# Pesquisa

Prof. Tiago Massoni

Engenharia da Computação

Poli - UPE

## Busca (pesquisa)

- Estudo de como recuperar informação a partir de uma grande massa de informação previamente armazenada
- A informação é dividida em **registros**
  - Cada registro possui uma chave para ser usada na pesquisa
- **Objetivo**
  - Encontrar uma ou mais ocorrências de registros com chaves iguais à chave de pesquisa.
- **Pesquisa com sucesso** X **Pesquisa sem sucesso**



## Conceitos

- Conjunto de registros ou arquivos: tabelas
- **Tabela:** associada a entidades de vida curta, criadas na memória interna durante a execução de um programa
- **Arquivo:** associado a entidades de vida mais longa, armazenadas em memória externa
  - **Distinção não é rígida**



## Tipos de pesquisa

- Depende principalmente
  - Quantidade dos dados envolvidos
  - Pode estar sujeito a inserções e retiradas freqüentes
- Se conteúdo do arquivo é estável é importante minimizar o tempo de pesquisa, sem preocupação com o tempo necessário para estruturar o arquivo



## TAD Dicionário

- Nome comumente utilizado para descrever uma estrutura de dados para pesquisa
- **Operações**
  1. Inicializa
  2. Pesquisa
  3. Insere
  4. Retira
- Analogia com um dicionário da língua portuguesa:
  - Chaves = palavras
  - Registros = entradas associadas com cada palavra
    - pronúncia
    - definição
    - sinônimos
    - outras informações



## Pesquisa seqüencial

- Método de pesquisa mais simples
- A partir do primeiro registro, pesquise seqüencialmente até encontrar a chave procurada; então pare
- Armazenamento de um conjunto de registros por meio do tipo estruturado arranjo
  - Se ordenado, então pesquisa sem sucesso é mais eficiente

## Pesquisa seqüencial

```
public class Tabela {
  private Comparable registros[]; private int n;
  public Tabela (int maxN){
    this.registros= new Comparable[maxN+1];
    this.n= 0;
  }

  public int pesquisa (Comparable reg){
    this.registros[0]= reg; //sentinela
    int i = this.n;
    while (this.registros[i].compareTo(reg)!=0)
      i--;
    return i ;
  }

  public void insere (Comparable reg){
   if(this.n == (this.registros.length-1))
     /*ERRO! Tabela cheia!!*/
   this.registros[++this.n]= reg;
  }
}
```

## Pesquisa seqüencial

- Cada registro contém um campo chave que identifica o registro
  - Além da chave, podem existir outros componentes em um registro, que não têm influência nos algoritmos
- O método pesquisa retorna o índice do registro que contém a chave no parâmetro; caso não esteja presente, o valor retornado é zero
- Essa implementação não suporta mais de um registro com a mesma chave
  - Formas alternativas: retornar lista de índices, por exemplo



## Pesquisa seqüencial

- Utilização de um registro sentinela na posição zero do array
  - Garante que a pesquisa sempre termina (se o índice retornado por pesquisa for zero, a pesquisa foi sem sucesso)
  - Não é necessário testar se i > 0, devido a isto
- O anel interno da pesquisa é extremamente simples
  - o índice i é decrementado e a chave de pesquisa é comparada com a chave que está no registro
- Isto faz com que esta técnica seja conhecida como pesquisa seqüencial rápida



## Custo da pesquisa sequencial

- Pesquisa com sucesso
  - melhor caso : $C(n) = 1$
  - pior caso : $C(n) = n$
  - caso médio : $C(n) = (n + 1)/2$
- Pesquisa sem sucesso:
  - $C(n) = n + 1$
- O algoritmo de pesquisa seqüencial é a **melhor escolha** para o problema de pesquisa em tabelas com até **25 registros**



## Pesquisa binária

- Pesquisa em tabela pode ser mais eficiente se registros forem mantidos em ordem
- Para saber se uma chave está presente na tabela:
  - Compare a chave com o registro que está na posição do meio da tabela
  - **Se** a chave é menor **então** o registro procurado está na primeira metade da tabela
  - **Se** a chave é maior **então** o registro procurado está na segunda metade da tabela
  - Repita o processo até que a chave seja encontrada, ou fique apenas um registro cuja chave é diferente da procurada, significando uma pesquisa sem sucesso



## Pesquisa binária

```
public int binaria(Comparable chave){
  if(this.n==0) return 0;
  int esq= 1 ,dir= this.n, i;
  do{
    i=(esq + dir)/2;
    if(chave.compareTo(this.registros[i])>0)
       esq= i + 1;
    else dir= i - 1;
  }while((chave.compareTo(this.registros[i])!=0)
        && (esq<=dir));
  if(chave.compareTo(this.registros[i])==0)
    return i;
  else return 0;
}
```

## Exemplo: procura pela chave G

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| Chaves iniciais: | *A* | *B* | *C* | *D* | *E* | *F* | *G* | *H* |
| | *A* | *B* | *C* | ***D*** | *E* | *F* | *G* | *H* |
| | | | | | *E* | ***F*** | *G* | *H* |
| | | | | | | | ***G*** | *H* |



## Análise da pesquisa binária

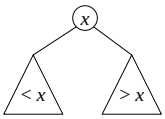
- A cada iteração do algoritmo, o tamanho da tabela é dividido ao meio
- O número de vezes que o tamanho da tabela é dividido ao meio é cerca de log n
- Ressalva: o custo para manter a tabela ordenada é alto
  - A cada inserção na posição p da tabela implica no deslocamento dos registros a partir da posição p para as posições seguintes
  - A pesquisa binária não deve ser usada em aplicações muito dinâmicas



## Árvore de busca binária

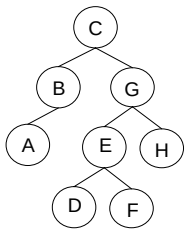
•Uma árvore binária é uma árvore de busca binária quando:

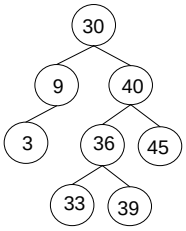
- todo elemento armazenado na subárvore esquerda é menor que a raiz x;
- nenhum elemento armazenado na subárvore direita é menor que a raiz x;
- as subárvores esquerda e direita também são árvores de busca binária.






## Exemplos






## Pesquisa em uma Árvore de Busca Binária

- Para encontrar um registro com uma chave reg
  - Compare-a com a chave que está na raiz .
  - Se é menor, vá para a subárvore esquerda.
  - Se é maior, vá para a subárvore direita.
- Repita o processo recursivamente, até que a chave procurada seja encontrada ou um nó folha é atingido
- Se a pesquisa tiver sucesso então o registro contendo a chave passada em reg é retornado

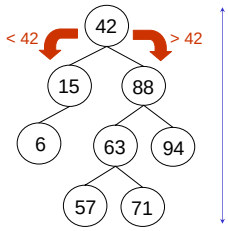


## Pesquisa em uma Árvore de Busca Binária

```
public class SearchTree {
  private Node root;
  ...
  Comparable pesquisa(Comparable reg,Node p){
    if (p== null) return null; //Nao encontrado
    else{
      if(reg.compareTo(p.getElement())<0)
        return pesquisa(reg,p.getLeft());
      else
        if(reg.compareTo(p.getElement())>0)
          return pesquisa(reg,p.getRight());
        else return p.getElement;
    }
  }
}
```

# Pesquisa em uma Árvore de Busca Binária

Complexidade da pesquisa de um objeto

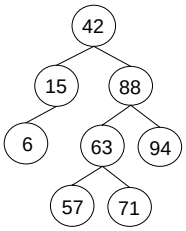



$O(\log_2 n)$



# Inclusão em uma Árvore de Busca Binária

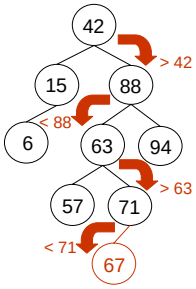
- Navegação na subárvore ESQ ou DIR até encontrar a posição correta para inclusão
- Complexidade da inclusão de um objeto → $O(\log_2 n)$




incluir chave 67



# Inclusão em uma Árvore de Busca Binária




➡ incluir chave 67



# Inclusão em uma Árvore de Busca Binária

```
Node insere(Comparable reg,Node p){
  if (p == null){
    p =new Node();
    p.setElement(reg);
    p.esq= null;
    p.dir= null;
  }
  else {
    if(reg.compareTo(p.getElement())<0){
      p.setLeft(insere(reg,p.getLeft());
    else
      if(reg.compareTo(p.getElement())>0)
        p.dir=insere(reg,p.getRight());
      else
        //Erro:Registro ja existe
  }
  return p;
}
```



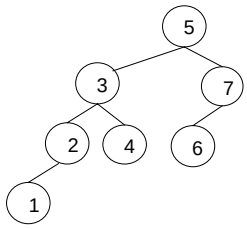
# Exclusão em uma Árvore de Busca Binária

- A retirada de um registro não é tão simples quanto a inserção
- Se o nó que contém o registro a ser retirado possui no máximo um descendente, a operação é simples
- No caso do nó conter dois descendentes o registro a ser retirado deve ser primeiro:
  - substituído pelo registro mais à direita na subárvore esquerda
  - ou pelo registro mais à esquerda na subárvore direita



# Exclusão

- Exemplos
  - caso 1: excluir chave 6 (simples)
  - caso 2: excluir chave 7 (passa filho para pai)
  - caso 2: excluir chave 5 (troca por 4 ou pelo 6)



## Exclusão

```
private Node retira(Comparable reg, Node p){

if(p==null)
    //Erro:Registro nao encontrado
else if(reg.compareTo(p.getElement())<0)
      p.setLeft(retira(reg,p.getLeft());

else if(reg.compareTo(p.getElement())>0)
      p.setRight(retira(reg,p.dir));

else {
      if(p.getRight() == null) p=p.getLeft();
      else if(p.getLeft()==null) p=p.getRight();
      else p.setLeft(antecessor(p,p.getLeft());
     }

return p;

}
```

## Exclusão (método auxiliar)

```
private Node antecessor(Node q, Node r){
  if (r.getRight()!=null)
    r.setRight(antecessor(q,r.getRight()));
  else{
    q.setElement(r.reg);
    r = r.getLeft();
  }
  return r ;
}
```

Escolha: elemento mais à direita da sub-árvore esquerda



## Impressão ordenada

```
public void imprime (){
  this.em_ordem(this.root);
}

private void em_ordem(Node p) {
  if (p != null){
    em_ordem(p.getLeft());
    System.out.println(p.getElement().toString());
    em_ordem(p.getRight());
  }
}
```



## Problema da Árvore de Busca Binária

- O desbalanceamento progressivo de uma árvore de busca binária tende a tornar linear a complexidade de pesquisa:

  $O(\log_2 n) \rightarrow O(n)$

- Exemplo:
  - ordem de inclusão: 1, 13, 24, 27, 56
  - complexidade da pesquisa: $O(n)$

Alternativa de solução: Árvore AVL

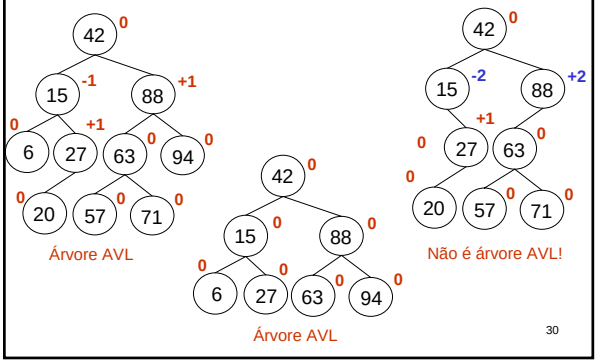


## Árvore AVL

- Uma árvore AVL A é uma árvore de busca binária tal que toda subárvore tem alturas das subárvores esquerda e direita diferindo de até 1
- Uma árvore AVL é uma Árvore de Busca Binária Balanceada
- AVL (Adelson-Velskii e Landis)



## Exemplos



## Operação de Rotação

- Como manter uma árvore AVL sempre balanceada após uma inclusão ou exclusão?
  - através de uma operação de Rotação
- Características da operação:
  - preserva a ordem das chaves
  - é preciso armazenar o nível em cada raíz, para poder definir o momento de fazer rotação (fatorB)
  - basta uma execução da operação de rotação para tornar a árvore novamente AVL



## Operação de Rotação

Tipos de rotação:

Rotação simples:
- para a direita
- para a esquerda

Rotação dupla:
- para a direita
- para a esquerda



## Rotação Simples

Executada toda vez que uma árvore fica desbalanceada com um fatorB:

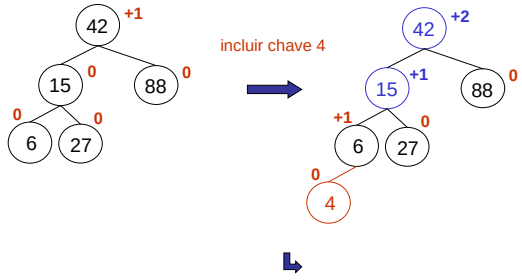
- positivo e sua subárvore ESQ também tem um fatorB positivo (Rotação Simples para a Direita)

ou

- negativo e sua subárvore DIR também tem um fatorB negativo (Rotação Simples para a Esquerda)

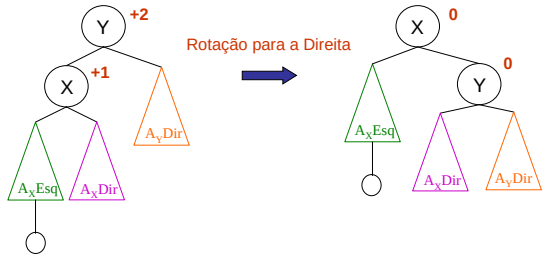


## Exemplo - Rotação Simples para a Direita



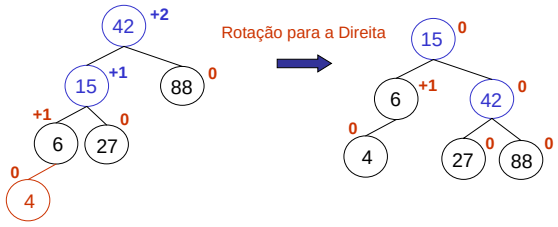



## Rotação Simples para a Direita






## Exemplo - Rotação Simples para a Direita



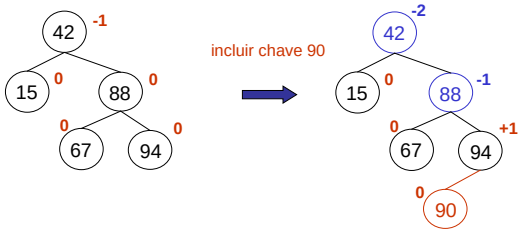
## Rotação simples para direita (algoritmo) em p

```
rotacaoDireita (Node p){
  q = p.getLeft();
  aux = q.getRight();
  q.setRight(p);
  p.setLeft(aux);
  root = q;
}
```

Complexidade: O(1)

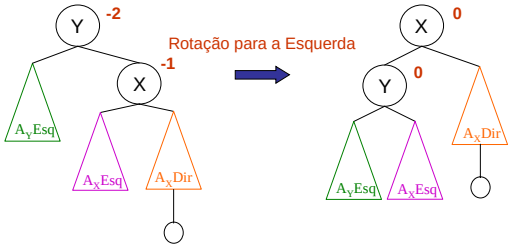


## Exemplo - Rotação Simples para a Esquerda



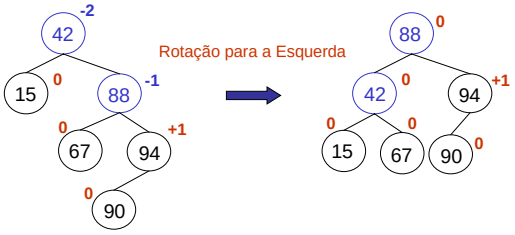



## Rotação Simples para a Esquerda






## Exemplo - Rotação Simples para a Esquerda






## Rotação Dupla

Executada toda vez que uma árvore fica desbalanceada com um fatorB:

- positivo e sua subárvore ESQ tem um fatorB negativo (Rotação Dupla para a Direita)

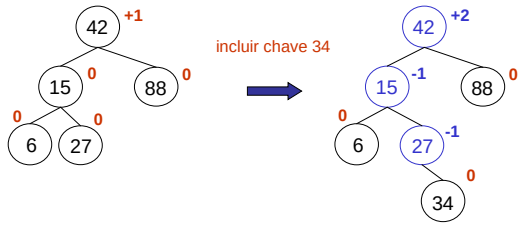
ou

- negativo e sua subárvore DIR tem um fatorB positivo (Rotação Dupla para a Esquerda)

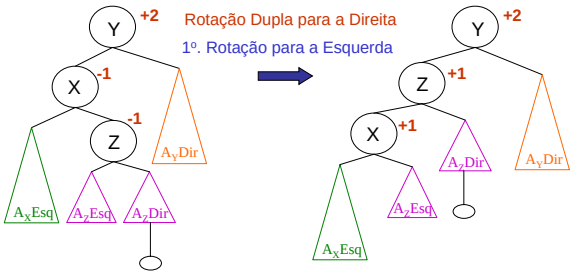
Complexidade: O(1)



## Exemplo - Rotação Dupla para a Direita

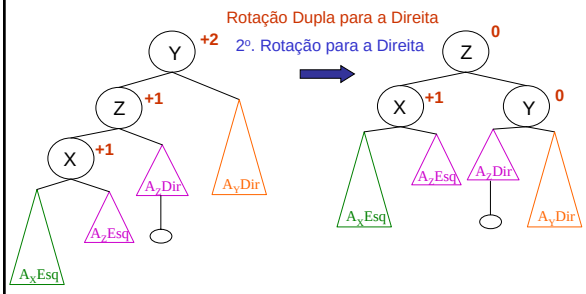


## Rotação Dupla para a Direita



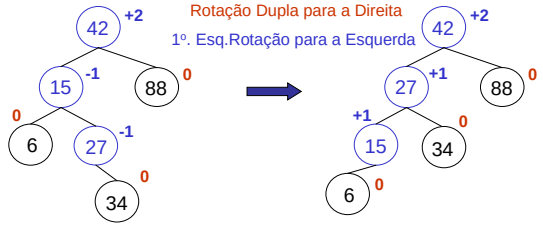



## Rotação Dupla para a Direita



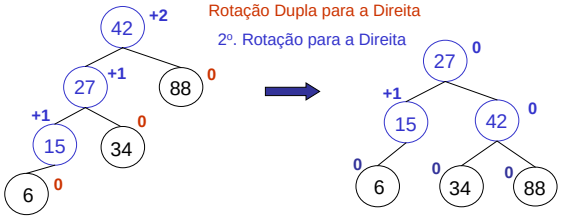



## Exemplo - Rotação Dupla para a Direita






## Exemplo - Rotação Dupla para a Direita






## Balanceamento

Quando realizar o balanceamento de uma arvore AVL?

• Sempre que a árvore apresentar um fatorB fora do intervalo [-1,1].

• O valor do fatorB deve ser testado sempre após uma operação de inclusão ou exclusão de nodo na árvore.



## Pesquisa digital

- Baseada na representação das chaves como uma seqüência de caracteres ou de dígitos
- Os métodos de pesquisa digital são particularmente vantajosos quando as chaves são grandes e de tamanho variável.
- Possibilidade de localizar ocorrências de uma determinada cadeia em um texto, com tempo de resposta logarítmico em relação ao tamanho do texto
  - Árvores Trie
  - Árvores Patrícia

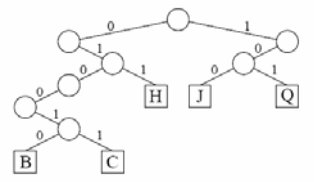
## Trie

- Árvore M-ária cujos nós são vetores de M componentes com campos dígitos ou caracteres que formam as chaves
  - Cada nó no nível i representa o conjunto de todas as chaves que começam com a mesma seqüência de i dígitos ou caracteres
- Este nó especifica uma ramificação com M caminhos dependendo do (i + 1)-ésimo dígito ou caractere de uma chave



## Trie

- Se chaves são seqüência de bits (M=2), o algoritmo é semelhante ao de pesquisa em árvore,
  - em vez de caminhar na árvore comparando chaves, caminha-se de acordo com os bits de chave

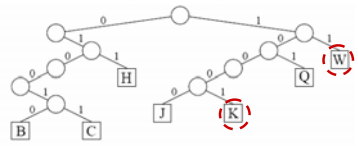



B = 010010
C = 010011
H = 011000
J = 100001
Q = 101000



## Inserção em Trie

- Faz-se uma pesquisa na árvore com a chave a ser inserida
  - Se o nó externo vazio, cria-se um novo nó externo nesse ponto com a nova chave
  - exemplo: inserção da chave W = 110110
- Se o nó externo tiver uma chave cria-se nós internos cujos descendentes têm a chave já existente e a nova chave
  - exemplo: inserção da chave K = 100010






## Desvantagem da Trie

- A formação de caminhos de uma só direção para chaves com um grande número de bits em comum
- Se duas chaves diferem no último bit, elas formam um caminho igual ao tamanho delas, não importando outras chaves
  - Ex: Caminho gerado pelas chaves B e C



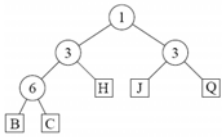
## Patrícia

- Practical Algorithm To Retrieve Information Coded In Alphanumeric
- Baseado no método de pesquisa digital, mas sem apresentar o inconveniente citado para o caso das tries
- Solução simples e elegante: cada nó interno da árvore contém o índice do bit a ser testado para decidir qual ramo tomar



## Patrícia

B = 010010
C = 010011
H = 011000
J = 100001
Q = 101000



## Pesquisa em Memória Secundária

- Arquivos contém mais registros do que a memória interna pode armazenar
- Custo para acessar um registro é algumas ordens de grandeza maior do que o custo de processamento na memória primária
  - Custo de trasferir dados entre principal e secundária (minimizar)
- Memórias secundárias: apenas um registro pode ser acessado em um dado momento (acesso seqüencial)
- Memórias primárias: acesso a qualquer registro de um arquivo a um custo uniforme (acesso direto)
- Em um método eficiente de pesquisa, o aspecto sistema de computação é importante
  - Algoritmos de pesquisa em memória secundária levam isso em consideração



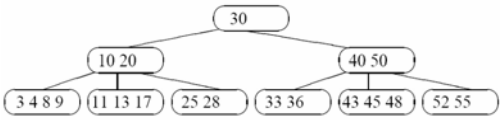
## Árvores B

- Árvores n-árias: mais de um registro por nó (=página)
- Em uma árvore B de ordem m:
  - página raiz: 1 - 2m registros
  - demais páginas: no mínimo m registros e m + 1 descendentes e no máximo 2m registros e 2m + 1 descendentes
- páginas folhas aparecem todas no mesmo nível
- Os registros aparecem em ordem crescente da esquerda para a direita



## Árvores B

- Árvore B de ordem m = 2 com três níveis






## Algoritmos para árvores B

```
public class ArvoreB {
  private static class Pagina {
    int n; Comparable[] r; Pagina p[];
    public Pagina (int mm) {
      this.n= 0;
      this.r= new Comparable[mm];
      this.p= new Pagina[mm+1];
    }
  }
  private Pagina root;
  private int m,mm;
  public ArvoreB (int m){
    this.root = null;
    this.m= m; this.mm= 2*m;
  }
  public Comparable pesquisa(Comparable reg){
    return this.pesquisa(reg,this.root);
  }
...
```

## Pesquisa em árvores B

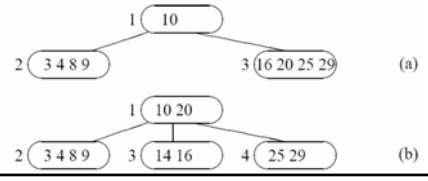
```
Comparable pesquisa(Comparable reg,Pagina ap){
  if (ap==null)return null; //não encontrado
  else{
   int i= 0;

   while((i<ap.n-1)&&(reg.compareTo(ap.r[i])>0))
     i++;

   if(reg.compareTo(ap.r[i])==0)
     return ap.r[i];
   else if(reg.compareTo(ap.r[i])<0)
     return pesquisa(reg,ap.p[i]);
   else return pesquisa (reg,ap.p[i+1]);
  }
}
```

## Árvore B: inserção

- Localizar a página apropriada onde o registro deve ser inserido
- Se o registro encontra uma página com menos de 2m registros, o processo de inserção termina
- Se o registro a ser inserido encontra uma página cheia, é criada uma nova página
  - no caso da página pai estar cheia o processo de divisão se propaga (balanceamento com metade do nó movendo, do meio sobe)
- Exemplo: Inserindo o registro com chave 14



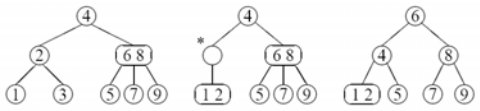
## Árvore B: remoção

- Se página com o registro a ser retirado é folha
  - retira-se o registro
  - se a página não possui pelo menos de m registros, a propriedade da árvore B é violada.
  - Pega-se um registro emprestado da página vizinha. Se não existir registros suficientes na página vizinha, as duas páginas devem ser fundidas em uma só.
- Se página com o registro não é folha
  - o registro a ser retirado deve ser primeiramente substituído por um registro contendo uma chave adjacente

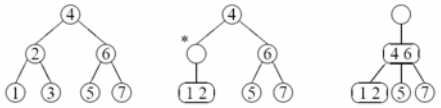


## Árvore B: remoção

- Retirando a chave 3



(a) Página vizinha possui mais do que $m$ registros

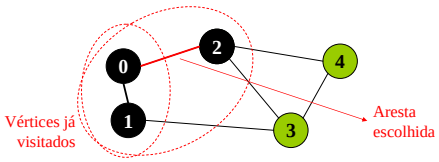


(b) Página vizinha possui exatamente $m$ registros



## Busca em Grafos

- Escolhendo um vértice inicial, é possível visitar os vértices seguindo uma determinada ordem.

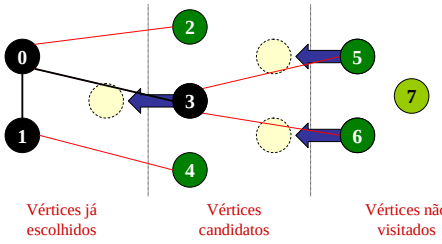



- A cada iteração, escolhemos uma aresta que parte de um vértice já visitado.



## Busca em Grafos

- A cada passo, os vértices são divididos em três grupos:






## Busca em Grafos

- Mais comuns
  - Busca em Profundidade (Depth-First Search)
    Escolhe arestas que partem do vértice mais recente
  - Busca em Largura (Breadth-First Search)
    Escolhe arestas que partem do vértice mais antigo

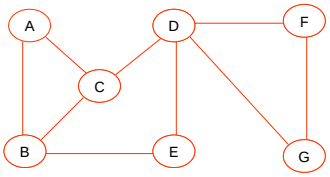


## Busca em Profundidade

- Nesse algoritmo a busca inicia a partir de um nó raiz e percorre cada caminho de forma a ir o mais longe possível antes de passar para outro caminho
- O caminho percorrido por esta busca forma uma árvore profunda
- Pode ser implementada com recursão (pilha)

## Busca em Profundidade

Pilha



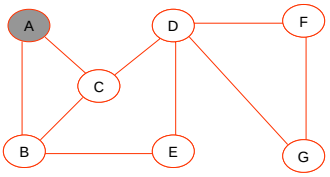

Quando um vértice é descoberto pela primeira vez ele torna-se cinza, e é tornado preto quando sua lista de adjacentes tenha sido completamente examinada



## Busca em Profundidade

Pilha

A



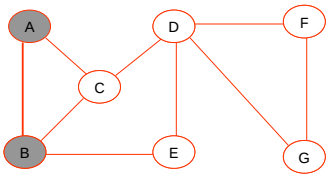



## Busca em Profundidade

Pilha

B

A





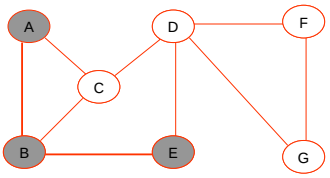

## Busca em Profundidade

Pilha

E

B

A






## Busca em Profundidade

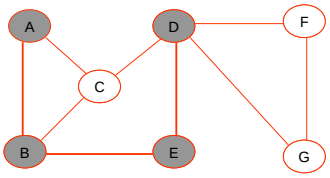
Pilha

D

E

B

A






## Busca em Profundidade

Pilha

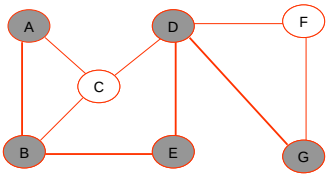
G

D

E

B

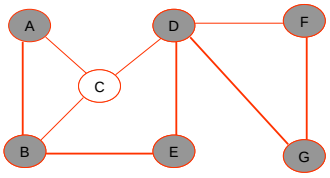
A



## Busca em Profundidade

Pilha

F
G
D
E
B
A

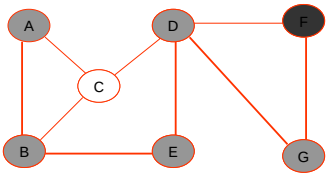





## Busca em Profundidade

Pilha

G
D
E
B
A

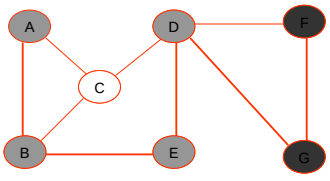





## Busca em Profundidade

Pilha

D
E
B
A

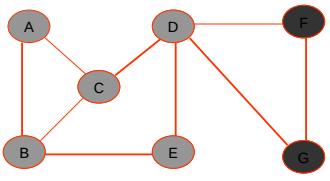





## Busca em Profundidade

Pilha

C
D
E
B
A

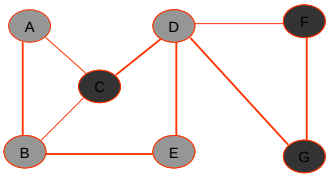





## Busca em Profundidade

Pilha

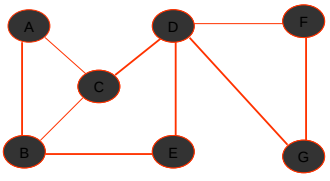
D
E
B
A






## Busca em Profundidade

Pilha

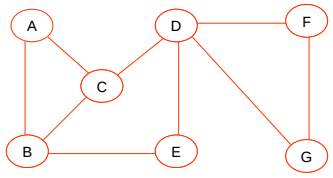


## Busca em Largura

- Outra forma de percorrer um grafo é através da busca em largura (também chamada de busca por nível)
- Esse tipo de busca visita primeiro todos os nós próximos da raiz da busca antes de visitar os mais distantes
- Pode ser implementada com ajuda de uma fila



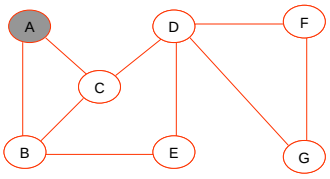
## Busca em Largura




**Fila**



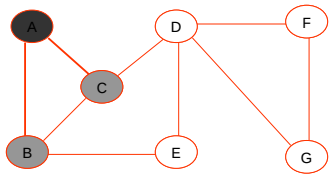
## Busca em Largura




**Fila** A



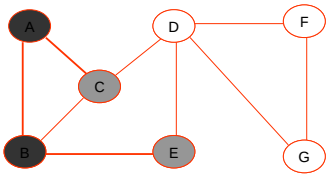
## Busca em Largura




**Fila** B C



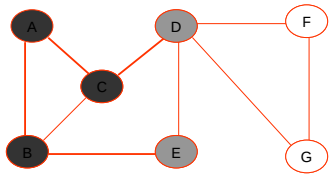
## Busca em Largura




**Fila** C E

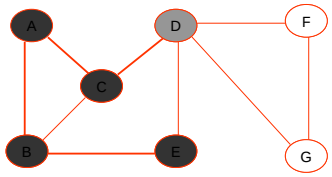


## Busca em Largura




**Fila** E D

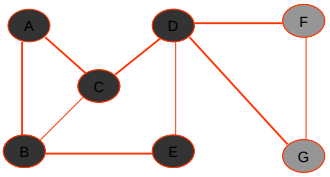
## Busca em Largura




**Fila** D



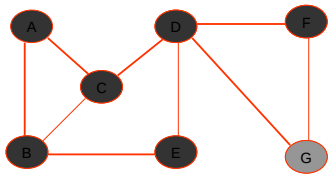
## Busca em Largura




**Fila** F G



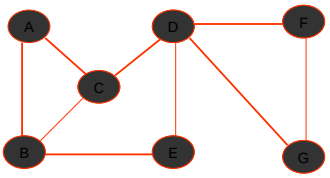
## Busca em Largura




**Fila** G



## Busca em Largura




**Fila**