Aula 10: 04/04/2019

Tópicos

- Árvores binárias de busca (BSTs)

Leitura

Árvores binárias de busca (BSTs), Binary Search Trees (S&W), slides (S&W)

Vídeo

Binary Search Trees (S&W)

Resultados experimentais

```
array> java Driver ../data/les-miserables.txt
Criando a ST com as palavras do arquivo '../data/les-miserables.txt' ...
ST criada em 61.672 segundos
ST contém 26764 itens
Início da consulta interativa. Tecle ctrl+D encerrar

binary-search> java Driver ../data/les-miserables.txt
Criando a ST com as palavras do arquivo '../data/les-miserables.txt' ...
ST criada em 1.549 segundos
ST contém 26764 itens
Início da consulta interativa. Tecle ctrl+D encerrar

linked-list> java Driver ../data/les-miserables.txt
Criando a ST com as palavras do arquivo '../data/les-miserables.txt' ...
ST criada em 145.048 segundos
ST contém 26764 itens
Início da consulta interativa. Tecle ctrl+D encerrar
>>>

sorted-linked-list> java Driver ../data/les-miserables.txt
Criando a ST com as palavras do arquivo '../data/les-miserables.txt' ...
ST criada em 121.631 segundos
ST contém 26764 itens
Início da consulta interativa. Tecle ctrl+D encerrar

skip-list> java Driver ../data/les-miserables.txt
Criando a ST com as palavras do arquivo '../data/les-miserables.txt' ...
ST criada em 1.063 segundos
ST contém 26764 itens
Início da consulta interativa. Tecle ctrl+D encerrar
```

```
>>>

binary-search-tree> java Driver ../data/les-miserables.txt
Criando a ST com as palavras do arquivo '../data/les-miserables.txt' ...
ST criada em 0.725 segundos
ST contém 26764 itens
Início da consulta interativa. Tecle ctrl+D encerrar
>>>
```

Árvores binárias de busca (BSTs)

Árvores binárias de busca (BSTs) servem para implementar TSs ordenadas. BSTs busca binária e listas ligadas ordenadas. BST implementam ST ordenadas.

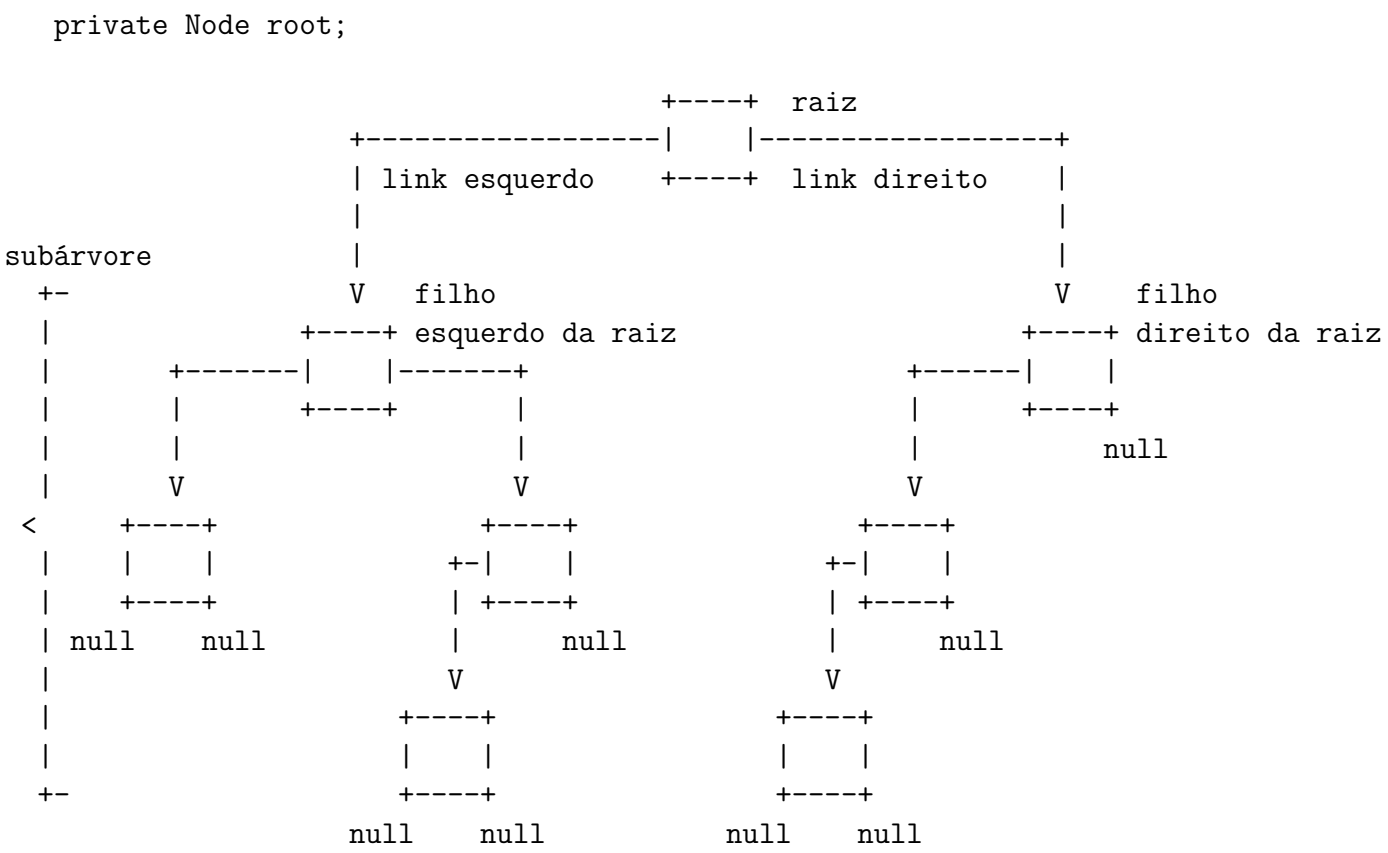
Árvores binárias

Em uma árvore binária cada nó tem no máximo dois filhos: um esquerdo e outro direito

```
+-----+-----+
|    item   |
|           |
+-----+-----+
|left |right|
|     |     |
+-----+-----+
```

```
private class Node {
   private Item  item;
   private Node left, right;
   public Node(Item item) {
      this.item = item;
   }
}
```

Anatomia de uma árvore binária



A **profundidade** (=*depth*) de um nó de uma BT é o número de links no caminho que vai da raiz até o nó.

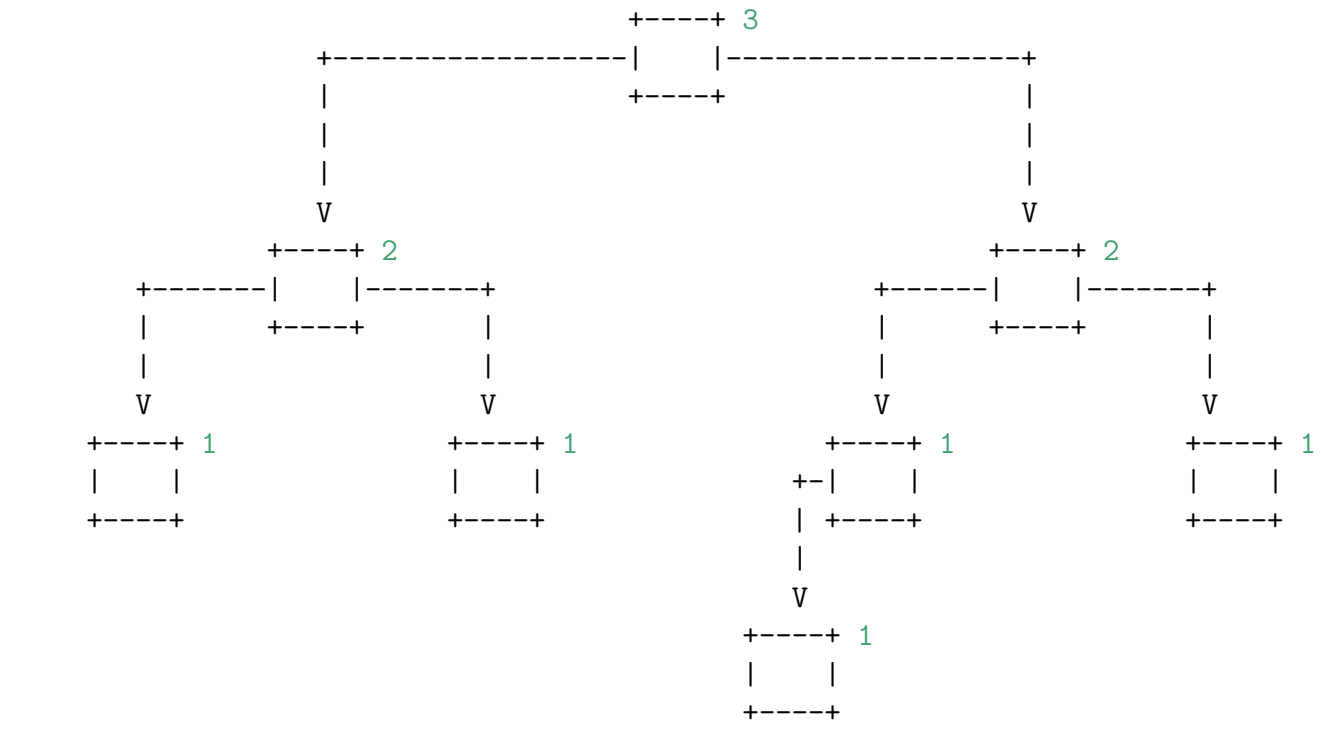
A **altura** (=*height*) de uma BT é o máximo das profundidades dos nós, ou seja, a profundidade do nó mais profundo.

```
private int altura(Node r) {
    if (r == null) return -1;
    int hLeft = altura(r.left);
    int hRight = altura(r.right);
    return Math.max(hLeft, hRight) + 1
}
```

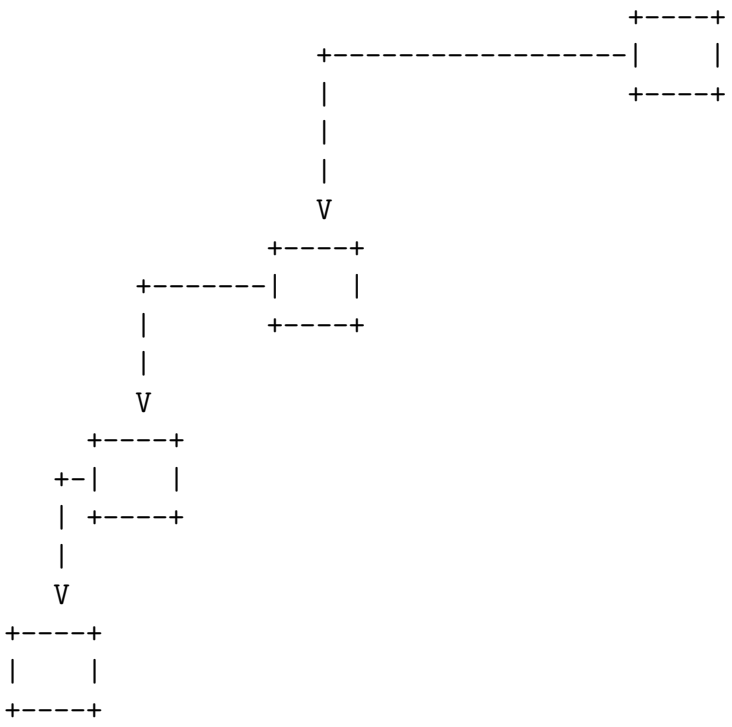
Uma BT com `n` nós, tem altura no máximo `n-1` e no mínimo $\lfloor \lg n \rfloor$.

Se a altura estiver perto de $\lg n$, a BT é **balanceada**.

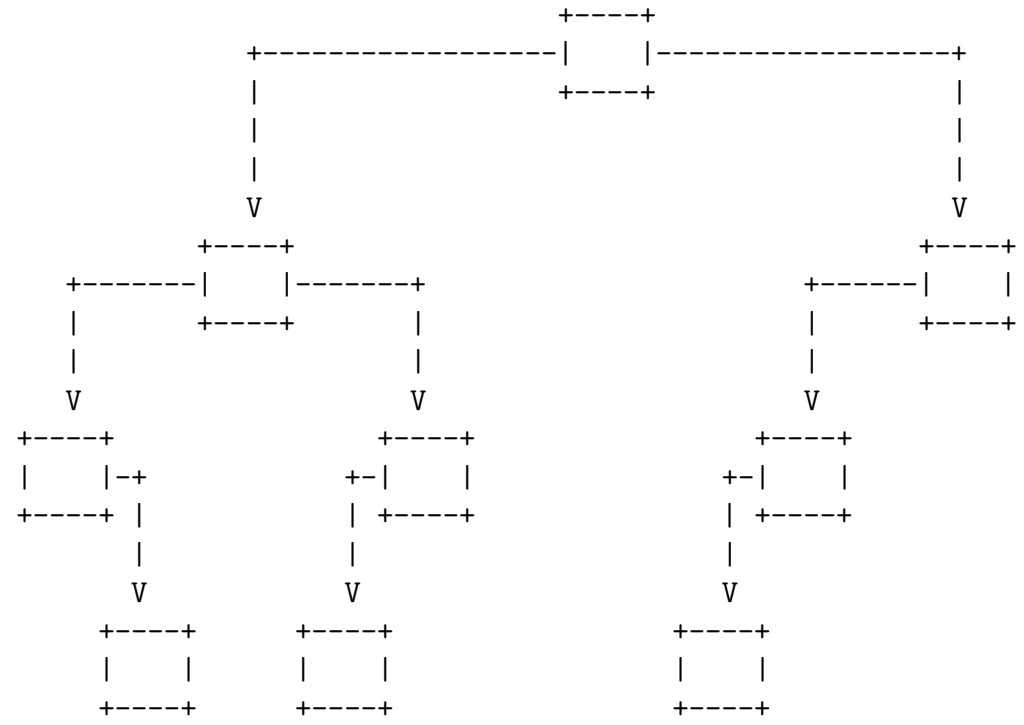
Árvore balanceada



Árvore de altura n



Árvores típica:

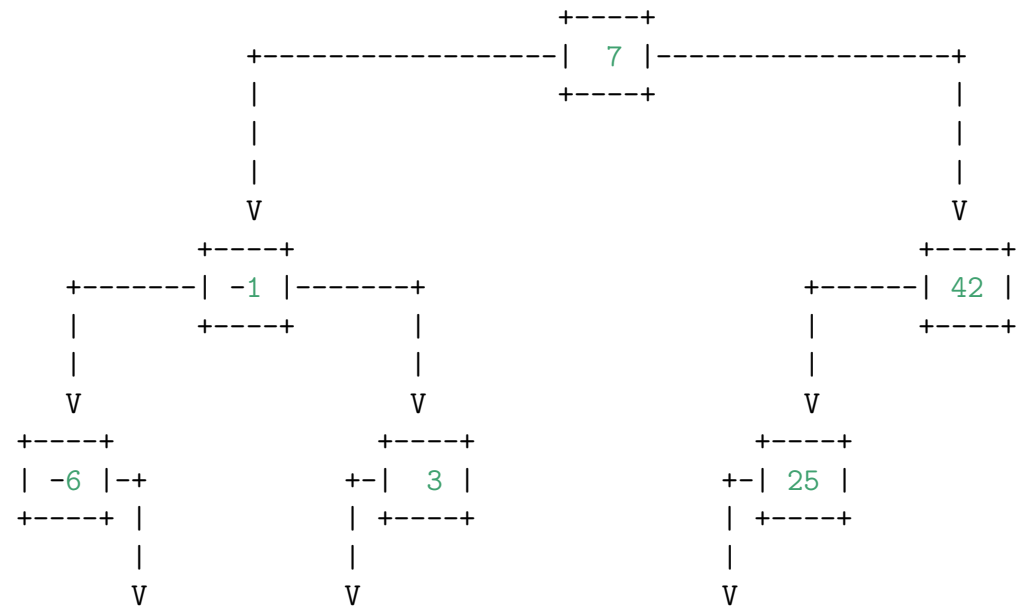


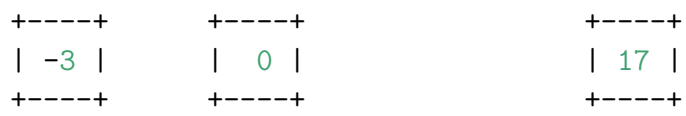
[!] O **comprimento interno** (internal path length) de uma BT é a soma das profundidades dos seus nós, ou seja, a soma dos comprimentos de todos os caminhos que levam da raiz até um nó. Esse conceito é usado para estimar o desempenho esperado de TSs implementadas com BSTs

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
      // Devolve 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;
   }
```

```
}
```

Desempenho no pior caso

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

Logo, o número de nós visitados não passa de 1 + h, sendo h a altura da BST.

Proposição E No pior caso, todas as operações sobre uma BST consomem tempo proporcional à altura da árvore.

Infelizmente, uma BST pode não estar balanceada: sua altura pode estar bem mais perto de N que de lg N.

Desempenho esperado (= médio)

Qual a altura esperada de uma BST aleatória? Resposta, aproximadamente $3 \lg n$. Veja animação no website do livro.

Portanto, o número esperado de nós visitados durante uma busca em uma BST aleatória não passa de $3 \lg n$.

O número esperado de nós visitados durante uma busca ou inserção em uma BST aleatória é menor que $3 \lg n$: ele tende a $1.4 \lg n$ quando `n` aumenta (veja Proposições C e D) no livro.

Resumo

	pior caso		caso medio			
	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()