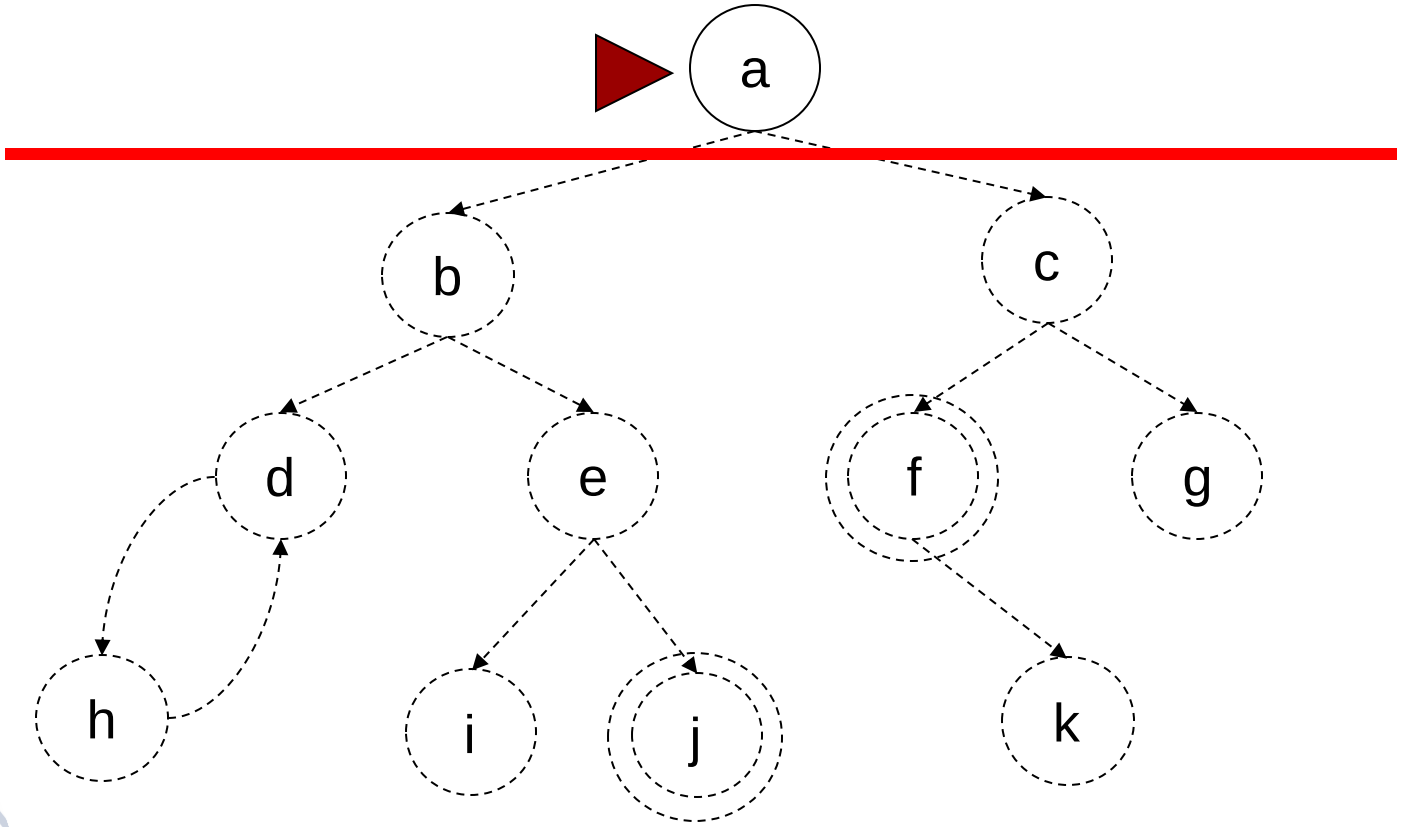
- **Completa**? Não: falha em espaços de profundidade infinita, espaços com loops. Modificar para evitar estados repetidos no caminho → completa em espaços finitos.
- **Tempo**? $O(b^m)$ terrível se m é muito maior que d, mas se soluções são densas, pode ser mais rápido que a busca em largura.
- **Espaço**? $O(b\ m)$ i. e. espaço linear!
- **Ótima**? Não.

b = fator de ramificação máximo
m = profundidade máxima do espaço de estados

# Busca Profundidade limitada

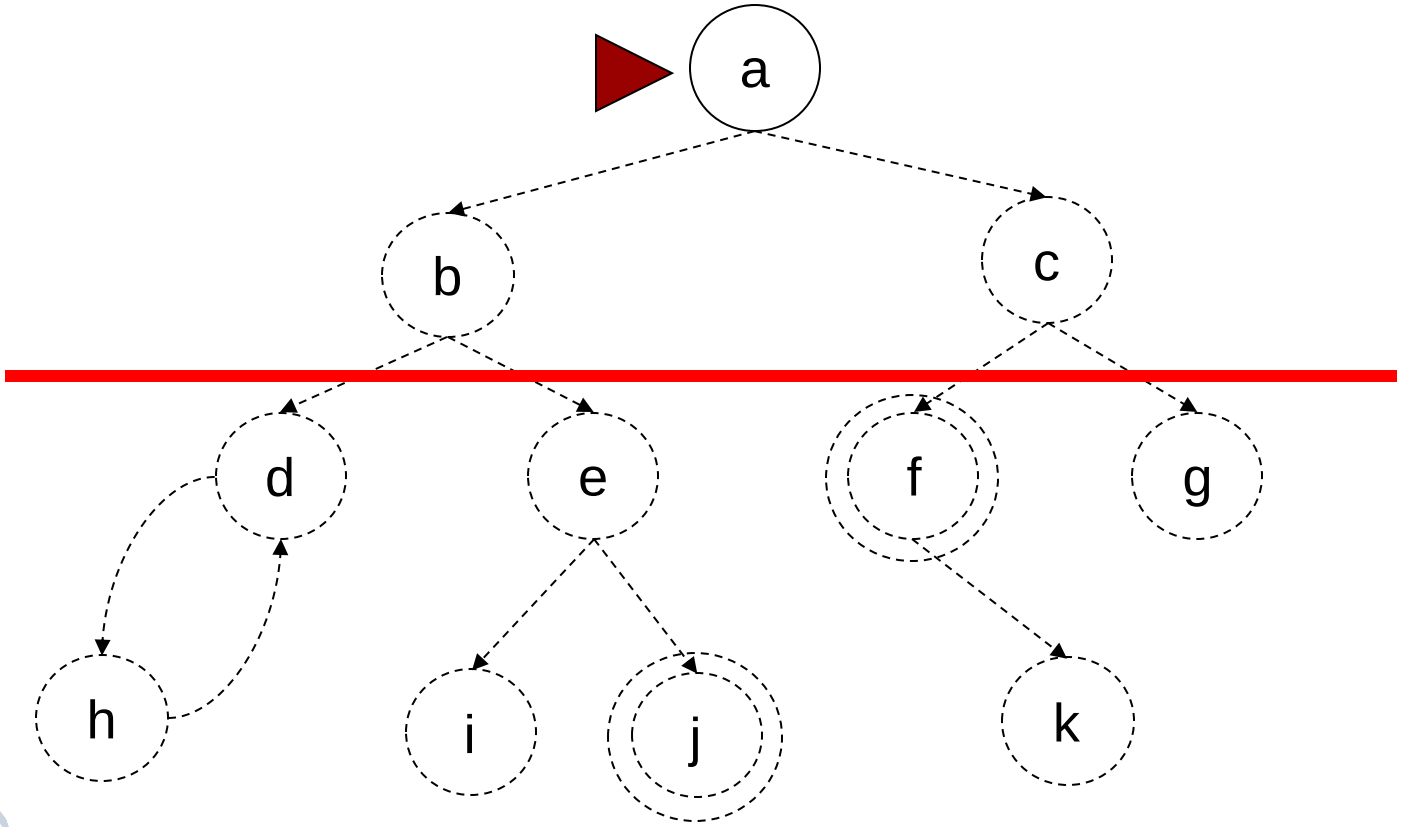
- Busca em Profundidade Limitada = Busca em Profundidade com limite de profundidade l, i.e., nós na profundidade l não têm sucessores.
- Implementação recursiva.

# Busca em Profundidade Limitada (L=0)



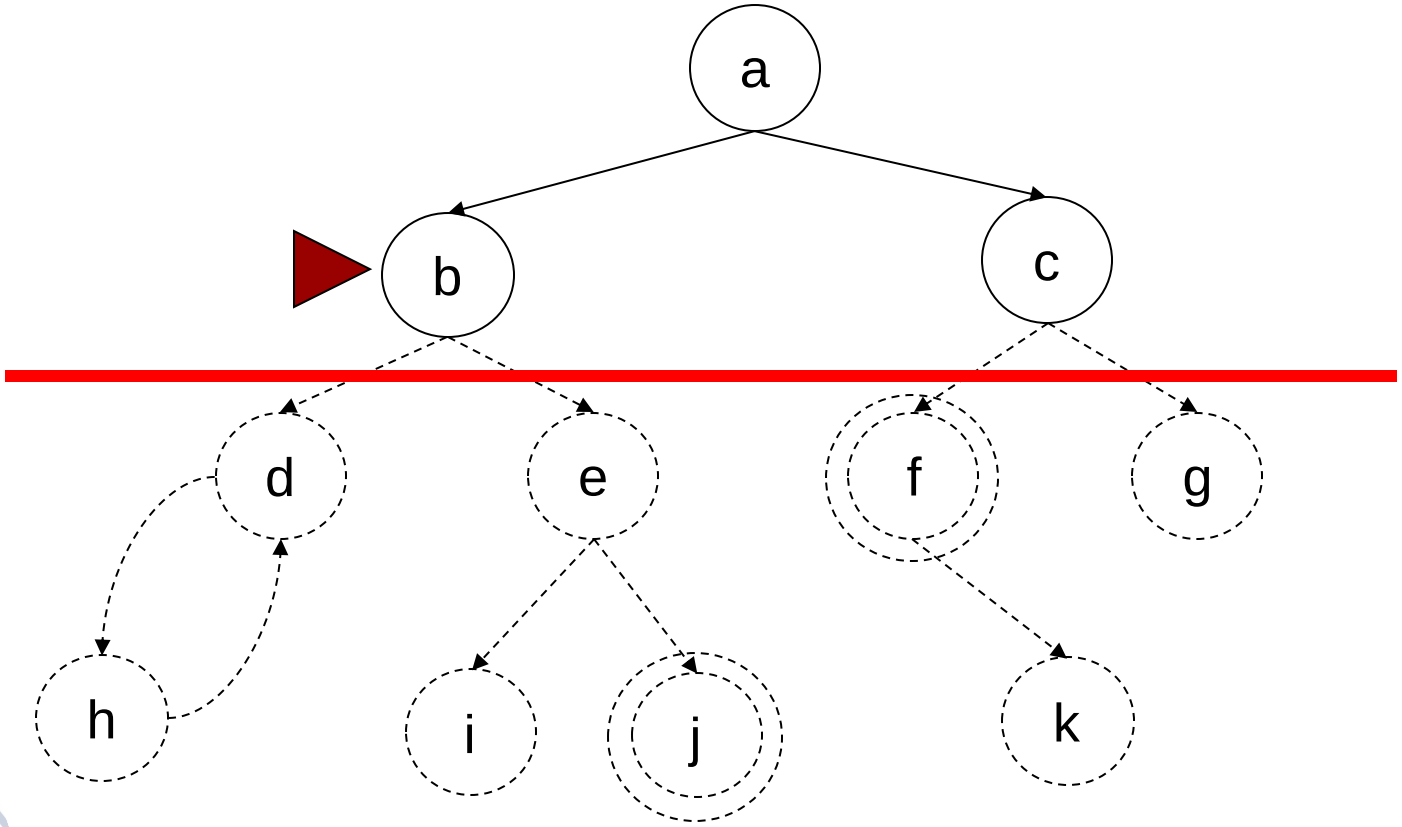

Inserir na frente, remover da frente: a

# Busca em Profundidade Limitada (L=1)



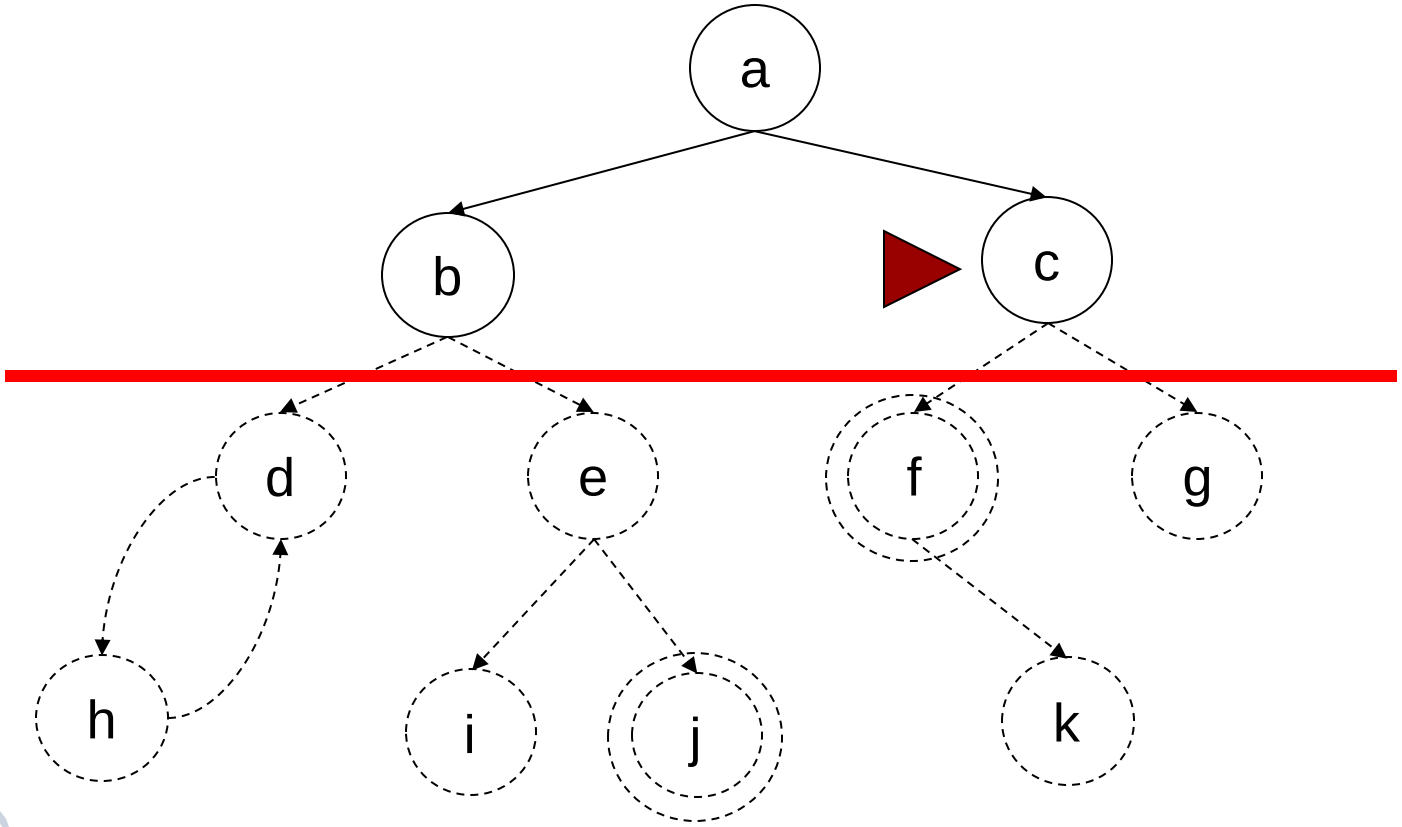

Inserir na frente, remover da frente: a

# Busca em Profundidade Limitada (L=1)



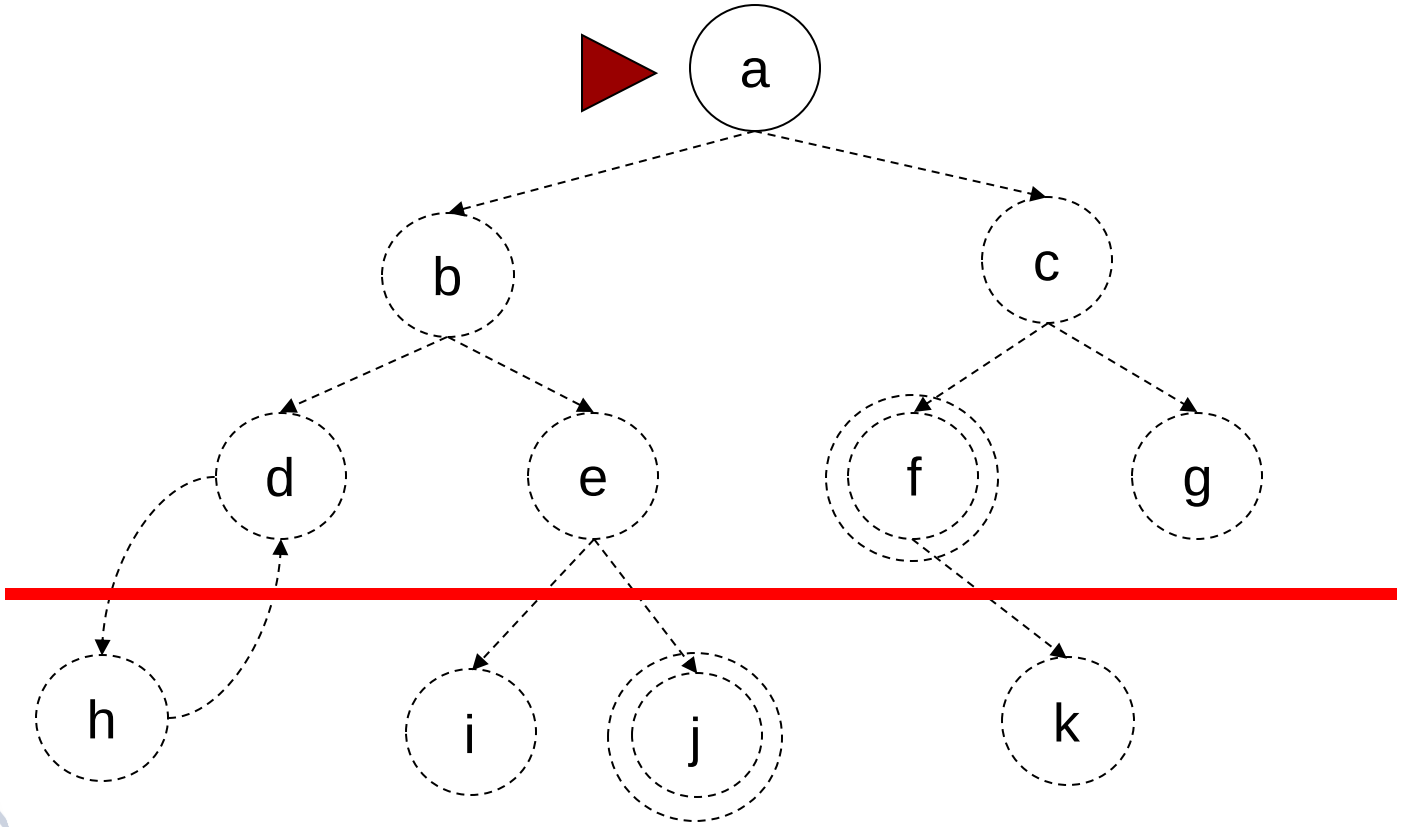

Inserir na frente, remover da frente: b, c

# Busca em Profundidade Limitada (L=1)



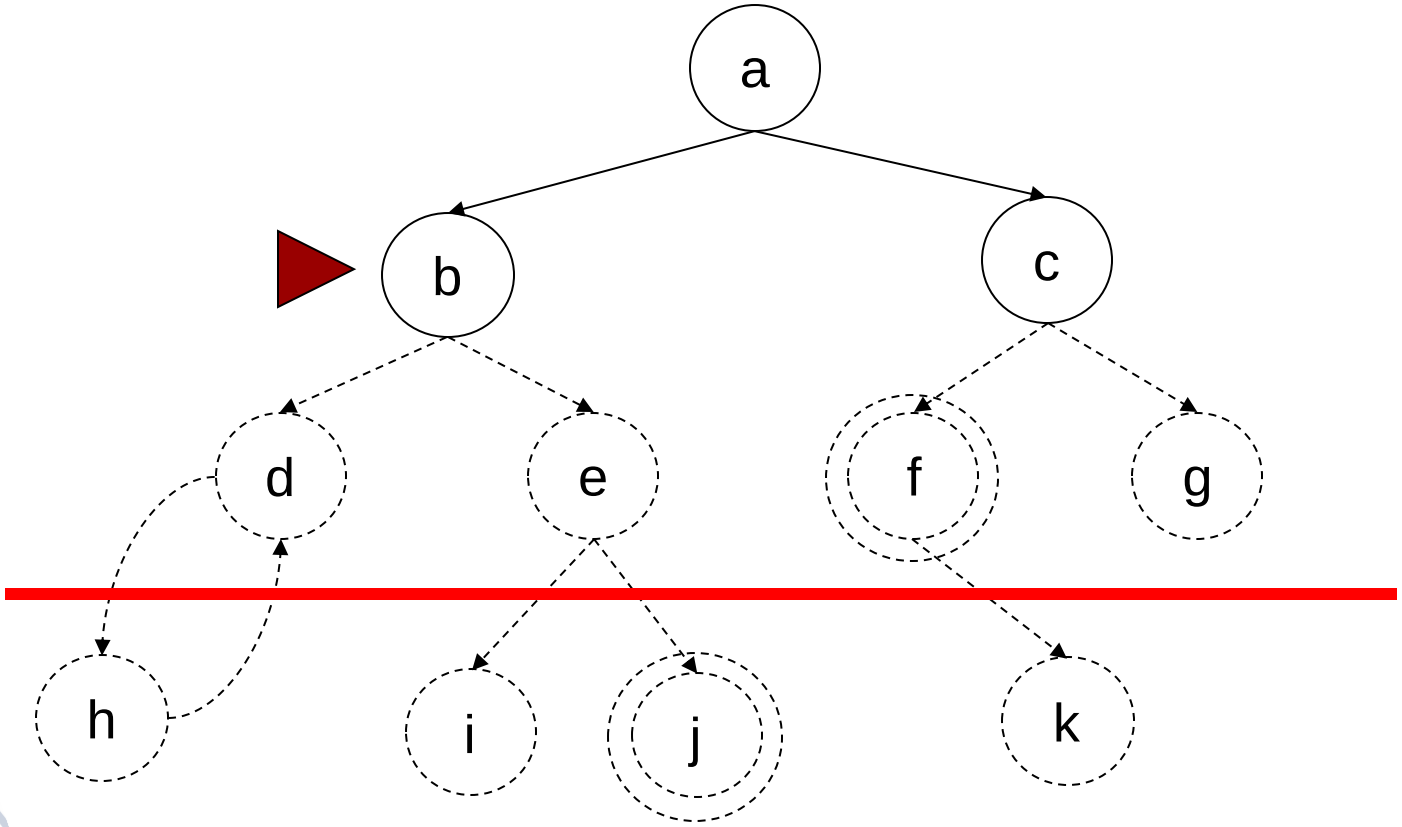

Inserir na frente, remover da frente: c

# Busca em Profundidade Limitada (L=2)



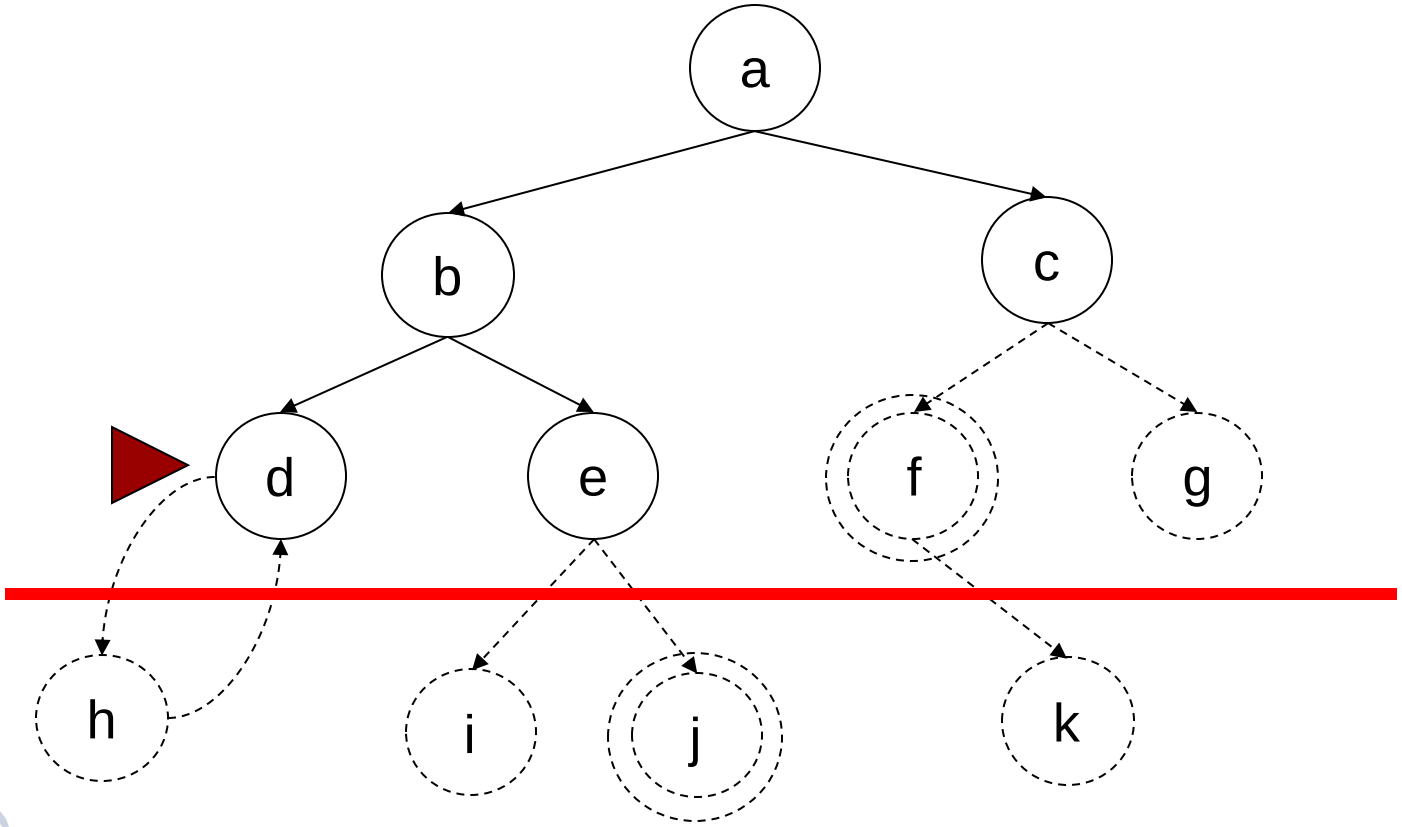

Inserir na frente, remover da frente: a

# Busca em Profundidade Limitada (L=2)



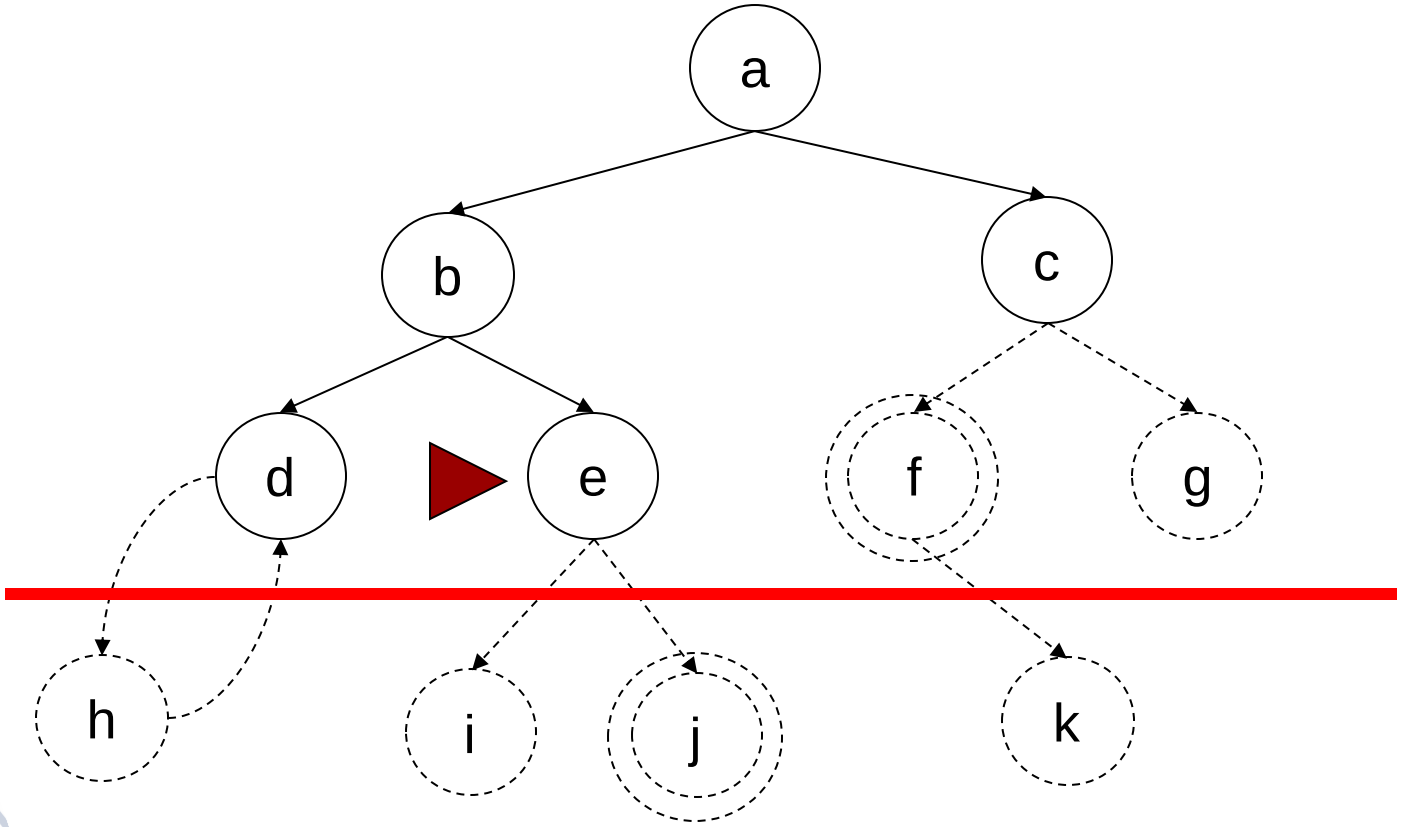

Inserir na frente, remover da frente: b, c

# Busca em Profundidade Limitada (L=2)



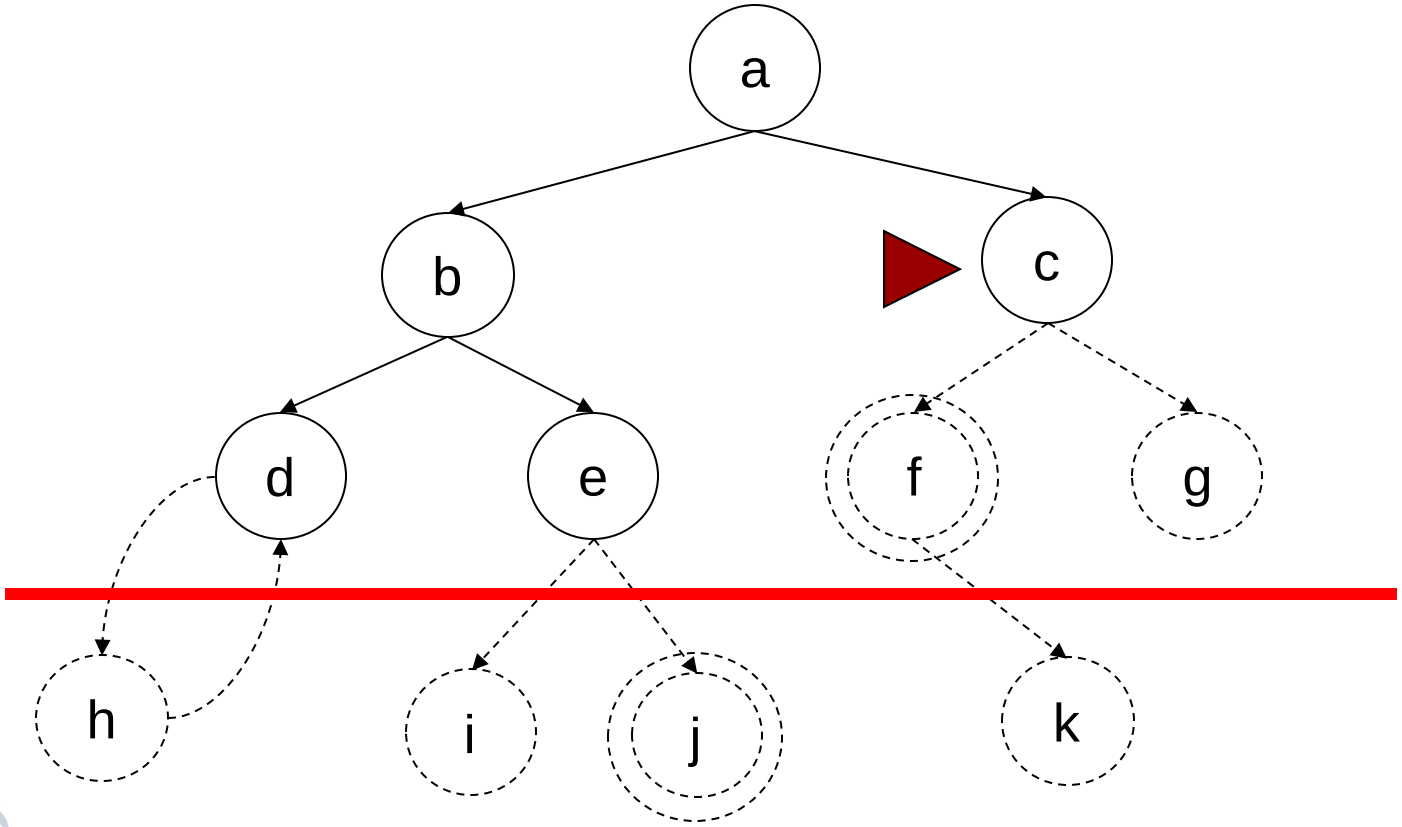

Inserir na frente, remover da frente: d, e, c

# Busca em Profundidade Limitada (L=2)



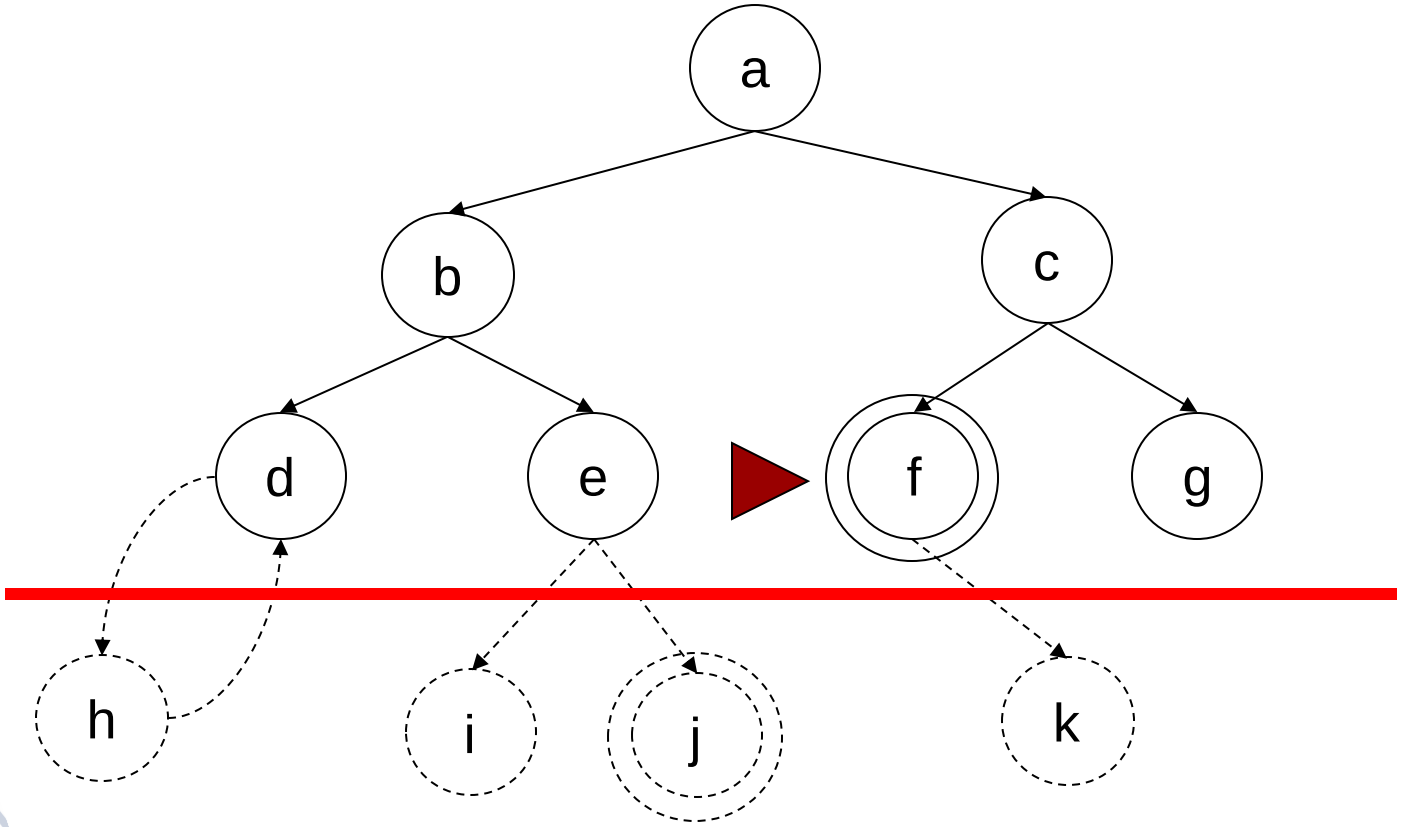

Inserir na frente, remover da frente: e, c

# Busca em Profundidade Limitada (L=2)



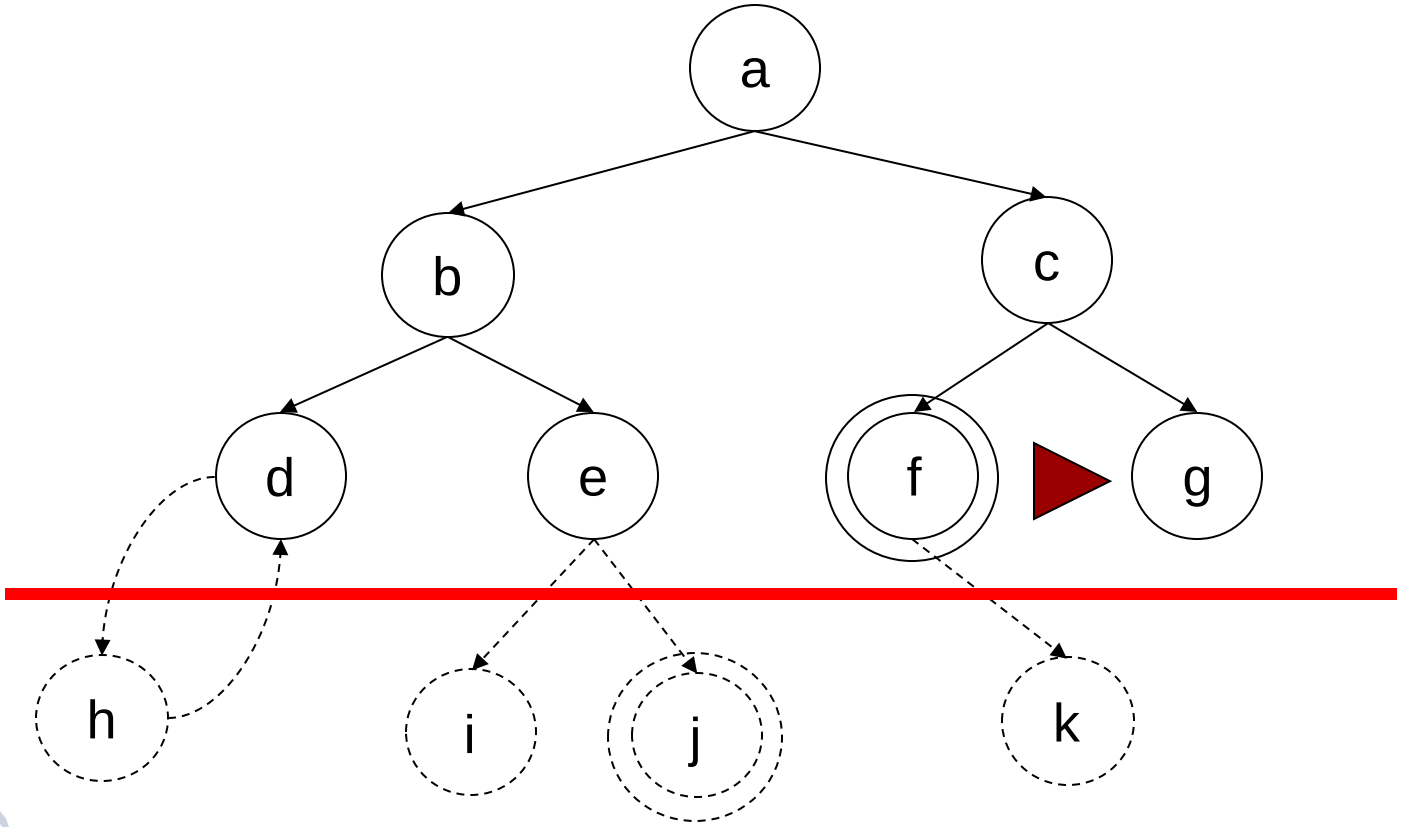

Inserir na frente, remover da frente: c

# Busca em Profundidade Limitada (L=2)




Inserir na frente, remover da frente: f, g

# Busca em Profundidade Limitada (L=2)




Inserir na frente, remover da frente: g

# Busca em Profundidade Limitada

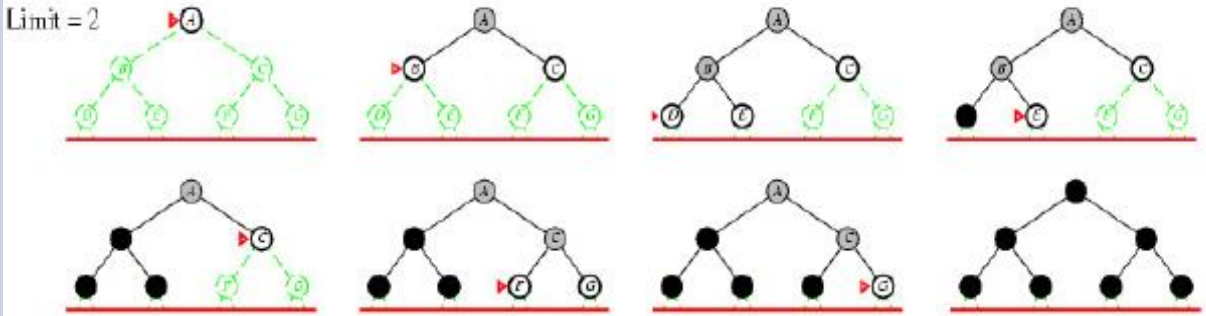
- **Um problema** com a busca em profundidade limitada é que **não se tem previamente um limite razoável**
  - Se o limite for muito pequeno (menor que qualquer caminho até uma solução) então a busca falha
  - Se o limite for muito grande, a busca se torna muito complexa
- Para **resolver este problema** a busca em profundidade limitada pode ser **executada de forma iterativa**, variando o limite durante a execução do procedimento: comece com um limite de profundidade pequeno e aumente gradualmente o limite até que uma solução seja encontrada

- Esta busca é denominada busca em **profundidade iterativa**

# Busca Aprofundamento Interativo

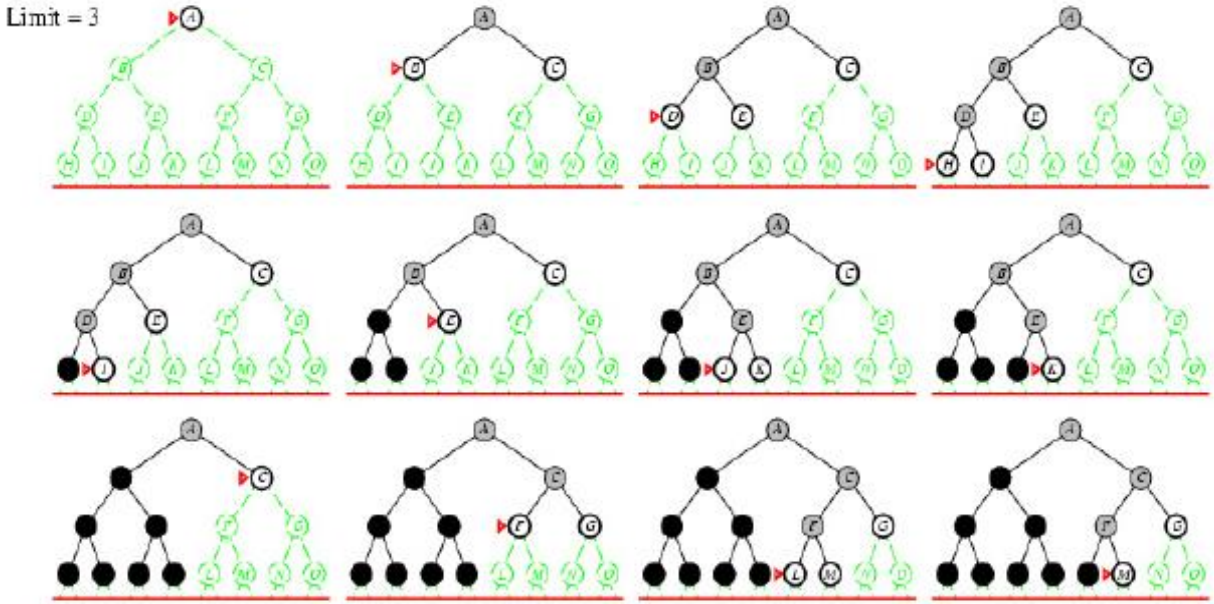
*l=0*


*l=1*


*l=2*

*l=3*



# Propriedades Busca Aprofundamento Interativo

- Número de nós gerados em uma busca em profundidade limitada até a profundidade $d$ com fator de ramificação $b$:
  - $N_{BPL} = b^0 + b^1 + b^2 + ... + b^{d-2} + b^{d-1} + b^d$
- Número de nós gerados em uma busca por aprofundamento iterativo até a profundidade $d$ com fator de ramificação $b$:
  - $N_{BAI} = (d+1)b^0 + db^1 + (d-1)b^2 + ... + 3b^{d-2} + 2b^{d-1} + 1b^d$
- Para $b = 10$, $d = 5$:
  - $N_{BPL} = 1 + 10 + 100 + 1.000 + 10.000 + 100.000 = 111.111$
  - $N_{BAI} = 6 + 50 + 400 + 3.000 + 20.000 + 100.000 = 123.456$
- *Overhead* = (123.456 - 111.111)/111.111 = 11%

- Completa? Sim.
- Tempo? $(d+1)b^0 + db^1 + (d-1)b^2 + ...b^d = \mathcal{O}(b^d)$.
- Espaço? $\mathcal{O}(b^d)$ (mantém todo nó na memória).
- Ótima? Sim (se custo = 1 por passo).

# Resumo dos Algoritmos

- b = número de caminhos alternativos/
  fator de bifurcação/ramificação (branching factor)
- d = profundidade da solução
- m = profundidade máxima da árvore de busca
- l = limite de profundidade

Table 1: Algoritmos de Busca Cega.

| *critério* | Largura | Profundidade | BPL | BAI |
|---|---|---|---|---|
| *Completa?* | Sim | Não | Não | Sim |
| *Tempo?* | $\mathcal{O}(b^{d+1})$ | $\mathcal{O}(b^m)$ | $\mathcal{O}(b^l)$ | $\mathcal{O}(b^d)$ |
| *Espaço?* | $\mathcal{O}(b^{d+1})$ | $\mathcal{O}(bm)$ | $\mathcal{O}(bl)$ | $\mathcal{O}(b^d)$ |
| *Ótimo?* | Sim | Não | Não | Sim |

- BPL = Busca em Profundidade Limitada
- BAI = Busca por Aprofundamento Iterativo

# Modele um problema como uma árvore de busca

- Use aplicativos para facilitar o entendimento dos algoritmos de busca

  http://www.aispace.org/search/search.jnlp

# Material usando implementações em PROLOG

# Busca em Espaço de Estados

- **Estratégias Básicas de Busca**

  (Busca não informada ou Cega)

  - As estratégias não-informadas usam apenas a informação disponível na definição do problema ou seja, **não utiliza** informações sobre o problema para guiar a busca
  - Estratégia de **busca exaustiva** aplicada até uma solução ser encontrada (ou falhar)
    - Busca em Profundidade (Depth-first)
    - Busca em Largura (Breadth-first)
    - Busca em profundidade limitada
    - Busca por aprofundamento iterativo

- **Estratégias Heurísticas de Busca**

  (Busca Informada)

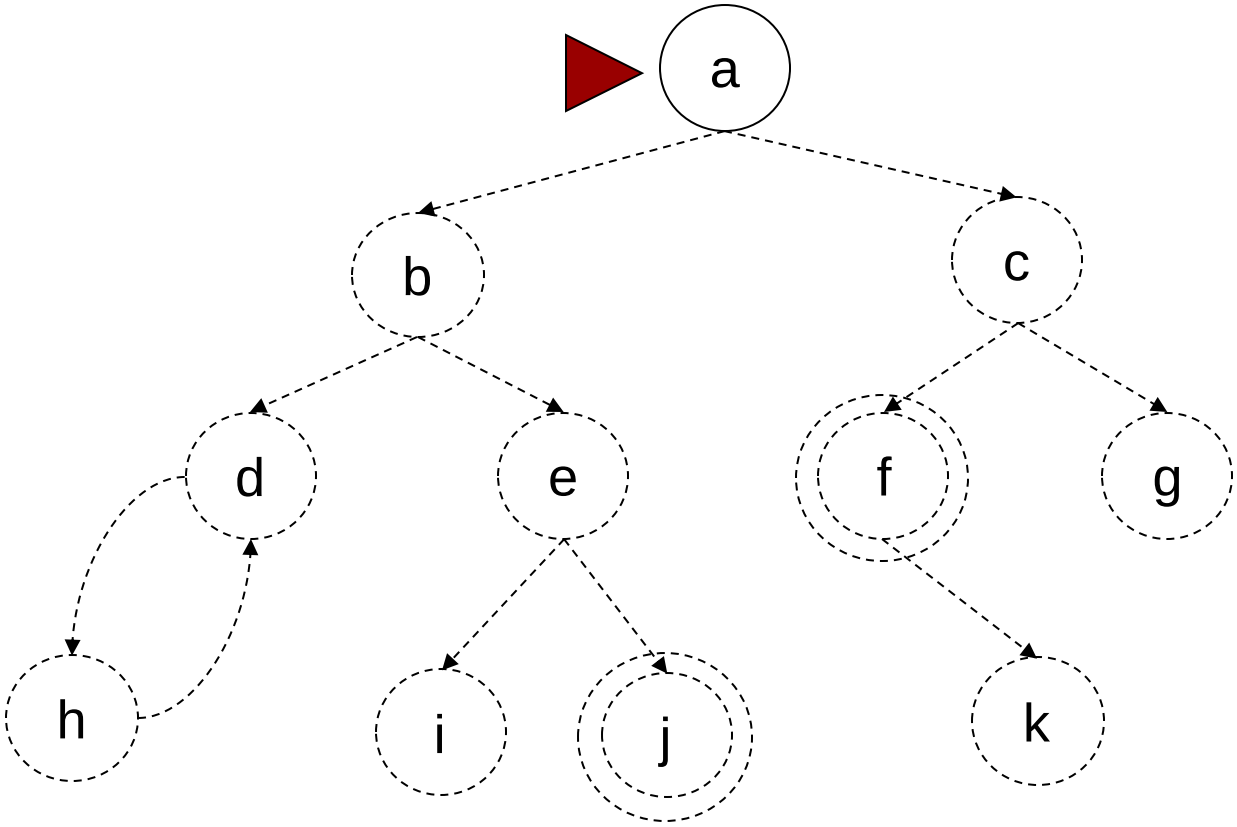
  - **utiliza** informações específicas do domínio para ajudar na decisão
    - Hill-Climbing
    - Best-First
    - A*

# Busca em Espaço de Estados

- Um espaço de estados pode ser representado pela relações
  - **pode_ir(X,Y)** que é verdadeira se há um movimento permitido no espaço de estados do nó X para o nó Y; neste caso, Y é um **sucessor** de X
  - **final(X)** que é verdadeira se X é um estado final
- Se houver custos envolvidos, um terceiro argumento pode ser adicionado, o custo do movimento
  - **pode_ir(X,Y,Custo)**

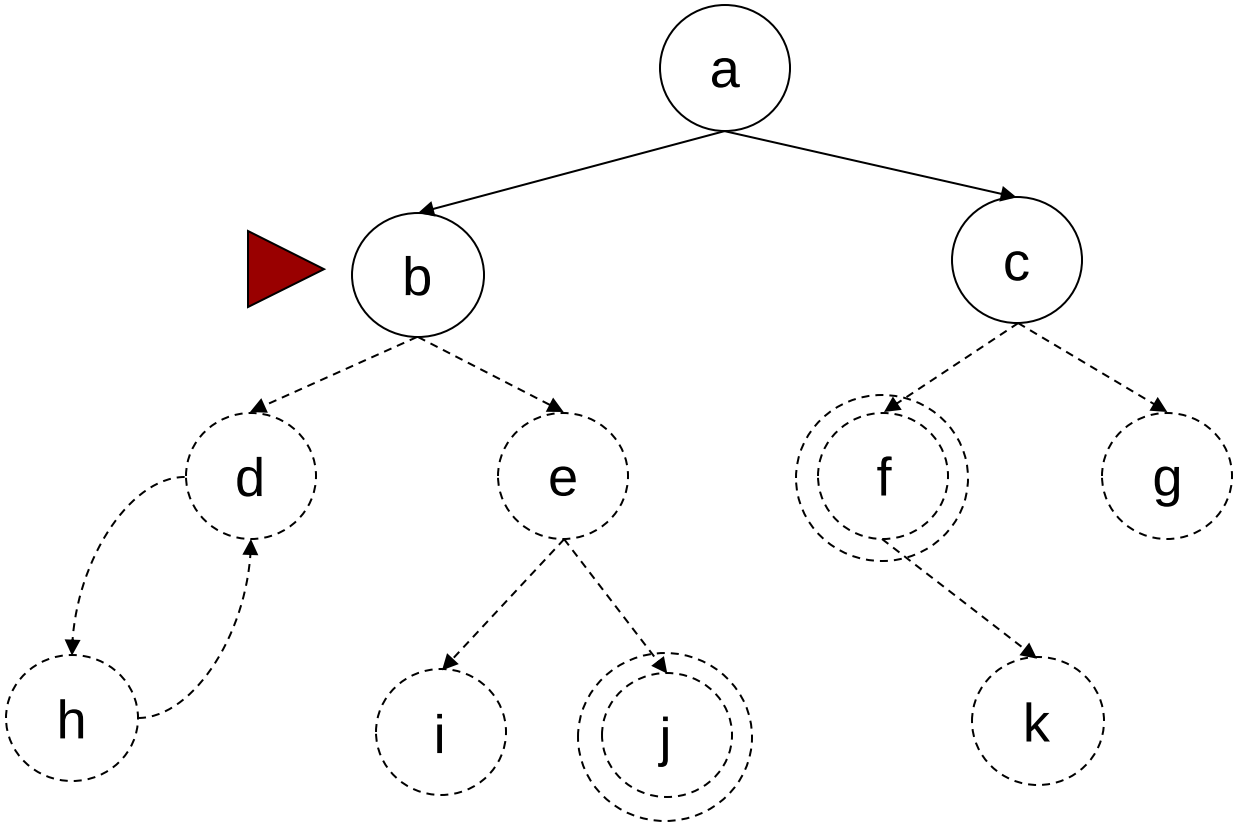
- A relação **pode_ir** pode ser representada explicitamente por um conjunto de **fatos**
- Entretanto, para espaços de estado complexos a relação **pode_ir** é usualmente definida implicitamente por meio de **regras** que permitam calcular o sucessor de um dado nó

# Busca em Profundidade



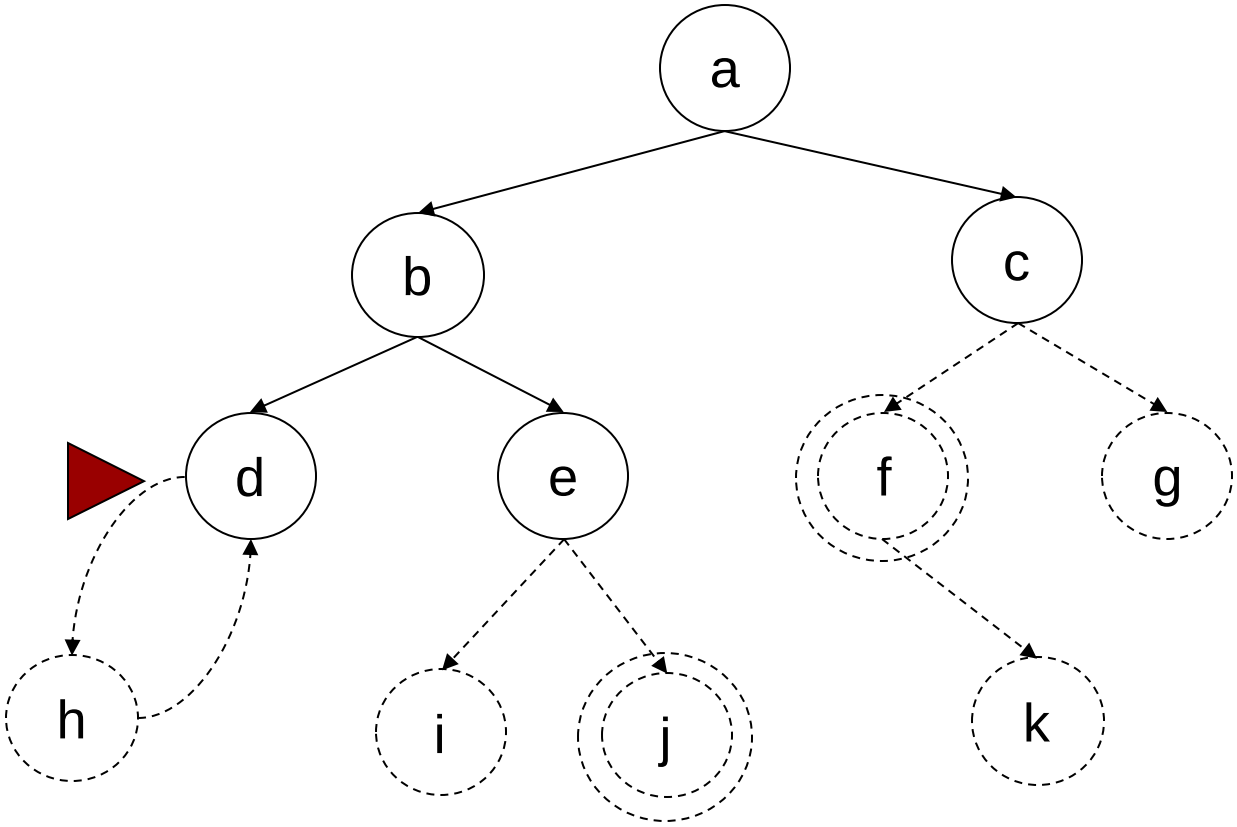

Inserir na frente, remover da frente: a

# Busca em Profundidade



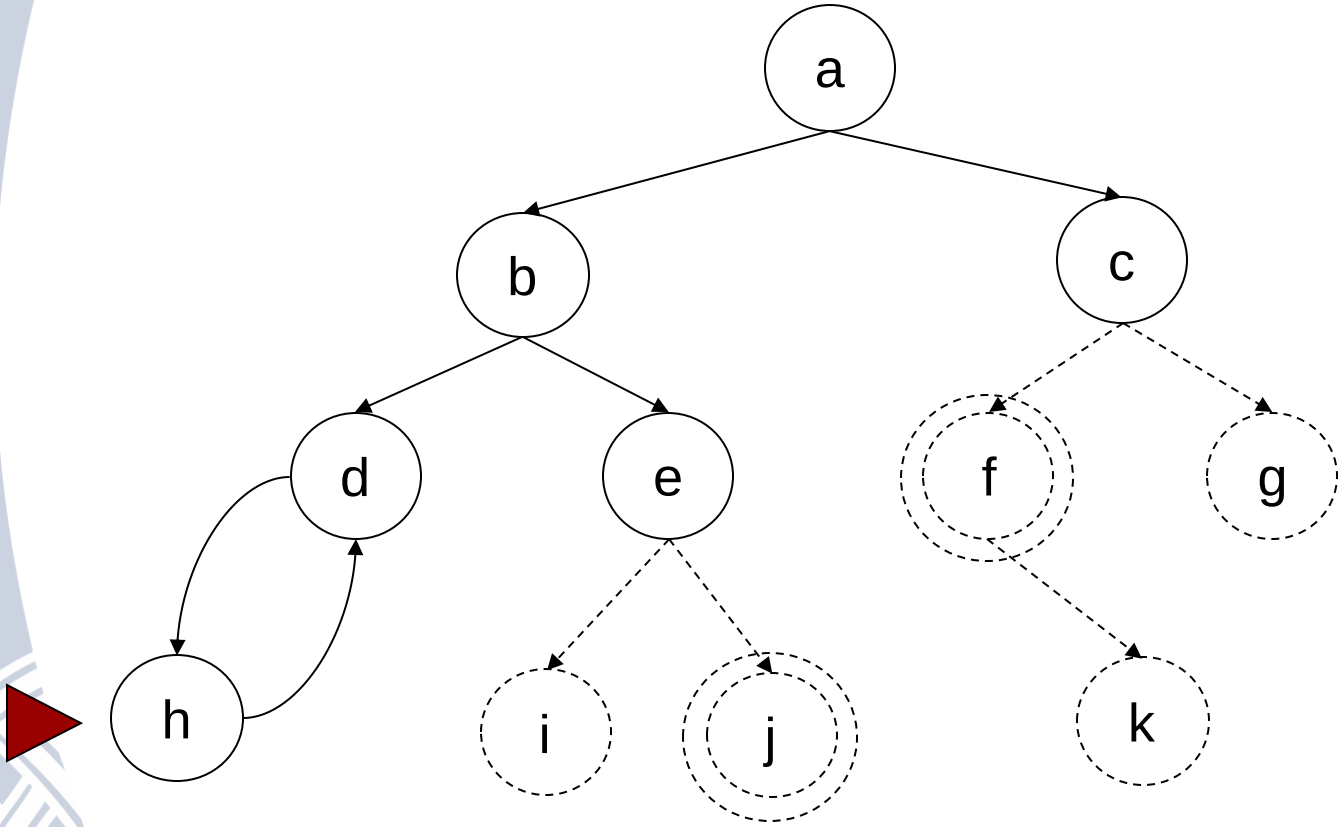

Inserir na frente, remover da frente: b, c

# Busca em Profundidade



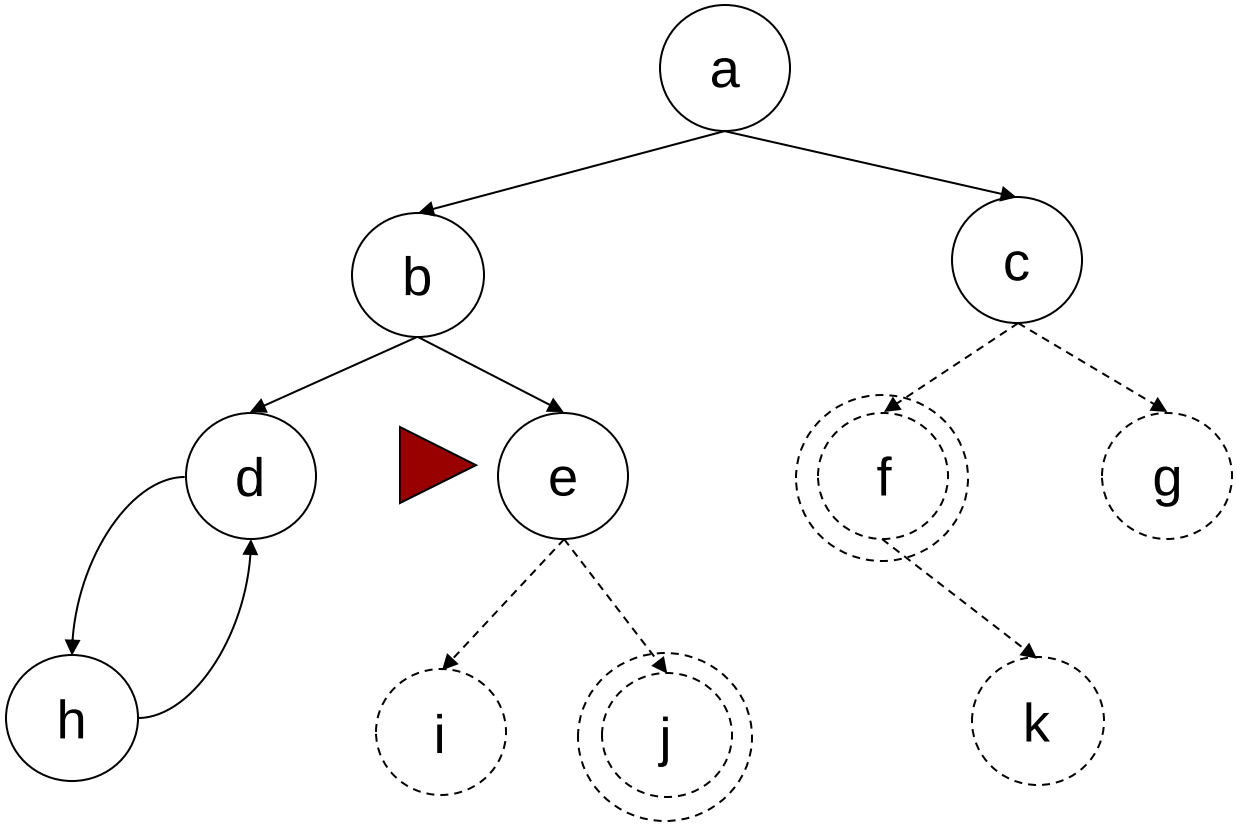

Inserir na frente, remover da frente: d, e, c

# Busca em Profundidade



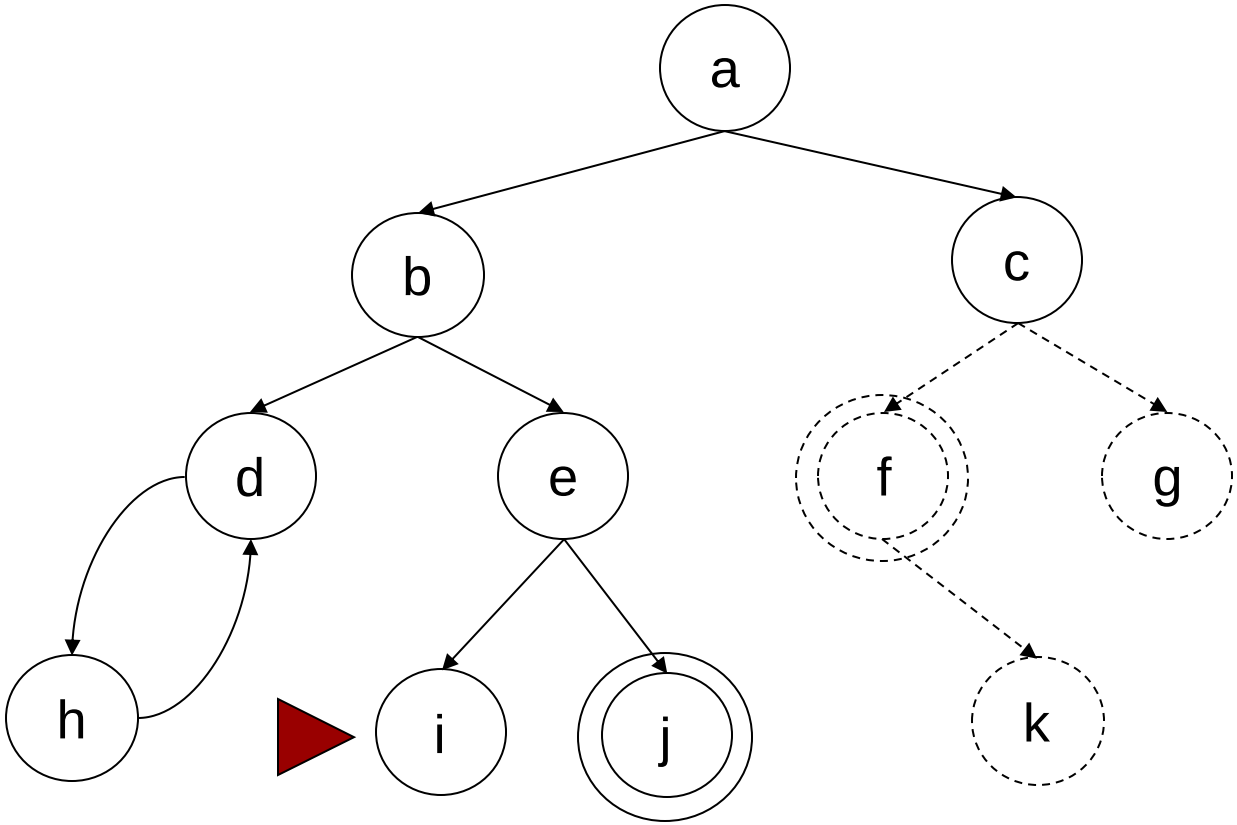

Inserir na frente, remover da frente: h, e, c

# Busca em Profundidade



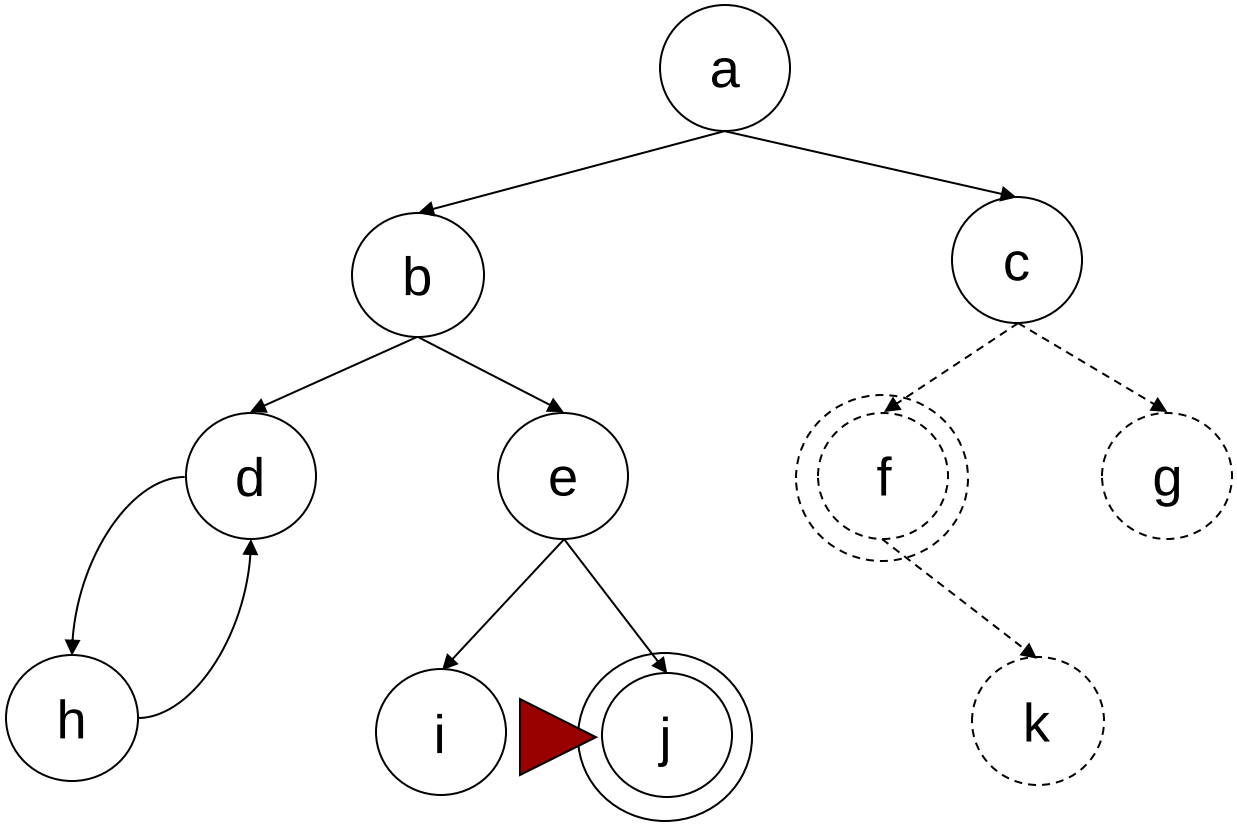

Inserir na frente, remover da frente: e, c

# Busca em Profundidade



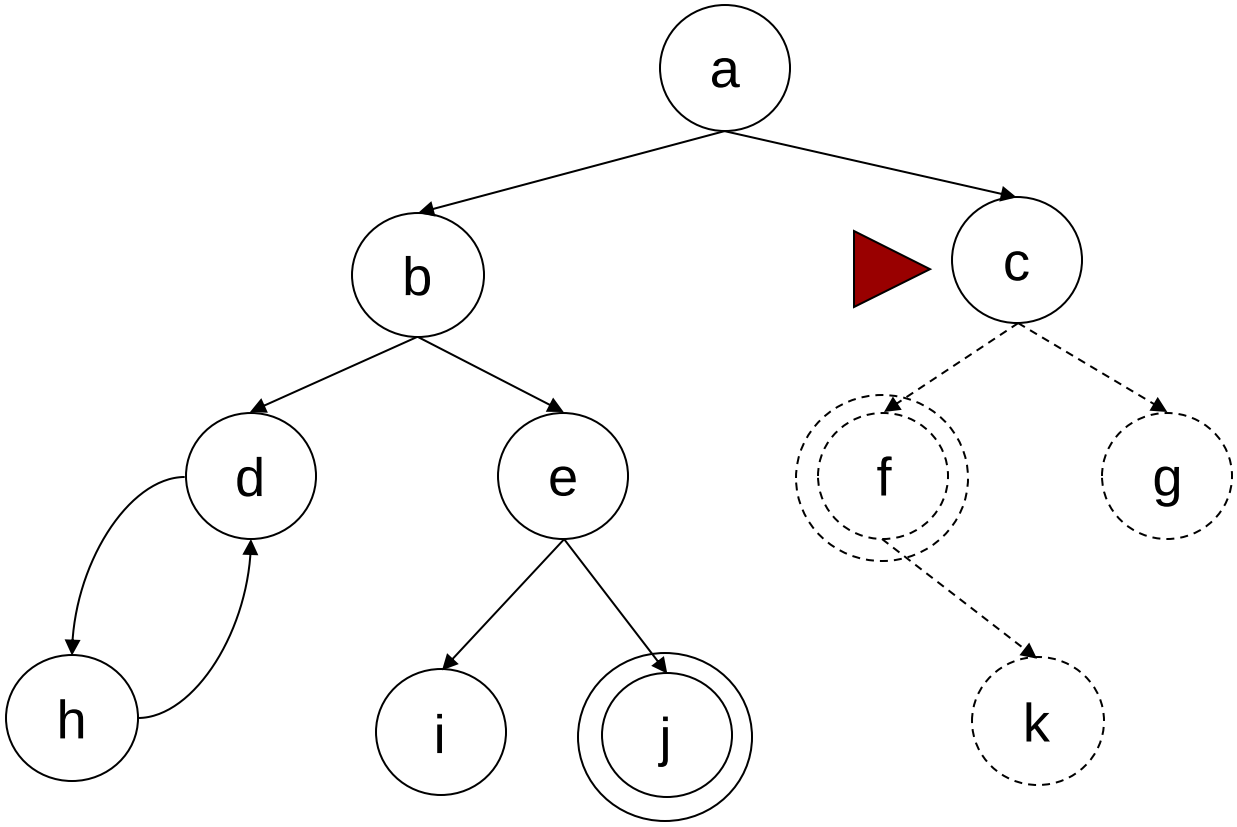

Inserir na frente, remover da frente: i, j, c

# Busca em Profundidade



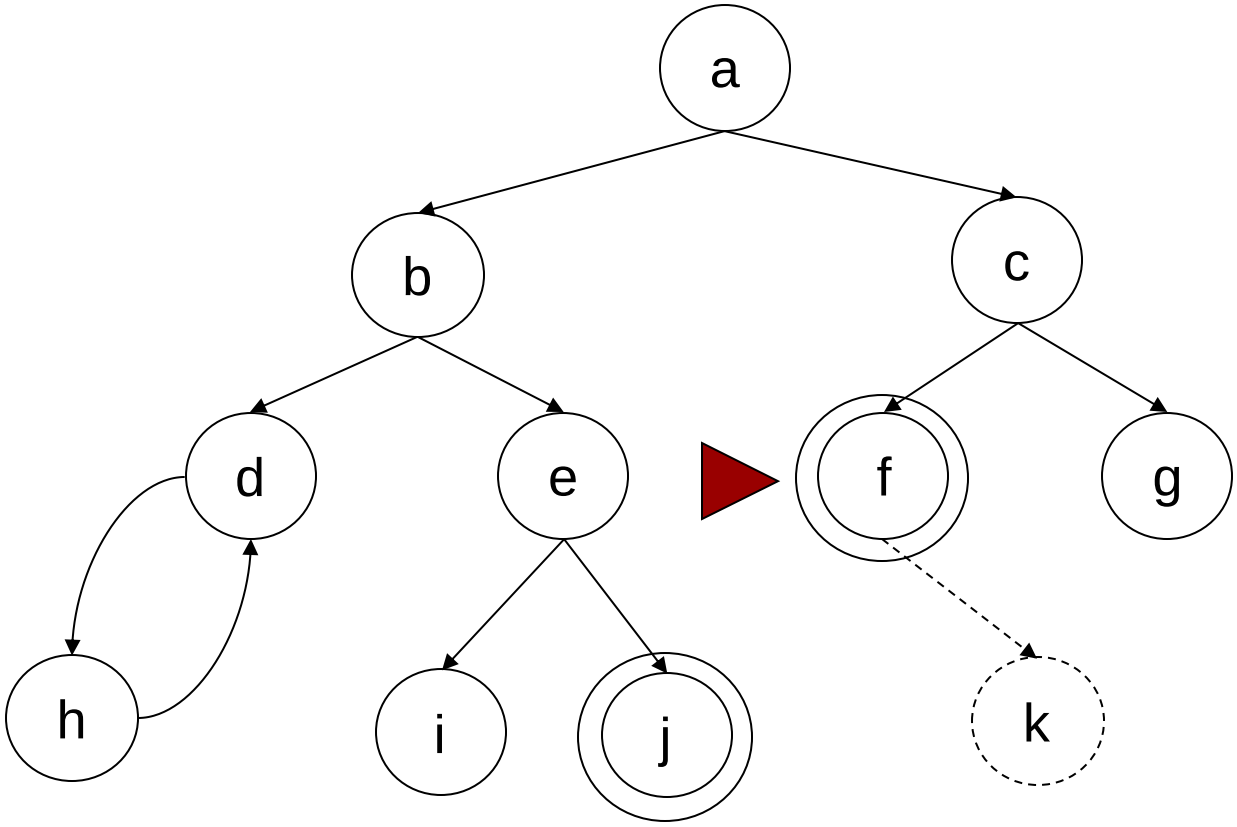

Inserir na frente, remover da frente: j, c

# Busca em Profundidade



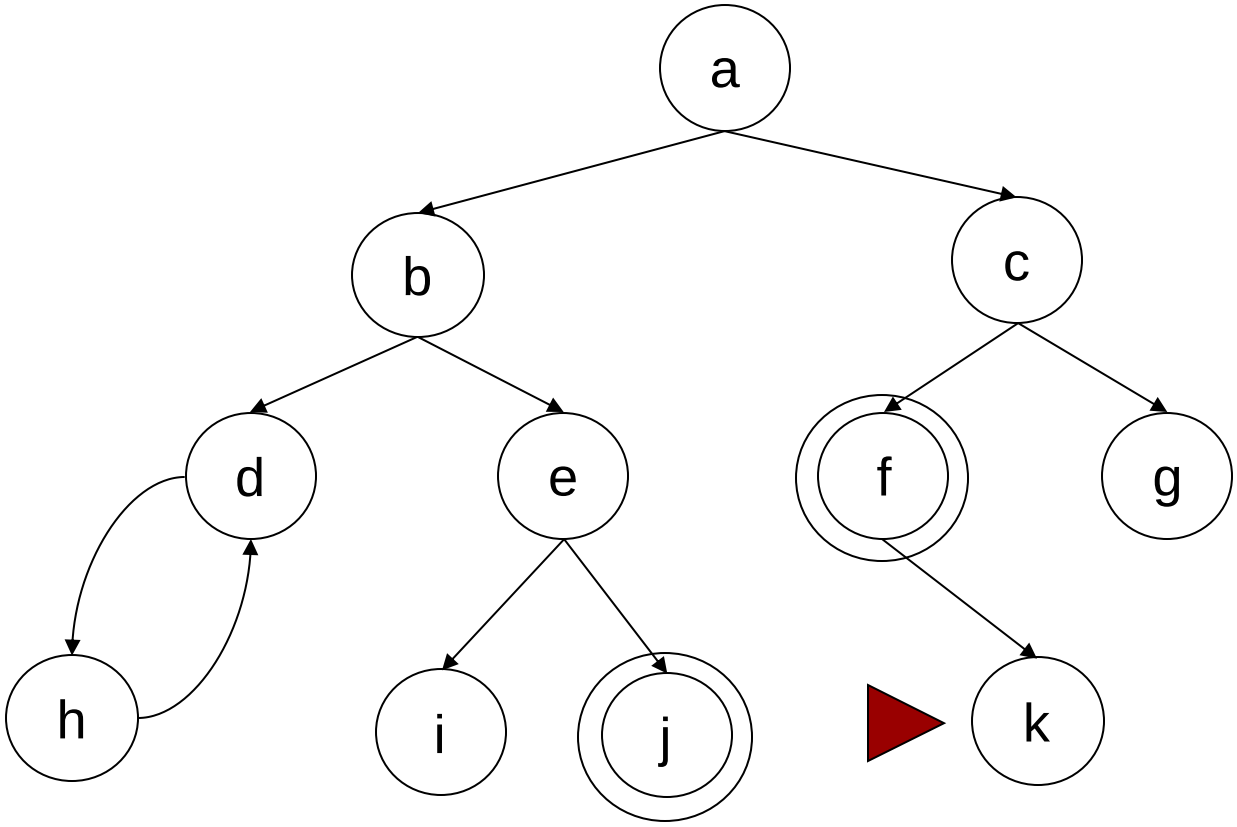

Inserir na frente, remover da frente: c

# Busca em Profundidade



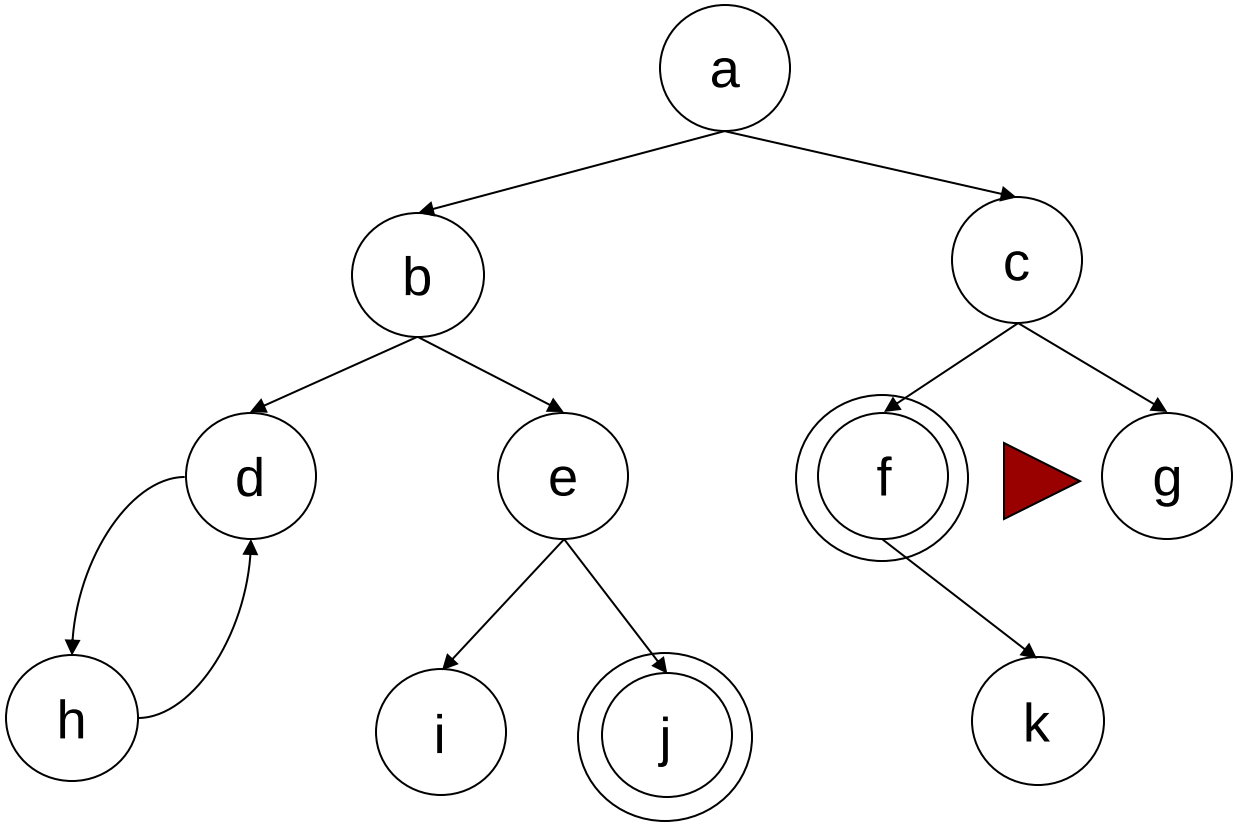

Inserir na frente, remover da frente: f, g

# Busca em Profundidade




Inserir na frente, remover da frente: k, g

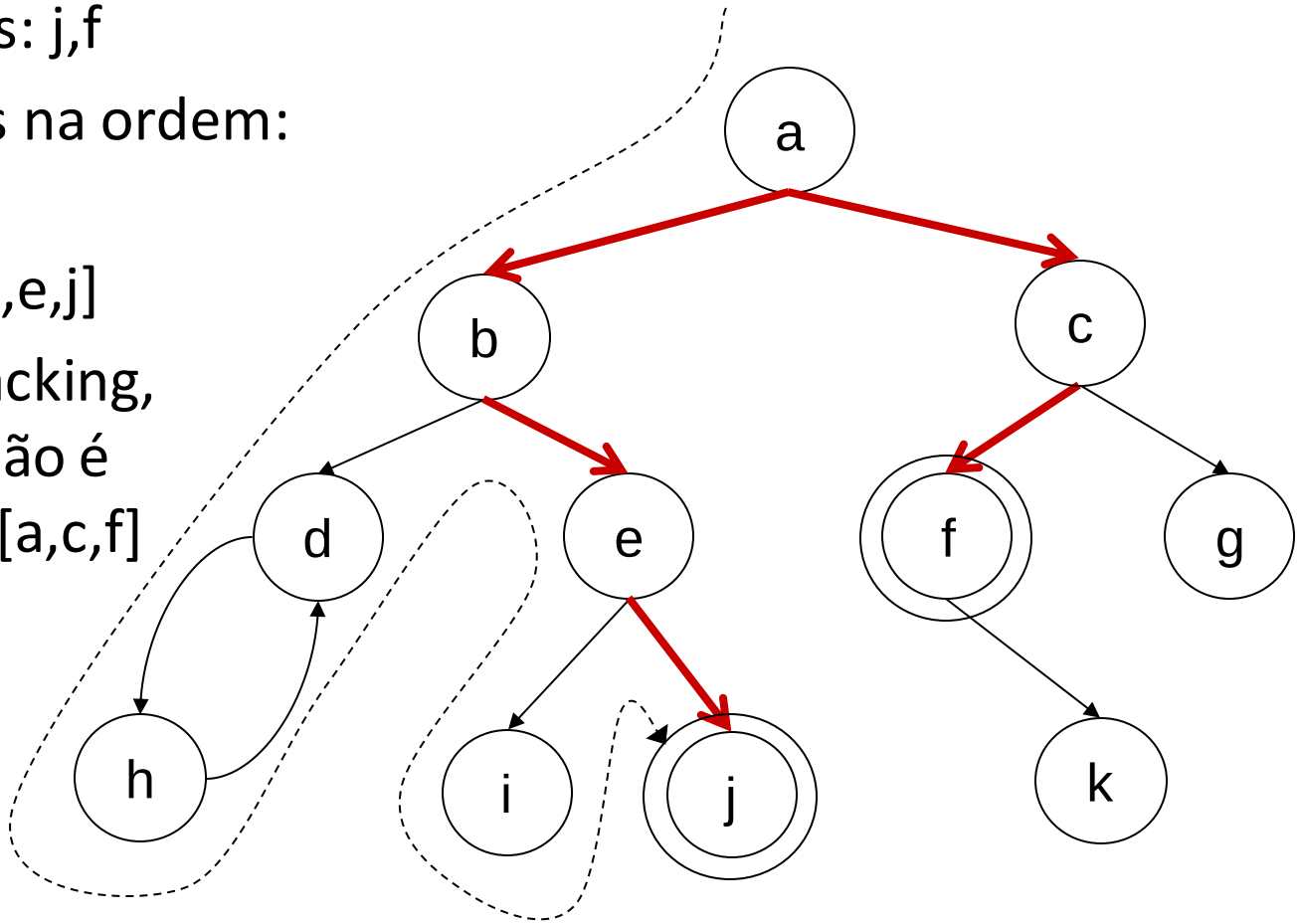
# Busca em Profundidade




Inserir na frente, remover da frente: g

# Busca em Profundidade

- Estado inicial: a
- Estados finais: j,f
- Nós visitados na ordem: a,b,d,h,e,i,j
- Solução: [a,b,e,j]
- Após backtracking, a outra solução é encontrada: [a,c,f]



# Busca em Profundidade

- **Algoritmo para busca em profundidade:** para encontrar uma **solução** (Caminho da solução) **de um estado inicial I para um estado final F (busca forward)**
  - Se **F foi alcançado** (o estado atual é F) o Caminho é uma solução
  - Se o **estado atual tem um sucessor**, e esse sucessor não faz parte do caminho até agora (detecção de loops) **insira no caminho até agora** e continue

- A **busca em profundidade** é o recurso mais simples em **programação recursiva** e, por isso, Prolog quando executa metas explora alternativas utilizando-a

# Busca em Profundidade *Backward*

```
caminho_cegoBPBackward(I,F,Cam):-
 caminho_b(I,[F],Cam).

caminho_b(I,[I|Caminho],[I|Caminho]).

caminho_b(I,[Ult_Estado|Caminho_ate_agora],Cam):-
 pode_ir(Est,Ult_Estado),
 not( pertence1(Est,Caminho_ate_agora)),
 caminho_b(I,[Est,Ult_Estado|Caminho_ate_agora],Cam).
```

```
pertence1(E,[E|_]):- !.
pertence1(E,[_|T]):-
   pertence1(E,T).
```

# Busca em Profundidade *Forward*

```
caminho_cegoBPForward(EI,EF,Cam):-
  caminho_f(EF,[EI],Cam).

  caminho_f(EF,[EF|Cam],[EF|Cam]).

  caminho_f(EF,[Eint|Caminho_ate_agora],Cam):-
    pode_ir(Eint,E),
    not(pertence1(E,Caminho_ate_agora)),
    caminho_f(EF,[E,Eint|Caminho_ate_agora], Cam).
```

# Algoritmo Busca em Profundidade

- **Um algoritmo de Busca em Profundidade pode ser:**

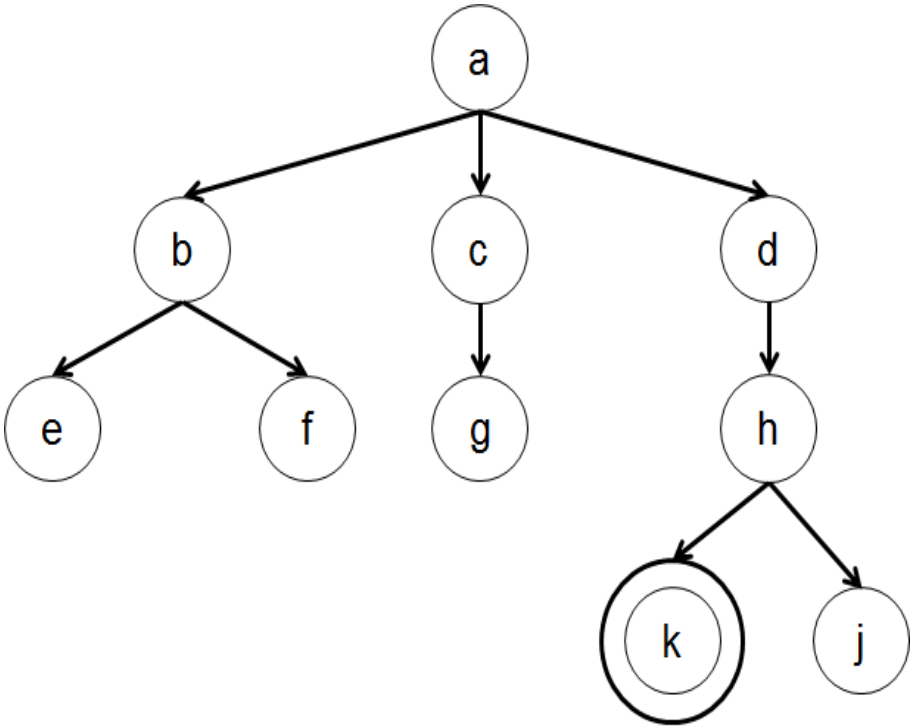
1) Empilhe um nó **v** origem na pilha **P** e marque-o como alcançado
2) Enquanto a pilha P não vazia faça
   **v** ← elemento do topo da pilha
   //(desempilhe)
   **se existe w** a partir de **v**, // (v,w)
   e **w** ainda não foi alcançado
   - marque **w** como alcançado
   - insira **w** na pilha **P** // empilhe(w)

# Considerações: Busca em Profundidade

- Um problema com a busca em profundidade é que pode existir espaços de estado nos quais o algoritmo se perde
- Muitos espaços de estado são infinitos e, nesse caso, o algoritmo de busca em profundidade pode perder um nó final, prosseguindo por um caminho infinito no grafo
- O algoritmo então explora esta parte infinita do espaço, nunca chegando perto de um nó final.
- Para ***n*** nós e ***a*** arestas, a complexidade de tempo é **O(*a*+*n*)**
- Para não prosseguir por um caminho infinito no grafo e perder uma eventual solução, uma proposta é limitar a busca.

# Exercício - Busca em Profundidade

- Considere a busca no espaço abaixo, em que **"a"** é o estado inicial e **"k"** é o estado final.




- **1.** Modele o espaço de busca com os fatos **pode_ir(X,Y)**.

# Exercício – Busca em Profundidade (continuação)

- **2.** Execute o **algoritmo em prolog de busca em profundidade** para encontrar a solução do problema. Mostre o passo a passo da execução, os nós visitados e o caminho encontrado.

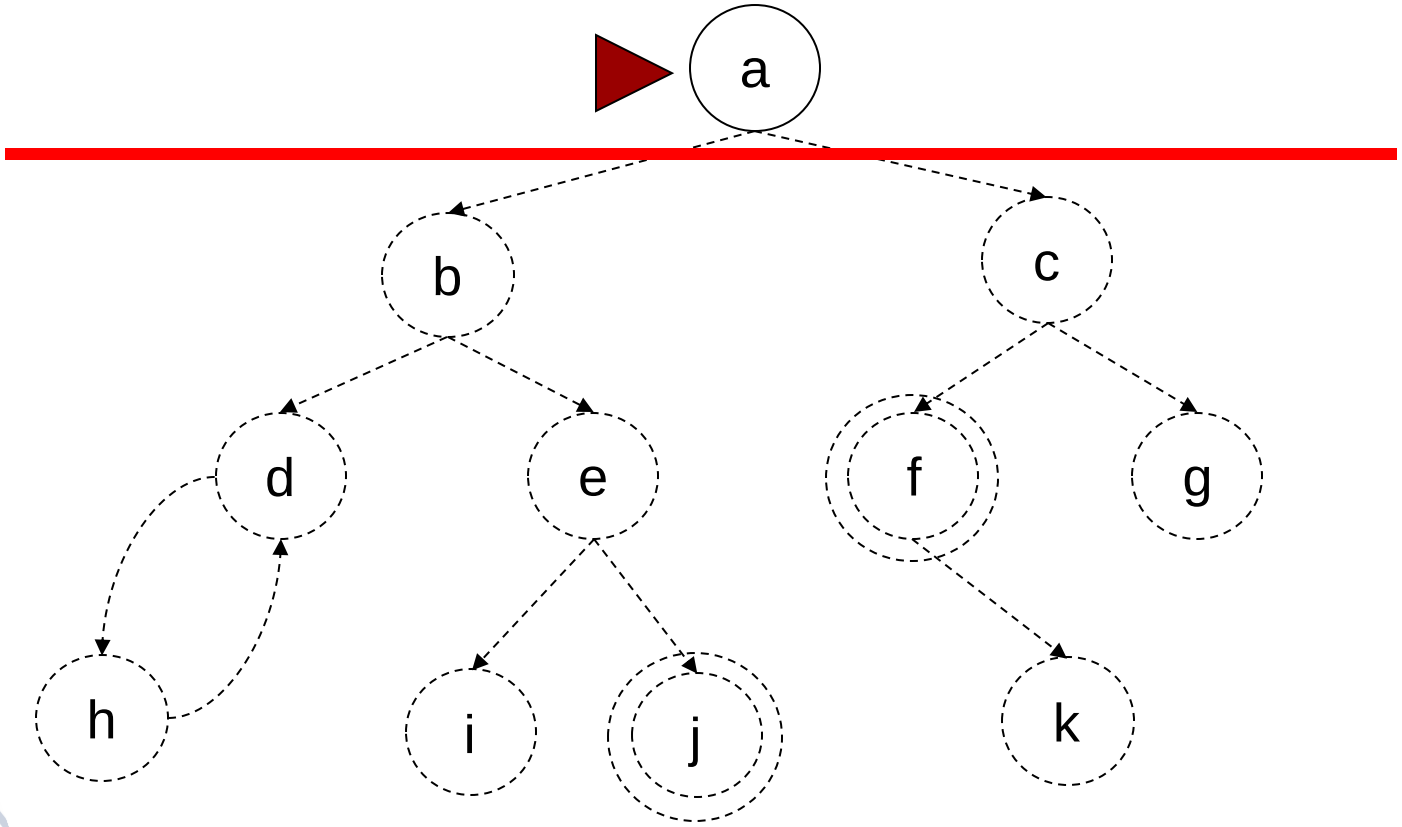
```
caminho_cegoBPBackward(I,F,Cam):-
 caminho_b(I,[F],Cam).

caminho_b(I,[I|Caminho],[I|Caminho]).

caminho_b(I,[Ult_Estado|Caminho_ate_agora],Cam):-
 pode_ir(Est,Ult_Estado),
 not( pertence1(Est,Caminho_ate_agora)),
 caminho_b(I,[Est,Ult_Estado|Caminho_ate_agora],Cam).
```

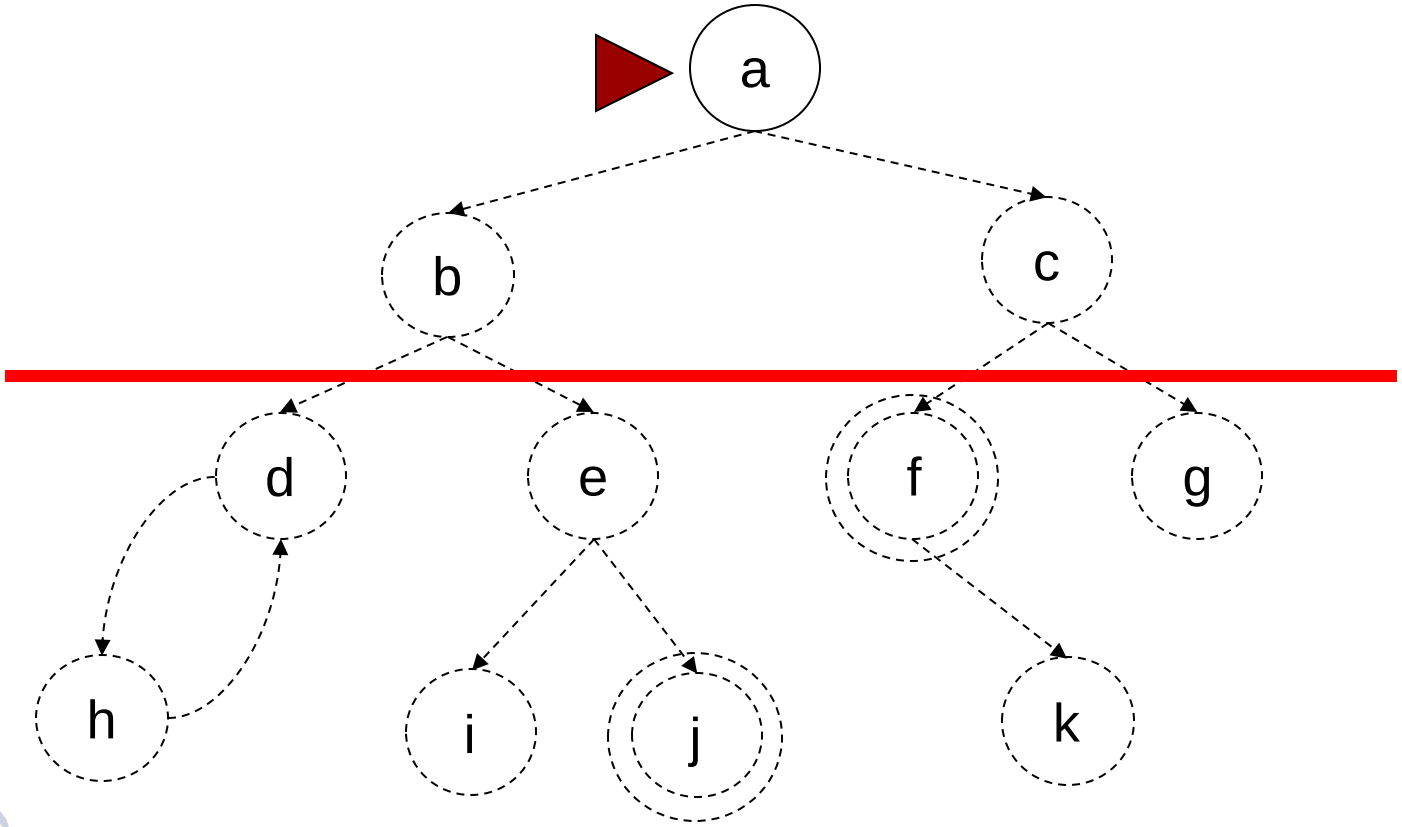
```
pertence1(E,[E|_]):- !.
pertence1(E,[_|T]):-
   pertence1(E,T).
```

# Busca em Profundidade Limitada (L=0)



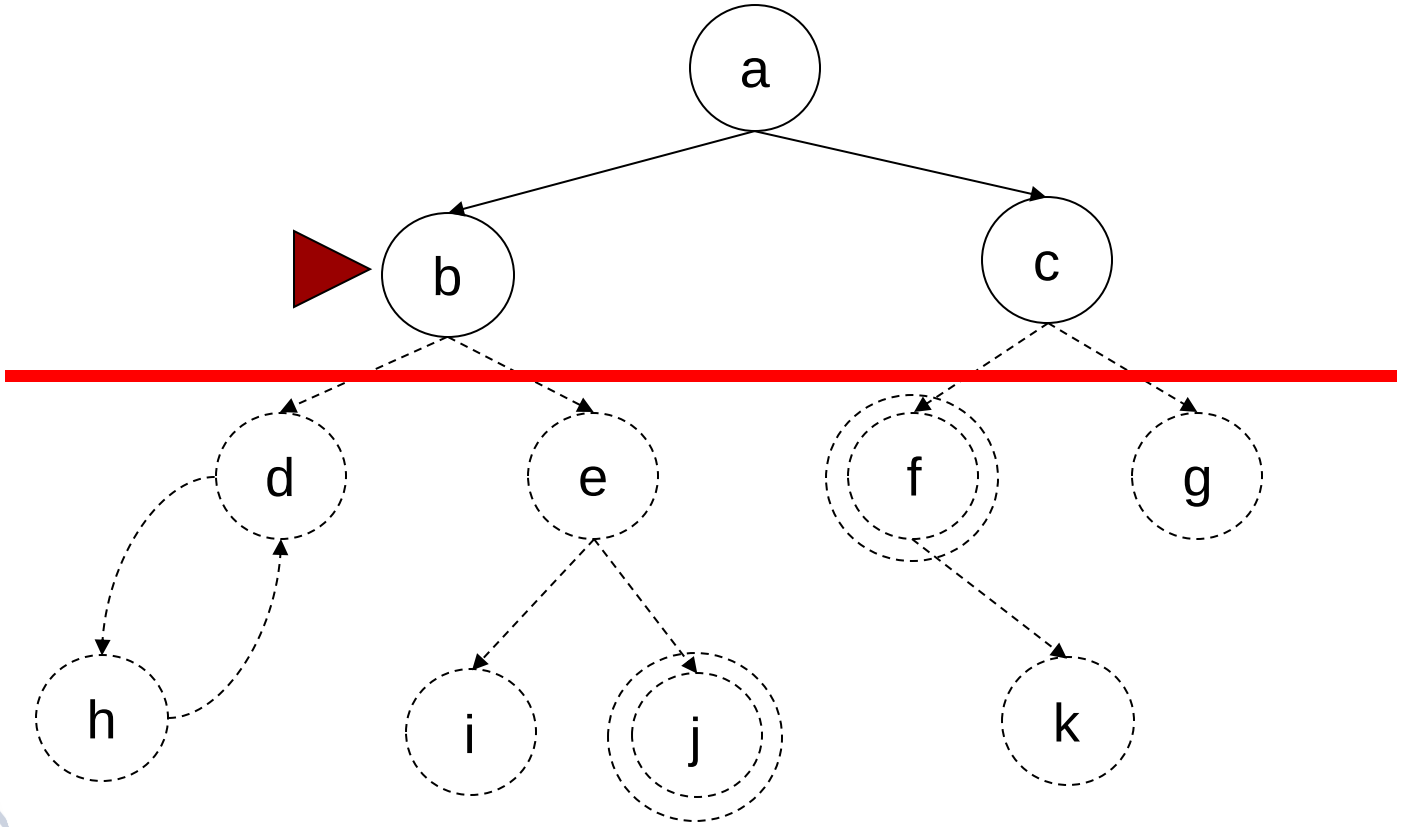

Inserir na frente, remover da frente: a

# Busca em Profundidade Limitada (L=1)



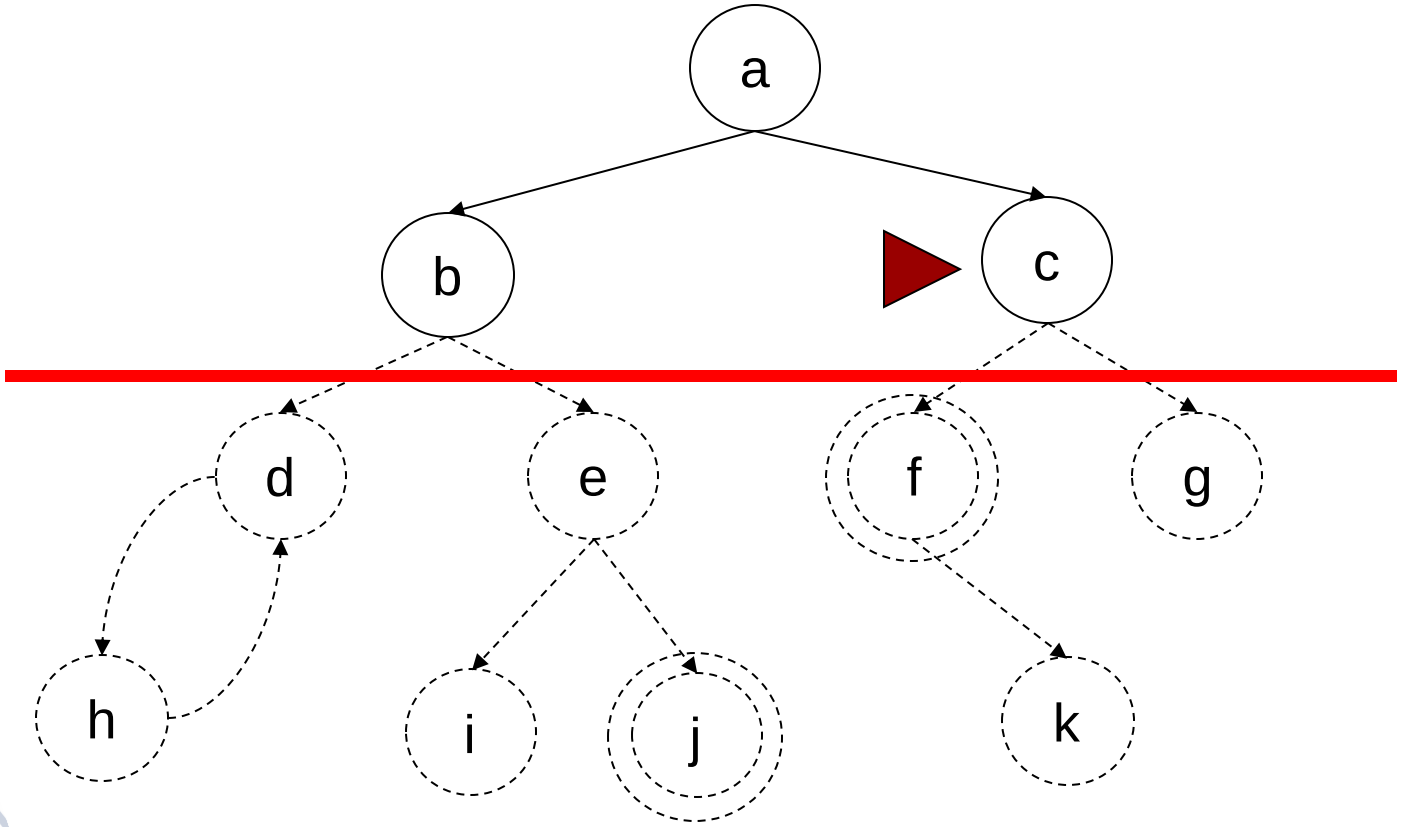

Inserir na frente, remover da frente: a

# Busca em Profundidade Limitada (L=1)



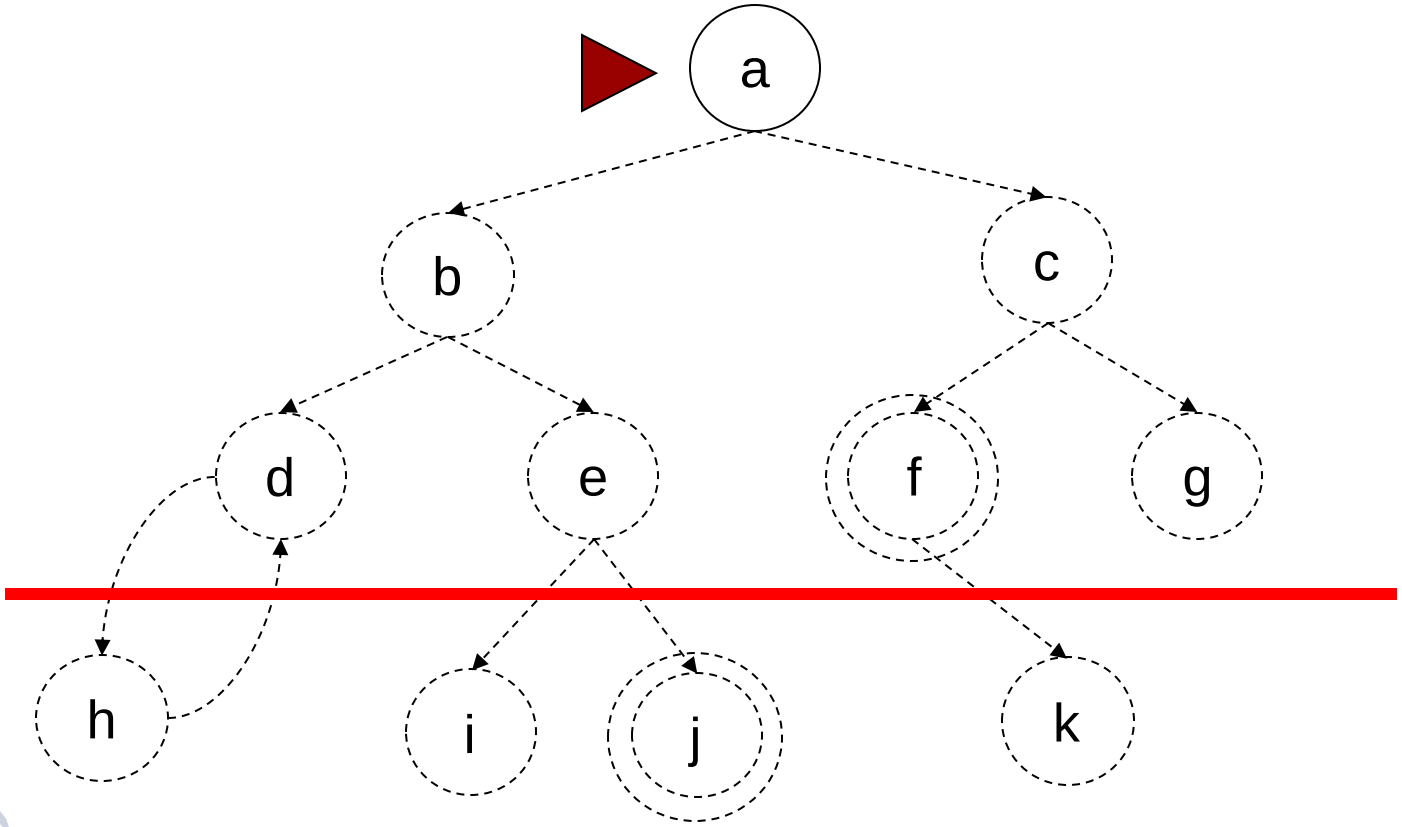

Inserir na frente, remover da frente: b, c

# Busca em Profundidade Limitada (L=1)



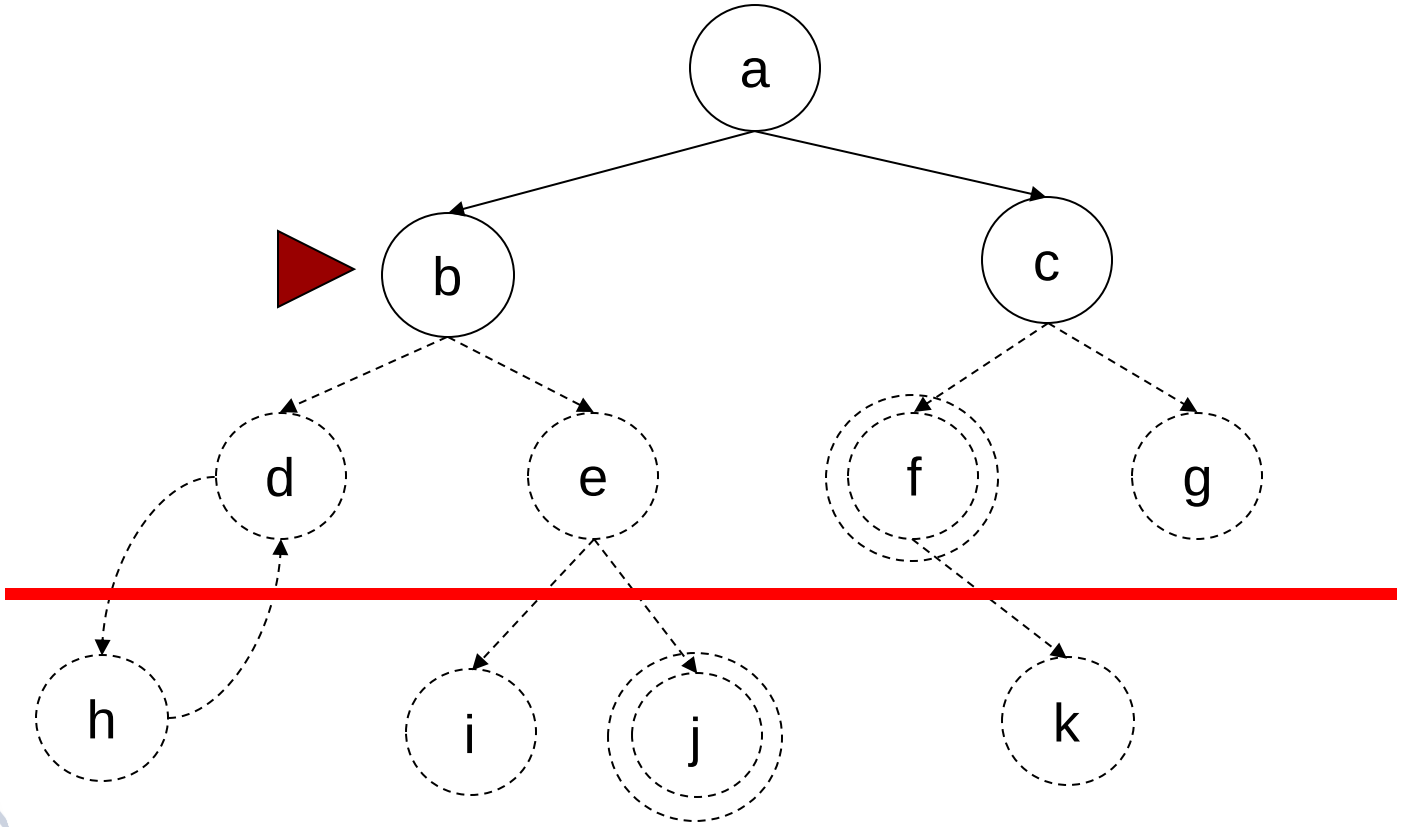

Inserir na frente, remover da frente: c

# Busca em Profundidade Limitada (L=2)



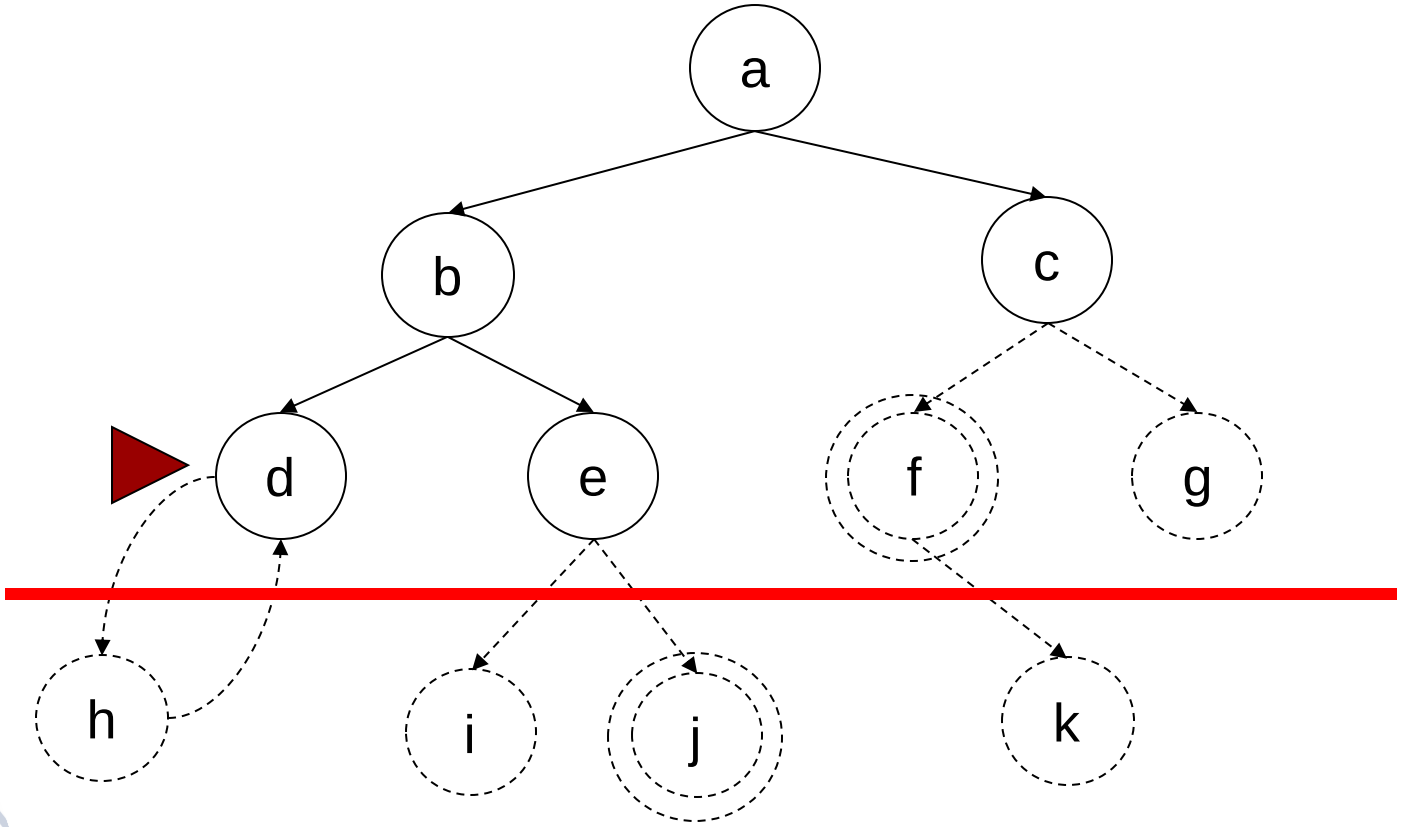

Inserir na frente, remover da frente: a

# Busca em Profundidade Limitada (L=2)



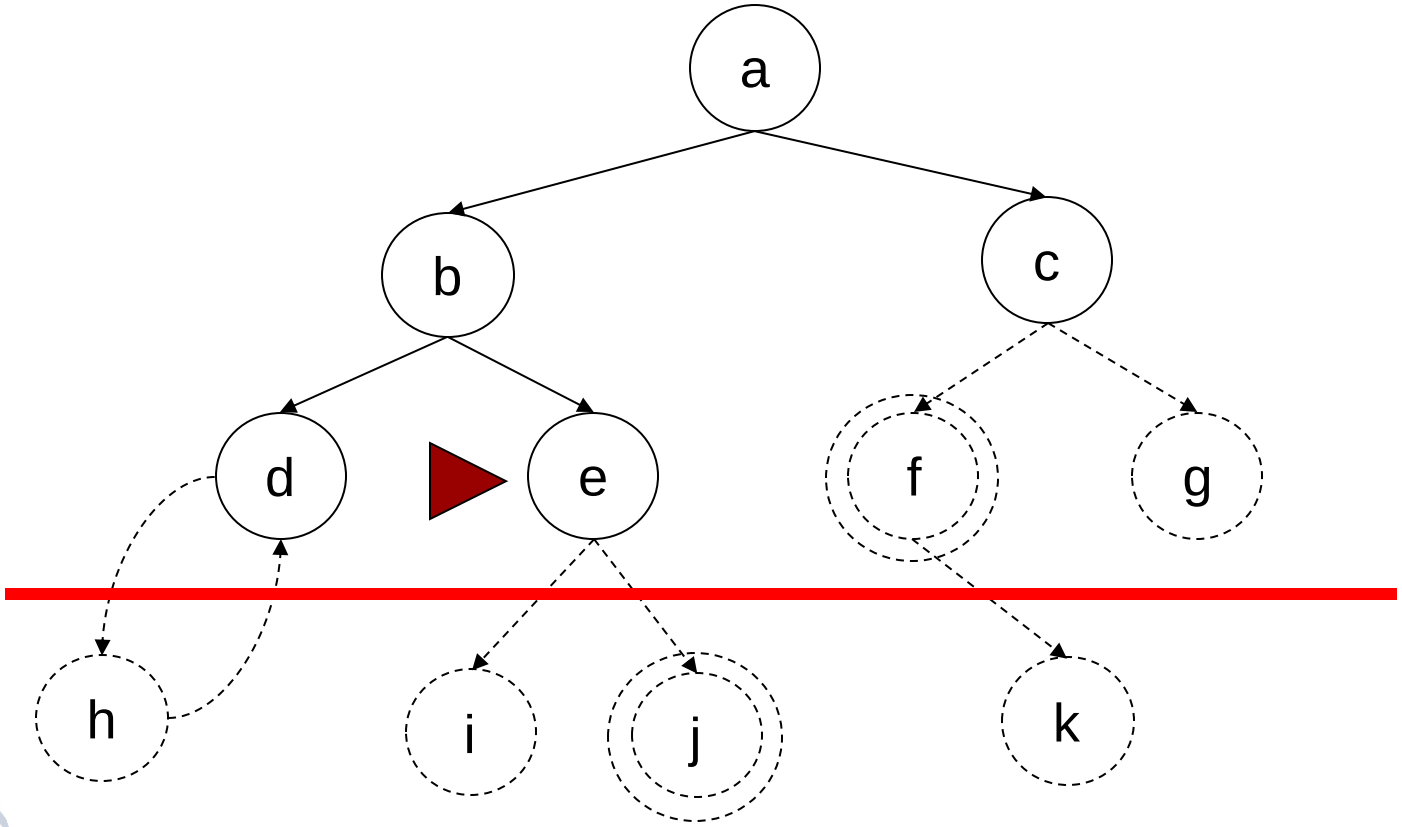

Inserir na frente, remover da frente: b, c

# Busca em Profundidade Limitada (L=2)



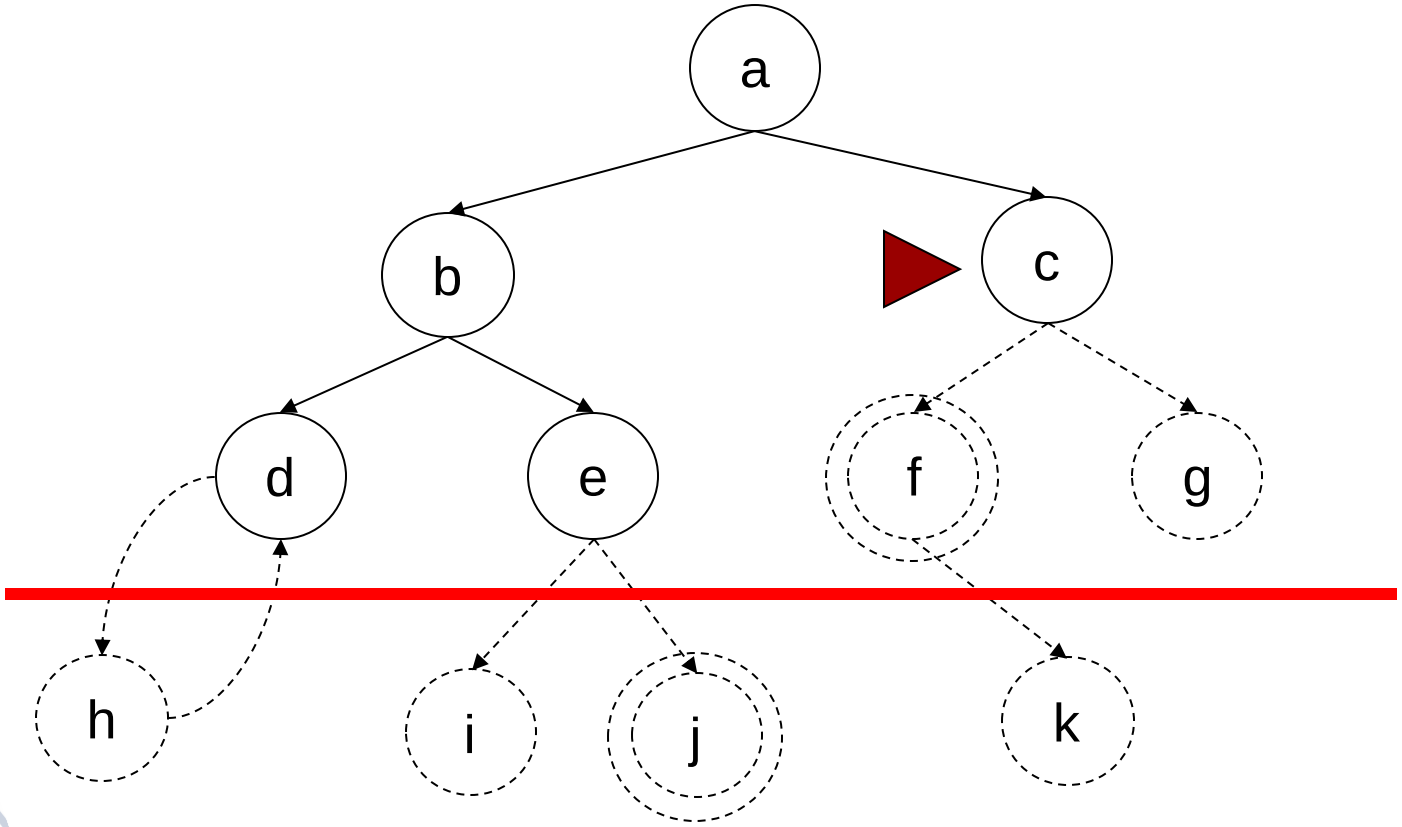

Inserir na frente, remover da frente: d, e, c

# Busca em Profundidade Limitada (L=2)



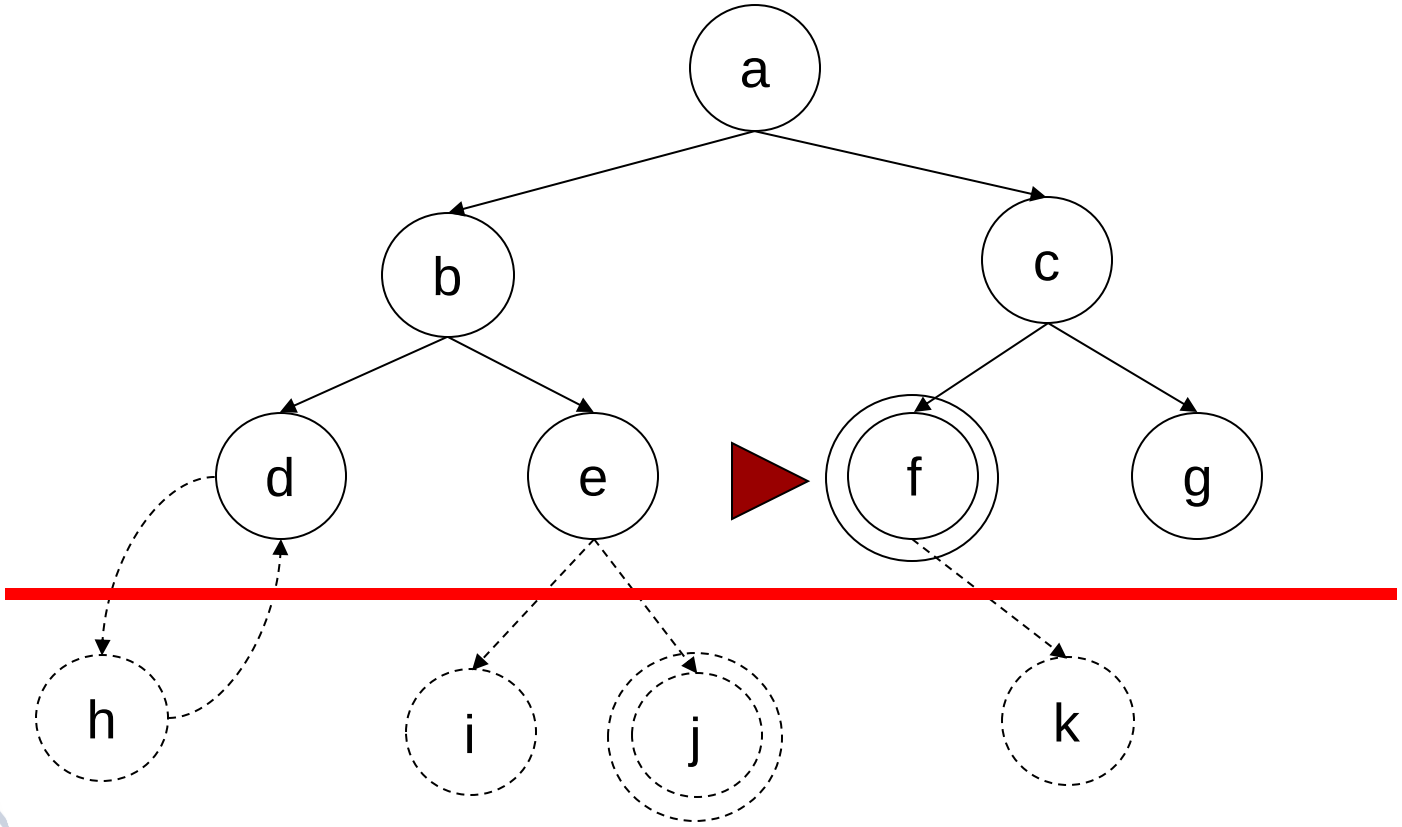

Inserir na frente, remover da frente: e, c

# Busca em Profundidade Limitada (L=2)



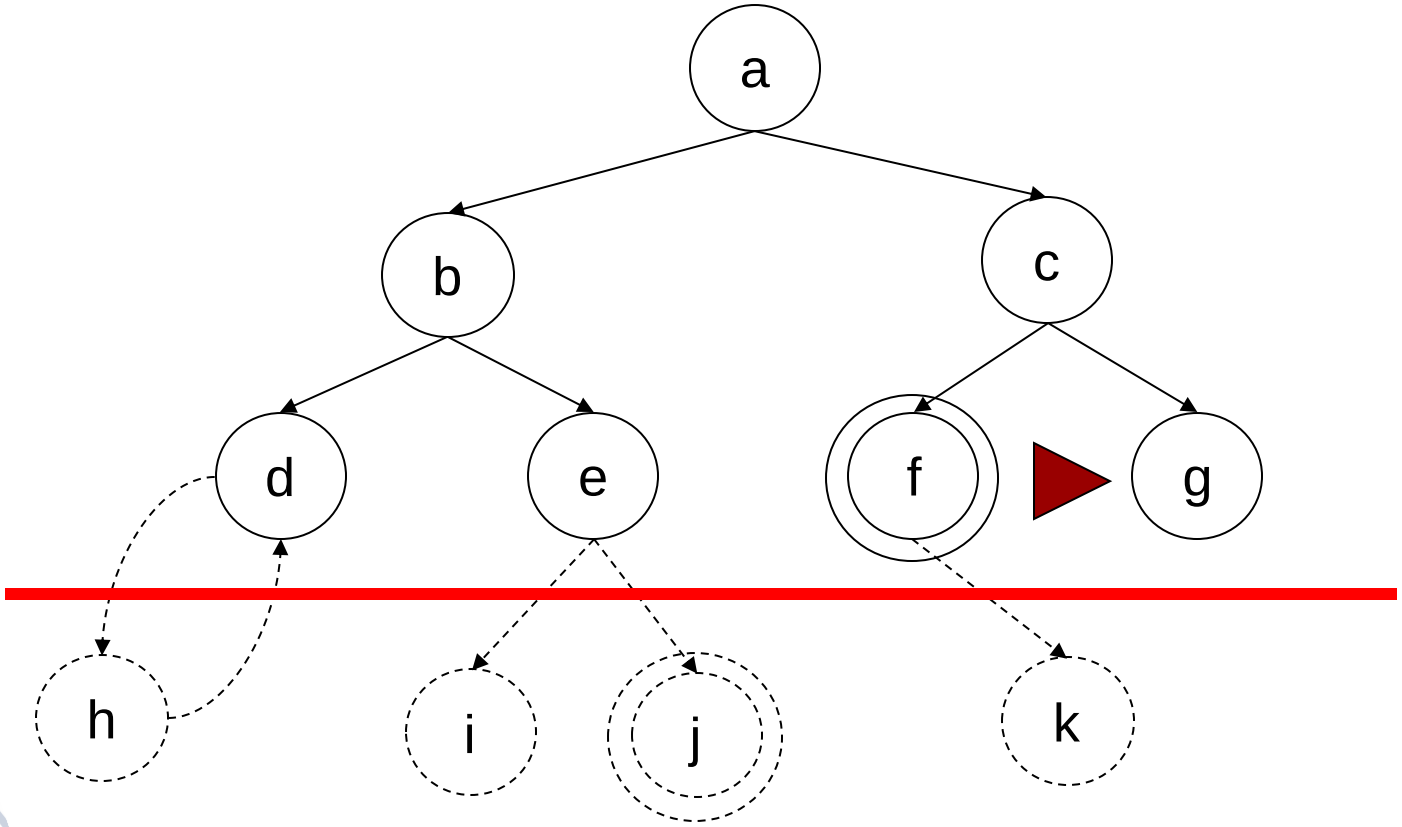

Inserir na frente, remover da frente: c

# Busca em Profundidade Limitada (L=2)




Inserir na frente, remover da frente: f, g

# Busca em Profundidade Limitada (L=2)




Inserir na frente, remover da frente: g

# Busca em Profundidade Limitada

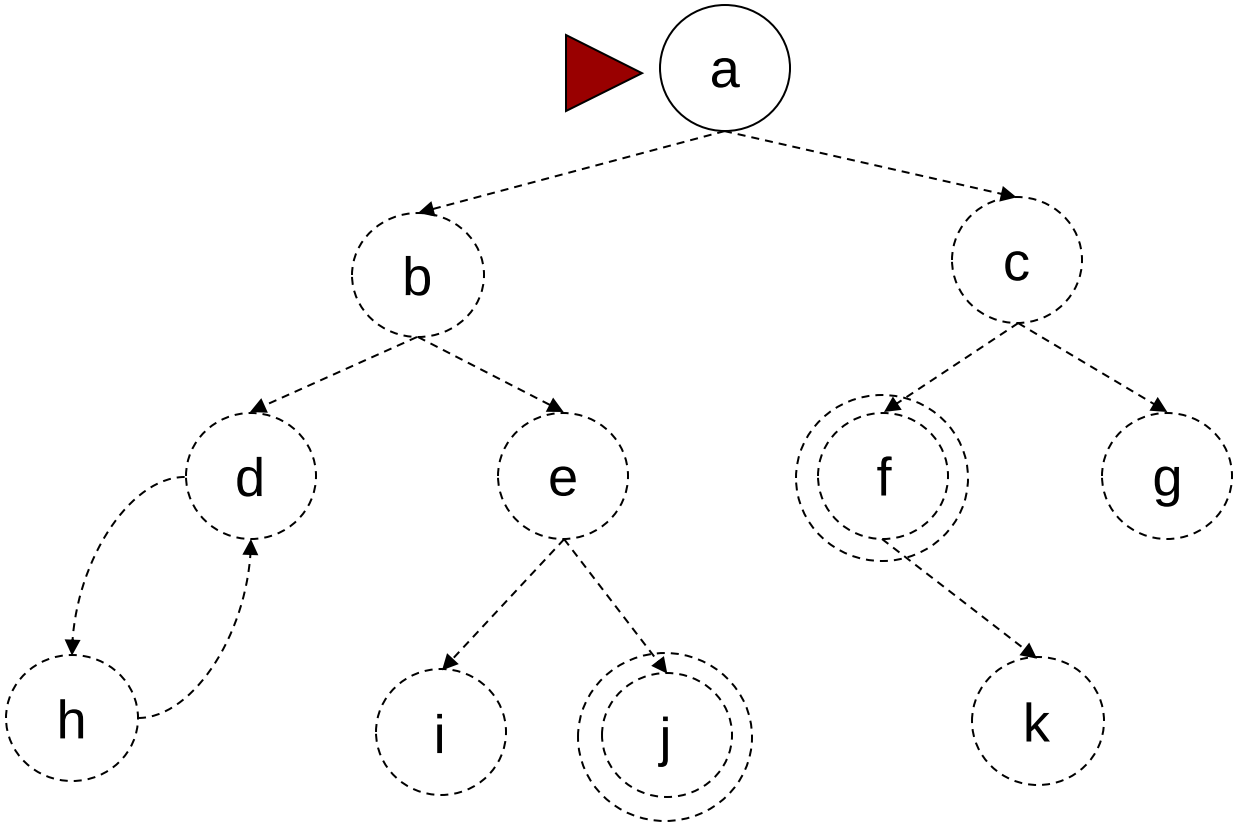
- **Um problema** com a busca em profundidade limitada é que **não se tem previamente um limite razoável**
  - Se o limite for muito pequeno (menor que qualquer caminho até uma solução) então a busca falha
  - Se o limite for muito grande, a busca se torna muito complexa
- Para **resolver este problema** a busca em profundidade limitada pode ser **executada de forma iterativa**, variando o limite durante a execução do procedimento: comece com um limite de profundidade pequeno e aumente gradualmente o limite até que uma solução seja encontrada

- Esta busca é denominada busca em **profundidade iterativa**

# Busca em Largura

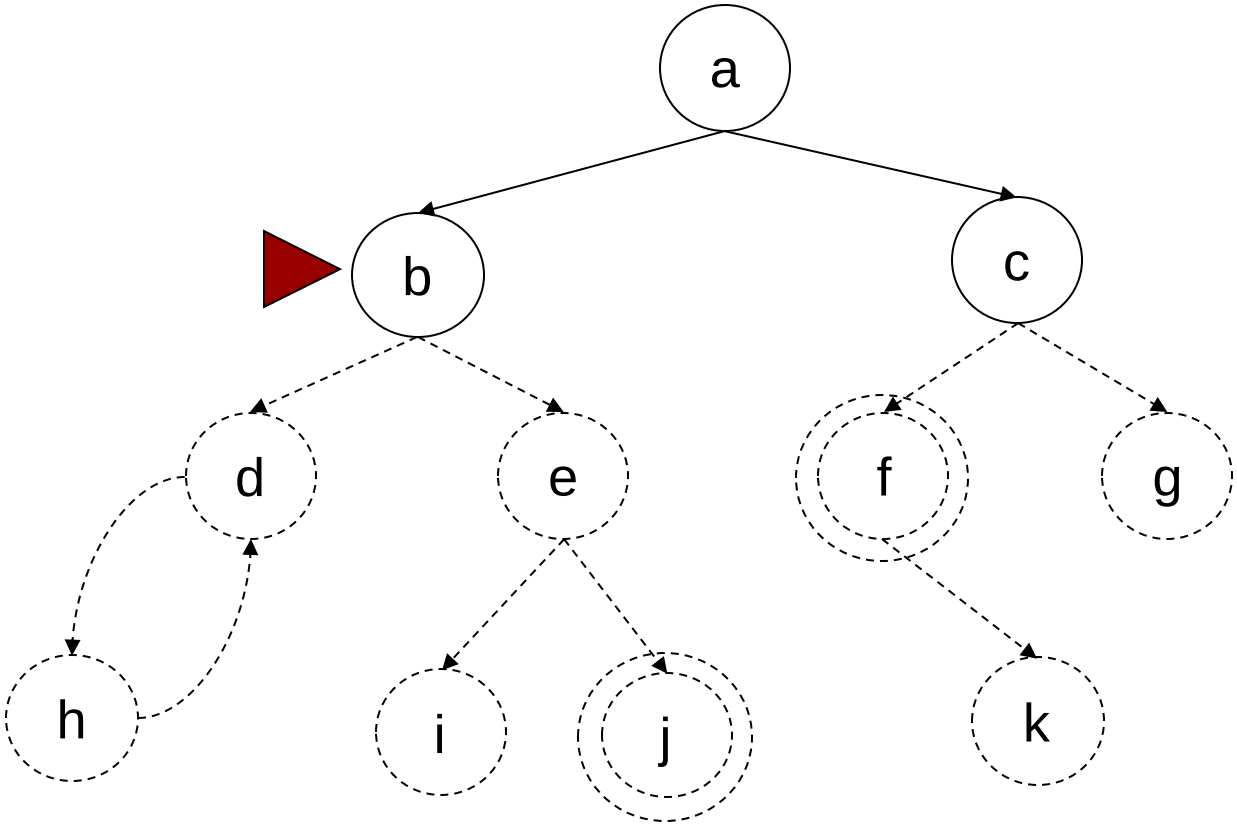
- Em contraste com a busca em profundidade, a **busca em largura escolhe** primeiro visitar aqueles nós **mais próximos do nó inicial**
- O algoritmo não é tão simples, pois **é necessário manter um conjunto de nós candidatos alternativos** e não apenas um único, como na busca em profundidade
- O **conjunto de nós candidatos** é todo o nível inferior da árvore de busca
- Além disso, só o conjunto é insuficiente se o caminho da solução também for necessário
- Assim, ao invés de manter um conjunto de nós candidatos, **é necessário manter um conjunto de caminhos candidatos**

# Busca em Largura



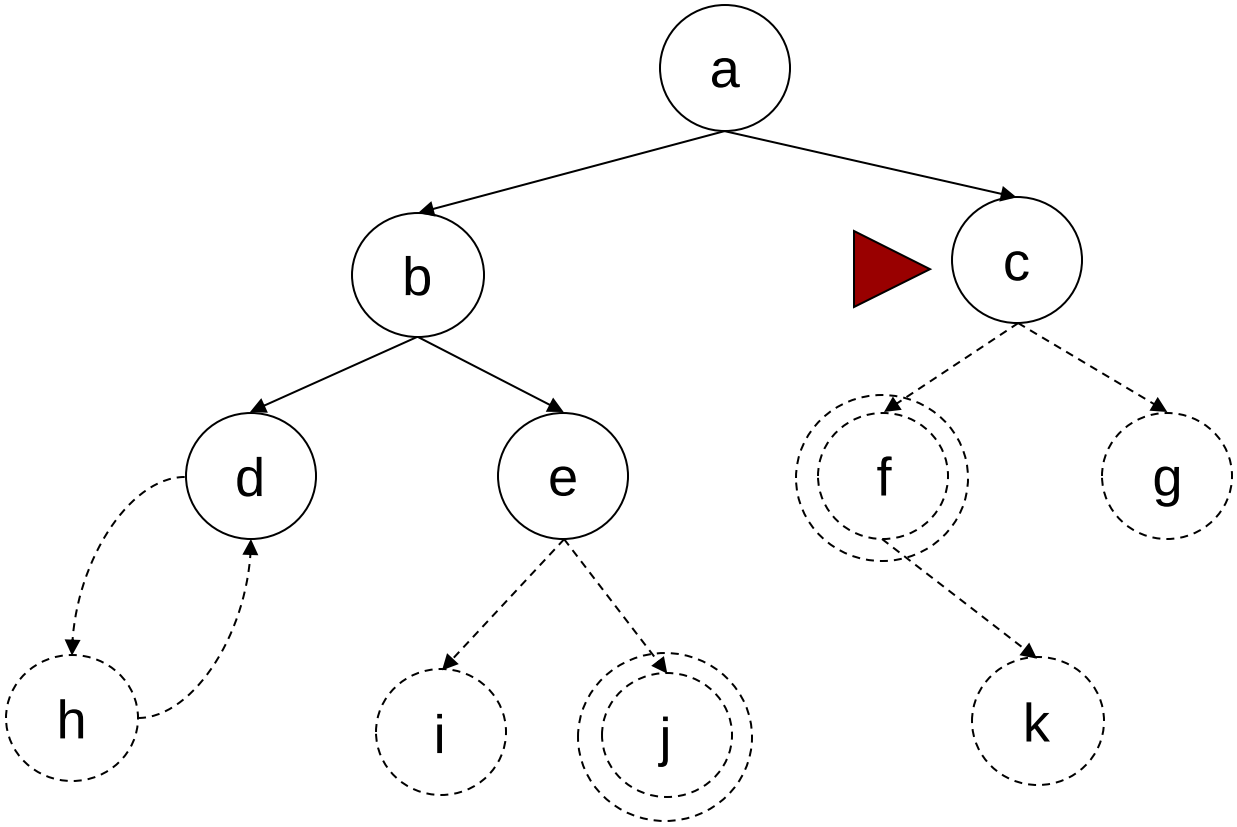

Inserir no final, remover da frente: a

# Busca em Largura



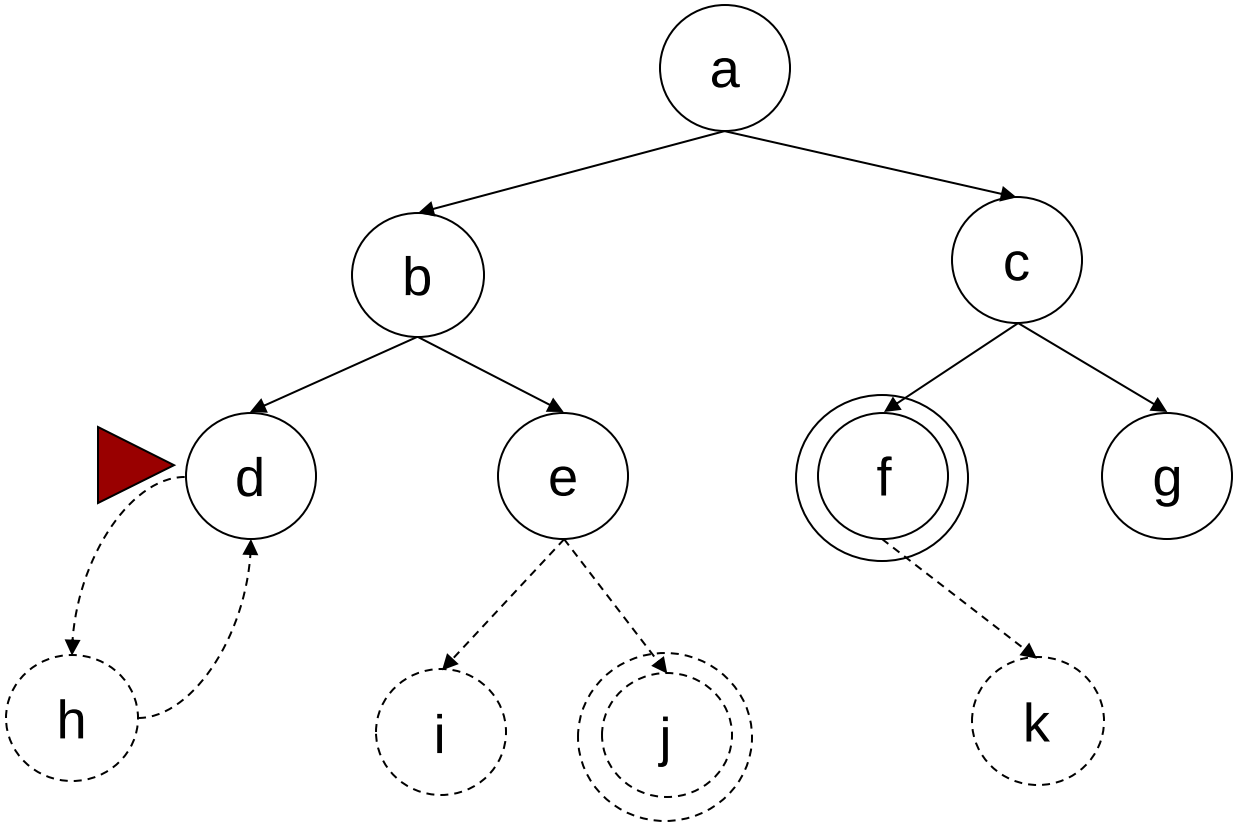

Inserir no final, remover da frente: b, c

# Busca em Largura



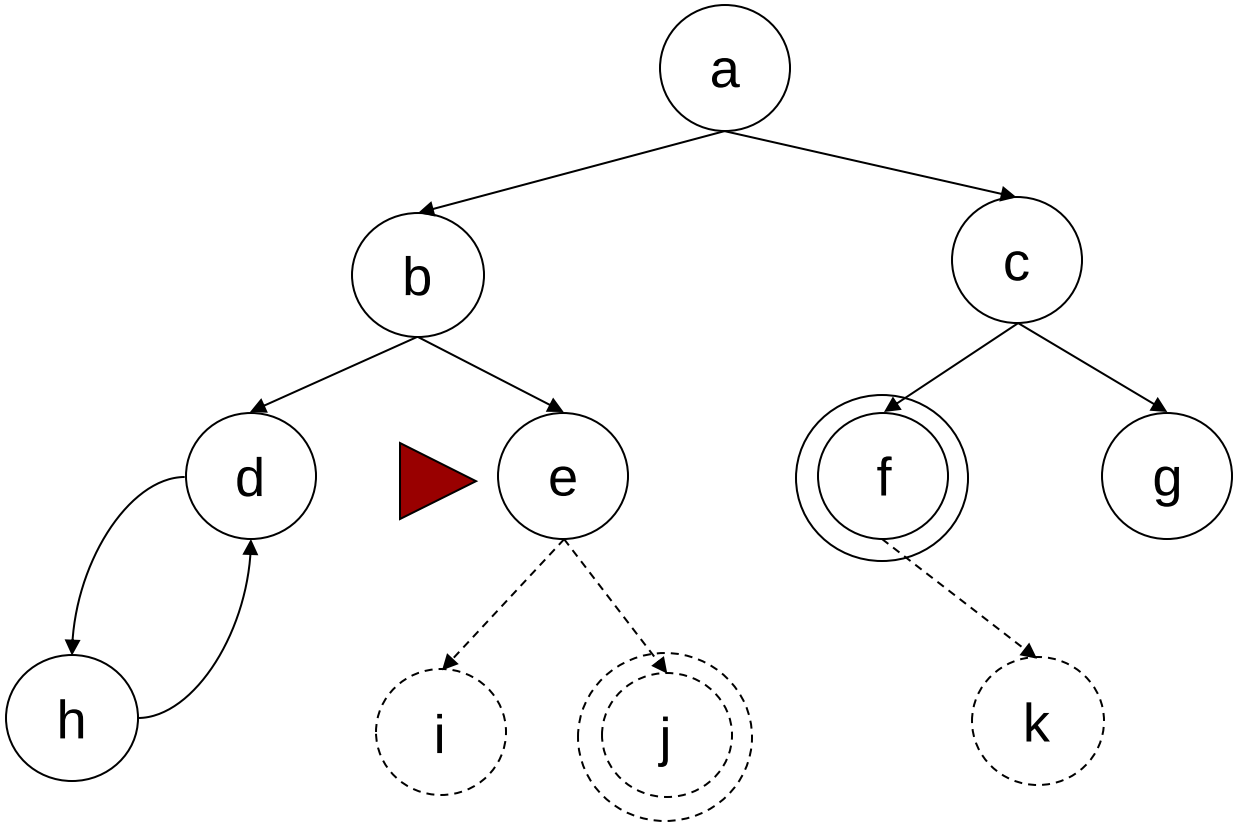

Inserir no final, remover da frente: c, d, e

# Busca em Largura



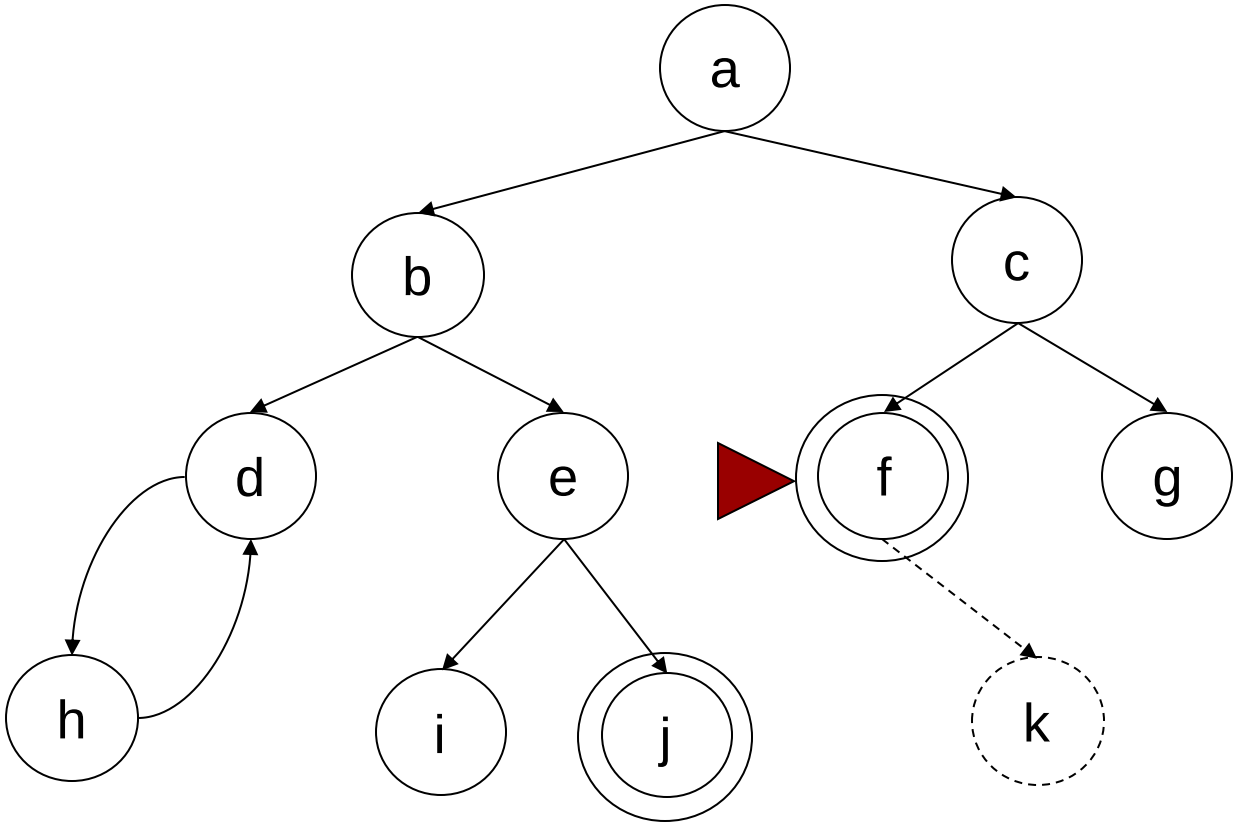

Inserir no final, remover da frente: d, e, f, g

# Busca em Largura



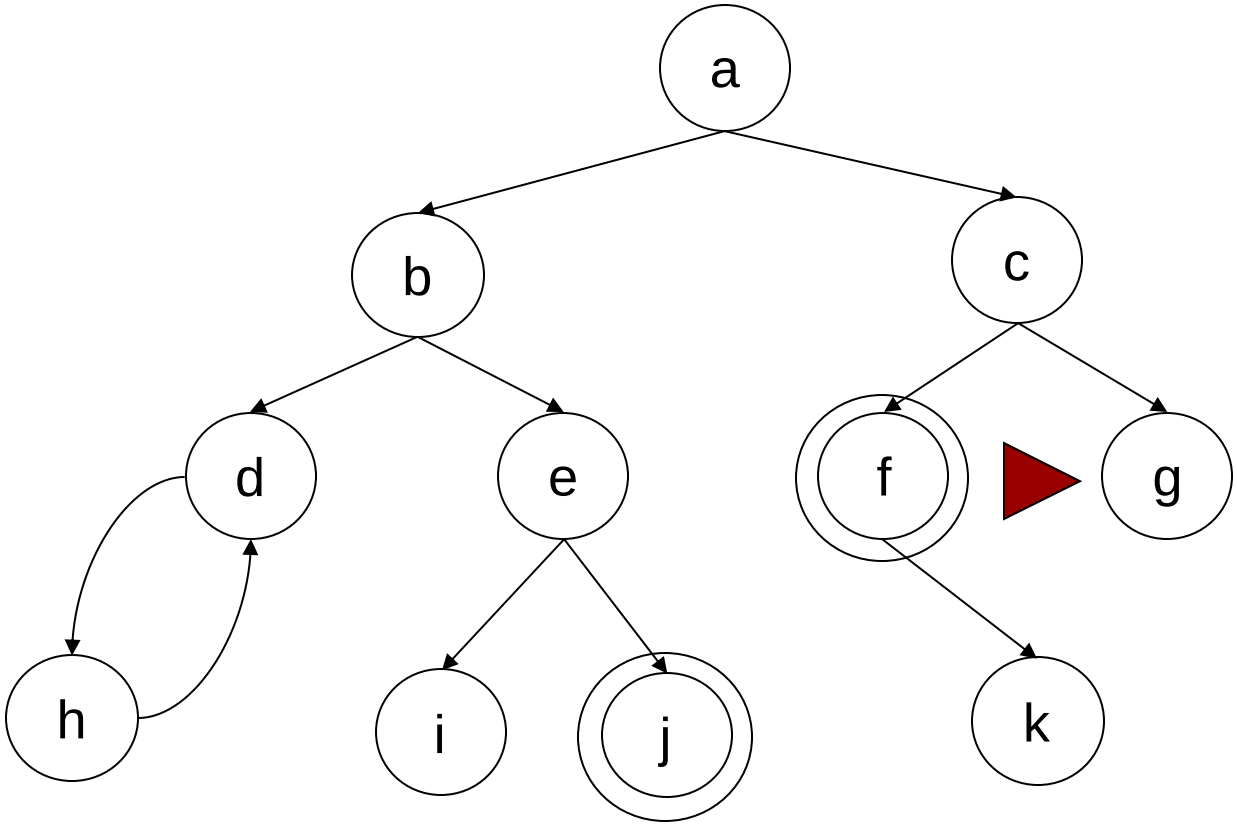

Inserir no final, remover da frente: e, f, g, h

# Busca em Largura



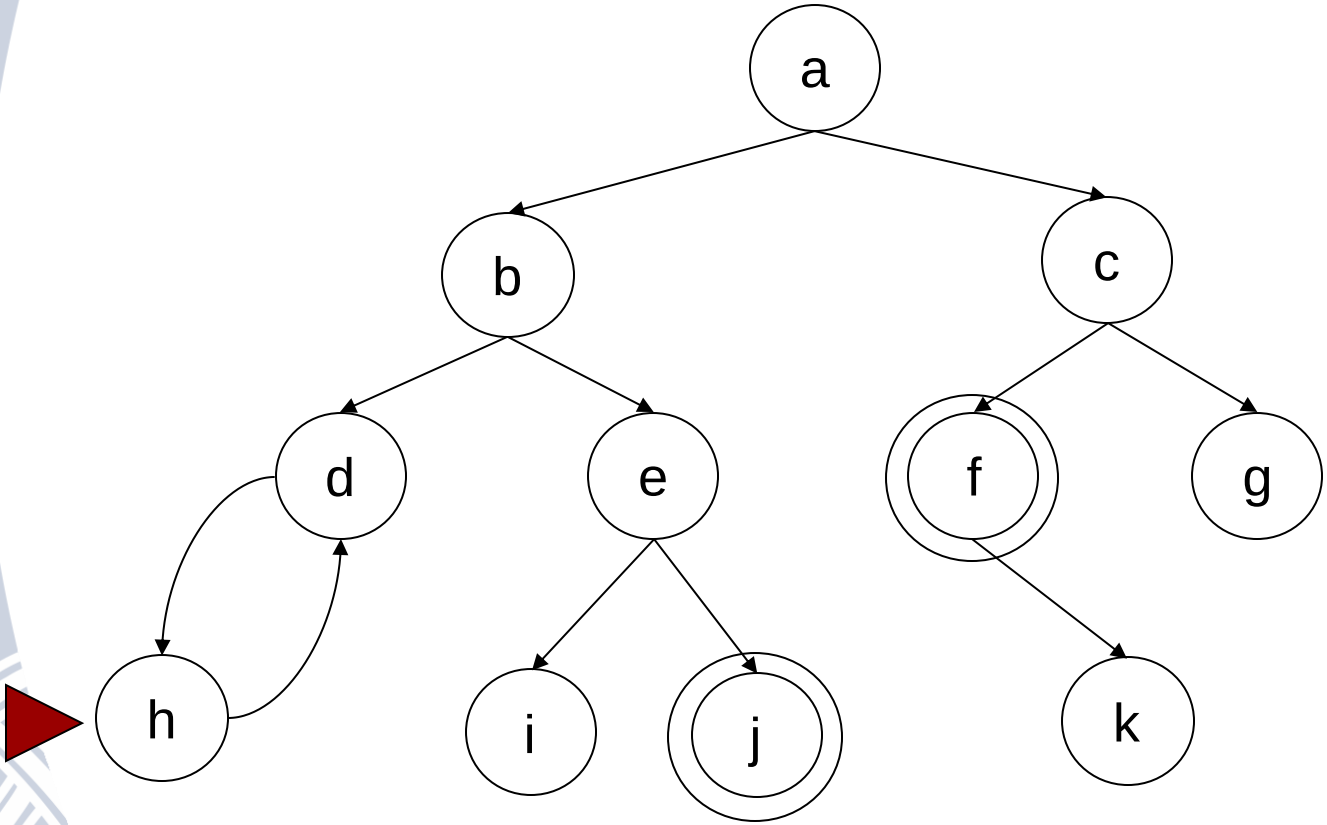

Inserir no final, remover da frente: f, g, h, i, j

# Busca em Largura



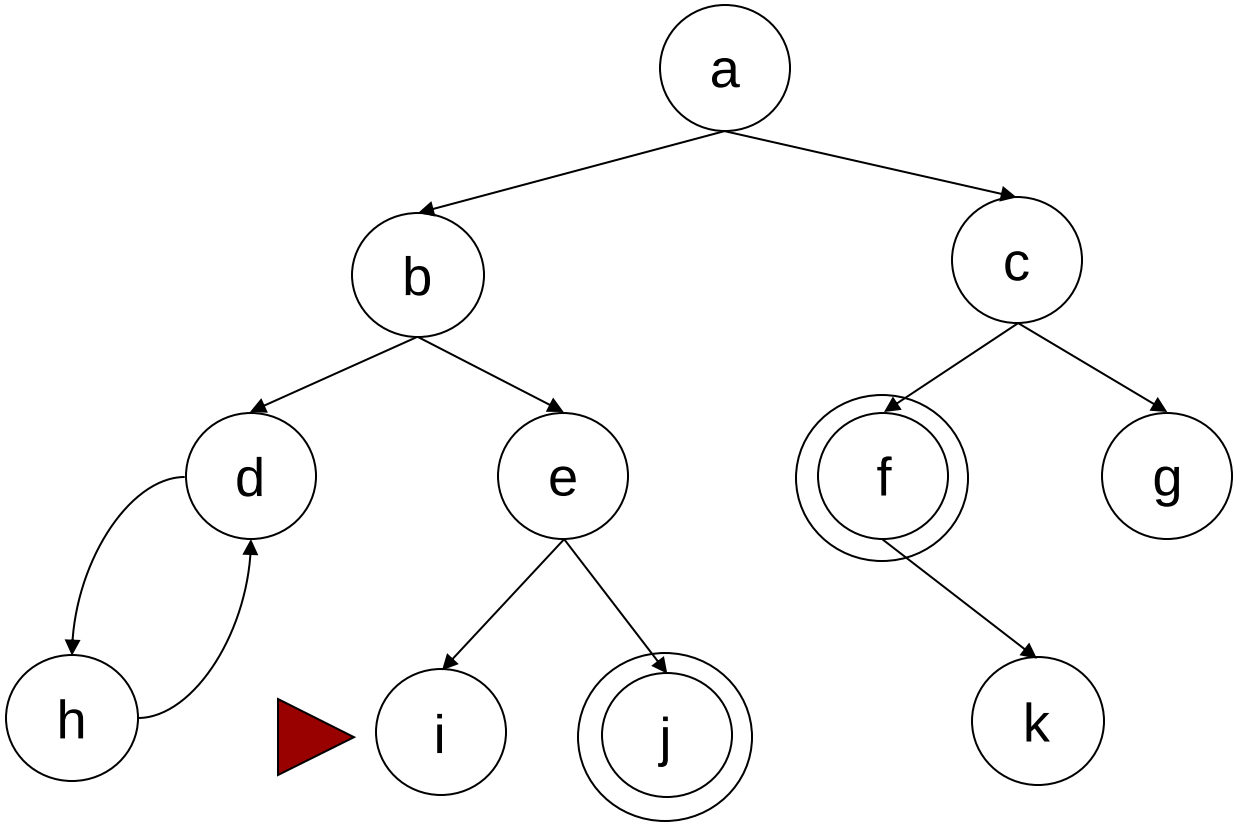

Inserir no final, remover da frente: g, h, i, j, k

# Busca em Largura



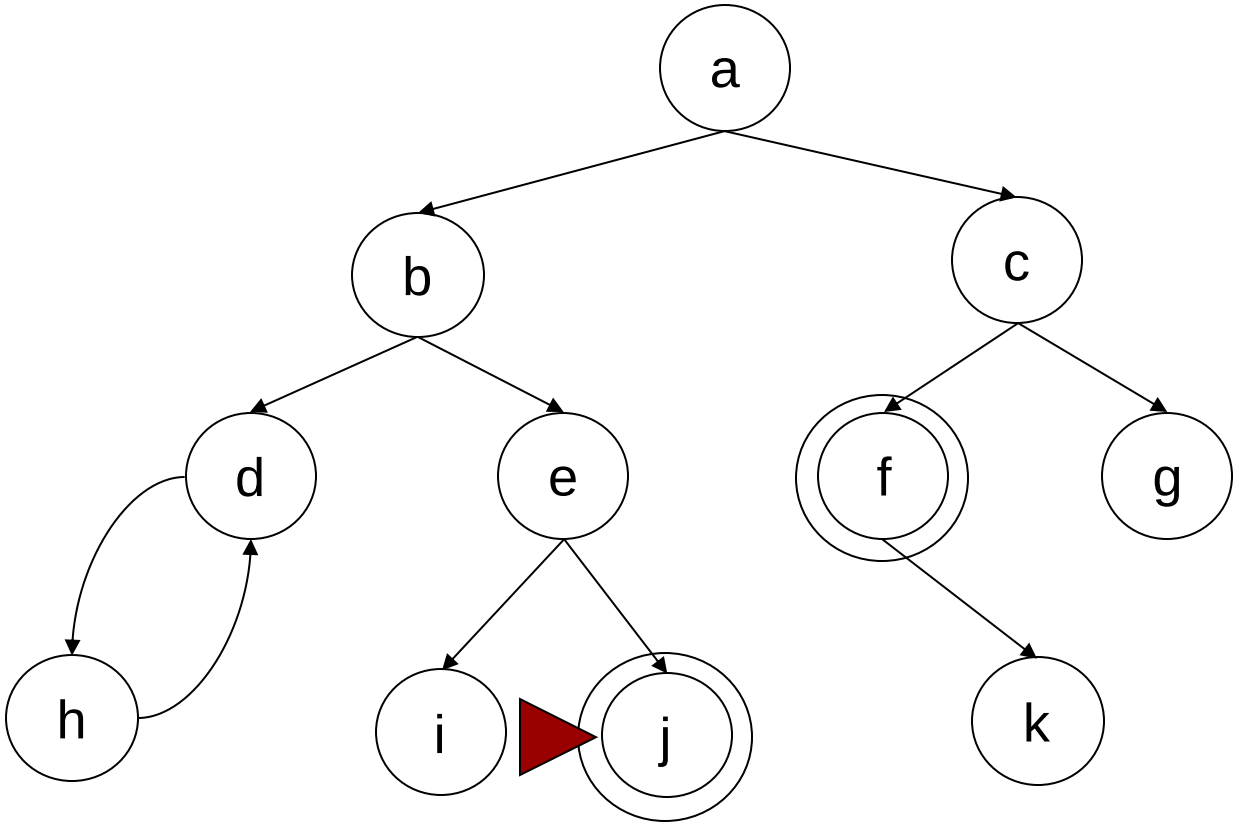

Inserir no final, remover da frente: h, i, j, k

# Busca em Largura



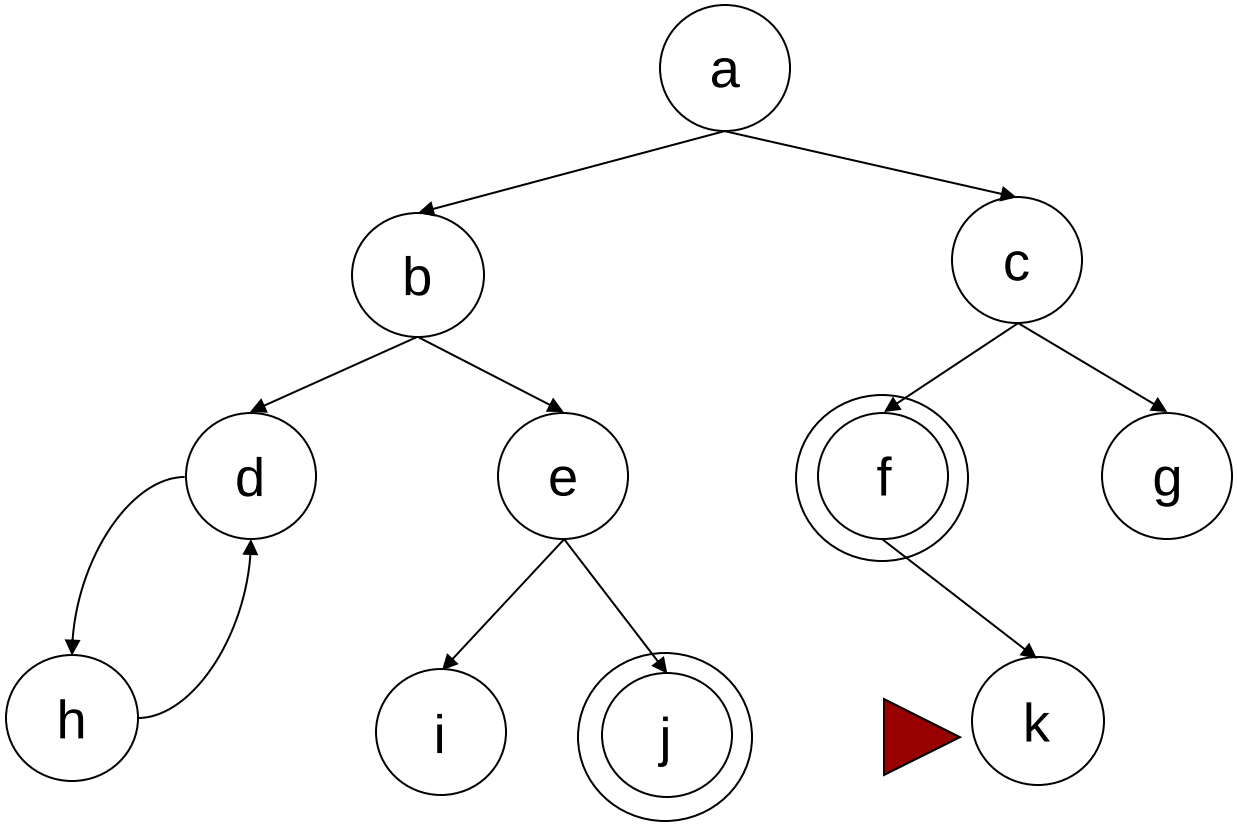

Inserir no final, remover da frente: i, j, k

# Busca em Largura




Inserir no final, remover da frente: j, k

# Busca em Largura

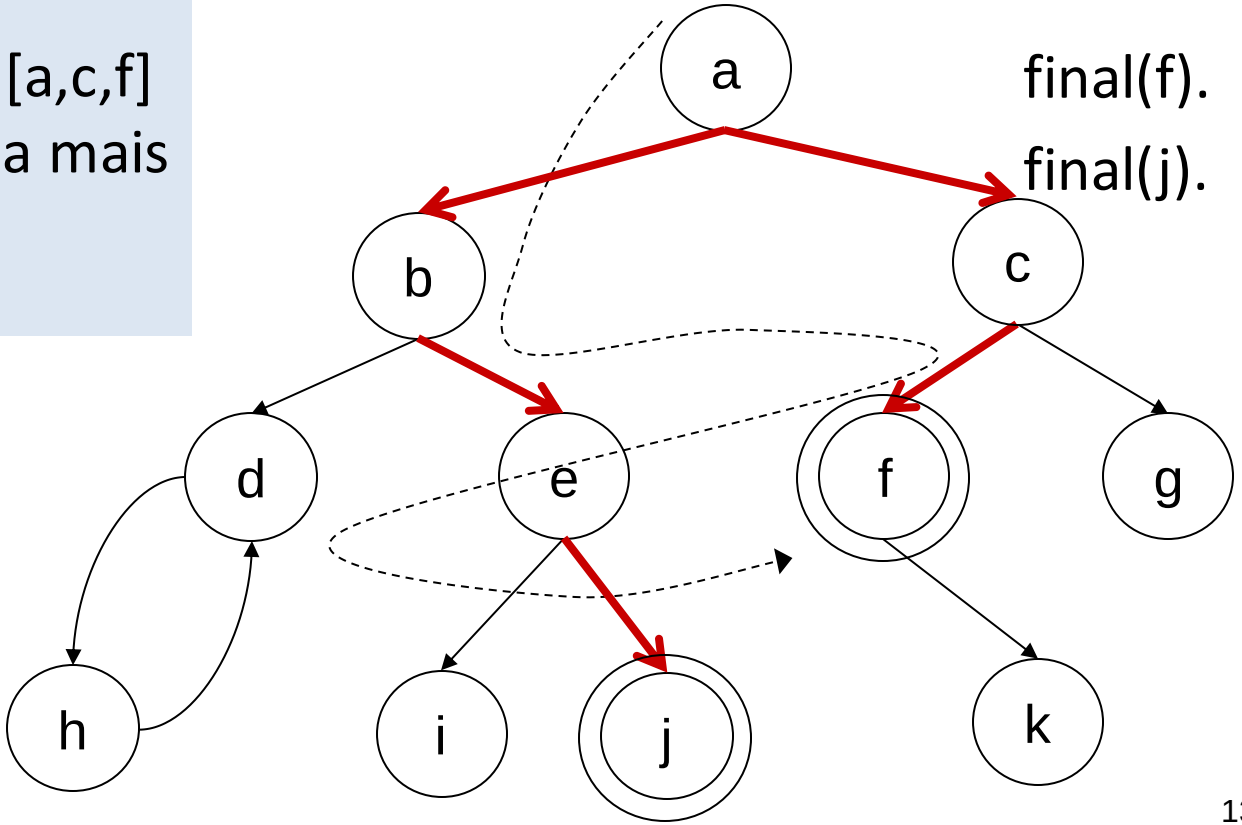



Inserir no final, remover da frente: k

# Busca em Largura

- Estado inicial: a
- Estados finais: j,f
- Nós visitados na ordem: a,b,c,d,e,f
- A solução mais curta [a,c,f] é encontrada antes da mais longa [a,b,e,j]

```
s(a,b).   s(c,f).   s(e,i).
s(a,c).   s(c,g).   s(e,j).
s(b,d).   s(d,h).   s(f,k).
s(b,e).   s(h,d).
```




```
final(f).
final(j).
```

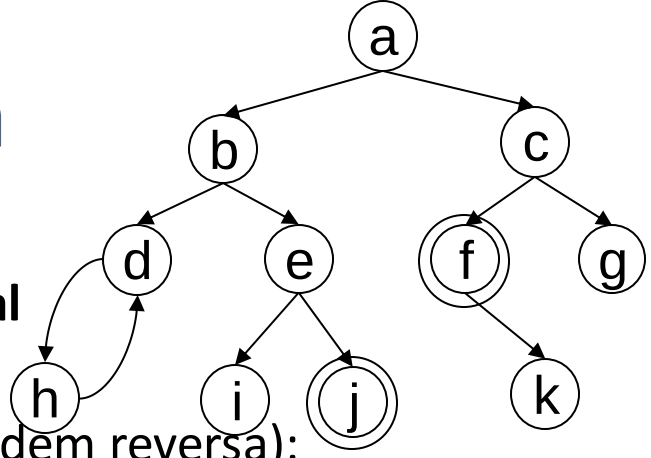
# Busca em Largura

- O **conjunto de caminhos** candidatos será representado como uma **lista de caminhos** e **cada caminho será uma lista de nós na ordem reversa**

- A **busca inicia** com um conjunto de um único candidato:
  - [ [Início] ]

- **Um algoritmo de busca em largura:**
  - Se a **cabeça** do primeiro caminho é um **nó final** então este caminho é uma solução; caso contrário
  - **Remova o primeiro caminho do conjunto de candidatos** e **gere o conjunto de todas as extensões** em um passo a partir deste caminho; adicione este conjunto de extensões ao **final do conjunto** de candidatos e execute busca em largura para atualizar este conjunto

# Busca em Largura




- Comece com o **conjunto de candidatos inicial**
  - [ [a] ]
- **Gere extensões** de [a] (note que estão em ordem reversa):
  - [ [b,a], [c,a] ]
- **Remova o primeiro candidato** [b,a] do conjunto de candidatos e gere extensões deste caminho
  - [d,b,a], [e,b,a]
- **Adicione as extensões ao final** do conjunto de candidatos
  - [ [c,a], [d,b,a], [e,b,a] ]
- Remova [c,a] e adicione as extensões ao final
  - [ [d,b,a], [e,b,a], [f,c,a], [g,c,a] ]
- Estendendo [d,b,a]
  - [ [e,b,a], [f,c,a], [g,c,a], [h,d,b,a] ]
- Estendendo [e,b,a]
  - [ [f,c,a], [g,c,a], [h,d,b,a], [i,e,b,a], [j,e,b,a] ]
- A busca **encontra a lista** [f,c,a] que **contém um nó final**, portanto o caminho é retornado como uma solução

# Implementação Prolog Busca em Largura

```
buscaLargura([[No|Caminho]|_],[No|Caminho]):-
  final(No).

buscaLargura([Caminho|OutrosCaminhos],Sol):-
     estender(Caminho,NovosCaminhos),
     concatenar(OutrosCaminhos,NovosCaminhos,
                                  Caminhos1),
     buscaLargura(Caminhos1,Sol).
```

# Implementação Prolog Busca em Largura

```
% resolva(No,Solucao) Solucao é um caminho
  acíclico (na ordem reversa) entre nó inicial No
  e uma solução.
 resolva(No,Solucao) :-
     buscaLargura([[No]],Solucao).
```

```
concatena([],L,L).
concatena([X|Y],L,[X|Lista]):-
      concatena(Y,L,Lista).
```

```
estender([No|Caminho],NovosCaminhos):-
      findall([NovoNo,No|Caminho],(s(No,NovoNo),
      not(pertence(NovoNo,[No|Caminho]))),
                                  NovosCaminhos).
```

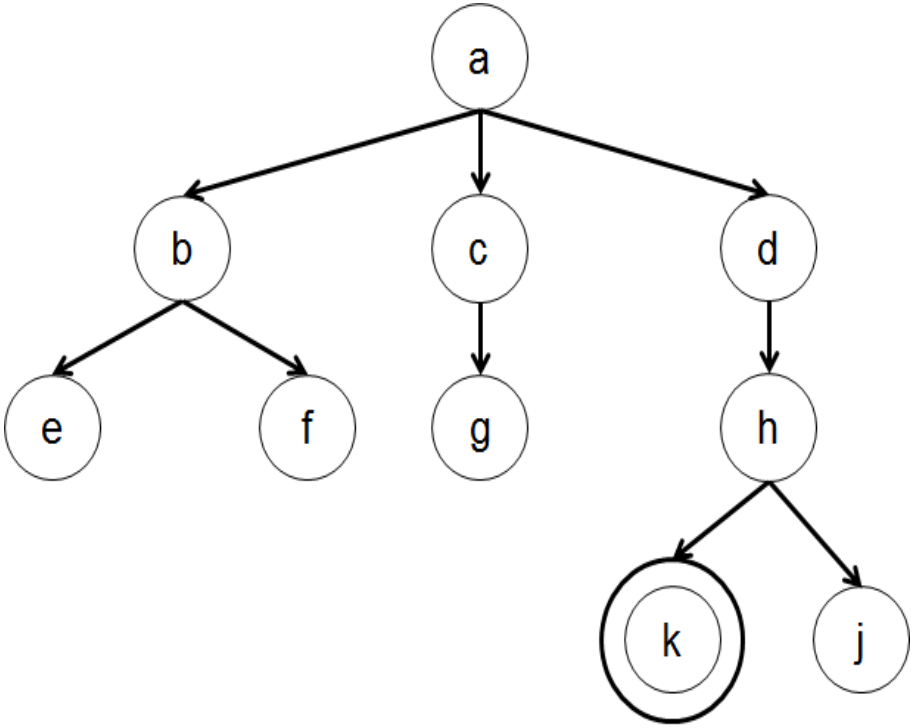
# Algoritmo Busca em Largura

- **Um algoritmo de Busca em Largura pode ser definida da seguinte forma:**

1) Insere na fila F o nó u e marque-o como alcançado
2) Enquanto fila F não vazia faça
   **v** ← elemento da frente da fila
   (retire v da fila)
   **para todo w** que partir de **v**,
   e **w** ainda não foi alcançado
   - marque **w** como alcançado
   - insira **w** na fila **F**

# Exercício – Busca em Largura

- Considere a busca no espaço abaixo, em que **"a"** é o estado inicial e **"k"** é o estado final.




- Considerando a modelagem do espaço de busca feita no exercício do slide 38, execute o **algoritmo em prolog de busca em largura** para encontrar a solução do problema. Mostre o passo a passo da execução, os nós visitados e o caminho encontrado.

# Complexidade dos Algoritmos de Busca

- b = número de caminhos alternativos/
  fator de bifurcação/ramificação (branching factor
- d = profundidade da solução
- m = profundidade máxima da árvore de busca
- l = limite de profundidade

| | **Tempo** | **Espaço** | **Solução menor altura garantida** | **Completa? (encontra uma solução quando ela existe)** |
|---|---|---|---|---|
| **Profundidade** | $O(b^m)$ | O(bm) | Não | Sim (espaços finitos)<br>Não (espaços infinitos) |
| Profundidade limitada | $O(b^l)$ | O(bl) | Não | Sim se $l \geq d$ |
| Profundidade iterativa | $O(b^d)$ | O(bd) | Sim | Sim |
| **Largura** | $O(b^d)$ | $O(b^d)$ | Sim | Sim |

# Modele um problema como uma árvore de busca

- Use aplicativos para facilitar o entendimento dos algoritmos de busca

  http://www.aispace.org/search/search.jnlp