Aula 16: Árvores binárias de busca

David Déharbe
Programa de Pós-graduação em Sistemas e Computação
Universidade Federal do Rio Grande do Norte
Centro de Ciências Exatas e da Terra
Departamento de Informática e Matemática Aplicada

Download me from `http://DavidDeharbe.github.io`.

Plano da aula

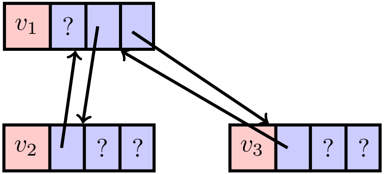
Árvores binárias

Árvores binárias de busca

Operações de consulta

Operações de mutação

Referência: Cormen et al. Capítulo 13.



- uma árvore binária é formada por células (nós) com atributos
  - $x.key$ (chave do dado armazenado),
  - $x.left$ (sub-árvore esquerda),
  - $x.right$ (sub-árvore direita),
  - $x.up$ (nó pai),
- a raiz é um nó destacado da árvore.
- NIL representa á árvore binária vazia (nenhuma célula)

Definição

Nós descendentes a partir de x:

$$\begin{aligned} nodes(x) &= \{x\} \cup nodes(x.left) \cup nodes(x.right) \\ nodes(\textsc{Nil}) &= \emptyset \end{aligned}$$

Definição

Chaves armazenados em uma árvore enraizada em x:

$$\begin{aligned} values(x) &= \{\{x.key \mid x \in nodes(x)\}\} \\ values(\textsc{Nil}) &= \emptyset \end{aligned}$$

Definição

Altura da sub-árvore enraizada em x:

$$\alpha(x) = \begin{array}{l} 0 \text{ se } x = \textsc{Nil}, \\ 1 + \max\{\alpha(x.left), \alpha(y.right)\} \text{ senão} \end{array}$$

- célula raiz: $\mathit{root}.\mathit{up} = \textsc{Nil}$
- ausência de ciclos
  $x \notin \mathit{nodes}(x.\mathit{left})$
  $x \notin \mathit{nodes}(x.\mathit{right})$
  $x.\mathit{left} \cap x.\mathit{right} = \emptyset$
- $x.\mathit{left} \neq \textsc{Nil} \Rightarrow x = x.\mathit{left}.\mathit{up}$
  $x.\mathit{right} \neq \textsc{Nil} \Rightarrow x = x.\mathit{right}.\mathit{up}$
- O número de atributos *left* e *right* iguais a $\textsc{Nil}$ é o número de nós mais um.

Aplicação de árvore binária

- Qualquer árvore pode ser representada através de uma árvore binária
  - $n.left$: primeiro nó descendente
  - $n.right$: próximo nó descendente
  - $n.up$: nó ancestral
- Árvores binárias de busca

- Representa coleção de dados $values(root)$
- Um iterador sobre a coleção é uma referência a um nó da árvore.
- Operações
  - Inserção de um dado;
  - Remoção de um dado;
  - Busca na coleção de um dado com uma determinada chave
  - Maior elemento
  - Menor elemento
  - Elemento seguinte
  - Elemento anterior
- $h = \alpha(root)$: altura da árvore:
  Custo no pior caso: $\Theta(h)$ em média.

Árvores binárias de busca

Especificação

- representa a coleção $\mathit{values}(\mathit{root})$
- árvore binária
- com a seguinte propriedade de ordenação:

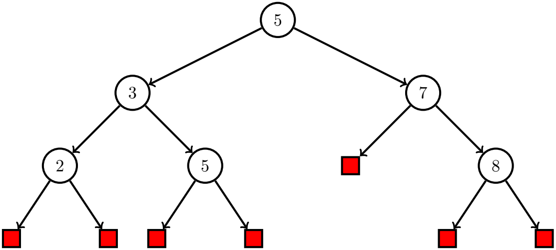
$$\begin{array}{rl} \forall x \cdot \forall y \cdot & y \in \mathit{nodes}(x.\mathit{left}) \Rightarrow x.\mathit{key} \geq y.\mathit{key} \quad \wedge \\ & y \in \mathit{nodes}(x.\mathit{right}) \Rightarrow x.\mathit{key} \leq y.\mathit{key} \end{array}$$

$$\begin{array}{rcll} \mathit{bst}(x) & \equiv & x = \text{NIL} \vee & (x.\mathit{key} \geq \max \mathit{values}(x.\mathit{left}) \quad \wedge \\ & & & x.\mathit{key} \leq \min \mathit{values}(x.\mathit{right}) \quad \wedge \\ & & & \mathit{bst}(x.\mathit{left}) \wedge \mathit{bst}(x.\mathit{right})) \end{array}$$

PPgSC

Coleção: $2, 3, 5, 5, 7, 8$

- Uma árvore binária de busca de altura 3

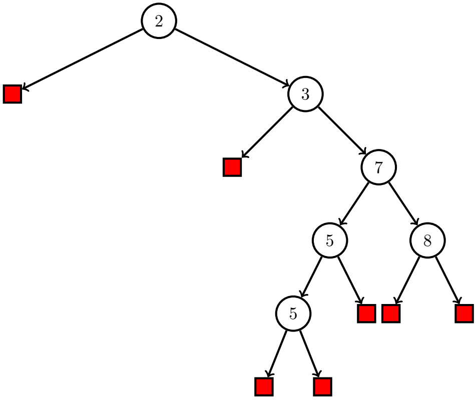


Árvores binárias de busca

Ilustração

Coleção: $2, 3, 5, 5, 7, 8$

- Uma árvore binária de busca de altura 5



- Processamento em ordem permite visitar os valores da coleção em ordem crescente de chaves.

IN-ORDER(x, f)
```
if x ≠ NIL
    IN-ORDER(x.left)
    f(x)
    IN-ORDER(x.right)
```

- Complexidade: $\Theta(n)$

Exercícios

1. Qual a altura máxima possível para uma árvore binária de busca representando uma coleção de n valores?
2. Seja $\textsc{Make-Node}(k, l, r, u)$ uma sub-rotina que constroi um nó x de árvore binária, tal que $x.key = k$, $x.left = l$, $x.right = r$, $x.up = u$.
   Projete um algoritmo utilize esta sub-rotina e ordenação em arranjo para construir uma árvore binária de busca a partir de uma coleção inicialmente em um arranjo. O algoritmo deverá ter complexidade $\Theta(n \lg n)$.
3. Projete um algoritmo não recursivo para processar os valores de uma árvore binária de busca em ordem, e em tempo $\Theta(n)$.

Operações de consulta

$\textsc{Search}(x, k)$

    // $bst(x)$
    if $x == \textsc{Nil}$ or $x.key == k$
        return x
    else
        if $k < x.key$
            return $\textsc{Search}(x.left, k)$
        else return $\textsc{Search}(x.right, k)$
    // $\textsc{Search}(x, k) = \textsc{Nil} \Leftrightarrow k \notin values(x)$
    // $\textsc{Search}(x, k) = y \neq \textsc{Nil} \Leftrightarrow y.key = k \land y \in nodes(x)$

Complexidade: $O(\alpha(root))$

Operações de consulta

$\textsc{Search}(x, k)$

$\quad$ // $bst(x)$
$\quad$ **if** $x == \textsc{Nil}$ or $x.key == k$
$\qquad$ **return** x
$\quad$ **else**
$\qquad$ **if** $k < x.key$
$\qquad\quad$ **return** $\textsc{Search}(x.left, k)$
$\qquad$ **else return** $\textsc{Search}(x.right, k)$
$\quad$ // $\textsc{Search}(x, k) = \textsc{Nil} \Leftrightarrow k \notin values(x)$
$\quad$ // $\textsc{Search}(x, k) = y \neq \textsc{Nil} \Leftrightarrow y.key = k \wedge y \in nodes(x)$

Complexidade: $O(\alpha(root))$
Exercício: escrever um algoritmo não recursivo.

Operações de consulta

$\text{MINIMUM}(x)$

// $bst(x) \wedge x \neq \text{NIL}$
if $x.left == \text{NIL}$
  return x
else return $\text{MINIMUM}(x.left)$
// $\text{MINIMUM}(x) = \min values(x)$

$\text{MAXIMUM}(x)$

// $bst(x) \wedge x \neq \text{NIL}$
if $x.right == \text{NIL}$
  return x
else return $\text{MAXIMUM}(x.right)$
// $\text{MAXIMUM}(x) = \max values(x)$

Complexidade: $O(\alpha(root))$

Operações de consulta

```
MINIMUM(x)
    // bst(x) ∧ x ≠ NIL
    if x.left == NIL
        return x
    else return MINIMUM(x.left)
    // MINIMUM(x) = min values(x)
```

```
MAXIMUM(x)
    // bst(x) ∧ x ≠ NIL
    if x.right == NIL
        return x
    else return MAXIMUM(x.right)
    // MAXIMUM(x) = max values(x)
```

Complexidade: $O(\alpha(root))$
Exercício: escrever algoritmos não recursivas.

Operações de consulta

SUCCESSOR(x)

// $x \neq$ NIL $\wedge$ $bst(x)$

if $x.right \neq$ NIL

  return MINIMUM($x.right$)

$y = x.up$

while $y \neq$ NIL and $x == y.right$

  $x = y$

  $y = x.up$

return y

Complexidade: $O(\alpha(root))$

Operações de consulta

```
SUCCESSOR(x)
    // x ≠ NIL ∧ bst(x)
    if x.right ≠ NIL
        return MINIMUM(x.right)
    y = x.up
    while y ≠ NIL and x == y.right
        x = y
        y = x.up
    return y
```

Complexidade: $O(\alpha(root))$
Exercício: escrever o algoritmo que encontra o nó predecessor (se existir).

Inserção

Operações de mutação

$\textsc{Insert}(A, k)$

$\quad$ // $bst(A.root)$

$\quad A.root = \textsc{Insert-Aux}(k, A.root, \textsc{Nil})$

$\quad$ // $bst(A'.root) \wedge values(A'.root) = values(A.root) \cup \{\{k\}\}$

$\textsc{Insert-Aux}(k, x, p)$

$\quad$ // $bst(x) \wedge$

$\quad$ // $(x = Nil \vee x.up = \textsc{Nil} \vee$

$\quad$ // $x.up = p \wedge (p.key \leq k \wedge p.right = x \vee p.key \geq k \wedge p.left = x))$

$\quad$ **if** $x == \textsc{Nil}$

$\qquad$ **return** $\textsc{Make-Node}(k, \textsc{Nil}, \textsc{Nil}, p)$

$\quad$ **else**

$\qquad$ **if** $k < x.key$

$\qquad\quad x.left = \textsc{Insert-Aux}(p, x.left, x)$

$\qquad$ **else** $x.right = \textsc{Insert-Aux}(p, x.right, x)$

Inserção

- $O(h)$ para encontrar a posição de inserção
- $\Theta(1)$ para criar o nó
- $O(h)$ para atualizar as referências para as sub-árvores

- Objetivo: remover a chave k da coleção representada
- Preâmbulo: buscar o nó, digamos z, com a chave k
- Remover o nó z.

- Objetivo: remover a chave k da coleção representada
- Preâmbulo: buscar o nó, digamos z, com a chave k
- Remover o nó z.

  z é uma folha:

  - eliminar z, e
  - modificar o nó $z.up$ (se existir) para ter NIL como sub-árvore, ao invés de z;

- Objetivo: remover a chave k da coleção representada
- Preâmbulo: buscar o nó, digamos z, com a chave k
- Remover o nó z.

  z tem apenas uma sub-árvore:

  - eliminar z
  - modificar o nó $z.up$ (se existir) para ter a sub-árvore de z como sub-árvore, ao invés de z.

- Objetivo: remover a chave k da coleção representada
- Preâmbulo: buscar o nó, digamos z, com a chave k
- Remover o nó z.
  z tem duas sub-árvores:
  - procurar o nó y, elemento sucessor de z na árvore
    necessariamente existe y (por quê?)
  - copiar $y.key$ em $z.key$
  - eliminar o nó y.
    necessariamente y tem pelo menos uma sub-árvore vazia. (por quê?)

Remoção

Operações de mutação

```
REMOVE-NODE(A, k)
z = SEARCH(A.root, k) // z: nó com o valor a eliminar
if z == NIL
    return
if z.left == NIL or z.right == NIL
    y = z
else y = SUCCESSOR(z) // y: nó a eliminar
if y.left ≠ NIL
    x = y.left
else x = y.right // x: sub-árvore não vazia de y ou NIL
if x ≠ NIL
    x.up = y.up
if y.up == NIL
    A.root = x
else if y == y.up.left
        y.up.left = x
    else y.up.right = x // y foi desconectado da árvore
if y ≠ z
    z.key = y.key
FREE-NODE(y)
```

- $O(h)$ para encontrar o nó a remover
- $O(h)$ para encontrar o nó sucessor
- $Theta(1)$ para realizar as modificações nas referências

Exercícios

1. O algoritmo de inserção apresentado realiza $O(h)$ atualizações de sub-árvore.
   Modificar o algoritmo para realizar $\Theta(1)$ atualizações de sub-árvore
2. Projetar um algoritmo de inserção não recursivo.
3. Verdadeiro ou falso?
   Remover a chave k_1, e então remover a chave k_2 deixa a árvore no mesmo estado de que remover primeiro k_2 e então k_1.
   Justifique.
4. Um algoritmo de ordenação de um arranjo consiste em
   4.1 inserir sucessivamente os valores do arranjo em uma árvore binária de busca
   4.2 percorrer a árvore em ordem, inserindo os valores no arranjo.

   Qual a complexidade deste algoritmo?

Conclusões

- Operações: $O(\alpha(A.root))$
- Em geral:
  - $\alpha(A.root) \in O(|values(A.root)|)$
  - $\alpha(A.root) \in \Omega(\lg|values(A.root)|)$
- Podemos fazer melhor?

PPgSC

Conclusões

- Operações: $O(\alpha(A.root))$
- Em geral:
  - $\alpha(A.root) \in O(|values(A.root)|)$
  - $\alpha(A.root) \in \Omega(\lg |values(A.root)|)$
- Podemos fazer melhor?
  - Sim!
    $\implies \alpha(A.root) \in \Theta(\lg |values(A.root)|)$
  - Árvores AVL, árvores rubro-negras.

PPgSC