

Aula 08: 07/04/2017

Tópico

- 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

```
linked-list> java Driver ../testes/les-miserables.txt
Criando a ST com as palavras do arquivo '../testes/les-miserables.txt' ...
ST criada em 92.785 segundos
ST contém 26764 itens
Início da consulta interativa. Tecla ctrl+D encerrar
```

```
sorted-linked-list> java Driver ../testes/les-miserables.txt
Criando a ST com as palavras do arquivo '../testes/les-miserables.txt' ...
ST criada em 84.192 segundos
ST contém 26764 itens
Início da consulta interativa. Tecla ctrl+D encerrar
>>>
```

```
array> java Driver ../testes/les-miserables.txt
Criando a ST com as palavras do arquivo '../testes/les-miserables.txt' ...
ST criada em 60.084 segundos
ST contém 26764 itens
Início da consulta interativa. Tecla ctrl+D encerrar
>>>
```

```
binary-search> java Driver ../testes/les-miserables.txt
Criando a ST com as palavras do arquivo '../testes/les-miserables.txt' ...
ST criada em 1.283 segundos
ST contém 26764 itens
Início da consulta interativa. Tecla ctrl+D encerrar
>>>
```

```
binary-search-tree> java Driver ../testes/les-miserables.txt
Criando a ST com as palavras do arquivo '../testes/les-miserables.txt' ...
ST criada em 0.687 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

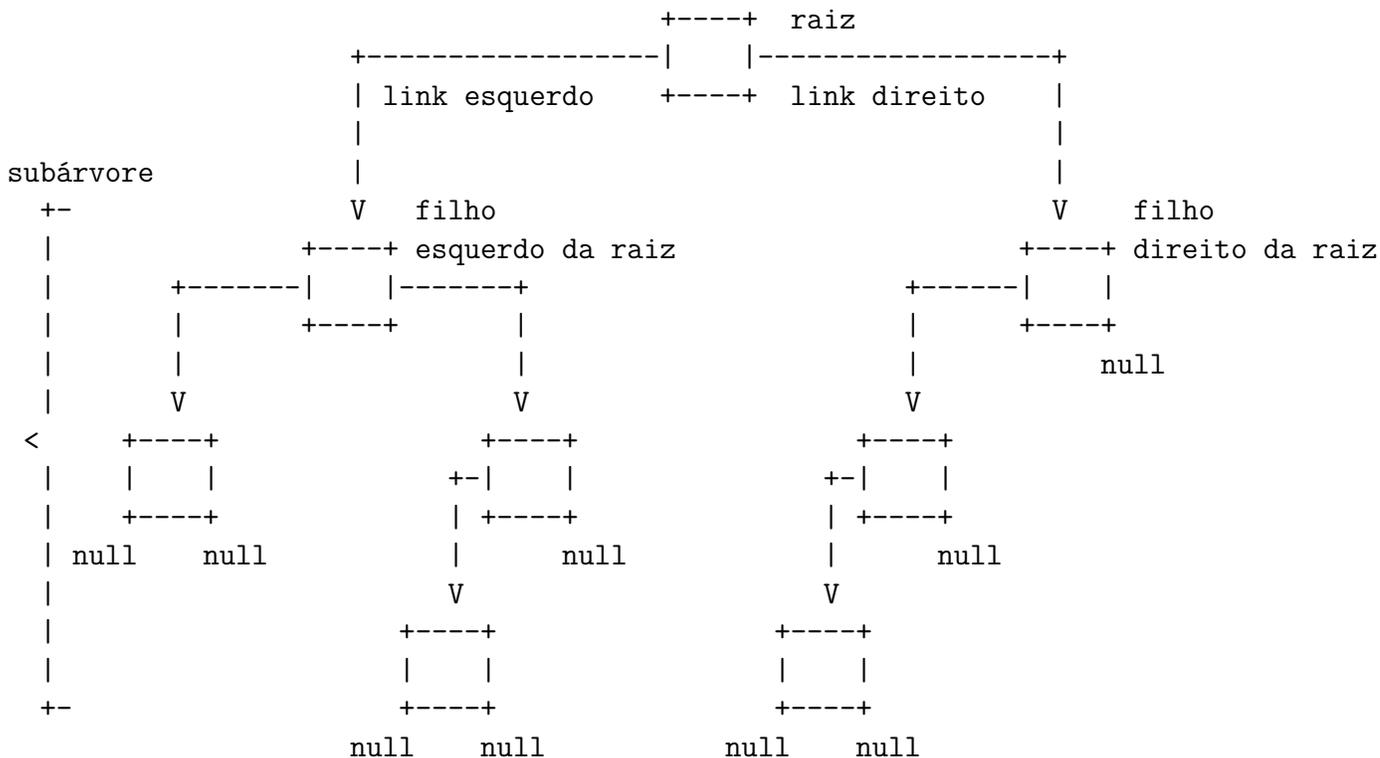
Em uma árvore binária cada nó tem no máximo dois filhos: um esquerdo e outro direito

```
+-----+-----+
|   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

```
private Node root;
```



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.

```
“java 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 }
```

Percursos (*traversals*)

Existem várias maneiras de percorrermos um árvore binária:

- *in-ordem*: ordem esquerda-raiz-direita;
- *pós-ordem*: ordem esquerda-direita-raiz; e
- *pré-ordem*: ordem raiz-esquerda-direita.

`preOrdem()`, `inOrdem()` e `posOrdem()` retorna todos os itens da árvore como `Iterable`. Para iterar sobre todos os itens de uma tabela de símbolos de nome `st` basta fazermos

```
for (Item item: st.preOrdem()) {  
    StdOut.println(item);  
}  
  
public Iterable<Item> preOrdem() {  
    Queue<Item> queue = new Queue<Item>();  
    preOrdem(root, queue)  
    return queue;  
}
```

```

}

private void preOrdem(Node r, Queue<Item> queue) {
    if (r != null) {
        queue.enqueue(r.item);
        preOrdem(r.left, queue);
        preOrdem(r.right, queue);
    }
}

private void inOrdem(Node r, Queue<Item> queue) {
    if (r != null) {
        inOrdem(r.left, queue);
        queue.enqueue(r.item);
        inOrdem(r.right, queue);
    }
}

private void posOrdem(Node r, Queue<Item> queue) {
    if (r != null) {
        posOrdem(r.left, queue);
        posOrdem(r.right, queue);
        queue.enqueue(r.item);
    }
}

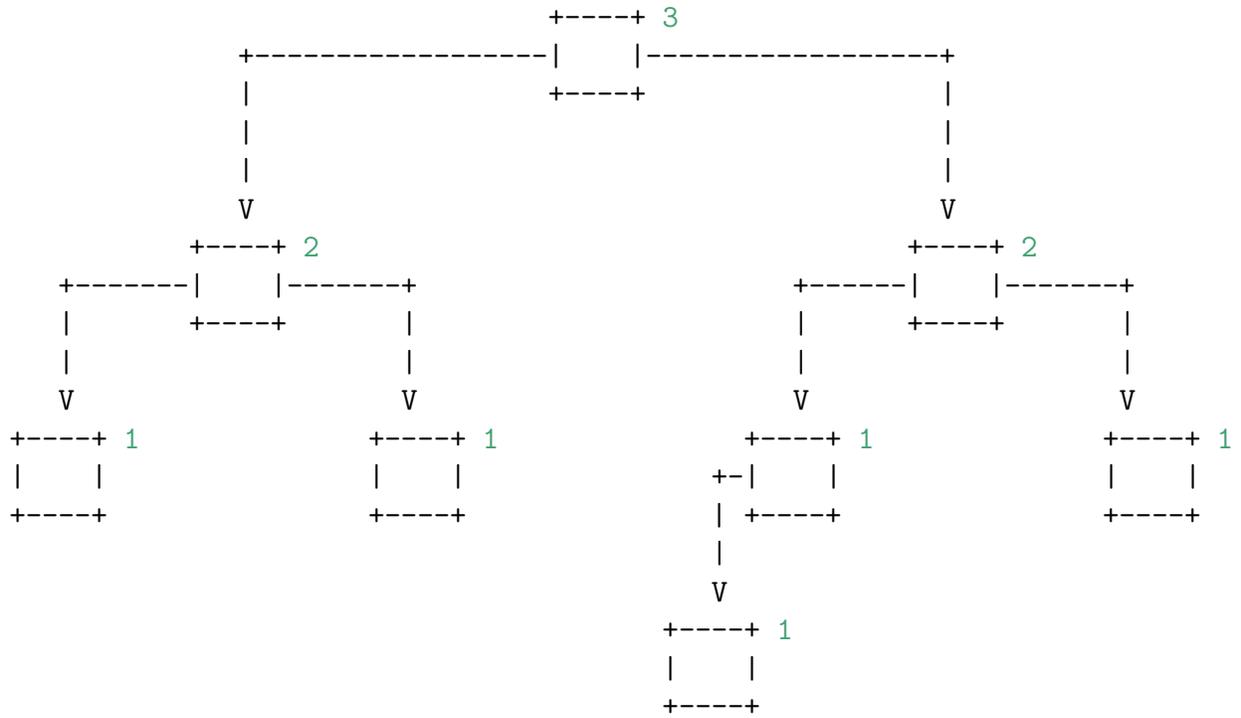
// versão iterativa
private void inOrdem(Node r, Queue<Item> queue) {
    Stack<Node> s = new Stack<Node>;
    while (r != null || !s.isEmpty()) {
        if (r != null) {
            s.push(r);
            r = r.left;
        }
        else {
            r = s.pop();
            queue.enqueue(r.item);
            r = r.right;
        }
    }
}

```

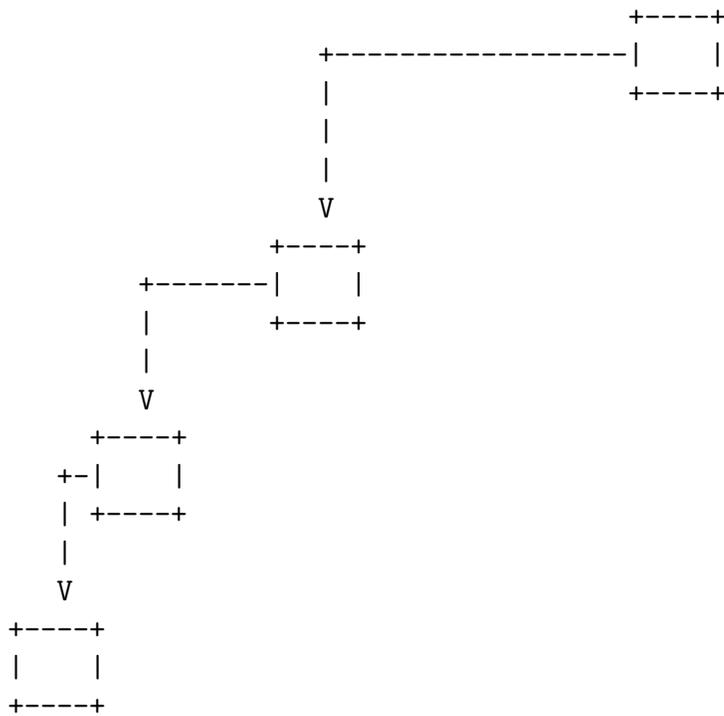
Uma BT com n nós, tem altura no máximo $n-1$ e no mínimo $\lceil \lg n \rceil$.

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

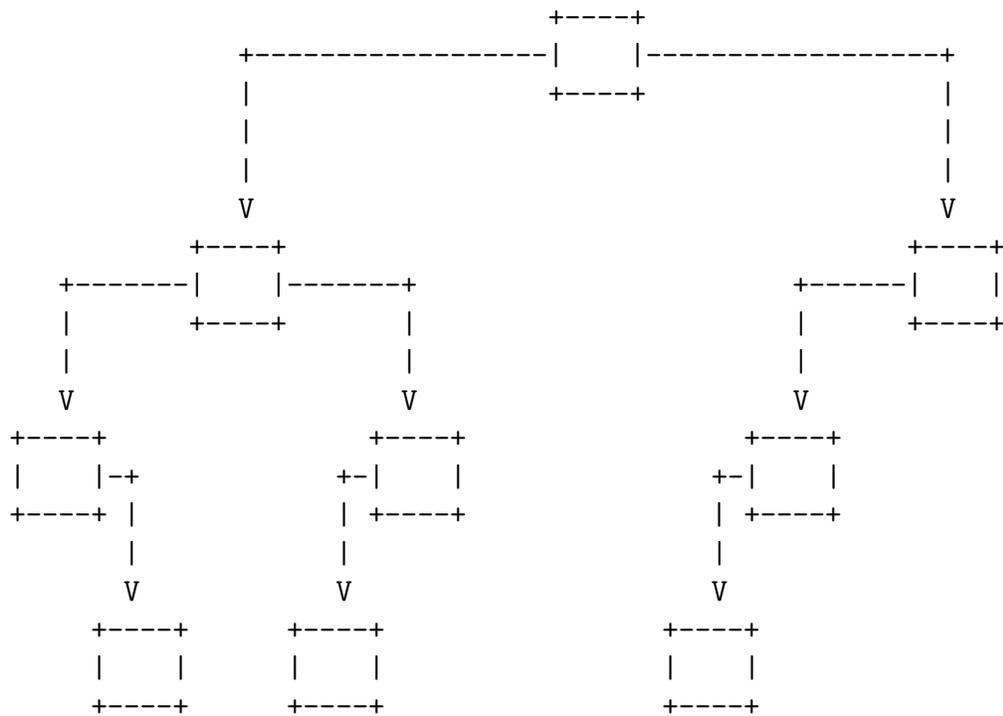
Árvore balanceada



Árvore de altura n



Árvores típica:

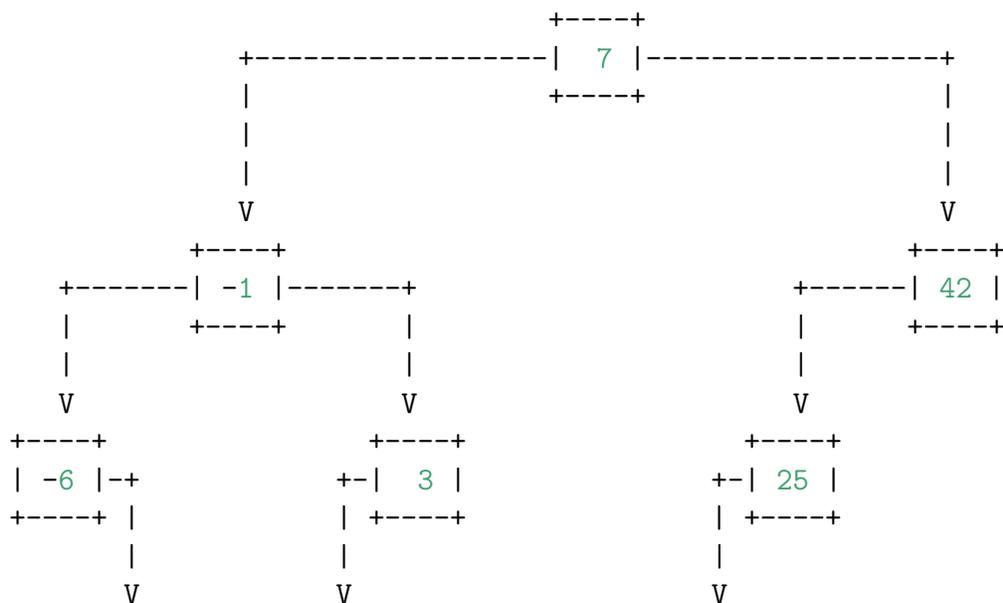


[!] O **comprimento interno** (internal path length) de uma BT é a soma das profundidades dos seus nós, ou seja, a soma dos comprimentos de todos os caminhos que levam da raiz até um nó. Esse conceito é usado para estimar o desempenho esperado de TSs implementadas com BSTs

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.



```

+-----+      +-----+      +-----+
|  -3  |      |   0  |      |  17  |
+-----+      +-----+      +-----+

```

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		ordenada	interface
	get()	put()	get()	put()		
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()