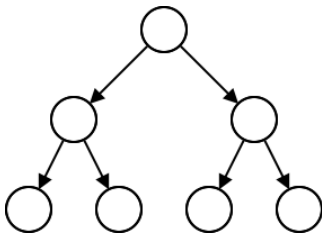


# Árvores Binárias

Pedro Ribeiro

DCC/FCUP

2021/2022



*(baseado e/ou inspirado parcialmente nos slides de Luís Lopes e de Fernando Silva)*

# Estruturas não lineares

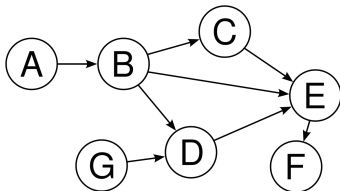
Os arrays e as listas são exemplos de estruturas de dados **lineares**.

Cada elemento tem:

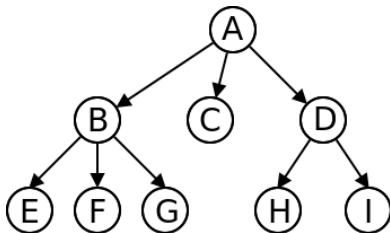
- um predecessor único (excepto o primeiro elemento da lista);
- um sucessor único (excepto o último elemento da lista).

Existem outros tipos de estruturas?

- um **grafo** é uma estrutura de dados **não-linear**, pois cada um dos seus elementos, designados por nós, pode ter mais do que um predecessor ou mais do que um sucessor.

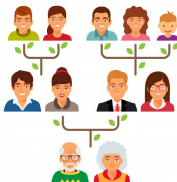
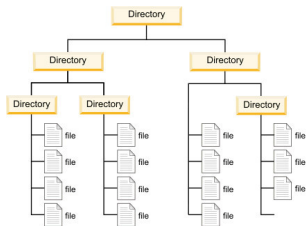


- Uma **árvore** é um tipo específico de grafo
- cada elemento, designado por **nó**, tem zero ou mais sucessores, mas apenas um predecessor (excepto o primeiro nó, a que se dá o nome de **raíz** da árvore);
- Um exemplo de árvore:

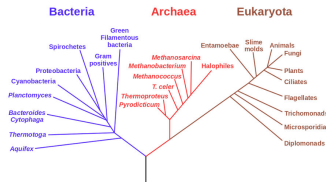


# Árvores - Exemplos

- As árvores são estruturas particularmente adequadas para representar informação organizada em **hierarquias**:
- Alguns exemplos:
  - ▶ a estrutura de directórios (ou pastas) de um sistema de ficheiros
  - ▶ uma árvore genealógica de uma família
  - ▶ uma árvore da vida

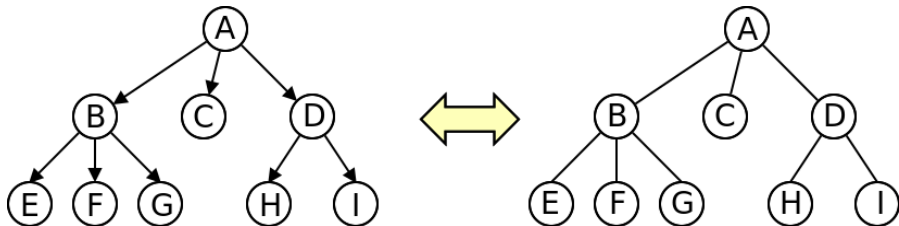


Phylogenetic Tree of Life

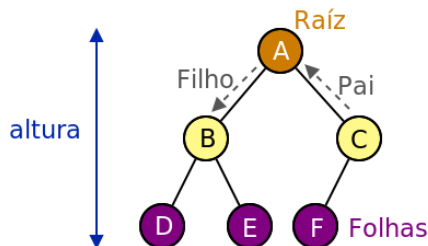


# Árvores - Visualização

- Muitas vezes não se incluem as "setas" nos arcos (ou ligações) pois fica claro pelo desenho quais nós descendem de quais:

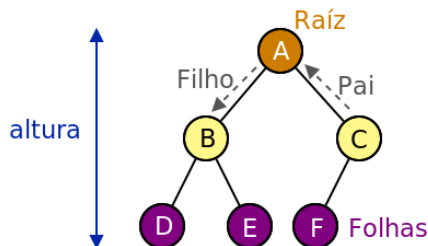


# Árvores - Terminologia



- Ao predecessor (único) de um nó, chamamos **pai**
  - ▶ Exemplo: O pai de B é A; o pai de C também é A
- Os sucessores de um nó são os seus **filhos**
  - ▶ Exemplo: Os filhos de A são B e C
- O **grau** de um nó é o seu número de filhos
  - ▶ Exemplo: A tem 2 filhos, C tem 1 filho
- Uma **folha** é um nó sem filhos, ou seja, de grau 0
  - ▶ Exemplo: D, E e F são nós folha
- A **raiz** é o único nó sem pai
- Uma **subárvore** é um subconjunto de nós (ligados) da árvore
  - ▶ Exemplo:  $\{B, D, E\}$  são uma sub-árvore

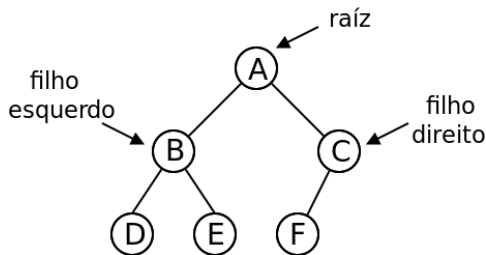
# Árvores - Terminologia



- Os arcos que ligam os nós, chamam-se **ramos**
- Chama-se **caminho** à sequência de ramos entre dois nós
  - ▶ *Exemplo: A-B-D é o caminho entre A e D*
- O **comprimento** de um caminho é o número de ramos nele contido;
  - ▶ *Exemplo: A-B-D tem comprimento 2*
- A **profundidade** de um nó é o comprimento do caminho desde a raiz até esse nó (a profundidade da raiz é zero);
  - ▶ *Exemplo: B tem profundidade 1, D tem profundidade 2*
- A **altura** de uma árvore é a profundidade máxima de um nó da árvore
  - ▶ *Exemplo: A árvore da figura tem altura 2*

# Árvores Binárias

- A **aridade** de uma árvore é o grau máximo de um nó
- Uma **árvore binária** é uma árvore de aridade 2, isto é, cada nó possui no máximo dois filhos, designados por filho **esquerdo** e **direito**.





# Árvores Binárias - Implementação

- Vamos então implementar uma árvore binária **genérica** (valores podem ser inteiros, strings ou qualquer outro tipo de objecto)
- Começemos por definir um **nó da árvore**:

```
class BTreeNode<T> {
    private T value;           // Valor guardado no nó
    private BTreeNode<T> left; // Filho esquerdo
    private BTreeNode<T> right; // Filho direito

    // Construtor
    BTreeNode(T v, BTreeNode<T> l, BTreeNode<T> r) {
        value = v; left = l; right = r;
    }

    // Getters e Setters
    public T getValue() {return value;}
    public BTreeNode<T> getLeft() {return left;}
    public BTreeNode<T> getRight() {return right;}
    public void setValue(T v) {value = v;}
    public void setLeft(BTreeNode<T> l) {left = l;}
    public void setRight(BTreeNode<T> r) {right = r;}
}
```

# Árvores Binárias - Implementação

- Vamos agora definir a **árvore** em si
- Do mesmo modo que uma lista ligada tem uma referência para o primeiro nó da lista, uma árvore deve ter uma referência para... a **raíz!**

```
public class BTree<T> {
    private BTNode<T> root; // raíz da árvore

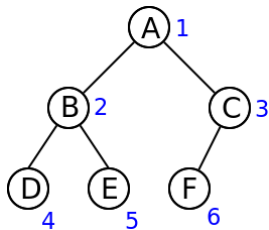
    // Construtor
    BTree() {
        root = null;
    }

    // Getter e Setter para a raíz
    public BTNode<T> getRoot() {return root;}
    public void setRoot(BTNode<T> r) {root = r;}

    // Verificar se árvore está vazia
    public boolean isEmpty() {
        return root == null;
    }
}
```

# Árvores Binárias - Número de nós

- Vamos criar alguns **métodos** para colocar na classe `BTree<T>`
- Um primeiro método tem como objectivo **contar o número de nós** de uma árvore. Por exemplo, a árvore binária seguinte tem 6 nós:



- Vamos criar um método **recursivo**:
  - ▶ **Caso base:** quando a árvore está vazia... tem 0 nós!
  - ▶ **Caso recursivo:** o  $n^{\circ}$  nós numa árvore não vazia é igual a 1 mais o  $n^{\circ}$  nós da subárvore esquerda, mais o  $n^{\circ}$  nós da subárvore direita
    - ★ Exemplo fig.:  $num\_nós = 1 + num\_nos(\{B,D,E\}) + num\_nos(\{C,F\})$

# Árvores Binárias - Número de nós

- Precisamos de começar a contar... a partir da raíz!
- Queremos ter um método `numberNodes()` na classe `BTree<T>`
  - ▶ *Exemplo: se  $t$  for uma árvore queremos poder chamar  $t.numberNodes()$*
- Vamos usar um **método auxiliar** (privado) que é recursivo

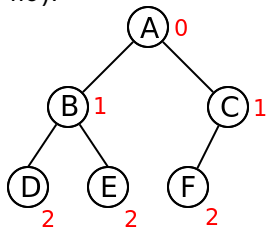
```
// Método principal (público)
public int numberNodes() {
    return numberNodes(root);
}

// Método auxiliar (privado)
private int numberNodes(BTNode<T> n) {
    if (n == null) return 0;
    return 1 + numberNodes(n.getLeft()) + numberNodes(n.getRight());
}
```

- Este **padrão** (*método principal que chama método auxiliar recursivo a partir da raíz*) pode ser usado para muitos tipos de métodos
- Vamos ver mais alguns exemplos...

# Árvores Binárias - Altura de uma árvore

- Vamos calcular a **altura de uma árvore** (profundidade máxima de um nó). Por exemplo, a árvore da figura tem altura 2 (a vermelho a profundidade de cada nó).



- Vamos criar um método **recursivo** muito parecido com o anterior:
  - ▶ **Caso recursivo:** a altura de uma árvore é igual 1 mais o máximo entre as alturas das subárvores esquerda e direita
    - ★ Exemplo fig.:  $altura = 1 + \max(altura(\{B,D,E\}), altura(\{C,F\}))$
- Qual deverá ser o **caso base?** Duas hipóteses:
  - ▶ Podemos parar numa folha: tem altura zero (0)
  - ▶ Se pararmos numa árvore nula, a altura tem de ser... **-1**
    - ★ Ex:  $altura \text{ árvore de } 1 \text{ nó} = 1 + \max(\text{null}, \text{null}) = 1 + \max(-1, -1) = 0$

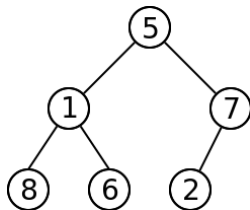
# Árvores Binárias - Altura de uma árvore

- Concretizando, com caso base da recursão do método auxiliar a ser a árvore nula (como no método do número de nós):

```
public int depth() {  
    return depth(root);  
}  
  
private int depth(BTNode<T> n) {  
    if (n == null) return -1;  
    return 1 + Math.max(depth(n.getLeft()), depth(n.getRight()));  
}
```

# Árvores Binárias - Procura de um elemento

- Vamos agora ver um método para **verificar se um elemento está ou não contido numa árvore**. Por exemplo, a árvore da figura seguinte contém o número 2, mas não contém o número 3:



- Vamos criar um método **recursivo** muito parecido com os anteriores:
  - ▶ **Caso base 1:** se a árvore é vazia, então não contém o valor que procuramos e devolvemos *false*
  - ▶ **Caso base 2:** se valor que procuramos está na raiz da árvore, então devolvemos *true*
  - ▶ **Caso recursivo:** se não está na raiz, então verificamos se está na subárvore esquerda **OU** na subárvore direita.

# Árvores Binárias - Procura de um elemento

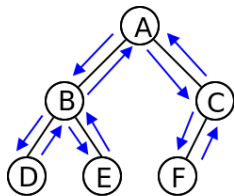
- Concretizando, e recordando que para comparar objectos devem usar o `.equals()` e não o `==`:

```
public boolean contains(T value) {  
    return contains(root, value);  
}  
  
private boolean contains(BTNode<T> n, T value) {  
    if (n==null) return false;  
    if (n.getValue().equals(value)) return true;  
    return contains(n.getLeft(), value) ||  
           contains(n.getRight(), value);  
}
```

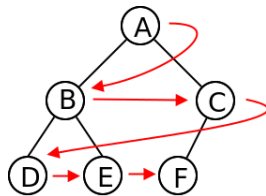


# Árvores Binárias - Escrita dos nós de uma árvore

- Como **escrever o conteúdo (os nós)** de uma árvore?
- Temos de **passar por todos os nós**. Mas por qual **ordem**?
- Vamos distinguir entre duas ordens diferentes:



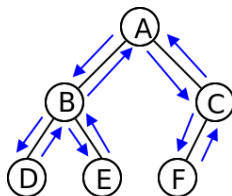
Pesquisa em Profundidade



Pesquisa em Largura

- **Pesquisa em Profundidade** (DFS: *depth-first-search*): visitar todos os nós da subárvore de um filho antes de visitar a subárvore do outro filho
- **Pesquisa em Largura** (BFS: *breadth-first-search*): visitar nós por ordem crescente de profundidade

# Árvores Binárias - Pesquisa em profundidade



Pesquisa em  
Profundidade

- Se escrevermos o nó da primeira vez que lá passamos, obtemos o seguinte para a figura: **A B D E C F**
- Isto equivale a fazer o seguinte:
  - 1 Escrever raíz
  - 2 Escrever toda a subárvore esquerda
  - 3 Escrever toda a subárvore direita
- Isto pode ser directamente convertido num **método recursivo!**

- Concretizando em código o que foi dito no slide anterior:

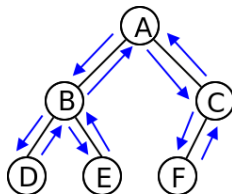
```
public void printPreOrder() {
    System.out.print("PreOrder:");
    printPreOrder(root);
    System.out.println();
}

private void printPreOrder(BTNode<T> n) {
    if (n==null) return;
    System.out.print(" " + n.getValue() );
    printPreOrder(n.getLeft());
    printPreOrder(n.getRight());
}
```

- Para a árvore anterior, iria ser escrito *"PreOrder: A B D E C F"*
- Chamamos a esta ordem **PreOrder**, porque escrevemos a raiz antes das duas subárvores

# Árvores Binárias - Pesquisa em profundidade

- Para além da **PreOrder**, podemos considerar também mais duas ordens em profundidade:
  - ▶ **InOrder**: raíz escrita *entre* as duas subárvores
  - ▶ **PostOrder**: raíz escrita *depois* das duas subárvores



Pesquisa em  
Profundidade

- Para a árvore da figura:
  - ▶ **PreOrder**: A B D E C F
  - ▶ **InOrder**: D B E A F C
  - ▶ **PostOrder**: D E B F C A

- Implementando a *InOrder*:

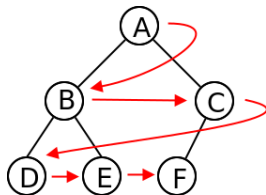
```
public void printInOrder() {
    System.out.print("InOrder:");
    printInOrder(root);
    System.out.println();
}

private void printInOrder(BTNode<T> n) {
    if (n==null) return;
    printInOrder(n.getLeft());
    System.out.print(" " + n.getValue());
    printInOrder(n.getRight());
}
```

- Implementando a *PostOrder*:

```
public void printPostOrder() {
    System.out.print("PostOrder:");
    printPostOrder(root);
    System.out.println();
}

private void printPostOrder(BTNode<T> n) {
    if (n==null) return;
    printPostOrder(n.getLeft());
    printPostOrder(n.getRight());
    System.out.print(" " + n.getValue());
}
```

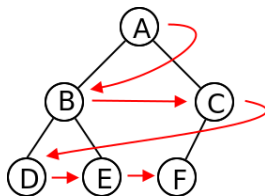


Pesquisa em  
Largura

- Para visitar em largura vamos usar o **TAD fila**
  - 1 Inicializar uma fila  $Q$  adicionando a raiz
  - 2 Enquanto  $Q$  não estiver vazia:
    - 3 Retirar primeiro elemento  $cur$  da fila
    - 4 Escrever  $cur$
    - 5 Adicionar filhos de  $cur$  ao fim da fila

# Árvores Binárias - Pesquisa em largura

- Vejamos um exemplo:



Pesquisa em  
Largura

- 1 Inicialmente temos que  $Q = \{A\}$
- 2 Retiramos e escrevemos **A**, adicionamos filhos **B** e **C**:  $Q = \{B, C\}$
- 3 Retiramos e escrevemos **B**, adicionamos filhos **D** e **E**:  $Q = \{C, D, E\}$
- 4 Retiramos e escrevemos **C**, adicionamos filho **F**:  $Q = \{D, E, F\}$
- 5 Retiramos e escrevemos **D**, não tem filhos:  $Q = \{E, F\}$
- 6 Retiramos e escrevemos **E**, não tem filhos:  $Q = \{F\}$
- 7 Retiramos e escrevemos **F**, não tem filhos:  $Q = \{\}$



# Árvores Binárias - Pesquisa em largura

- Concretizando em código:

(Nota: vamos usar a nossa implementação de filas, mas poderíamos também ter usado as filas do Java)

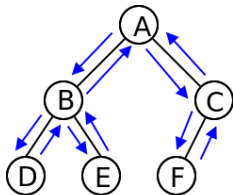
```
public void printBFS() {
    System.out.print("BFS:");

    MyQueue<BTNode<T>> q = new LinkedListQueue<BTNode<T>>();
    q.enqueue(root);
    while (!q.isEmpty()) {
        BTNode<T> cur = q.dequeue();
        if (cur != null) {
            System.out.print(" " + cur.getValue());
            q.enqueue(cur.getLeft());
            q.enqueue(cur.getRight());
        }
    }
    System.out.println();
}
```

- Nesta versão deixamos *null* entrar na fila, mas depois ignoramos. Também poderíamos só ter adicionado se não fosse *null*.

# Árvores Binárias - BFS vs DFS

- Se em vez de uma **fila**  $Q$  (*FIFO*) tivéssemos usado uma **pilha**  $S$  (*LIFO*), em vez de BFS estaríamos a fazer... um DFS!



## Pesquisa em Profundidade

- Inicialmente temos que  $S = \{A\}$
- $pop$  e escrita de **A**, push de filhos  $B$  e  $C$ :  $S = \{B, C\}$
- $pop$  e escrita de **C**, push do filho  $F$ :  $S = \{B, F\}$
- $pop$  e escrita de **F**, não tem filhos:  $S = \{B\}$
- $pop$  e escrita de **B**, push de filhos  $D$  e  $E$ :  $S = \{D, E\}$
- $pop$  e escrita de **E**, não tem filhos:  $S = \{D\}$
- $pop$  e escrita de **D**, não tem filhos:  $S = \{\}$

# Árvores Binárias - Pesquisa em profundidade

- Concretizando em código:

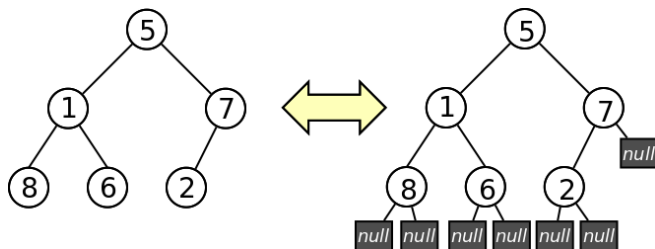
(Nota: vamos usar a nossa implementação de pilhas, mas poderíamos também ter usado as pilhas do Java)

```
public void printDFS() {
    System.out.print("DFS:");

    MyStack<BTNode<T>> q = new LinkedListStack<BTNode<T>>();
    q.push(root);
    while (!q.isEmpty()) {
        BTNode<T> cur = q.pop();
        if (cur != null) {
            System.out.print(" " + cur.getValue());
            q.push(cur.getLeft());
            q.push(cur.getRight());
        }
    }
    System.out.println();
}
```

# Árvores Binárias - Leitura PreOrder

- Como **ler uma árvore**?
- Uma hipótese é usar **PreOrder**, representando explicitamente os *nulls*
- Note que as duas representações seguintes referem-se à mesma árvore:

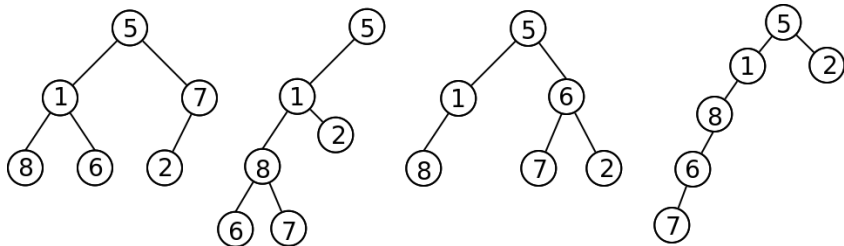


- Se representarmos *null* por **N**, então a árvore em *PreOrder* ficaria representada por:

5 1 8 N N 6 N N 7 2 N N N

# Árvores Binárias - Leitura PreOrder

- Note como os *nulls* são necessários.
- Ex: sem *nulls*, a seguinte representação inorder podia referir-se a qualquer uma das 4 árvores (entre outras): 5 1 8 6 7 2



- Com os *nulls*, as 4 árvores ficam diferentes:
  - ▶ 1ª Árvore: 5 1 8 N N 6 N N 7 2 N N N
  - ▶ 2ª Árvore: 5 1 8 6 N N 7 N N 2 N N N
  - ▶ 3ª Árvore: 5 1 8 N N N 6 7 N N 2 N N
  - ▶ 4ª Árvore: 5 1 8 6 7 N N N N N 2 N N

# Árvores Binárias - Leitura PreOrder

- Implementando em código (numa classe utilitária com métodos estáticos) uma leitura preorder de uma árvore de inteiros:

```
import java.util.Scanner;

class LibBTree {
    public static BTree<Integer> readIntTree(Scanner in) {
        BTree<Integer> t = new BTree<Integer>();
        t.setRoot(readIntNode(in));
        return t;
    }

    private static BTreeNode<Integer> readIntNode(Scanner in) {
        String s = in.next();
        if (s.equals("N")) return null;
        Integer value = Integer.parseInt(s);
        BTreeNode<Integer> left = readIntNode(in);
        BTreeNode<Integer> right = readIntNode(in);
        return new BTreeNode<Integer>(value, left, right);
    }
}
```

# Árvores Binárias - Testando tudo o que fizemos

- Testando tudo o que foi implementado:

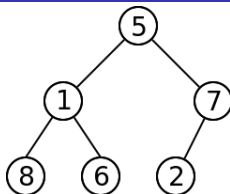
```
import java.util.Scanner;

class TestBTree {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        BTree<Integer> t = LibBTree.readIntTree(in);

        System.out.println("numberNodes = " + t.numberNodes());
        System.out.println("depth = " + t.depth());
        System.out.println("contains(2) = " + t.contains(2));
        System.out.println("contains(3) = " + t.contains(3));

        t.printPreOrder();
        t.printInOrder();
        t.printPostOrder();
        t.printBFS();
        t.printDFS();
    }
}
```

# Árvores Binárias - Testando tudo o que fizemos



- Chamando com o input da árvore da figura colocado num ficheiro `input.txt`

```
5 1 8 N N 6 N N 7 2 N N N
```

- `java TestBTree < input.txt` daria como resultado:

```
numberNodes = 6
depth = 2
contains(2) = true
contains(3) = false
PreOrder: 5 1 8 6 7 2
InOrder: 8 1 6 5 2 7
PostOrder: 8 6 1 2 7 5
BFS: 5 1 7 8 6 2
DFS: 5 7 2 1 6 8
```



# Árvores Binárias - Complexidade dos métodos

- Qual a **complexidade temporal** dos métodos que implementamos?
  - ▶ `numberNodes()`
  - ▶ `depth()`
  - ▶ `contains()`
  - ▶ `printPreOrder()`
  - ▶ `printInOrder()`
  - ▶ `printPostOrder()`
  - ▶ `printBFS()`
  - ▶ `printDFS()`
  - ▶ `readIntTree(Scanner in)`
- Todos eles passam uma única vez por cada nó da árvore (para *contains()* esse é o pior caso, nos outros métodos é sempre assim), e gastam um número constante de operações nesse nó.
- Todos estes métodos têm portanto complexidade linear  $\mathcal{O}(n)$ , onde  $n$  é o número de nós da árvore
- Será possível melhorar esta complexidade?

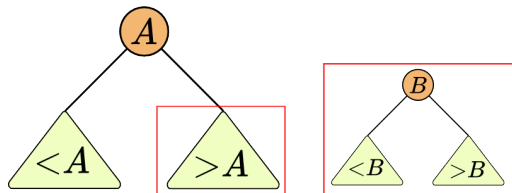
## Árvores Binárias de Pesquisa

# Árvores Binárias de Pesquisa - Motivação

- Seja  $S$  um conjunto de objectos/itens "**comparáveis**":
  - ▶ Sejam  $a$  e  $b$  dois objectos.  
São "**comparáveis**" se for possível dizer se  $a < b$ ,  $a = b$  ou  $a > b$ .
  - ▶ Um exemplo seriam números, mas poderiam ser outra coisa (alunos com um nome e n.º mecanográfico; equipas com pontos, golos marcados e sofridos, ...)
- Alguns possíveis **problemas** de interesse:
  - ▶ Dado um conjunto  $S$ , determinar se **um dado item está em  $S$**
  - ▶ Dado um conjunto  $S$  **dinâmico** (que sofre alterações: adições e remoções), determinar se **um dado item está em  $S$**
  - ▶ Dado um conjunto  $S$  **dinâmico** determinar **o maior/menor** item de  $S$
  - ▶ **Ordenar** um conjunto  $S$
  - ▶ ...
- **Árvores Binárias de Pesquisa!**

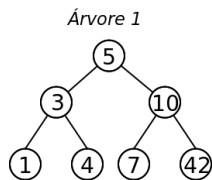
# Árvores Binárias de Pesquisa - Conceito

- Para **todos** os nós da árvore, deve acontecer o seguinte:  
**o nó é maior que todos os nós da sua subárvore esquerda e menor que todos os nós da sua subárvore direita**

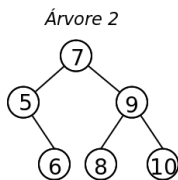


# Árvores Binárias de Pesquisa - Exemplos

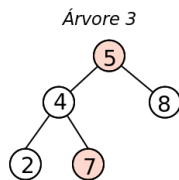
- Para **todos** os nós da árvore, deve acontecer o seguinte:  
**o nó é maior que todos os nós da sua subárvore esquerda e menor que todos os nós da sua subárvore direita**



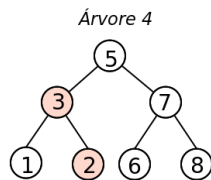
Árvore Binária de Pesquisa



Árvore Binária de Pesquisa



Não é Árvore Binária de Pesquisa

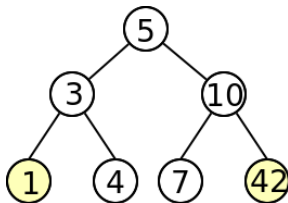


Não é Árvore Binária de Pesquisa

- Nas árvores 1 e 2 as condições são respeitadas
- Na árvore 3 o nó 7 está à esquerda do nó 5 mas  $7 > 5$
- Na árvore 4 o nó 2 está à direita do nó 3 mas  $2 < 3$

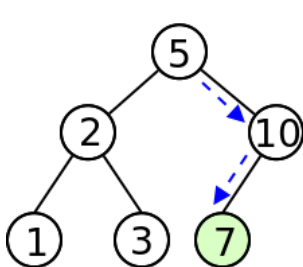
# Árvores Binárias de Pesquisa

## Algumas consequências



- O **menor** elemento de todos está... no **nó mais à esquerda**
- O **maior** elemento de todos está... no **nó mais à direita**

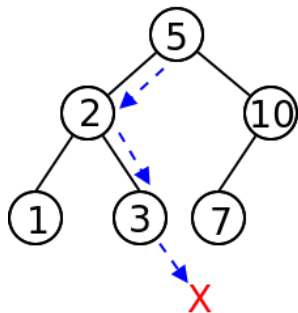
# Árvores Binárias de Pesquisa - Pesquisar um valor



contains(7) ?

true

- Começar na raiz, e ir percorrendo a árvore
- Escolher ramo esquerdo ou direito consoante o valor seja menor ou maior que o nó "actual"

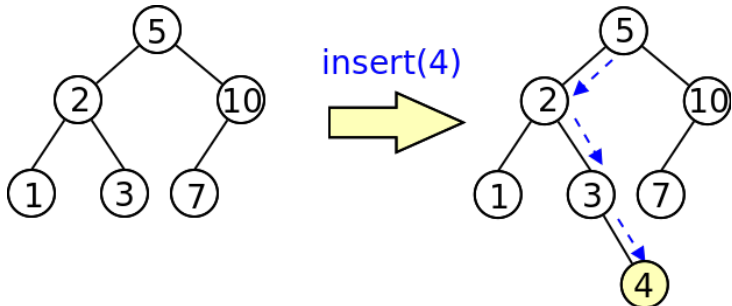


contains(4) ?

false

- Começar na raiz, e ir percorrendo a árvore
- Escolher ramo esquerdo ou direito consoante o valor seja menor ou maior que o nó "actual"

# Árvores Binárias de Pesquisa - Inserir um valor

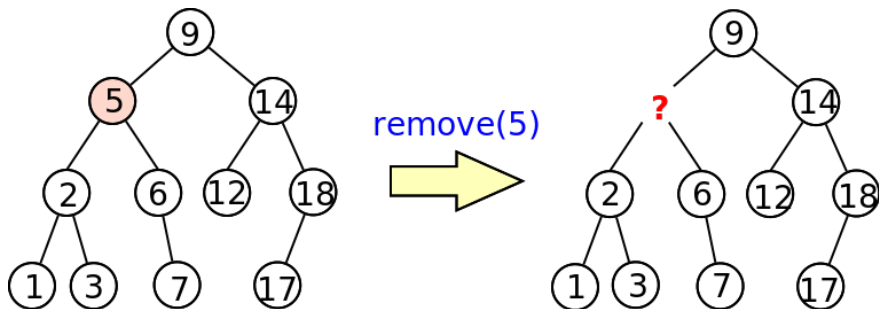


- Começar na raiz, e ir percorrendo a árvore
- Escolher ramo esquerdo ou direito consoante o valor seja menor ou maior que o nó "actual"
- Inserir na posição folha correspondente

**Nota:** Normalmente **se valor for igual a um já existente...** não se insere. Caso desejemos ter valores repetidos (um *multiset*) temos de ser coerentes e assumir sempre uma posição (ex: sempre à esquerda, ou seja, nós do ramo esquerdo seriam  $\leq$  e nós do ramo direito seriam  $>$ )



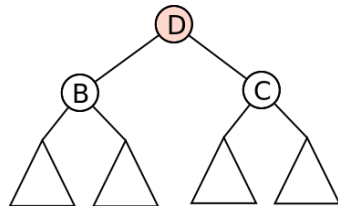
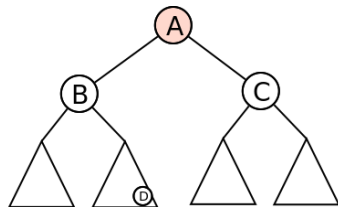
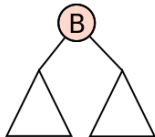
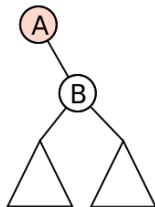
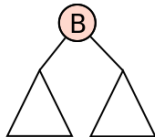
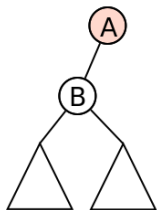
# Árvores Binárias de Pesquisa - Remover um valor



- Começar na raiz, e ir percorrendo a árvore até encontrar o valor
- Ao retirar o valor o que fazer a seguir?
  - ▶ Se o nó que retiramos só tiver um ramo filho, basta "subir" esse filho até à posição correspondente
  - ▶ Se tiver dois ramos filhos, os candidatos a ficarem nessa posição são:
    - ★ O maior nó do ramo esquerdo, ou
    - ★ o menor nó do ramo direito

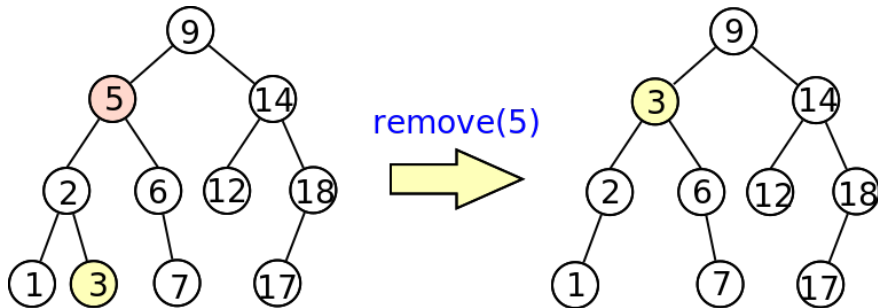
# Árvores Binárias de Pesquisa - Remover um valor

- Depois de encontrar o nó é preciso decidir **como o retirar**
  - ▶ 3 casos possíveis:



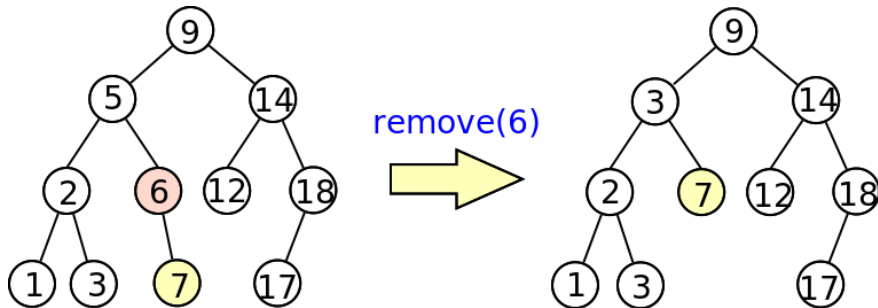
# Árvores Binárias de Pesquisa - Remover um valor

Exemplo com dois filhos



# Árvores Binárias de Pesquisa - Remover um valor

Exemplo só com um filho



# Árvores Binárias de Pesquisa - Visualização

- Podem visualizar a pesquisa, inserção e remoção (experimentem o url indicado):

<https://www.cs.usfca.edu/~galles/visualization/BST.html>

**Binary Search Tree**

Insert Delete Find Print

Searching for 0007 : 0007 = 0007 (Element found!)

```
graph TD; 0010((0010)) --> 0005((0005)); 0010 --> 0014((0014)); 0005 --> 0002((0002)); 0005 --> 0007((0007));
```

Animation Paused

Skip Back Step Back play Step Forward Skip Forward Animation Speed

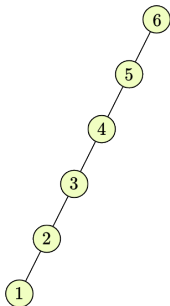
- Como caracterizar o **tempo que cada operação demora**?
  - ▶ Todas as operações procuram um nó percorrendo a **altura** da árvore

## Complexidade de operações numa árvore binária de pesquisa

Seja  $h$  a altura de uma árvore binária de pesquisa  $T$ . A complexidade de descobrir o mínimo, o máximo ou efetuar uma pesquisa, uma inserção ou uma remoção em  $T$  é  $\mathcal{O}(h)$ .

# Desiquilíbrio numa Árvore Binária de Pesquisa

- O **problema** do método anterior:

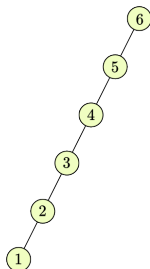


A altura da árvore pode ser da ordem de  $\mathcal{O}(n)$  ( $n$ , número de elementos)

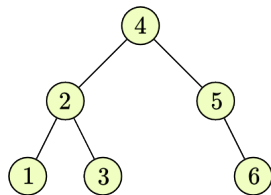
*(a altura depende da ordem de inserção e existem ordens "más")*

# Árvores equilibradas

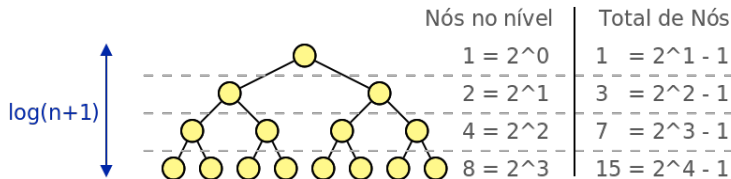
- Queremos árvores... **equilibradas**



vs



- Numa árvore equilibrada com  $n$  nós, a altura é ...  $\mathcal{O}(\log n)$

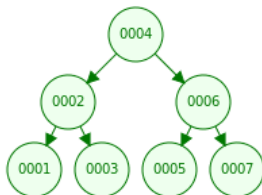




# Árvores equilibradas

Dado um conjunto de números, **por que ordem inserir** numa árvore binária de pesquisa para que fique o mais equilibrada possível?

**Resposta:** “*pesquisa binária*” - se os números estiverem ordenados, inserir o elemento do meio, partir a lista restante em duas nesse elemento e inserir os restantes elementos de cada metade pela mesma ordem



# Estratégias de Balanceamento

- E se não soubermos os elementos todos à partida e tivermos de **dinamicamente** ir inserindo e removendo elementos?

**Nota:** não vamos falar de como implementar nenhuma das estratégias seguintes deste slide, mas para ficarem com uma ideia de que existem e quais os seus nomes

(vão falar de algumas delas noutra unidades curriculares como *Desenho e Análise de Algoritmos*, *Algoritmos* ou *Tópicos Avançados em Algoritmos*)

- Existem estratégias para garantir que a complexidade das operações de pesquisar, inserir e remover são melhores que  $\mathcal{O}(n)$

Árvores equilibradas:  
(altura  $\mathcal{O}(\log n)$ )

- ▶ Red-Black Trees
- ▶ AVL Trees
- ▶ Splay Trees
- ▶ Treaps

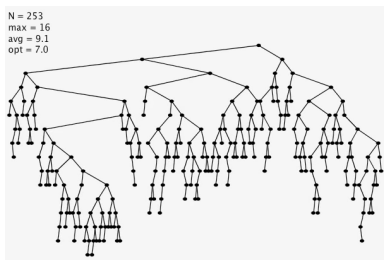
Outras estruturas de dados:

- ▶ Skip List
- ▶ Hash Table
- ▶ Bloom Filter

# Altura para uma ordem aleatórios

## Altura de uma árvore com elementos aleatórios

Se inserirmos  $n$  elementos por uma ordem completamente aleatória numa árvore binária de pesquisa, a sua *altura esperada* é  $\mathcal{O}(\log n)$



- Se tiverem curiosidade em ver uma prova espreitem o capítulo 12.4 do livro "*Introduction to Algorithms*" (não é necessário saber para exame)
- Para dados puramente *aleatórios*, a altura média é portanto  $\mathcal{O}(\log n)$  à medida que vamos inserindo e removendo

- Como implementar árvores binárias de pesquisa em **Java**?
- As árvore binárias *normais* apenas precisavam do método `equals()` para saber se um dado elemento estava na árvore.
- As árvores binárias de pesquisa necessitam contudo de elementos que sejam **comparáveis**:
  - ▶ Precisamos de saber se um elemento é *menor*, *igual* ou *maior* que outro
- Como fazer isto em java para um objecto genérico?  
Exemplo: não é possível comprar duas strings com operador `<`

```
String s1 = "ola", s2 ="mundo";  
if (s1 < s2) System.out.println("menor");
```

Em Java isto gera o erro: **bad operand types for binary operator '<'**

# Interface Comparable

- Em Java, a maneira correcta de comparar dois objectos é usar o método `compareTo()` (definido no interface *Comparable*).
- Sejam *o1* e *o2* dois objectos comparáveis do mesmo tipo. `o1.compareTo(o2)` devolve:
  - ▶ um valor negativo ( $< 0$ ) se *o1* é menor que *o2*
  - ▶ zero (0) se *o1* for igual a *o2*
  - ▶ um valor positivo ( $> 0$ ) se *o1* é maior que *o2*
- Os *wrappers* dos tipos primitivos implementam o interface `Comparable`, sendo que podemos directamente usá-lo:

```
String s1 = "ola", s2 = "mundo";
System.out.println(s1.compareTo(s2)); // Número positivo
Integer i1 = 42, i2 = 42;
System.out.println(i1.compareTo(i2)); // Zero
Double d1 = 0.5, d2 = 1.0;
System.out.println(d1.compareTo(d2)); // Número negativo
```

# Interface Comparable

- Muitos métodos de Java necessitam que os objectos seja comparáveis
- Um exemplo é o método `Arrays.sort` para ordenar um array. Como os wrappers são comparáveis, podemos chamar sem fazer mais nada:

```
import java.util.Arrays;

class TestSort {
    public static void main(String[] args) {
        Integer[] v1 = {4,6,7,3,1,8,2};
        Arrays.sort(v1);
        System.out.println(Arrays.toString(v1));
        String[] v2 = {"quarenta","e","dois","sentido","vida"};
        Arrays.sort(v2);
        System.out.println(Arrays.toString(v2));
    }
}
```

```
[1, 2, 3, 4, 6, 7, 8]
[dois, e, quarenta, sentido, vida]
```

*Nota: Sem a chamada a `Arrays.toString`, apenas seria imprimida a referência para o Array*

# Interface Comparable

- Se tentarmos ordenar uma classe "nossa", o Java vai-se queixar caso não seja implementado o interface *Comparable*:

```
import java.util.Arrays;

class Person {
    String name;
    int age;
    Person(String n, int a) { name = n; age = a;}
    public String toString() {return "(" + name + "," + age + ");}
}

class TestSort {
    public static void main(String[] args) {
        Person[] v = new Person[3];
        v[0] = new Person("Mary",23);
        v[1] = new Person("John",42);
        v[2] = new Person("X",31);
        Arrays.sort(v);
        System.out.println(Arrays.toString(v));
    }
}
```

Exception in thread "main" java.lang.ClassCastException: Person cannot be cast to java.lang.Comparable

# Interface Comparable

- Para ser usado com métodos que necessitem de objectos comparáveis... a nossa classe deve implementar o interface Comparable, que "exige" a implementação do método **compareTo**
- Por exemplo, se quisermos que a pessoa seja comparada tendo em conta a ordem alfabética do seu nome:

```
class Person implements Comparable<Person> {
    String name;
    int age;

    Person(String n, int a) { name = n; age = a; }
    public String toString() {return "(" + name + "," + age + ")"; }
    // Método compareTo para implementar o interface Comparable
    public int compareTo(Person p) { return name.compareTo(p.name); }
}
```

- O programa do slide anterior, com esta classe, já compila e dá o seguinte *output* quando executado:

```
[(John,42), (Mary,23), (X,31)]
```



# Interface Comparable

- Se quisermos então criar uma classe com genéricos que precise de comparar elementos, precisamos então "apenas" de dizer que o tipo genérico tem de implementar o interface comparable:

```
// Classe exemplo para mostrar uso do interface Comparable: erro de compilação
class TestComparable<T> {
    boolean lessThan(T o1, T o2) {
        if (o1.compareTo(o2) < 0) return true;
        else return false;
    }
}
```

Devolve erro dizendo que não consegue encontrar método **compareTo**

```
// Classe exemplo para mostrar uso do interface Comparable: ok!
class TestComparable<T extends Comparable<T>> {
    boolean lessThan(T o1, T o2) {
        if (o1.compareTo(o2) < 0) return true;
        else return false;
    }
}
```

Compila sem problemas, porque T implementa o interface Comparable

# Interface Comparable

- Um último detalhe:

(assumindo que temos as classes *Person* e *TestComparable* dos slides anteriores)

```
// Classe que herda os métodos de Person
class Student extends Person {
    Student(String n, int a) {super(n,a);}
}

class Test {
    public static void main(String[] args) {
        Student s1 = new Student("Paul", 22);
        Student s2 = new Student("Sarah", 25);

        // Esta linha compila bem porque Student herda
        // o método compareTo a partir de Person
        int resul = s1.compareTo(s2);

        // Esta linha dá erro de compilação
        // porque TestComparable exige algo que Comparable<Student>
        TestComparable<Student> t;
    }
}
```

Erro porque não é implementado directamente **comparable<Student>**

# Interface Comparable

- Se mudarmos a classe TestComparable para:

```
// O tipo T tem de implementar o interface Comparable
// (ou tê-lo herdado de uma super classe).
class TestComparable<T extends Comparable<? super T>> {
    boolean lessThan(T o1, T o2) {
        if (o1.compareTo(o2) < 0) return true;
        else return false;
    }
}
```

- Agora o o programa anterior já **compila sem problemas**, pois a linha `<T extends Comparable<? super T>>` admite que não seja o tipo T a implementar directamente *(o que importa é que seja possível chamar `T.compareTo()`)*.

# Árvores Binárias de Pesquisa - Implementação

- Estamos finalmente prontos para implementar uma árvore binária de pesquisa com objectos de tipo genérico.
- Começemos por um **nó**, que tem de ser de objectos... comparáveis:

```
class BSTNode<T extends Comparable<? super T>> {
    private T value;           // Valor guardado no nó
    private BSTNode<T> left;  // Filho esquerdo
    private BSTNode<T> right; // Filho direito

    // Construtor
    BSTNode(T v, BSTNode<T> l, BSTNode<T> r) {
        value = v;
        left = l;
        right = r;
    }

    // Getters e Setters
    public T getValue() {return value;}
    public BSTNode<T> getLeft() {return left;}
    public BSTNode<T> getRight() {return right;}
    public void setValue(T v) {value = v;}
    public void setLeft(BSTNode<T> l) {left = l;}
    public void setRight(BSTNode<T> r) {right = r;}
}
```

# Árvores Binárias de Pesquisa - Implementação

- A árvore contém uma referência para a raiz (tal como uma árvore binária normal)

```
public class BSTree<T extends Comparable<? super T>> {
    private BSTNode<T> root; // raiz da árvore

    // Construtor
    BSTree() {
        root = null;
    }

    // Verificar se árvore está vazia
    public boolean isEmpty() {
        return root == null;
    }

    // Limpa a árvore (torna-a vazia)
    public void clear() {
        root = null;
    }

    // (...)
}
```

# Árvores Binárias de Pesquisa - Implementação

- O que muda são os outros métodos, tal como **contains**, **insert** e **remove**, que se podem aproveitar do facto de ser árvore binária de pesquisa,
- Estes são métodos da classe BSTree. Para uma explicação visual do que fazem, podem ver slides anteriores (slides 38 a 45)
- Começemos pelo **contains**

```
// O elemento value está contido na árvore?  
public boolean contains(T value) {  
    return contains(root, value);  
}  
  
private boolean contains(BSTNode<T> n, T value) {  
    if (n==null) return false;  
    if (value.compareTo(n.getValue()) < 0) // menor? sub-árvore esquerda  
        return contains(n.getLeft(), value);  
    if (value.compareTo(n.getValue()) > 0) // maior? sub-arvore direita  
        return contains(n.getRight(), value);  
    return true; // se não é menor ou maior, é porque é igual  
}
```

- No **insert** vamos aproveitar o valor de retorno da função recursiva

```
// Adicionar elemento a uma árvore de pesquisa
// Devolve true se conseguiu inserir, false caso contrário
public boolean insert(T value) {
    if (contains(value)) return false;
    root = insert(root, value);
    return true;
}

private BSTNode<T> insert(BSTNode<T> n, T value) {
    if (n==null)
        return new BSTNode<T>(value, null, null);
    else if (value.compareTo(n.getValue()) < 0)
        n.setLeft(insert(n.getLeft(), value));
    else if (value.compareTo(n.getValue()) > 0)
        n.setRight(insert(n.getRight(), value));
    return n;
}
```

# Árvores Binárias de Pesquisa - Implementação

- **remove**: substituir removido pelo maior valor da subárvore esquerda

```
// Remover elemento de uma árvore de pesquisa
// Devolve true se conseguiu remover, false caso contrário
public boolean remove(T value) {
    if (!contains(value)) return false;
    root = remove(root, value);
    return true;
}
// Assume-se que elemento existe (foi verificado antes)
private BSTNode<T> remove(BSTNode<T> n, T value) {
    if (value.compareTo(n.getValue()) < 0)
        n.setLeft(remove(n.getLeft(), value));
    else if (value.compareTo(n.getValue()) > 0)
        n.setRight(remove(n.getRight(), value));
    else if (n.getLeft() == null) n = n.getRight(); // Sem filho esq.
    else if (n.getRight() == null) n = n.getLeft(); // Sem filho dir.
    else { // Dois filhos: ir buscar máximo do lado esquerdo
        BSTNode<T> max = n.getLeft();
        while (max.getRight() != null) max = max.getRight();
        n.setValue(max.getValue()); // Substituir valor removido
        n.setLeft(remove(n.getLeft(), max.getValue()));
    }
    return n;
}
```



# Árvores Binárias de Pesquisa - Teste

- Um exemplo de utilização (assumindo implementação usual de PreOrder, InOrder e PostOrder):

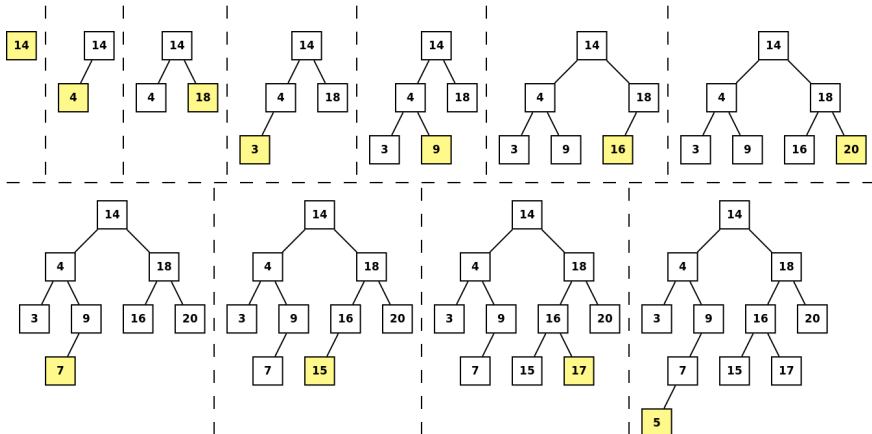
```
class TestBSTree {
    public static void main(String[] args) {

        // Criação da Árvore
        BSTree<Integer> t = new BSTree<Integer>();
        // Inserindo 11 elementos na árvore binária de pesquisa
        int[] data = {14, 4, 18, 3, 9, 16, 20, 7, 15, 17, 5};
        for (int i=0; i<data.length; i++) t.insert(data[i]);

        // Escrever resultado de chamada a alguns métodos
        System.out.println("numberNodes = " + t.numberNodes());
        System.out.println("depth = " + t.depth());
        System.out.println("contains(2) = " + t.contains(2));
        System.out.println("contains(3) = " + t.contains(3));
        // Escrever nós da árvore seguindo várias ordens possíveis
        t.printPreOrder(); t.printInOrder(); t.printPostOrder();
        // Experimentando remoção
        t.remove(14);
        t.printPreOrder(); t.printInOrder(); t.printPostOrder();
    }
}
```

# Árvores Binárias de Pesquisa - Teste

- O código do slide anterior cria a seguinte árvore binária de pesquisa:



- Inorder, esta árvore fica: 3 4 5 7 9 14 15 16 17 18 20

Quando imprimidos *inOrder*, os elementos de uma árvore binária de pesquisa ficam sempre por ordem crescente

# Árvores Binárias de Pesquisa e TAD Conjunto

- Quase no início do semestre falamos do **TAD Conjunto**, que tinha as seguintes operações básicas (ou seja, o seu *interface*):
  - ▶ **boolean contains(x)** - verifica se o elemento  $x$  está no conjunto. Retorna *true* se o elemento está no conjunto e *false* caso contrário.
  - ▶ **boolean add(x)** - adiciona o elemento  $x$  ao conjunto. Retorna *true* se foi adicionado ou *false* caso  $x$  já esteja no conjunto.
  - ▶ **boolean remove(x)** - remove o elemento  $x$  do conjunto. Retorna *true* se foi removido ou *false* caso  $x$  não esteja no conjunto.
  - ▶ **int size()** - retorna o número de elementos do conjunto.
  - ▶ **void clear()** - limpa o conjunto (torna-o vazio)
- Na altura era apenas um *IntSet* (números inteiros - ainda não conheciam *genéricos*) e vimos como implementá-lo usando... arrays
- As **árvores binárias de pesquisa** são outra estrutura de dados que suporta todas estas operações de um TAD Conjunto (literalmente existem métodos com o mesmo nome, com exceção do *add*, que chamamos de *insert*, e do *size*, que chamamos de *numberNodes*)

# TAD Conjunto - Comparação de Implementações

- Comparemos algumas implementações possíveis do **TAD Conjunto**.
- Para **qualquer tipo**, verificando igualdade com *equals*:
  - ▶ **Lista Desordenada**: array ou lista ligada contendo todos os elementos do conjunto sem nenhuma ordem. Procurar implica pesquisa linear.
- Para **tipos comparáveis** (onde se possa usar o *compareTo*)
  - ▶ **Lista Ordenada**: array por ordem crescente. Pode procurar-se com pesquisa binária, mas ao inserir/remover temos de manter ordenado.
  - ▶ **Arv. Pesquisa**: árvore binária de pesquisa. As operações dependem da altura da árvore, que tem:
    - ★ Valor esperado de  $\mathcal{O}(\log n)$  para dados "aleatórios" (pior caso é  $\mathcal{O}(n)$ )
    - ★ É possível garantir  $\mathcal{O}(\log n)$  para quaisquer dados usando estratégias de balanceamento (ex: AVL Trees, Red-Black Trees)
- Para **números inteiros** (vimos anteriormente, mas não é *genérica*)
  - ▶ **BooleanArray**: array de booleanos - posição  $x$  é *true/false* dependendo se o elemento  $x$  está ou não no conjunto.

# TAD Conjunto - Comparação de Implementações

- **n**: número de elementos do conjunto
- **k**: tamanho do maior elemento

	Complexidade Temporal					Complex. Espacial
	contains	insert	remove	size	clear	
<b>Lista Desordenada</b>	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$
<b>Lista Ordenada</b>	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$
<b>Arv. Pesquisa</b>	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$
<b>BooleanArray</b>	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(k)$	$\mathcal{O}(k)$

- Notem que  $\log n$  é muito mais pequeno que  $n$  (ex:  $\log(1 \text{ milhão})$  é  $\sim 20$ )
- A maioria das linguagens de programação tem disponível o TAD Conjunto precisamente como árvore binária de pesquisa (ex: Java TreeSet e C++ set)
- A última solução é mais rápida, mas gasta muita memória e não é genérica (é possível ter mais genérico e que ocupe menos memória e com operações em  $\mathcal{O}(1)$  usando *hash tables*: a sua ideia básica é ter maneira de converter valores em posições usando para isso uma função de *hash* - vão falar disto noutras UCs )

# Exemplo de aplicação do TAD Conjunto

- Imaginemos que temos disponível uma **lista de nomes de pessoas** (poderiam ser por exemplo alunos de EDados, ou actores de filmes)

persons.txt

```
Antonio Augusto Martins  
Ricardo Jorge Oliveira  
Carla Silva Barbosa  
Antonio Sousa Cunha  
Judite Pinto Rodrigues  
...
```

- Como poderíamos calcular **quantos primeiros nomes diferentes existem?** (ou quantos apelidos diferentes existem?)
  - ▶ Uma solução simples usando um **TAD conjunto**  $s$  seria simplesmente **inserir** todos nomes em  $s$  e no final verificar o **tamanho** de  $s$
  - ▶ **Eficiência** depende da implementação usada (da sua complexidade)

"Get your data structures correct first, and the rest of the program will write itself."  
*David Jones*

# Exemplo de aplicação do TAD Conjunto

- Contar quantos nomes existem resume-se agora a chamadas a métodos da BSTree em duas linhas de código:
  - ▶  $n$  **inserts** para adicionar os nomes ao conjunto
  - ▶ uma chamada a **numberNodes** no final

```
public static void main(String[] args) {
    Scanner in = new Scanner(System.in);

    // Usei <> em vez de <String> para mostrar "diamond notation": a partir
    // do Java 1.7 passou a ser possível fazer assim e o Java infere o tipo
    BSTree<String> set = new BSTree<>();

    while (in.hasNextLine()) {
        String line = in.nextLine();           // Ler uma linha
        String[] names = line.split(" ");     // Separar por palavras
        set.insert(names[0]);                 // Inserir primeiro nome
    }
    System.out.println("Nomes diferentes = " + set.numberNodes());
}
```

- E saber quantas **quantas ocorrências existem de cada nome?**
  - ▶ É preciso de associar "dados" a cada "elemento" do conjunto

## TAD Dicionário (ou Mapa, ou Array Associativo)

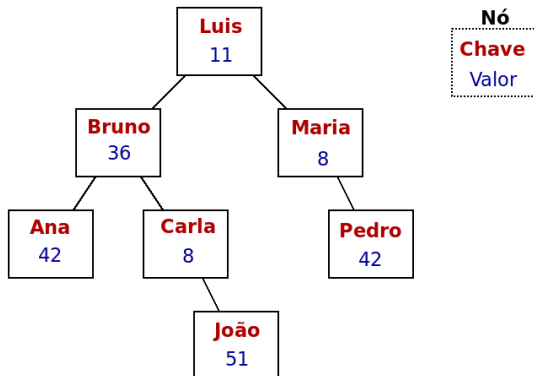
Um TAD Dicionário armazena pares (**chave, valor**). Uma chave apenas aparece uma vez e tem associada a si um valor.

- Metáfora de array associativo: `ocorrencias["pedro"] = 42`
- As operações típicas de um dicionário são:
  - ▶ **Adicionar** um par ao dicionário (chave e valor)
  - ▶ **Remover** um par do dicionário (dando a chave)
  - ▶ **Modificar** um par do dicionário (dar chave e novo valor)
  - ▶ **Devolver valor** associado a uma dada chave



# Implementação do TAD Dicionário

- Obviamente, também podemos usar **árvores binárias de pesquisa** para este efeito, com pequenas alterações ao código:
  - ▶ Cada nó vai ter como atributos uma chave e um valor
  - ▶ O elemento que determina a posição na árvore é a chave



Exemplo de um dicionário com chaves do tipo *String* e valores do tipo *Integer*

# Dicionário com Árvores de Pesquisa - Nó

- Vejamos como ficaria um **nó**:  
(implementação completa disponível no site)

```
// K é o tipo da chave (key) e V o tipo do valor (value)
// O tipo K tem de implementar o interface Comparable
public class BSTMapNode<K extends Comparable<? super K>, V> {
    private K key;           // chave
    private V value;        // valor
    private BSTMapNode<K,V> left; // Filho esquerdo
    private BSTMapNode<K,V> right; // Filho direito

    // Construtor
    BSTMapNode(K k, V v, BSTMapNode<K,V> l, BSTMapNode<K,V> r) {
        key = k;
        value = v;
        left = l;
        right = r;
    }

    // Getters e Setters habituais
    // ...
}
```

# Dicionário com Árvores de Pesquisa - BSTMap

- O dicionário em si é apenas uma árvore binária de pesquisa.
- Vamos espreitar um pouco da implementação e as assinaturas dos principais métodos: (implementação completa no site)

```
public class BSTMap<K extends Comparable<? super K>,V> {
    private BSTMapNode<K,V> root; // raiz da árvore

    BSTMap() {root = null;} // Construtor
    public boolean isEmpty() {return root == null;} // Dicionário vazio?
    public void clear() {root = null;} // Limpar dicionário
    public int size() { /*(...)*/ } // Número de chaves da árvore

    // Devolver o valor associado a uma chave (ou null caso não exista)
    public V get(K key) { /*(...)*/ }

    // Adicionar par (chave,valor) - se chave já existir, substituir valor antigo
    public void put(K key, V value) { /*(...)*/ }

    // Remover uma chave do dicionário - devolve true se conseguiu remover
    public boolean remove(K key) { /*(...)*/ }

    // Devolver lista ligada das chaves (usando listas do Java)
    public LinkedList<K> keys() { /*(...)*/ }
}
```

# Dicionário com Árvores de Pesquisa - get

- **get** é muito similar ao **contains** da *BSTree*
- **put** é muito similar ao **insert** da *BSTree*
- **remove** é muito similar ao **remove** da *BSTree*
- Vamos espreitar o **get()**:

```
// Devolver o valor associado a uma chave (ou null caso não exista)
public V get(K key) {
    return get(root, key);
}

private V get(BSTMapNode<K,V> n, K key) {
    if (n==null) return null;
    if (key.compareTo(n.getKey()) < 0) return get(n.getLeft(), key);
    if (key.compareTo(n.getKey()) > 0) return get(n.getRight(), key);
    return n.getValue(); // se não é menor ou maior, é porque é igual
}
```

# Dicionário com Árvores de Pesquisa - put

- Vamos espreitar o **put()**:

```
// Adicionar par (chave,valor) ao dicionário
// Se chave já existir, substitui o valor antigo pelo novo
public void put(K key, V value) {
    root = put(root, key, value);
}

private BSTMapNode<K,V> put(BSTMapNode<K,V> n, K key, V value) {
    if (n==null)
        return new BSTMapNode<K,V>(key, value, null, null);
    else if (key.compareTo(n.getKey()) < 0)
        n.setLeft(put(n.getLeft(), key, value));
    else if (key.compareTo(n.getKey()) > 0)
        n.setRight(put(n.getRight(), key, value));
    else
        n.setValue(value);
    return n;
}
```

- Vamos espreitar o **keys()**:
- Usamos listas ligadas do Java para ilustrar, mas poderíamos também ter usado as listas que criamos nesta UC

```
// Devolver lista ligada das chaves (usando listas do Java)
public LinkedList<K> keys() {
    LinkedList<K> list = new LinkedList<K>();
    keys(root, list);
    return list;
}

private void keys(BSTMapNode<K,V> n, LinkedList<K> l) {
    if (n==null) return;
    keys(n.getLeft(), l);
    l.addLast(n.getKey());
    keys(n.getRight(), l);
}
```

# Dicionário com Árvores de Pesquisa - Exemplo

- Vamos espreitar um exemplo de uso:

```
public class TestBSTMap {
    public static void main(String[] args) {
        // Criação de um dicionário
        BSTMap<String,Integer> map = new BSTMap<String,Integer>();

        // Inserindo alguns pares (chave,valor)
        map.put("Life", 42);
        map.put("Zero", 0);
        map.put("Infinity", 999);
        // Tamanho e conteúdo
        System.out.println("size = " + map.size());
        System.out.println("Value of \"Life\" = " + map.get("Life"));
        System.out.println("Value of \"Data\" = " + map.get("Data"));
        System.out.println(map.keys());

        // Modificando um valor
        map.put("Life", 22);
        System.out.println("Value of \"Life\" = " + map.get("Life"));
        // Apagando um valor
        map.remove("Life");
        System.out.println("Value of \"Life\" = " + map.get("Life"));
    }
}
```

# Frequência dos nomes usando Dicionários

- Como contar então a frequência dos vários nomes?
- Ter um dicionário que associa a cada nome uma contagem, ou seja, um dicionário de pares (*String,Integer*)

```
public static void main(String[] args) {
    Scanner in = new Scanner(System.in);

    // Criar dicionário de pares (String,Integer)
    BSTMap<String,Integer> map = new BSTMap<>();

    while (in.hasNextLine()) {
        String line = in.nextLine();           // Ler uma linha
        String[] names = line.split(" ");     // Separar por palavras
        Integer i = map.get(names[0]);        // Contagem actual
        if (i==null) map.put(names[0], 1);    // Se não existe, colocar a 1
        else      map.put(names[0], i+1);     // Se já existia, incrementar
    }

    LinkedList<String> names = map.keys();    // Ir buscar lista de nomes
    for (String s : names) // Percorrer lista e imprimir
        System.out.println(map.get(s) + " " + s);
}
```



# Classes do Java

- Vimos implementações nossas dos TADs conjunto e dicionário
- A linguagem Java também tem disponíveis estes TADs:
  - ▶ interface `Set<E>`:  
<https://docs.oracle.com/javase/8/docs/api/java/util/Set.html>
  - ▶ interface `Map<K,V>`:  
<https://docs.oracle.com/javase/8/docs/api/java/util/Map.html>
- Estes interfaces têm implementações com árv. binárias de pesquisa:
  - ▶ classe `TreeSet<E>`:  
<https://docs.oracle.com/javase/8/docs/api/java/util/TreeSet.html>
  - ▶ classe `TreeMap<T>`:  
<https://docs.oracle.com/javase/8/docs/api/java/util/TreeMap.html>
- A documentação diz que são usadas árvores equilibradas, garantindo tempos logarítmicos (imagem é excerto da documentação):

## Class `TreeMap<K,V>`

A **Red-Black tree** based `NavigableMap` implementation. The map is sorted according to the **natural ordering** of its keys, or by a `Comparator` provided at map creation time, depending on which constructor is used.

This implementation provides **guaranteed  $\log(n)$**  time cost for the `containsKey`, `get`, `put` and `remove` operations. Algorithms are adaptations of those in Cormen, Leiserson, and Rivest's *Introduction to Algorithms*.

# Exemplo de uso das classes do Java

- O programa anterior com as classes do Java:

```
public static void main(String[] args) {
    Scanner in = new Scanner(System.in);

    // Criar dicionário de pares (String,Integer)
    Map<String,Integer> map = new TreeMap<>();

    while (in.hasNextLine()) {
        String line = in.nextLine();           // Ler uma linha
        String[] names = line.split(" ");     // Separar por palavras
        Integer i = map.get(names[0]);        // Contagem actual
        if (i==null) map.put(names[0], 1);    // Se não existe, colocar a 1
        else      map.put(names[0], i+1);     // Se já existia, incrementar
    }
    Set<String> names = map.keySet();         // Ir buscar conjunto de nomes
    for (String s : names) // Percorrer conjunto e imprimir
        System.out.println(map.get(s) + " " + s);
}
```