

# Classificação e Pesquisa de Dados

Aulas 16 e 17  
Introdução a Árvores de Pesquisa e  
Árvores Binárias de Pesquisa

UFRGS

INF01124

1

## Conceitos

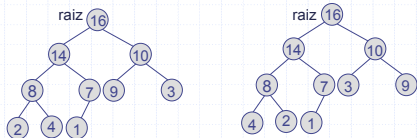
- Um grafo não orientado é um par  $(V, E)$ , onde  $V$  é um conjunto finito de vértices e  $E$  é um conjunto de arestas
- Cada aresta  $e \in E$  é um par não ordenado de vértices  $(u, v)$ , com  $u, v \in V$
- Um grafo não orientado é conexo se cada par de vértices é conectado por meio de um caminho
- Exemplos de grafos conexo e não conexo



2

## Conceitos (Cont.)

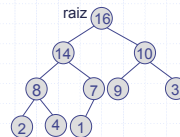
- Uma árvore livre é grafo não orientado, conexo e acíclico
- Uma árvore com raiz (*rooted tree*) é uma árvore livre em que um de seus vértices é designado como raiz da árvore
- Uma árvore ordenada é uma árvore com raiz na qual os filhos de cada um dos vértices (ou nodos) encontram-se ordenados (posicionalmente)
- Exemplo: Árvores iguais se não consideradas como árvores ordenadas



3

## Árvores Binárias

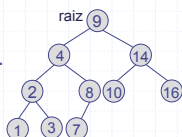
- Uma árvore binária é uma estrutura definida sobre um conjunto finito de nodos que:
  - Não contém nenhum nodo (árvore vazia), ou
  - Constitui-se de três conjuntos disjuntos de nodos: um nodo raiz, uma sub-árvore esquerda e uma sub-árvore direita



4

## Árvores Binárias de Pesquisa (ABP)

- Uma árvore binária de pesquisa (ou árvore binária de busca) obedece à seguinte **propriedade**:
  - Seja  $x$  um nodo de uma árvore binária de pesquisa. Se  $y$  é um nodo pertencente à sub-árvore esquerda de  $x$ , então  $\text{chave}[y] \leq \text{chave}[x]$ . Se  $y$  é um nodo pertencente à sub-árvore direita de  $x$ , então  $\text{chave}[y] > \text{chave}[x]$ .
- Além do valor de chave, armazenam os endereços do filho da esquerda, do filho da direita e do nodo pai
- O tempo para realização de operações básicas (consulta, inserção e exclusão de nodos) tem custo proporcional à altura da árvore.



5

## Caminhamento Central

- A propriedade da árvore binária de busca permite listar os valores de todos os nodos da árvore utilizando um caminhamento central

```
Proc caminhamento_central ( x );  
    { x : raiz da (sub-)árvore }  
begin  
    if x ≠ nil then  
        begin  
            caminhamento_central(left[x]);  
            print(chave[x]);  
            caminhamento_central(right[x]);  
        end  
    end;  
end;
```

Custo  $\Theta(n)$ : a partir do nodo raiz, o procedimento é chamado duas vezes para cada nodo da árvore.

6

## Exercícios

- ◆ Escreva algoritmos para realizar caminhamento em pré-ordem e em pós-ordem.

7

## Pesquisa em ABPs

- ◆ Versão recursiva:

```
Proc pesquisa_recursiva_arvore ( x, k );
{ x : raiz da (sub-)árvore }
{ k : valor de chave a ser pesquisado }
```

```
begin
  if x = nil or k = chave[x] then
    return (x);
  if k < chave[x] then
    return (pesquisa_recursiva_arvore (left[x], k));
  else
    return (pesquisa_recursiva_arvore (right[x], k));
end;
```

Custo  $O(h)$ , onde  $h$  é a altura da árvore.

8

## Pesquisa em ABPs

- ◆ Versão iterativa:

```
Proc pesquisa_iterativa_arvore ( x, k );
{ x : raiz da (sub-)árvore }
{ k : valor de chave a ser pesquisado }
```

```
begin
  while ( x ≠ nil and k ≠ chave[x] ) do
    begin
      if k < chave[x] then
        x ← left[x];
      else
        x ← right[x];
      end
    return (x);
  end;
```

Custo  $O(h)$ , onde  $h$  é a altura da árvore.

9

## Máximo e Mínimo

- ◆ O valor mínimo é encontrado seguindo-se sempre o ponteiro para o filho da esquerda, começando na raiz
- ◆ O valor máximo é encontrado seguindo-se sempre o ponteiro para o filho da direita, começando na raiz
- ◆ Em ambos os casos, o custo é  $O(h)$

```
Proc minimo_da_arvore ( x );
{ x : raiz da (sub-)árvore }
begin
  y ← nil;
  while ( x ≠ nil ) do
    y ← x;
    x ← left[x];
  return (y);
end;
```

```
Proc maximo_da_arvore ( x );
{ x : raiz da (sub-)árvore }
begin
  y ← nil;
  while ( x ≠ nil ) do
    y ← x;
    x ← right[x];
  return (y);
end;
```

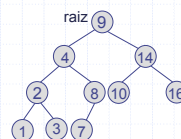
## Sucessor e Predecessor

- ◆ Caso as chaves sejam todas distintas:
  - O sucessor de um nodo  $x$  é o nodo  $y$ , tal que  $\text{chave}[y]$  é o menor valor maior que  $\text{chave}[x]$
  - O predecessor de um nodo  $x$  é o nodo  $y$ , tal que  $\text{chave}[y]$  é o maior valor menor que  $\text{chave}[x]$
- ◆ É possível determinar o sucessor e o predecessor de um nodo sem realizar comparações entre valores de chaves

11

## Sucessor na Árvore

- ◆ Para o sucessor de um nodo  $x$ , existem dois casos a serem tratados:
  - 1) Se a sub-árvore da direita não é vazia, o sucessor é o mínimo desta sub-árvore
  - 2) Do contrário, caso o sucessor exista, este será o ancestral mais próximo de  $x$ , tal que seu filho da esquerda também seja um ancestral de  $x$  ( $x$  é dito um ancestral não-próprio dele mesmo)
- ◆ Exercício: Identifique os sucessores dos nodos com valores de chave 4 e 8.



12

## Sucessor na Árvore

**Proc** sucessor\_arvore ( x );  
 { x : raiz da (sub-)árvore }

```
begin
  if right[x] ≠ nil then
    return minimo_da_arvore (right[x]);
  y ← parent[x];
  while (y ≠ nil and x = right[y]) do
    begin
      x ← y;
      y ← parent[y];
    end
  return (y)
end;
```

**Caso (1):** Trivial

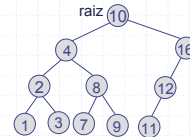
**Caso (2):** Suba na árvore, a partir do nodo x, até encontrar um nodo k que constitua a raiz de uma sub-árvore esquerda (i.e.  $k = \text{left}[\text{parent}[k]]$ ). Neste caso, o sucessor de x é o pai de k.

O custo do procedimento é  $O(h)$ , onde  $h$  é a altura da árvore

13

## Exercícios

- Verifique que o custo do procedimento `sucessor_arvore` é  $O(h)$ , encontrando os sucessores para os nodos com valores 9 e 10 na árvore abaixo



- Implemente o procedimento "predecessor\_arvore". Ele é simétrico ao procedimento "sucessor\_arvore" e também apresenta custo  $O(h)$

14

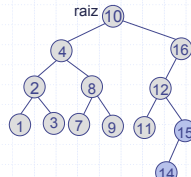
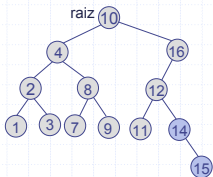
## Inserção e Remoção de Nodos

- Estas operações devem respeitar a propriedade das árvores binárias de pesquisa

- Inserção:** identifica a posição apropriada e insere

- A ordem em que os valores são inseridos é relevante

- Exemplo:** Inserir os nodos 14 e 15 na árvore abaixo



15

## Inserção de um Nodo

**Proc** insere\_nodo ( T, z );  
 { T : árvore que receberá um novo nodo }  
 { nodo a ser inserido }

```
begin
  y ← nil;
  x ← root[T];
  while (x ≠ nil) do
    begin
      y ← x;
      if (chave[z] < chave[x])
        then x ← left[x];
      else x ← right[x];
    end
  parent[z] ← y;
  if (y = nil)
    then root[T] ← z;
  else if (chave[z] < chave[y])
    then left[y] ← z;
  else right[y] ← z;
end;
```

*/\* encontra a posição de inserção \*/*

*O custo do procedimento é  $O(h)$ , determinado pelo laço while – um novo nodo é sempre inserido como filho de um nodo folha*

*/\* atualiza o pai de z \*/*

*/\* árvore estava vazia \*/*

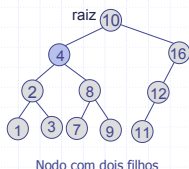
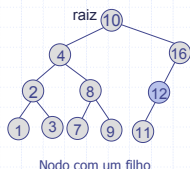
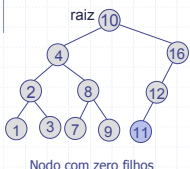
*/\* insere na sub-árvore esquerda \*/*

*/\* insere na sub-árvore direita \*/*

16

## Remoção de Nodos

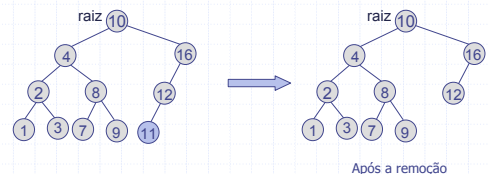
- Três casos distintos a serem tratados: nodo a ser removido tem zero, um ou dois filhos



17

## Remoção de Nodos

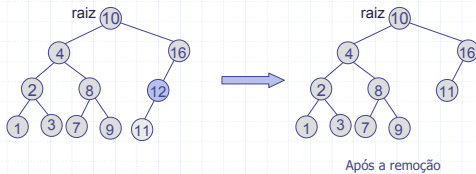
- Caso 1:** nodo a ser removido tem zero filhos
  - Simplesmente remove o nodo



18

## Remoção de Nodos

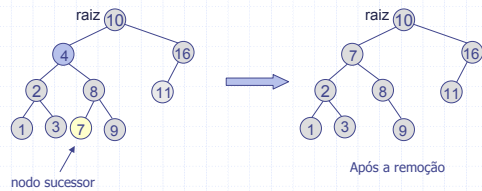
- ◆ Caso 2: nodo a ser removido tem um filho
  - Substitui o nodo por seu filho



19

## Remoção de Nodos

- ◆ Caso 3: nodo a ser removido tem dois filhos
  - Substitui o nodo por seu sucessor



Perguntas: Como podemos garantir que sempre existirá um sucessor?  
Poderíamos ter feito a substituição pelo nodo antecessor?

20

## Remoção de um Nodo

O custo do procedimento é  $O(h)$ ,  
determinado pela chamada ao  
procedimento `sucessor_arvore`

```

Proc remove_nodo ( T, z );
{ T : árvore que receberá um novo nodo; z : nodo a ser removido }
begin
  if (left[z] = nil or right[z] = nil)
  then y ← z;
  else y ← sucessor_arvore(z);
  if (left[y] ≠ nil)
  then x ← left[y];
  else x ← right[y];
  if (x ≠ nil)
  then parent[x] ← parent[y];
  if (parent[y] = nil)
  then root[T] ← x;
  else if (y = left[parent[y]])
  then left[parent[y]] ← x;
  else right[parent[y]] ← x;
  if (y ≠ z)
  then chave[z] ← chave[y];
  return(y);
end;
```

21

## Pergunta

No caso do nodo a ser removido conter dois nodos filhos, aquele deve ser substituído pelo seu sucessor (antecessor). Note que o algoritmo **remove\_nodo** não contém chamadas recursivas. Como podemos garantir que o sucessor (antecessor) do nodo a ser removido também não contém dois filhos?

22