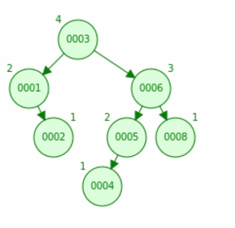
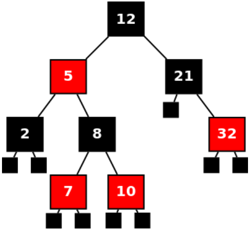
# Árvores Binárias de Pesquisa Equilibradas

Pedro Ribeiro

DCC/FCUP

2021/2022

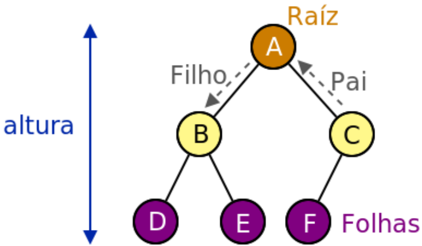


# Árvores Binárias de Pesquisa - Motivação

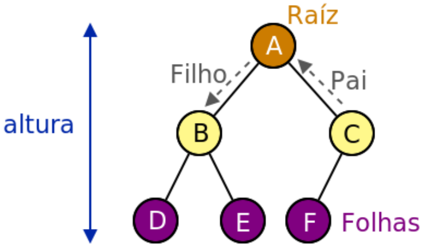
- Seja $S$ um conjunto de objectos/itens **"comparáveis"**:
  - Sejam $a$ e $b$ dois objectos.
    São "comparáveis" se for possível dizer se $a < b$, $a = b$ ou $a > b$.
  - Um exemplo seriam números, mas poderiam ser outra coisa (alunos com um nome e nº mecanográfico; equipas com pontos, golos marcados 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 $S$
  - Dado um conjunto $S$ **dinâmico** determinar os elementos que estão contido num intervalo $[a, b]$
  - **Ordenar** um conjunto S
  - ...
- **Árvores Binárias de Pesquisa!**

# Árvores Binárias - 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*

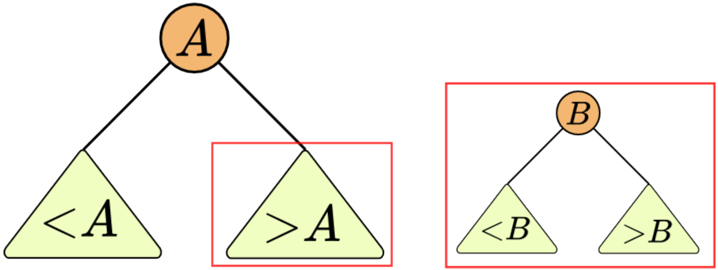
# Árvores Binárias - Terminologia




- 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 raíz 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*

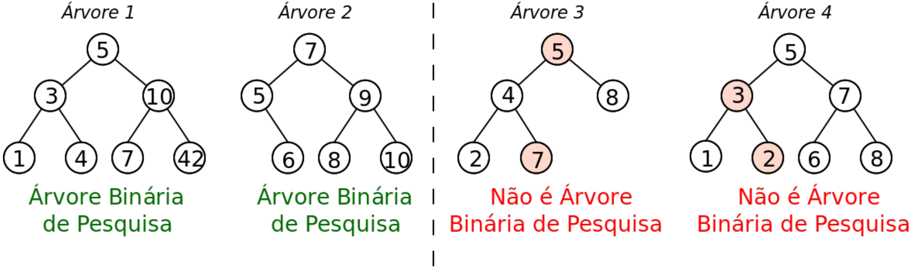
# Árvores Binárias de Pesquisa - Conceito

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



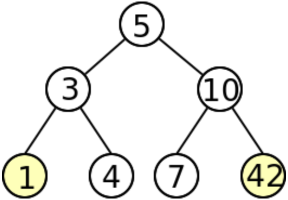
# Árvores Binárias de Pesquisa - Exemplos

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



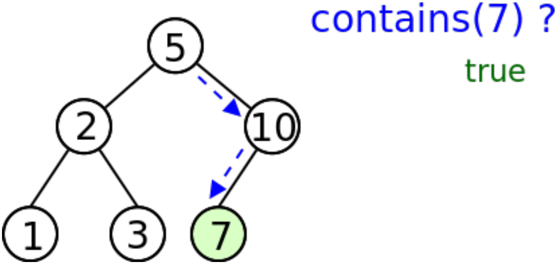

- Nas árvores 1 e 2 as condições são respeitadas
- Na árvore 3 o nó 7 está à esquerda do nó 5 mas $7 > 5$
- Na árvore 4 o nó 2 está à direito do nó 3 mas $2 < 3$

# Árvores Binárias de Pesquisa
## Consequências



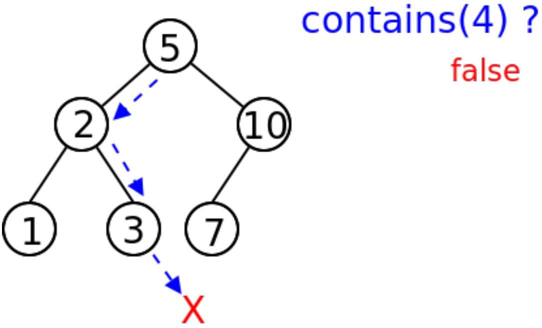

- O **menor** valor de todos está... no **nó mais à esquerda**
- O **maior** valor de todos está... no **nó mais à direita**

# Árvores Binárias de Pesquisa - Pesquisar um valor




- Começar na raíz, 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 - Pesquisar um valor




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

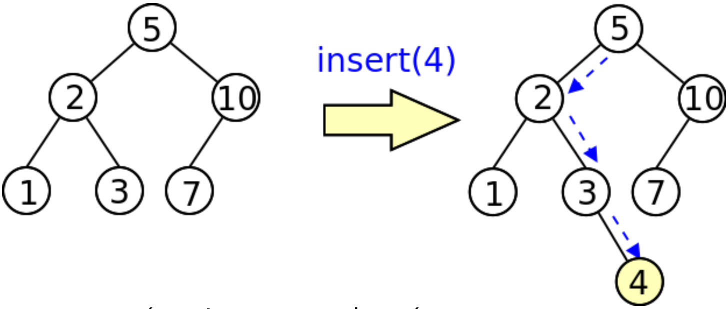
# Árvores Binárias de Pesquisa - Pesquisar um valor

- **Pesquisar valores** em árvores binárias de pesquisa:

**Pesquisa numa árvore binária de pesquisa**

**contains**$(T, v)$:
  **Se Nulo**$(T)$ **então**
    **devolver** *false*
  **Senão se** $v < T.valor$ **então**
    **devolver contains**$(T.esquerdo, v)$
  **Senão se** $v > T.valor$ **então**
    **devolver contains**$(T.direito, v)$
  **Senão**
    **devolver** *true*

# Árvores Binárias de Pesquisa - Inserir um valor




- Começar na raíz, 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 $>$)

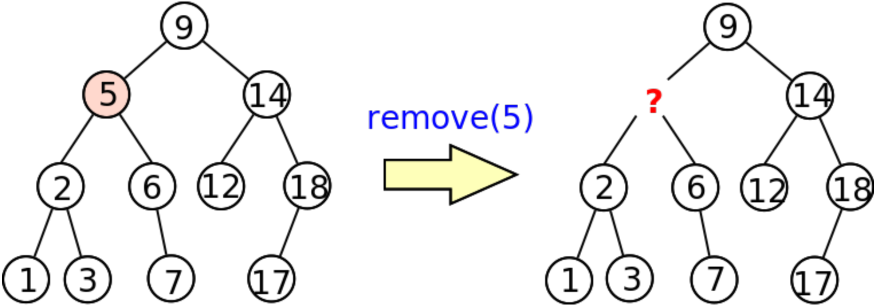
# Árvores Binárias de Pesquisa - Inserir um valor

- **Inserir valores** em árvores binárias de pesquisa:

**Inserir numa árvore binária de pesquisa**

**insert**($T, v$):
    **Se Nulo**($T$) **então**
        **devolver Novo Nó**(v)
    **Senão se** $v < T.valor$ **então**
        **devolver** $T.esquerdo =$ **insert**($T.esquerdo, v$)
    **Senão se** $v > T.valor$ **então**
        **devolver** $T.direito =$ **insert**($T.direito, v$)
    **Senão**
        **devolver** T

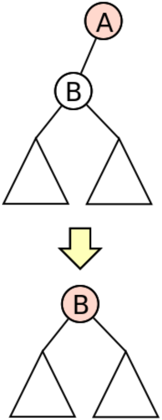
# Árvores Binárias de Pesquisa - Remover um valor

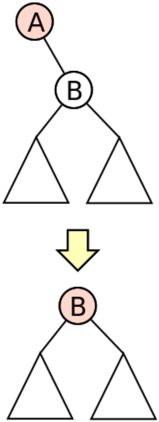


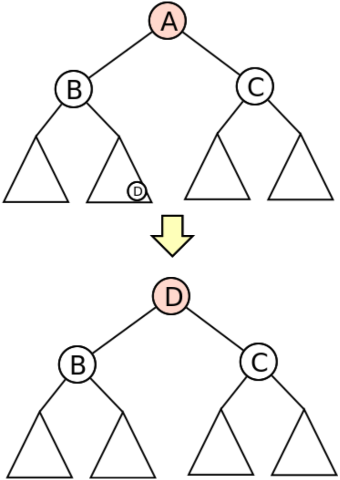

- Começar na raíz, e ir percorrendo a árvore até encontrar o valor
- Ao retirar o valor o que fazer a seguir?
  - Se o nó que retiramos só tiver um ramo filho, basta *"subir"* esse filho até à posição correspondente
  - Se tiver dois ramos filhos, os candidatos a ficarem nessa posição são:
    - O maior 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**
  - Casos possíveis (se náo não for folha):

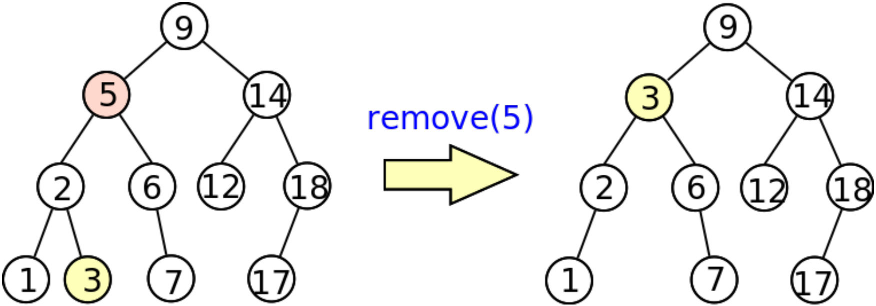


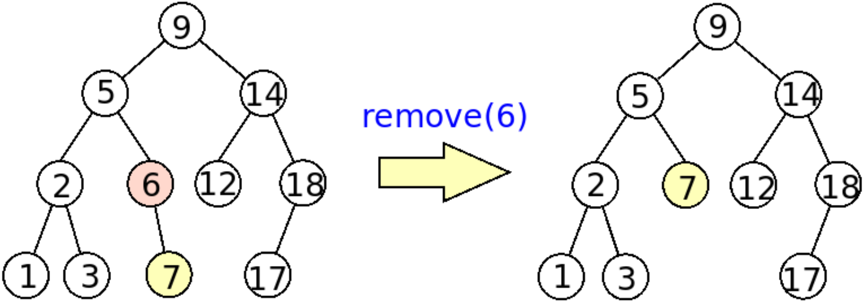
# Árvores Binárias de Pesquisa - Remover um valor

## Exemplo com dois filhos



# Árvores Binárias de Pesquisa - Remover um valor

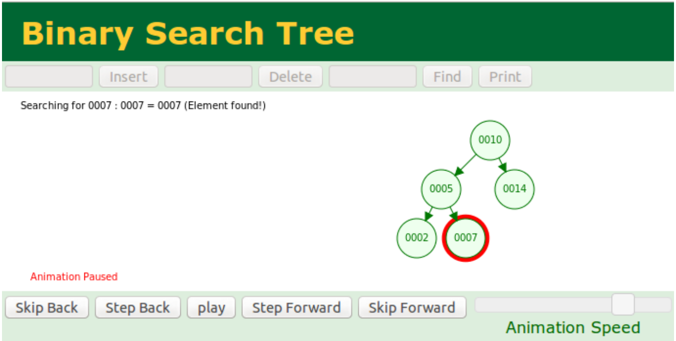
## Exemplo só com um filho



# 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



# Árvores Binárias de Pesquisa - Complexidade

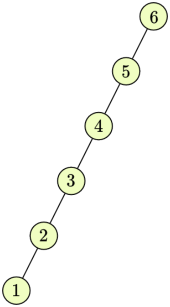
- Como caracterizar o **tempo que cada operação demora**?
  - Todas as operações procuram um nó percorrendo a **altura** da árvore

**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 $\mathcal{O}(h)$.

# Desequilíbrio numa Árvore Binária de Pesquisa

- O **problema** do método anterior:

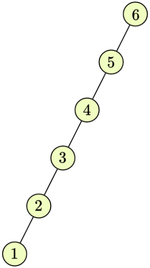


A altura da árvore pode ser da ordem de $\mathcal{O}(n)$ ($n$, número de elementos)

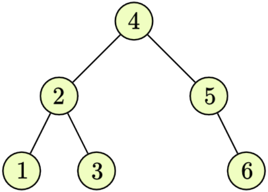
*(a altura depende da ordem de inserção e existem ordens "más")*

# Árvores equilibradas

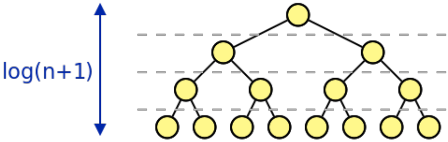
- Queremos árvores... **equilibradas**



vs



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

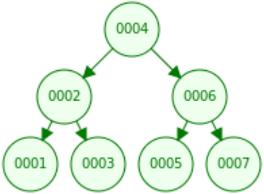



| Nós no nível | Total de Nós |
|---|---|
| 1 = 2^0 | 1 = 2^1 - 1 |
| 2 = 2^1 | 3 = 2^2 - 1 |
| 4 = 2^2 | 7 = 2^3 - 1 |
| 8 = 2^3 | 15 = 2^4 - 1 |

# Árvores equilibradas

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

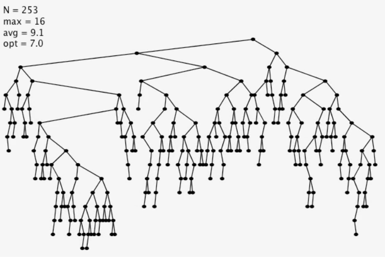
**Resposta**: *"pesquisa binária"* - se os números estiverem ordenados, inserir o elemento do meio, partir a lista restante em duas nesse elemento e inserir os restantes elementos de cada metade pela mesma ordem



# Altura para uma ordem aleatória

**Altura de uma árvore com elementos aleatória**

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

# Estratégias de Balanceamento

- E se não soubermos os elementos todos à partida e tivermos de **dinamicamente** ir inserindo e removendo elementos?
  *(e querendo precaver contra "más ordens" de inserção/remoção*

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

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

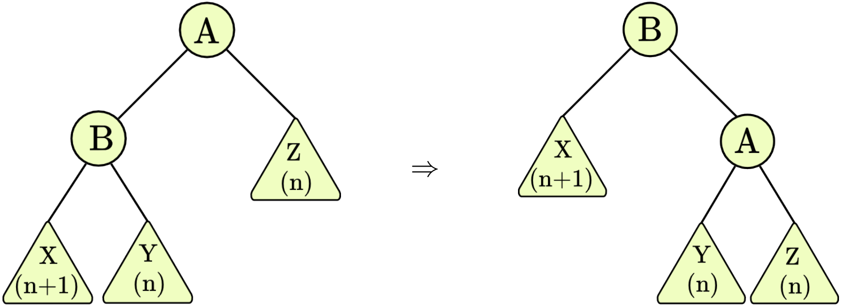
- **AVL Trees**
- **Red-Black Trees**
- Splay Trees
- Treaps

Outras estruturas de dados:

- Skip List
- Hash Table
- Bloom Filter

# Estratégias de Balanceamento

- Caso simples: **como balancear** a árvore seguinte (entre parentesis está a altura):

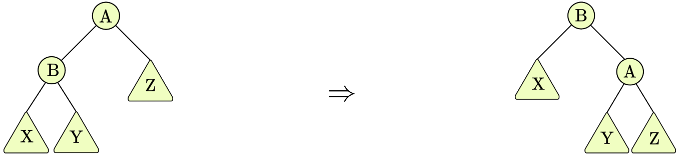



Esta operação base chama-se de **rotação à direita**

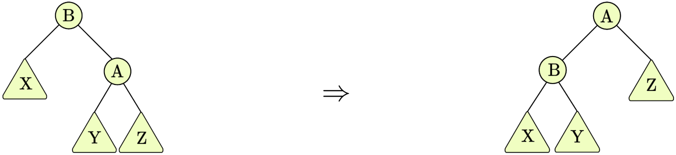
# Estratégias de Balanceamento

- As operações de rotação relevantes são as seguintes:
  - Note que é preciso não quebrar a condição de ser árvore binária de pesquisa

Rotação à direita



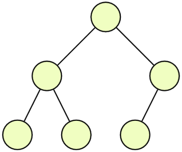

Rotação à esquerda



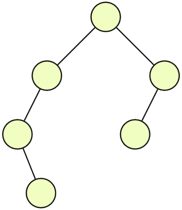
# Árvores AVL

## Árvore AVL

É uma árvore binária de pesquisa que garante que para cada nó da árvore, a altura da subárvore da esquerda e da subárvore da direita **diferem no máximo em uma unidade** (**invariante de altura**).



Árvore AVL

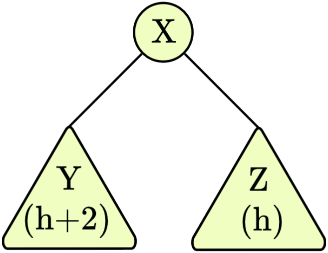


Não Árvore AVL

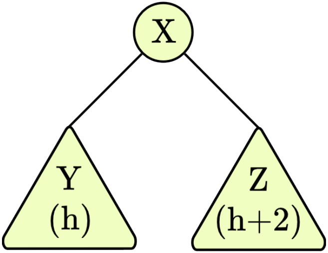
- Quando se inserem ou removem nós, alteramos a árvore de forma a manter o invariante da altura

# Árvores AVL

- **Inserir** numa árvore AVL funciona como inserir numa árvore binária de pesquisa qualquer, porém, a árvore pode deixar de ser balanceada (de acordo com o invariante de altura)
- Os seguintes casos podem ocorrer:




Desnível de 2 para a esquerda

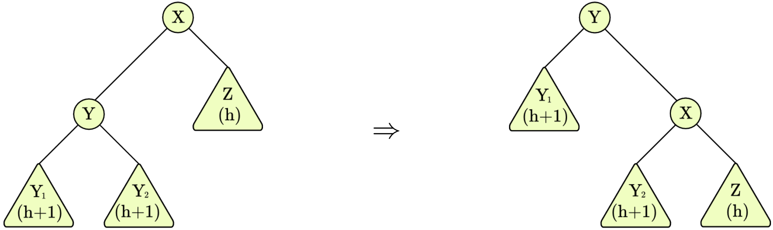



Desnível de 2 para a direita

- Vejamos como corrigir o primeiro com as rotações simples, **corrigir o segundo é análogo** mas com as rotações para o lado contrário

# Árvores AVL

- Dentro do primeiro caso, temos duas possibilidades da forma da árvore AVL
- A primeira:

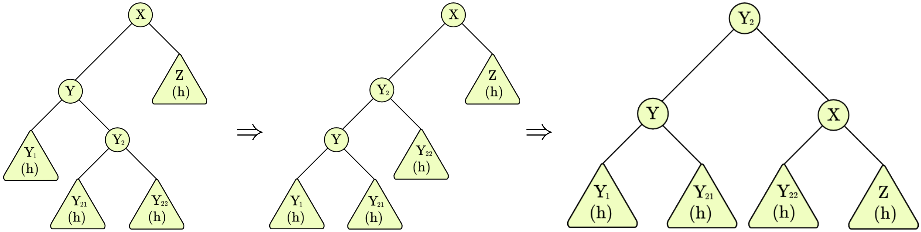



**Correção à esquerda, caso 1**

Corrige-se efetuando uma rotação para a direita a começar em $X$

- Nota: a altura de $Y_2$ pode ser $h+1$ ou $h$, esta correção funciona para ambos os casos

- A segunda:




**Correção à esquerda, caso 2**

Corrige-se efetuando uma rotação para a esquerda a começar em $Y$, seguida de uma rotação à direita a começar em $X$

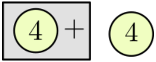
- Nota: a altura de $Y_{12}$ **ou** de $Y_{22}$ pode ser $h$ ou $h-1$, esta correção funciona para ambos os casos

# Árvores AVL

- Ao inserir nós **causamos desníveis** nas alturas das subárvores de nós
- Para os corrigir, aplicam-se operações de rotação **ao longo do caminho** onde foi inserido o novo nó
- Existem **dois tipos de desnível**: um à esquerda e um à direita, análogos
- Cada tipo de desnível tem **dois casos possíveis**, que se resolvem com diferentes aplicações de rotações

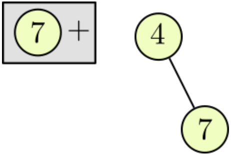
# Árvores AVL

- **Exemplo** de inserções de nós:



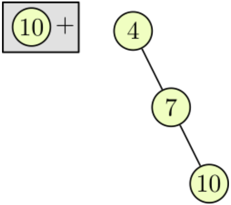
# Árvores AVL

- **Exemplo** de inserções de nós:



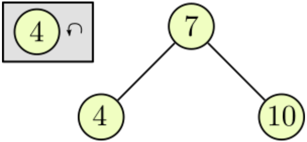
# Árvores AVL

- **Exemplo** de inserções de nós:



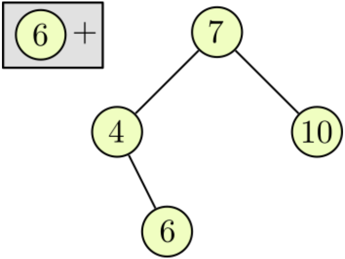
# Árvores AVL

- **Exemplo** de inserções de nós:



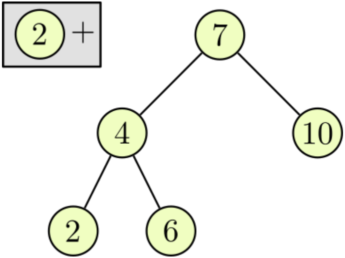
# Árvores AVL

- **Exemplo** de inserções de nós:



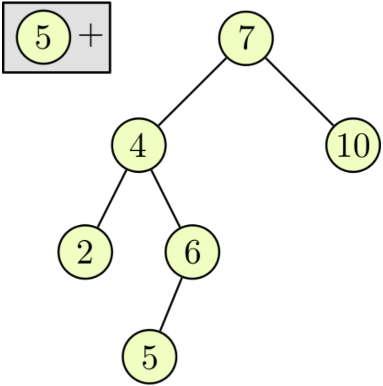
# Árvores AVL

- **Exemplo** de inserções de nós:



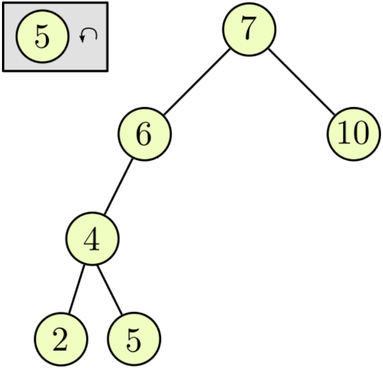
# Árvores AVL

- **Exemplo** de inserções de nós:



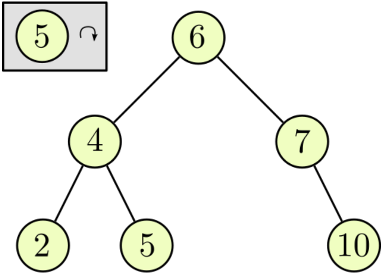
# Árvores AVL

- **Exemplo** de inserções de nós:



# Árvores AVL

- **Exemplo** de inserções de nós:



# Árvores AVL

- Para **remover elementos**, aplicamos a mesma ideia da inserção
- Primeiro, encontra-se o nó a remover
- Aplica-se uma das modificações vistas para o caso das árvores binárias de pesquisa
- Aplicam-se rotações conforme os casos analizados ao longo do caminho do nó removido até à raiz

# Árvores AVL

- Para a operação de **procura**, apenas se percorre no máximo a altura da árvore
- Para a operação de **inserção**, percorre-se a altura da árvore e aplicam-se no máximo duas rotações (porquê só duas?), que duram tempo $\mathcal{O}(1)$
- Para a operação de **remoção**, percorre-se a altura da árvore e aplicam-se no máximo duas rotações por cada nó no caminho da altura
- Conclui-se que a complexidade de cada operação é $\mathcal{O}(h)$, onde $h$ é a altura da árvore

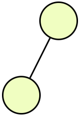
Qual é a altura máxima da árvore?

# Árvores AVL

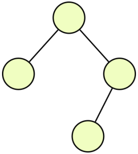
- Para calcular o **pior caso** da altura de uma árvore a qualquer momento faremos o seguinte exercício:
  - Qual a menor árvore AVL (válida segundo o invariante da altura) com altura exatamente $h$?
  - Chamaremos ao número de nós numa árvore com altura $h$ de $N(h)$

# Árvores AVL

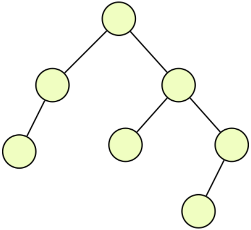
Altura 0



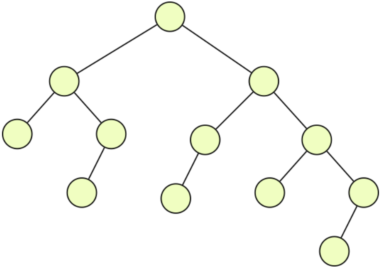
Altura 1



Altura 2



Altura 3



Altura 4

# Árvores AVL

- Sumarizando:
  - $N(0) = 1$
  - $N(1) = 2$
  - $N(2) = 4$
  - $N(3) = 7$
  - $N(4) = 12$
  - $\ldots$
  - $N(h) = N(h-2) + N(h-1) + 1$
- Tem um comportamento semelhante à da sequência de Fibonacci!
- Recordando das aulas de álgebra linear:
  - $N(h) \approx \phi^h$
  - $\log(N(h)) \approx \log(\phi)h$
  - $h \approx \frac{1}{\log(\phi)} \log(N(h))$

A altura $h$ de uma árvore AVL com $n$ nós obedece a: $h \leq 1.44 \log(n)$

# Árvores AVL

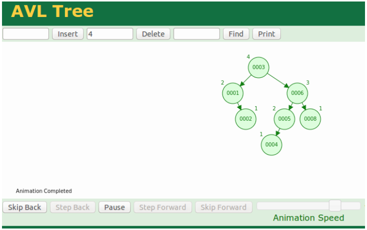
- **Vantagens** das árvores AVL:
  - Operações de pesquisa, inserção e remoção com complexidade garantida de $\mathcal{O}(\log n)$;
  - Pesquisa muito eficiente (em relação a outras estruturas semelhantes), pois o limite para a altura de $1.44 \log(n)$ é pequeno;
- **Desvantagens** das árvores AVL:
  - Implementação complexa (podemos simplificar a remoção usando uma técnica de *lazy delete*, semelhante à ideia de reconstruir);
  - Implementação requer mais dois *bits* de memória por nó (para guardar o desnível de alturas);
  - Inserção e remoção menos eficientes (em relação a outras estruturas semelhantes) por ter de garantir uma altura máxima menor;
  - As rotações alteram frequentemente a estrutura da árvore (o que não é bom para estruturas que sejam guardadas em disco);

# Árvores AVL

- O nome AVL vem dos autores: G. **A**delson-**V**elsky e E. **L**andis. O artigo original que as descreve é de 1962 (*"An algorithm for the organization of information"*, Proceedings of the USSR Academy of Sciences)

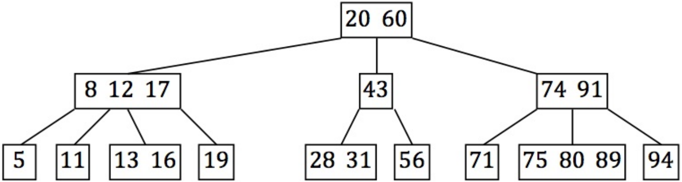
- Podem usar um visualizador de AVL Trees para "brincar" um pouco com o conceito e verem como são feitas as inserções, remoções, rotações, etc.

  https://www.cs.usfca.edu/~galles/visualization/AVLtree.html



# Árvores *Red-Black*

- Vamos explorar agora um outro tipo de árvores binárias "equilibradas" conhecidas como **red-black** trees
- Este tipo de árvores surgiu como uma "adaptação" da ideia das **árvores 2-3-4** para árvores binárias



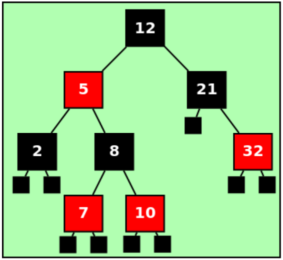

- O artigo original é de 1978 e foi escrito por L. Guibas e R. Sedgewick (*"A Dichromatic Framework for Balanced Trees"*)
- Os autores dizem que se usaram as cores preta e vermelha porque eram as que ficavam melhor quando impressas e eram as cores das canetas que tinham para desenhar as árvores :)

# Árvores *Red-Black*

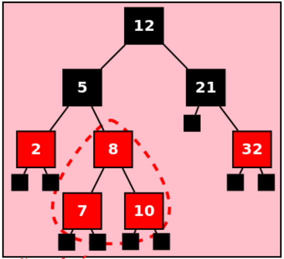
## Árvore *Red-Black*

É uma árvore binária de pesquisa onde cada nó é preto ou vermelho e:

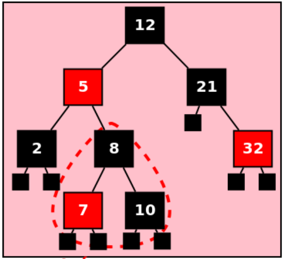
- **(root property)** A raíz da árvore é preta
- **(leaf property)** As folhas são nós (nulos/vazios) pretos
- **(red property)** Os filhos de um nó vermelho são pretos
- **(black property)** Para cada da nó, um caminho para qualquer uma das suas folhas descendentes tem o mesmo número de nós pretos




Árvore Red-Black



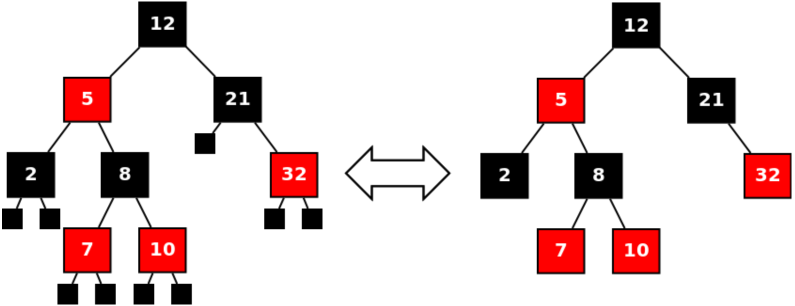

Não é Árvore Red-Black
("red property" não respeitada)




Não é Árvore Red-Black
("black property" não respeitada)

# Árvores *Red-Black*

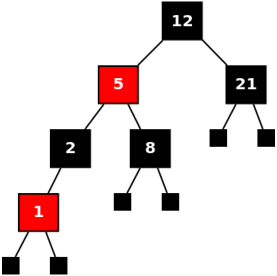
- Por uma questão visual, por vezes as imagens mostradas podem não conter os nós "nulos", mas podem assumir que eles existem
  Aos nós não nulos chamamos de **nós internos**.




- O nº de nós pretos no caminho de um nó $n$ até às suas folhas (não incluindo o próprio nó) é conhecido como **black height** e pode ser escrito como $bh(n)$
  - Ex: $\rightarrow bh(12) = 2$ e $bh(21) = 1$

# Árvores *Red-Black*

- Que tipo de "balanceamento" garantem as restrições dadas?
- Se $bh(n) = k$, então um caminho do nó $n$ até uma folha tem:
  - No mínimo $k$ nós (todos pretos)
  - No máximo $2k$ nós (alternando vermelho e preto)
    *[relembra que não podem existir dois nós vermelhos seguidos]*
- A altura de um ramo pode então ser duas vezes maior que a de um ramo irmão (mas não mais que isso)
  *[nas árvores AVL a diferença máxima de alturas não excedia 1]*



# Red-Black Trees

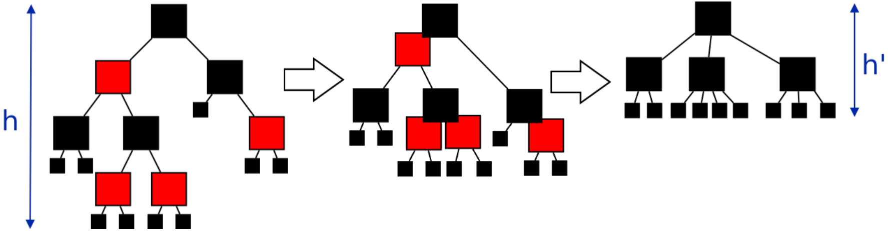
**Teorema - Altura de uma árvore *Red-Black***

Uma árvore red-black com $n$ nós tem altura $\mathbf{h} \leq \mathbf{2} \times \log_2(\mathbf{n}+\mathbf{1})$

*[ou seja, a altura h de uma red-black tree é* $\mathcal{O}(\log n)$*]*

**Intuição:**

Vamos fazer *merge* dos nós "vermelhos" com os seus pais "pretos":

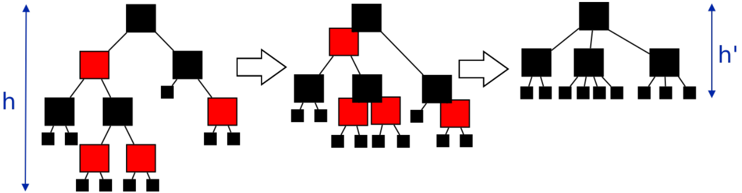



- Este processo produz uma árvore onde os nós têm 2, 3 ou 4 filhos
- Esta árvore 2-3-4 tem as folhas a uma altura uniforme de **h'** (onde h' é a *black height*)

**Teorema - Altura de uma árvore *Red-Black***

Uma árvore red-black com $n$ nós tem altura $\mathbf{h} \leq \mathbf{2} \times \log_{\mathbf{2}}(\mathbf{n}+\mathbf{1})$

*[ou seja, a altura h de uma red-black tree é $\mathcal{O}(\log n)$]*




- A altura da árvore é pelo menos metade da original: $h' \geq h/2$
- Uma árvore binária completa de altura $h'$ tem $2^{h'} - 1$ nós internos (não nulos)
- O número de nós da nova árvore é $\geq 2^{h'} - 1$ (é uma árvore 2-3-4)
- A árvore original tinha ainda mais nós que esta nova árvore: $n \geq 2^{h'} - 1$
- $n + 1 \geq 2^{h'}$
- $\log_2(n+1) \geq h' \geq h/2$
- $h \leq 2\log_2(n+1)$ $\square$

# Árvores *Red-Black*

- Como fazer uma **inserção**?

**Inserção de um nó numa árvore *red-black* não vazia**

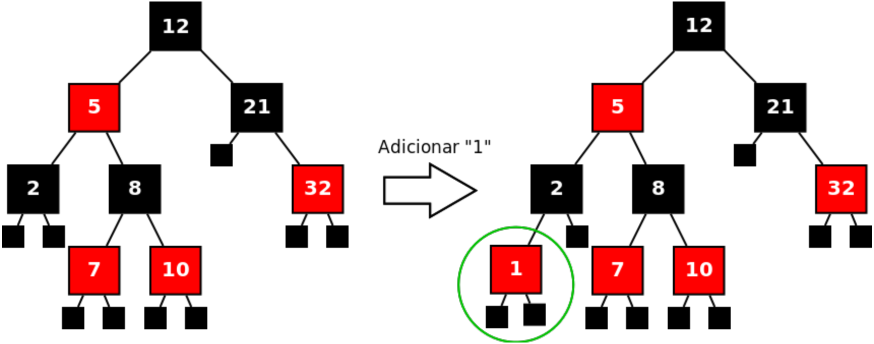
- Inserir como numa qualquer árvore binária de pesquisa
- Colorir o nó inserido de vermelho (acrescentando os nós folha "nulos")
- Recolorir e restruturar se necessário (restaurar invariantes)

- Como a árvore é não vazia não violamos a **root property**
- Como o nó inserido é vermelho não violamos a **black property**
- A única invariante que pode ser quebrada é a **red property**
  - Se o pai do elemento inserido for **preto** não é preciso fazer nada
  - Se o pai for **vermelho** ficamos com dois vermelho seguidos

# Árvores *Red-Black*

Quando o pai do nó inserido é um nó **preto** não é preciso fazer nada:

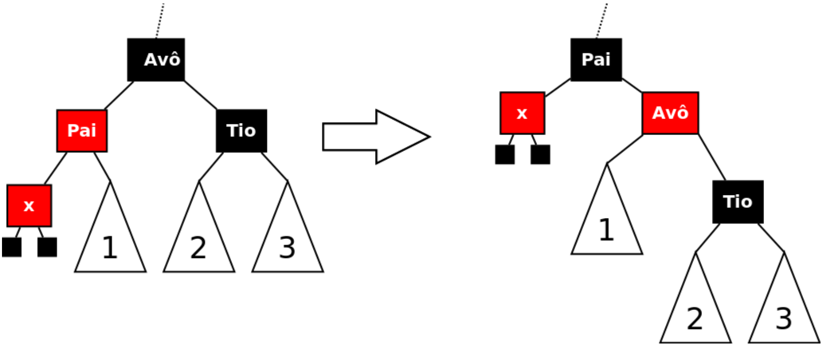
Exemplo:



# Árvores *Red-Black*

Situação de vermelho-vermelho depois de inserção (pai vermelho)

- Caso 1.a) O tio é um nó **preto** e o nó inserido $x$ é filho esquerdo

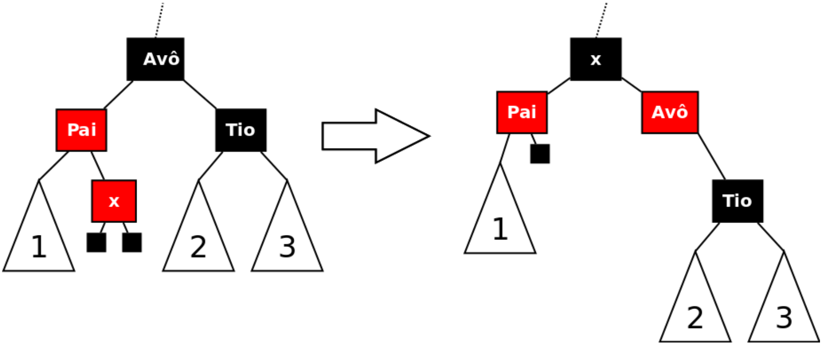



Descrição: rotação de avô à direita seguida de troca de cores entre pai e avô

# Árvores *Red-Black*

Situação de vermelho-vermelho depois de inserção (pai vermelho)

- Caso 1.b) O tio é um nó **preto** e o nó inserido $x$ é filho direito



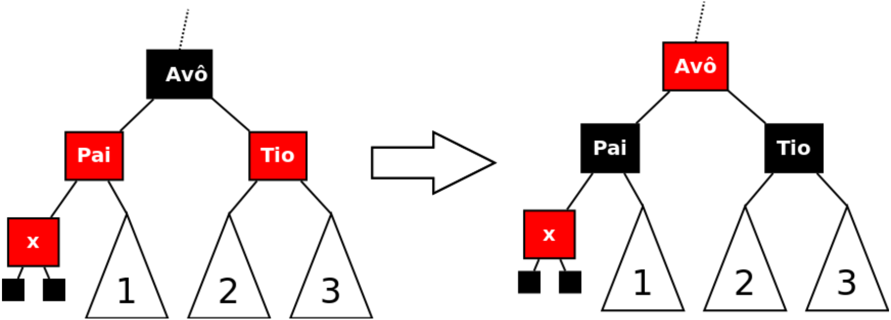

Descrição: rotação à esquerda de pai, seguida dos movimentos de 1.a

[Se o pai fosse o filho direito do avô tínhamos casos semelhantes mas simétricos em relação a estes]

# Árvores *Red-Black*

Situação de vermelho-vermelho depois de inserção (pai vermelho)

- Caso 2: O tio é um nó **vermelho**, sendo $x$ o nó inserido



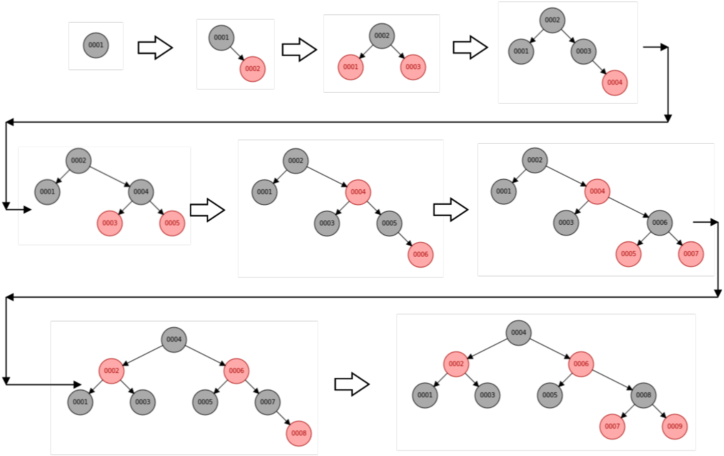

Descrição: trocar cores de pai, tio e avô

Agora, se o pai do avô for vermelho temos nova situação de vermelho-vermelho e basta voltar a aplicar um dos casos que já conhecemos (se avô for raíz, colocamos a preto)

# Árvores *Red-Black*

- Vamos visualizar algumas inserções (experimentem o url indicado):

  https://www.cs.usfca.edu/~galles/visualization/RedBlack.html



# Árvores *Red-Black*

- O custo de uma **inserção** é portanto $\mathcal{O}(\log n)$
  - $\mathcal{O}(\log n)$ para chegar ao local a inserir
  - $\mathcal{O}(1)$ para eventualmente recolorir e re-estruturar

- As **remoções** são parecidas em espírito mas um pouco mais complicadas, sendo que gastam também $\mathcal{O}(\log n)$ (não vamos detalhar aqui na aula - podem experimentar visualizar)

# Árvores *Red-Black*

- **Comparação** de árvores Red-Black (RB) com árvores AVL
  - Ambas são implementações de árvores binárias de pesquisa balanceadas (pesquisa, inserção e remoção em $\mathcal{O}(\log n)$)
  - RB são um pouco menos balanceadas no pior caso
    RB com altura $\sim 2\log(n)$ *vs* AVL com altura $\sim 1.44\log(n)$
  - RB demoram um pouco mais a pesquisar elementos
    (no pior caso, por causa da altura)
  - RB são um pouco mais rápidas a inserir/remover
    (rebalanceamento mais "leve")
  - Ocupam um pouco menos de memória
    (RB só precisam da cor, AVL precisam do desnível)
  - RB são (provavelmente) mais usadas nas linguagens usuais
    Exemplos de estruturas de dados que usam RB:
    - C++ STL: set, multiset, map, multiset
    - Java: java.util.TreeMap , java.util.TreeSet
    - Linux kernel: scheduler, linux/rbtree.h

# Outros tipos de árvores

- Existem muitos mais tipos de árvores de pesquisa ou outras estruturas de dados com o mesmo tipo de finalidade (*find, insert, remove*)
- Um exemplo são as **splay trees** (com um comportamento adaptativo):
  - Quando um elemento é procurado ou inserido, fica no topo da árvore
  - Para isso é usada uma operação chamada de *splay* (semelhante a rotações sucessivas para trazer o elemento para a raíz)
  - Se um elemento for frequentemente acedido, gasta-se menos para chegar a ele. Isto pode ser útil em várias situações.
    Ex: um router precisa de converter IPs em conexões físicas de saída. Quando um pacote com um IP chega, é provável que o mesmo IP volte a aparecer muitos vezes nos próximos pacotes.

    Espreitem uma visualização:

    https://www.cs.usfca.edu/~galles/visualization/SplayTree.html

# Uso em C/C++ e Java

- Qualquer linguagem "que se preze" tem a sua implementação de árvores binárias de pesquisa equilibradas
- Na próxima aula prática terão oportunidade de interagir com essas APIs e com código exemplo
- As principais estruturas de dados são:
  - **set**: inserir, remover e procurar elementos
  - **multiset**: um *set* com possibilidade de ter elementos repetidos
  - **map**: array associativo (associa uma chave a um valor)
    ex: associar *strings* a *ints*)
  - **multimap**: um *map* com possibilidade de ter chaves repetidas
- Os nós podem conter quaisquer tipos desde que sejam **comparáveis**
- Como existe ordem, podem-se usar **iteradores** para percorrer as árvores de forma ordenada.