Aula 09: 26/04/2017

Tópico

- mais árvores binárias de busca (BSTs): desempenho, outras operações

Referências

- Árvores binárias de busca (PF),
- BSTs: operações adicionais (PF),
- Binary Search Trees (S&W),
- slides (S&W)

Vídeo

Binary Search Trees (S&W)

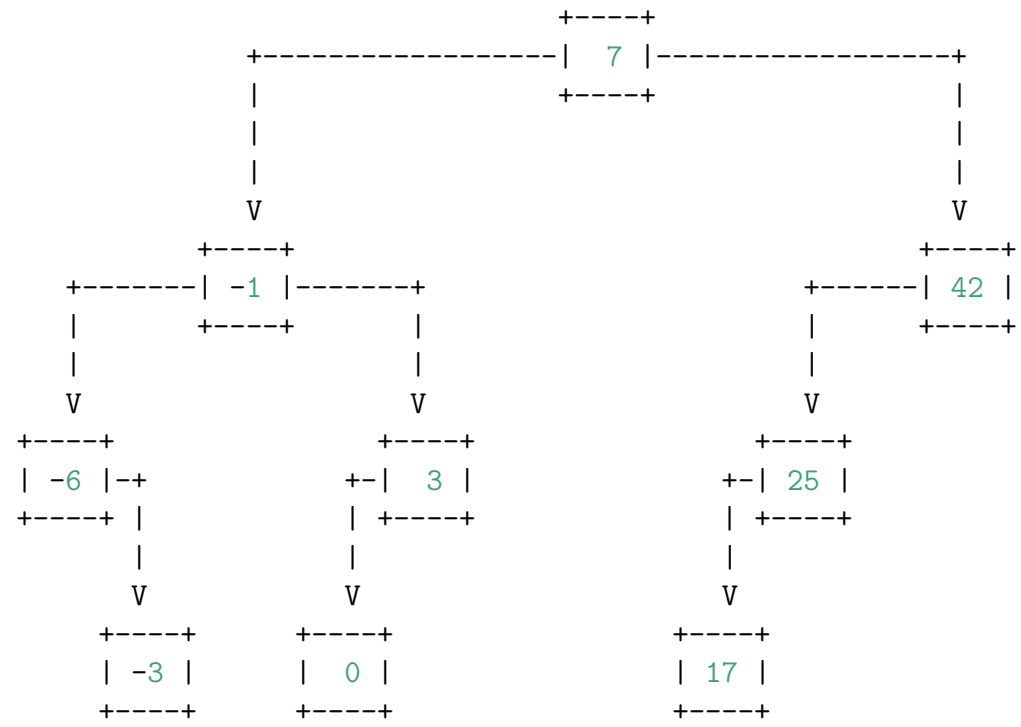
Resultados experimentais

```
linked-list> java Driver ../testes/les-miserables.txt
ST criada em 92.785 segundos
ST contém 26764 itens

sorted-linked-list> java Driver ../testes/les-miserables.txt
ST criada em 84.192 segundos
ST contém 26764 itens

array> java Driver ../testes/les-miserables.txt
ST criada em 60.084 segundos
ST contém 26764 itens

binary-search> java Driver ../testes/les-miserables.txt
ST criada em 1.283 segundos
ST contém 26764 itens

binary-search-tree> java Driver ../testes/les-miserables.txt
ST criada em 0.687 segundos
ST contém 26764 itens
```

Árvore binária de busca

Uma **árvore binária de busca** (=*binary search tree*) é um tipo especial de BT: para cada nó `x`, todos os nós na subárvore esquerda de `x` têm chave menor que `x.key` e todos os nós na subárvore direita de `x` têm chave maior que `x.key`.

As chaves de uma BST precisam ser comparáveis.



Veja a animação da busca e inserção em uma BST.

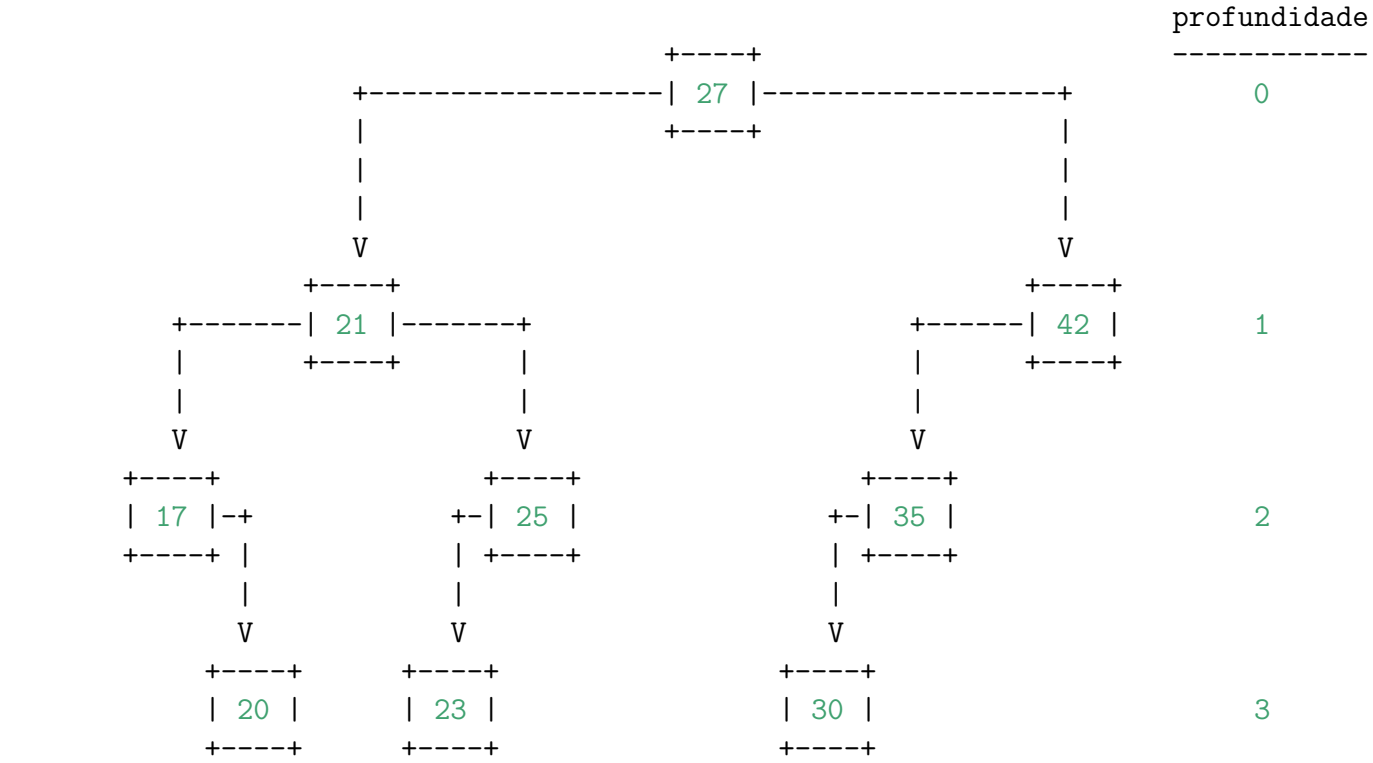
BST.java

```
public class BST <Key extends Comparable<Key>, Value> {

   private Node root;

   private class Node {
      private Key key;
      private Value val;
      private Node left, right;
      public Node(Key key, Value val) {
         this.key = key;
         this.val = val;
      }
   }

   public Value get(Key key) {
      return get(root, key);
   }

   private Value get(Node x, Key key) {
      // Considera apenas a subárvore que tem raiz x
      if (x == null) return null;
      int cmp = key.compareTo(x.key);
      if      (cmp < 0) return get(x.left, key);
      else if (cmp > 0) return get(x.right, key);
      else return x.val;
```

```
   }

   public void put(Key key, Value val) {
      root = put(root, key, val);
   }

   private Node put(Node x, Key key, Value val) {
      // Considera apenas a subárvore com raiz x
      // Retorna a raiz da nova subárvore
      if (x == null) return new Node(key, val);
      int cmp = key.compareTo(x.key);
      if      (cmp < 0) x.left = put(x.left, key, val);
      else if (cmp > 0) x.right = put(x.right, key, val);
      else x.val = val;
      return x;
   }
}
```

Simulação

Ordem de inserção: 27 21 25 42 17 20 23 35 30



Desempenho esperado (= médio)

A ideia de BST é realizarmos uma espécie de "busca binária dinâmica". Gostaríamos de combinar a eficiência da busca por chaves ordenados (`get()`) com a eficiência da inserção (`put()` e remoção (`delete()`).

Qual é o número de chaves examinadas/comparadas em uma busca com sucesso?

Busca com sucesso = busca em que a chave procurada está na BST.

Toda operação de busca ou inserção visita $1 + p$ nós, sendo p a profundidade do último nó visitado.

Na BST acima:

- a busca por 27 requer 1+0 comparações
- a busca por 21 requer 1+1 comparações
- a busca por 42 requer 1+1 comparações
- a busca por 17 requer 1+2 comparações
- a busca por 25 requer 1+2 comparações
- a busca por 35 requer 1+2 comparações
- a busca por 20 requer 1+3 comparações
- a busca por 23 requer 1+3 comparações
- a busca por 30 requer 1+3 comparações

Assim, o número médio de comparações necessários para uma busca com sucesso na BST acima é

$$(1 + 2 + 2 + 3 + 3 + 3 + 4 + 4 + 4)/9 \; 2.8$$

O **comprimento interno** (= *internal path length*) de uma BT é a soma das profundidades dos seus nós, ou seja, a soma dos comprimentos de todos os caminhos que levam da raiz até um nó. O comprimento interno da árvore mostrada anteriormente é

$$0 + 1 + 1 + 2 + 2 + 2 + 3 + 3 + 3$$

Uma **BST aleatória** é uma BST que se obtém inserindo `n` chaves distintas em ordem aleatória numa árvore inicialmente vazia.

Qual é o número esperado de comparações em uma busca com sucesso em uma BST aleatória com `n` chaves?

Fato. Buscas com sucesso numa BST aleatória com n chaves requer cerca de $2 \lg n$ comparações na média.

Demonstração: O número de comparações para encontrados uma dada chave é 1 mais a profundidade do nó que contém a chave. Se somarmos todas as profundidades dos nós de uma árvore obtemos o comprimento interno C da árvore. O número médio de comparações para uma busca com sucesso nessa árvore é portanto $1 + C/n$.

Seja C_n o comprimento interno de uma BST aleatória com n chaves. Temos que $C_0 = 0$, $C_1 = 0$ e

$$C_n = ((C_0 + C_{n-1} + n - 1) + (C_1 + C_{n-2} + n - 1) + \cdots + (C_{n-1} + C_0 + n - 1))/n.$$

Cada $n - 1$ é devido a contribuição da raiz às profundidades das duas subárvores.

Reescrevendo a terceira igualdade obtemos

$$nC_n = 2(C_0 + C_1 + \ldots + C_{n-1}) + n(n - 1).$$

Agora vem um truque meio manjado. Primeiro, reescrevemos a última igualdade temos que

$$(n - 1)C_{n-1} = 2(C_0 + C_1 + \ldots + C_{n-2}) + (n - 1)(n - 2).$$

Segundo, fazemos

$$nC_n - (n-1)C_{n-1} = 2C_{n-1} + 2(n-1).$$

Rearranjando os termos e dividindo por $n(n+1) obtemos ou seja

$$\frac{C_n}{n+1} = \frac{C_{n-1}}{n} + \frac{2(n-1)}{n(n+1)} < \frac{C_{n-1}}{n} + 2\frac{1}{n}.$$

Essa é uma recorrência que podemos "desenrolar" até o caso base

$$\frac{C_n}{n+1} < \frac{C_1}{2} + 2\left(\frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{n-1} + \frac{1}{n}\right).$$

Assim,

$$C_n < 2(n+1)\left(\frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{n-1} + \frac{1}{n}\right).$$

Sabemos que

$$\sum_2^n \frac{1}{n} < \int_1^n \frac{1}{x}\,\mathrm{d}x = \ln n - \ln 1 = \ln n.$$

Logo,

$$C_n < 2(n+1)\ln n = 2(n+1)\frac{\lg n}{\lg e} < 1.39(n+1)\lg n \quad \left(\frac{2}{\lg e} = 1.3862\ldots\right)$$

Portanto, concluímos que na média o número de comparações em uma BST aleatória é

$$1 + \frac{1.39(n+1)\lg n}{n} \sim 2\lg n.$$

Qual a altura esperada de uma BST aleatória?

Resposta, aproximadamente $3\lg n$.

Conclusão: o número esperado de nós visitados durante uma busca em uma BST aleatória não passa de $3\lg n$

min()

```
   // Retorna a menor chave da tabela de símbolos
   // ou null se a tabela estiver vazia.

   public Key min() {
      if (root == null) return null;
      return min(root);
   }

   // Retorna a menor chave da subárvore cuja raiz é x.
   // (Supõe que x não é null.)
   private Key min(Node x) {
      if (x.left == null) return x.key;
      return min(x.left);
   }
```

`max()` é semelhante

floor()

`floor(k)` é a maior chave da BST que é menor que ou igual a k.

```
   // Retorna null se key não tem piso nesta BST.
   public Key floor(Key key) {
      Node x = floor(root, key);
      if (x == null) return null;
      return x.key;
   }

   // Retorna o nó que contém o piso de key
   // na subárvore com raiz x.
   // Retorna null se esse piso não existe.
   private Node floor(Node x, Key key) {
      if (x == null) return null;
      int cmp = key.compareTo(x.key);
      if (cmp == 0) return x;
      if (cmp < 0) return floor(x.left, key);
      Node t = floor(x.right, key);
      if (t != null) return t;
      return x;
   }
```

`ceil()` é semelhante

deleteMin()

Como aquecimento, comecemos com `deleteMin()`.

```
// Remove o nó que tem a menor chave.
public void deleteMin() {
   if (root == null) return;
   root = deleteMin(root);
}

// Remove o nó que tem a menor chave
// na subárvore (não vazia) cuja raiz é x
// e retorna a raiz da subárvore resultante.
private Node deleteMin(Node x) {
   if (x.left == null) return x.right;
   x.left = deleteMin(x.left);
   return x;
}
```

Lição: remoção quando não tem um filho esquerdo é fácil.

Bem, na verdade, remoção quando não tem algum filho é fácil.

`deleteMax()` é semelhante.

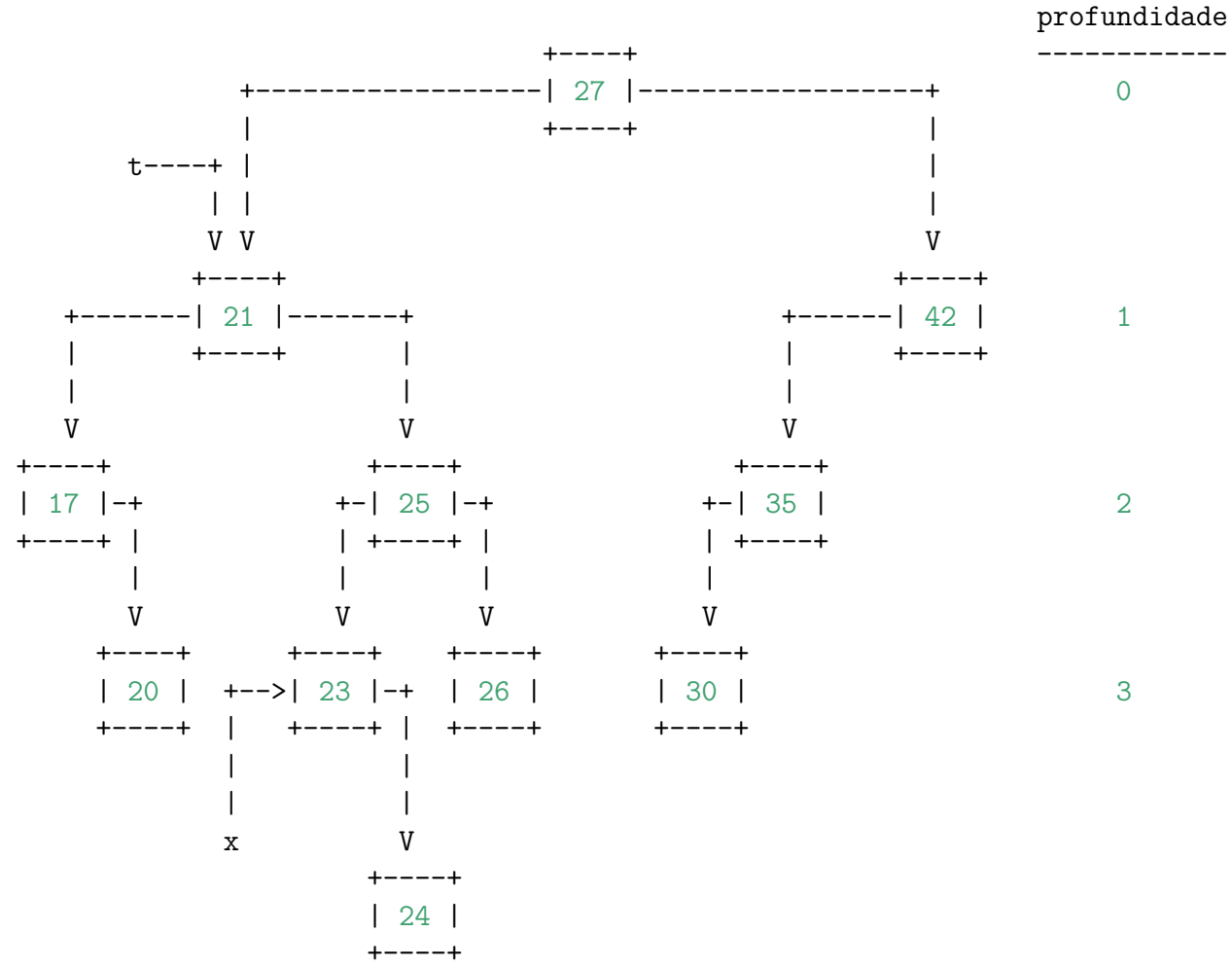
delete()

Solução proposta por pot T. Hibbard em 1962. Faz uso de `min()` e `deleteMin()`.

```
// Remove o nó que contém a chave key.
// Se nenhum nó contém key, não faz nada.
public void delete(Key key) {
   root = delete(root, key);
}

// Remove da subárvore cuja raiz é x
// o nó que contém a chave key
// e retorna a raiz da subárvore resultante.
// (Se key não está na subárvore,
// não faz nada e retorna x.)
private Node delete(Node x, Key key) {
   if (x == null) return null;
   int cmp = key.compareTo(x.key);
   if (cmp < 0) x.left = delete(x.left, key);
   else if (cmp > 0) x.right = delete(x.right, key);
   else {
      // não tem um filho, fazemos como em deleteMin()
      if (x.right == null) return x.left;
      if (x.left  == null) return x.right;

      // nesse ponto sabemos que x tem ambos os filhos
      Node t = x;
      x = min(t.right);  // note que x.left == null
      x.right = deleteMin(t.right); // hmm. precisa atualizar este antes
      x.left = t.left;
   }
   return x;
```

Simulação

`delete(21)`:



Passos:

1. encontre nó com 21
2. chame esse nó de t (referencia/apelido)
3. encontre o menor nó na subárvore direita (= min(t.right))
4. chame esse nó de x
5. remova x da árvore e ajuste x.right (x.right = delmin(t.right))
6. ajustes as referências x.left (x.letf = t.left)
7. retorne x e note que o nó referencia por t está pronto para o coletor de lixo assim que esse nível de chamada da função retornar e t deixar de existir.

Resumo

	pior caso		caso médio			
	get()	put()	get()	put()	ordenada	interface
SequentialSearchST	n	n	n	n	não	equals()
BinarySearchST	lg n	n	lg n	n	sim	compareTo()
BST	n	n	lg n	lg n	sim	compareTo()