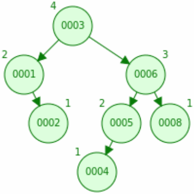
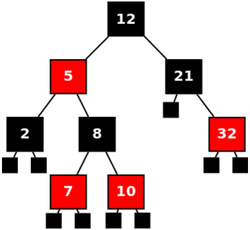
Árvores Binárias de Pesquisa Equilibradas

P. Ribeiro & P. Paredes

DCC/FCUP

2016/2017



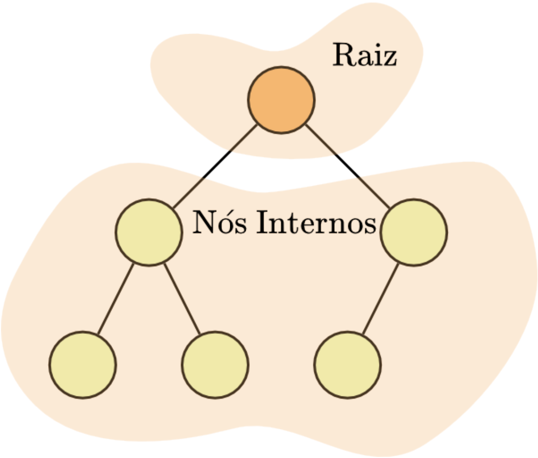


Árvores Binárias de Pesquisa

- **Problemas** de interesse:
  - Dada uma lista de números S, determinar se **um dado número está em** S
  - Dada uma lista de números S **dinâmica** (que sofre alterações: adições e remoções), determinar se **um dado número está em** S
  - Dada uma lista de números S **dinâmica** (que sofre alterações: adições e remoções), determinar se **o maior número de** S
  - **Ordenar** uma lista de números
- **Árvores Binárias de Pesquisa!**

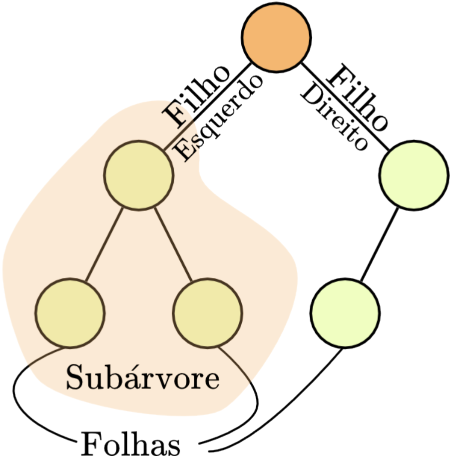
Árvores Binárias de Pesquisa

- Resumo da **notação** de árvores binárias:



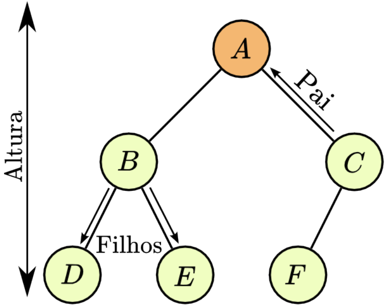
Árvores Binárias de Pesquisa

- Resumo da **notação** de árvores binárias:



Árvores Binárias de Pesquisa

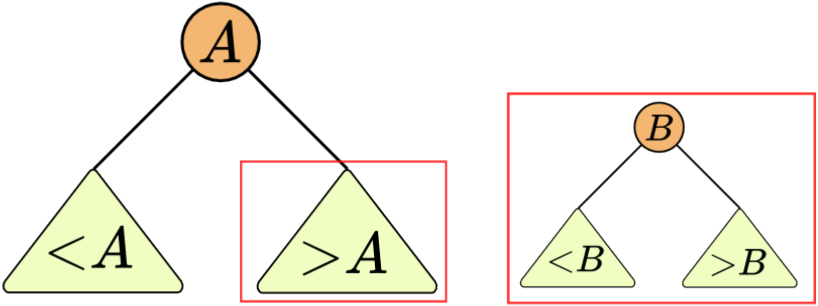
- Resumo da **notação** de árvores binárias:



- O nó A é pai do nó C
- Os nós D e E são filhos do nó B
- O nó B é irmão do C
- ...

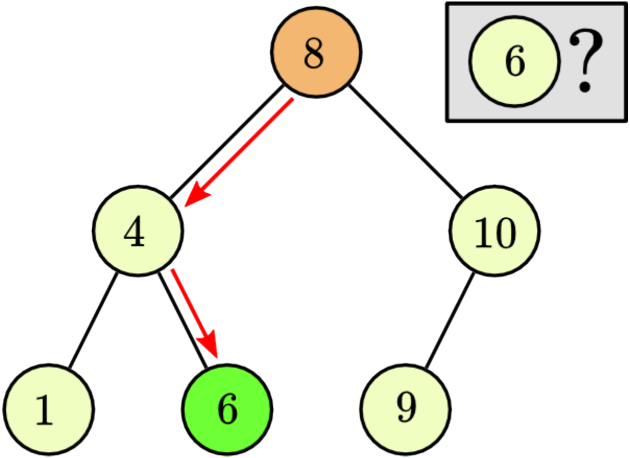
Árvores Binárias de Pesquisa

- Resumo de **árvores binárias de pesquisa**:



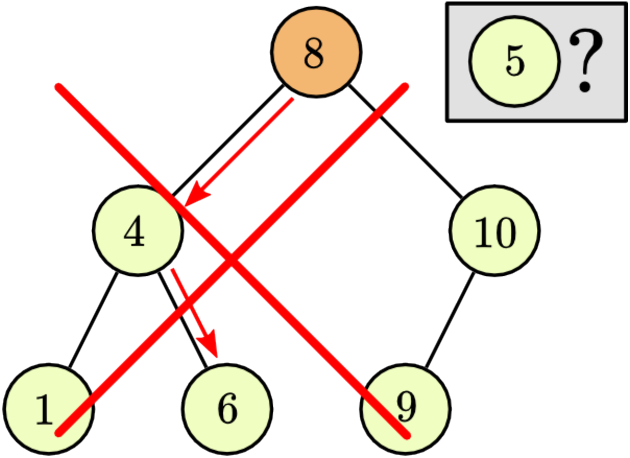
Árvores Binárias de Pesquisa

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



Árvores Binárias de Pesquisa

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



Árvores Binárias de Pesquisa

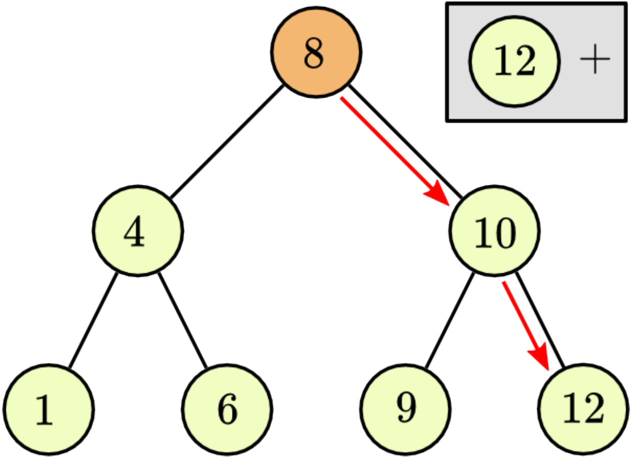
- **Pesquisar valores** em árvores binárias de pesquisa:

Pesquisa numa árvore binária de pesquisa

Pesquisa(T, v**)**:
  Se Nulo(T**) então**
    retorna $NÃO$
  Se $v < T.valor$ **então**
    retorna Pesquisa($T.esquerdo, v$**)**
  Senão $v > T.valor$ **então**
    retorna Pesquisa($T.direito, v$**)**
  Senão
    retorna SIM

Árvores Binárias de Pesquisa

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



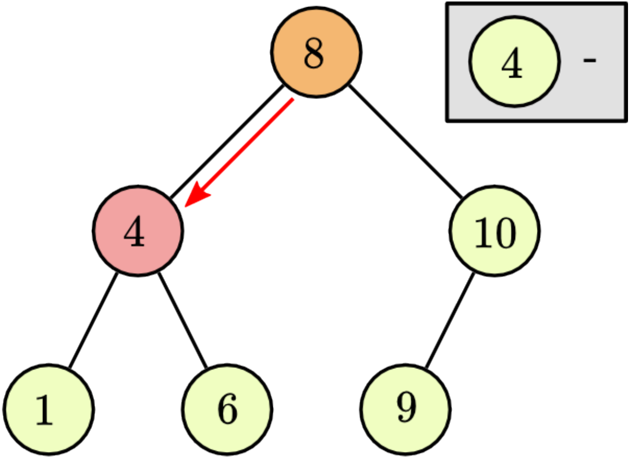
- **Inserir valores** em árvores binárias de pesquisa:

Inserir numa árvore binária de pesquisa

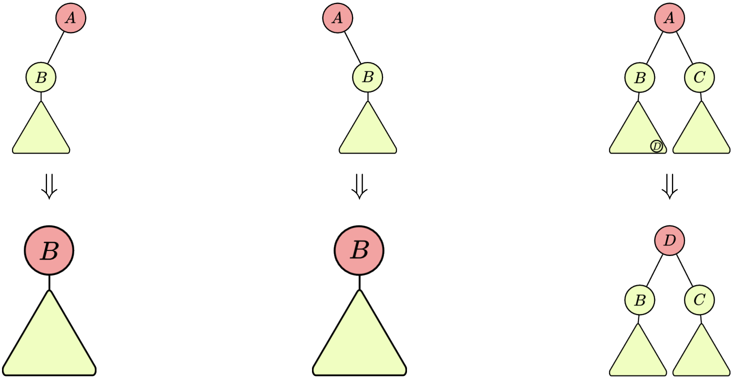
Insere(T, v**)**:
  Se Nulo(T**) então**
    retorna Novo No(v**)**
  Se $v < T.valor$ **então**
    retorna $T.esquerdo =$ **Insere(**$T.esquerdo, v$**)**
  Senão $v > T.valor$ **então**
    retorna $T.direito =$ **Insere(**$T.direito, v$**)**
  Senão
    retorna T

Árvores Binárias de Pesquisa

- **Remover valores** de árvores binárias de pesquisa:



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



Árvores Binárias de Pesquisa

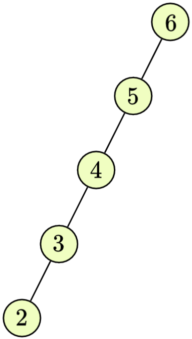
- Como caracterizar o **tempo que cada operação demora**?
  - Todas as operações procuram um nó percorrendo a **altura** da árvore

Complexidade de operações numa árvore binária de pesquisa

Seja H a altura de uma árvore binária de pesquisa T, a complexidade de efetuar uma pesquisa, uma inserção ou uma remoção em T é $\mathcal{O}(H)$.

Estratégias de Balanceamento

- O **problema** do método anterior:



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

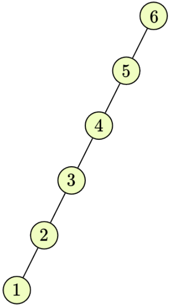
Estratégias de Balanceamento

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

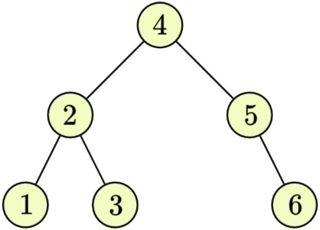
  - **AVL Tree**
  - **Red-black Tree**
  - Splay Tree
  - Treap
  - Skip List
  - Hash Table
  - Bloom Filter

Estratégias de Balanceamento

- Uma estratégia simples: **reconstruir** a árvore de vez em quando



$\Rightarrow$



Estratégias de Balanceamento

Dada uma lista ordenada de números, **por que ordem inserir** numa árvore binária de pesquisa para que fique o mais balanceada possível?

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

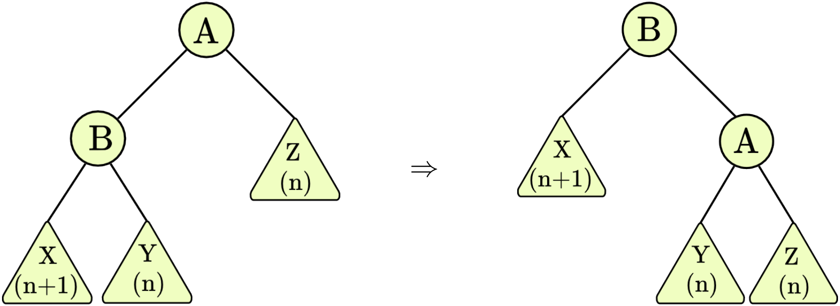
Com que frequência se deve reconstruir a árvore binária para garantir maior eficiência?

- Se reconstruirmos muitas vezes, temos muitas operações $\mathcal{O}(N)$
- Se reconstruirmos poucas vezes, a árvore pode ficar mal balanceada (como no exemplo anterior)

Uma resposta possível: a cada $\mathcal{O}(\sqrt{N})$ inserções

Estratégias de Balanceamento

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

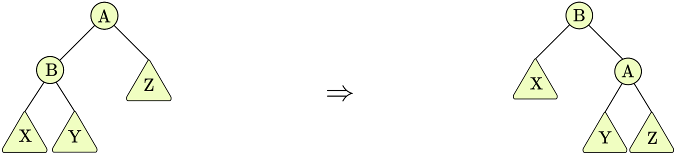


Esta operação base chama-se de **rotação à direita**

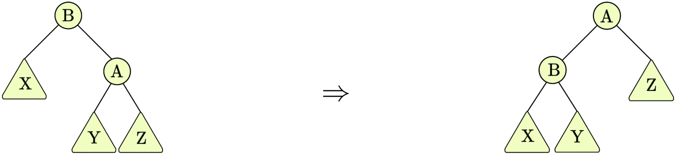
Estratégias de Balanceamento

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

Rotação à direita



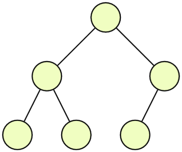
Rotação à esquerda



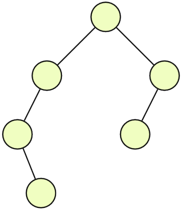
Árvores AVL

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 de uma unidade** (invariante de altura).



Árvore AVL

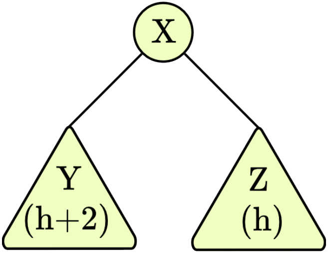


Não Árvore AVL

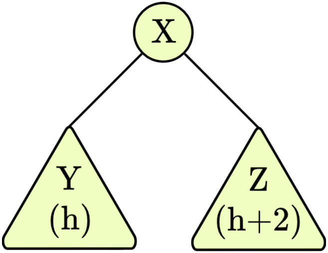
- Quando se inserem ou removem nós, alteramos a árvore de forma a manter o invariante da altura

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

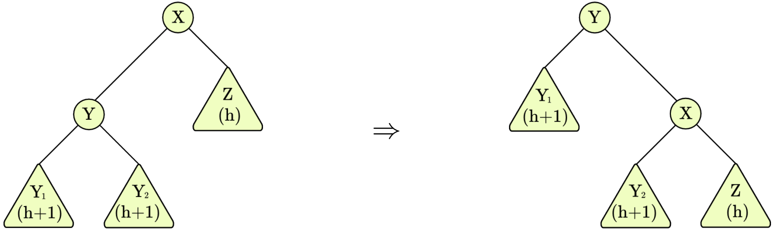


Desnível de 2 para a direita

- 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:



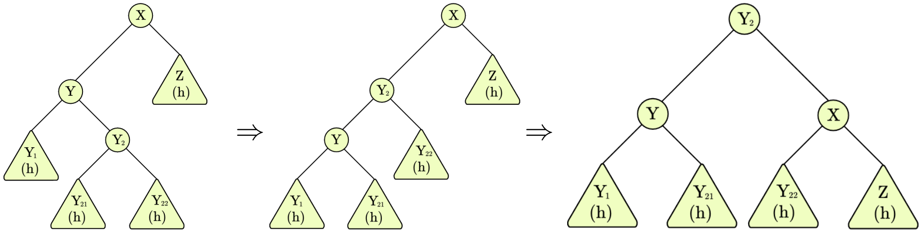
Correção à esquerda, caso 1

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

Árvores AVL

- 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

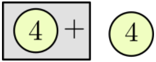
- Nota: a altura de Y_{12} **ou** de Y_{22} pode ser h ou $h-1$, esta correção funciona para ambos os casos

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

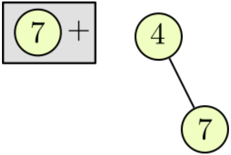
Árvores AVL

- **Exemplo** de inserções de nós:



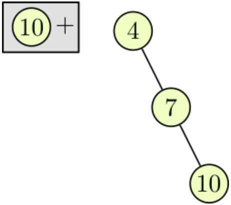
Árvores AVL

- **Exemplo** de inserções de nós:



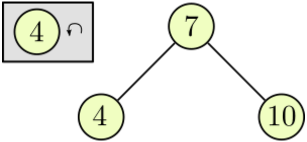
Árvores AVL

- **Exemplo** de inserções de nós:

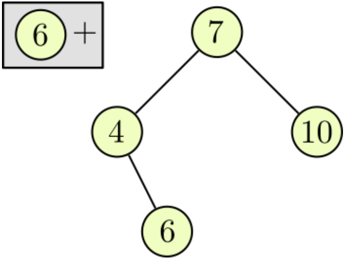


Árvores AVL

- **Exemplo** de inserções de nós:

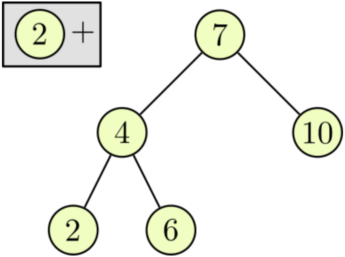


- **Exemplo** de inserções de nós:



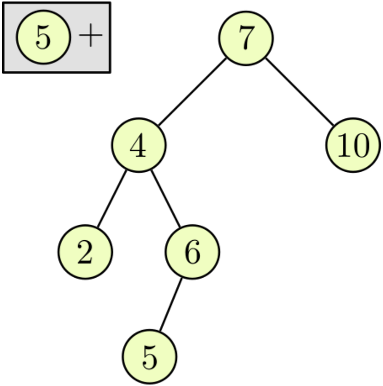
Árvores AVL

- **Exemplo** de inserções de nós:



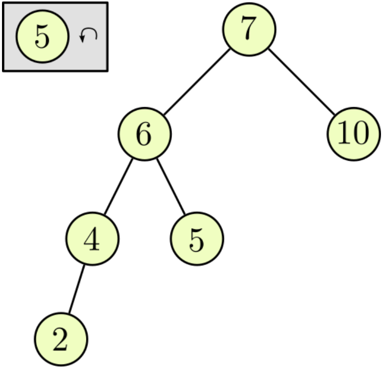
Árvores AVL

- **Exemplo** de inserções de nós:



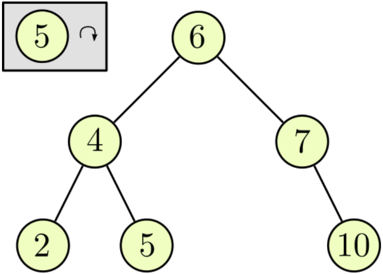
Árvores AVL

- **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

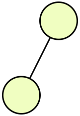
Qual é a altura máxima da árvore?

Árvores AVL

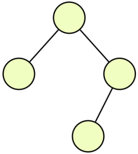
- Para calcular o **pior caso** da altura de uma árvore a qualquer momento faremos o seguinte exercício:
  - Qual a menor árvore AVL (válida segundo o invariante da altura) com altura exatamente h?
  - Chamaremos ao número de nós numa árvore com altura h de $N(h)$

Árvores AVL

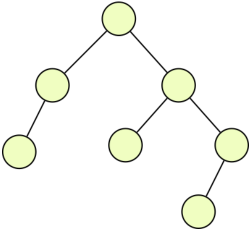
Altura 1



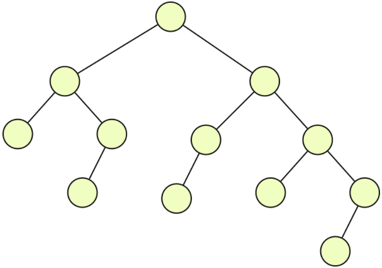
Altura 2



Altura 3



Altura 4



Altura 5

Árvores AVL

- Sumarizando:
  - $N(1) = 1$
  - $N(2) = 2$
  - $N(3) = 4$
  - $N(4) = 7$
  - $N(5) = 12$
  - $\ldots$
  - $N(h) = N(h-2) + N(h-1) + 1$
- 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: $h \leq 1.44 \log(n)$

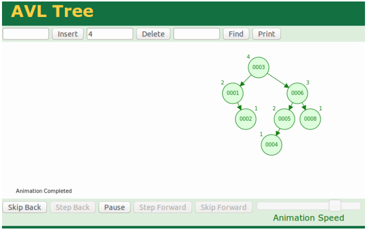
Árvores AVL

- **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

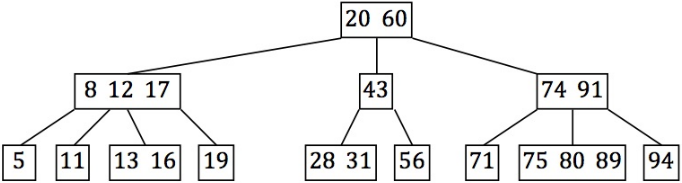
- O nome AVL vem dos autores: G. **A**delson-**V**elsky e E. **L**andis. O artigo original que as descreve é de 1962 (*"An algorithm for the organization of information"*, Proceedings of the USSR Academy of Sciences)

- 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



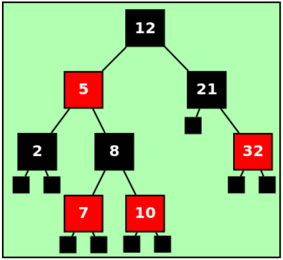
- O artigo original é de 1978 e foi escrito por L. Guibas e R. Sedgewick (*"A Dichromatic Framework for Balanced Trees"*)
- Os autores dizem que se usaram as cores preta e vermelha porque eram as que ficavam melhor quando impressas e eram as cores das canetas que tinham para desenhar as árvores :)

Árvores Red-Black

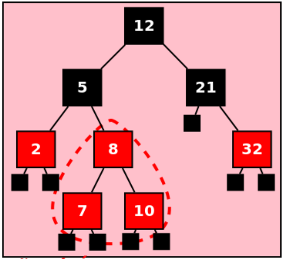
Árvore Red-Black

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

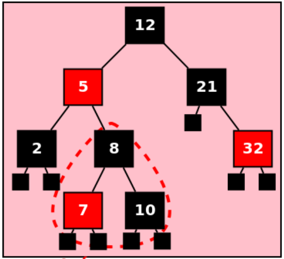
- **(root property)** A raíz da árvore é preta
- **(leaf property)** As folhas são nós (nulos/vazios) pretos
- **(red property)** Os filhos de um nó vermelho são pretos
- **(black property)** Para cada da nó, um caminho para qualquer uma das suas folhas descendentes tem o mesmo número de nós pretos



Árvore Red-Black



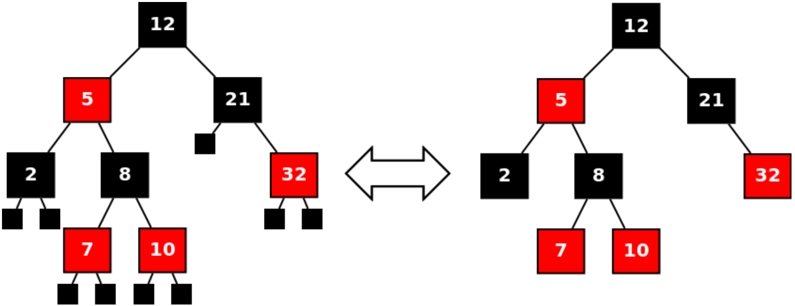
Não é Árvore Red-Black
("red property" não respeitada)



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

Árvores Red-Black

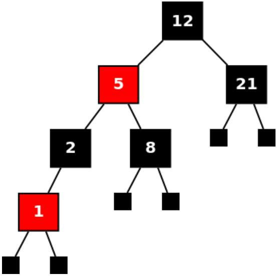
- Por uma questão visual, por vezes as imagens mostradas podem não conter os nós "nulos", mas podem assumir que eles existem
  Aos nós não nulos chamamos de **nós internos**.



- 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]



Árvores Red-Black

Teorema - Altura de uma árvore Red-Black

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

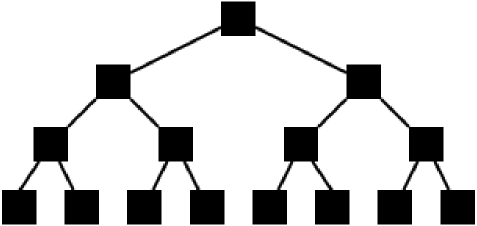
[ou seja, a altura de uma red-black tree é $\mathcal{O}(\log n)$]

Passo 1

A subárvore de um qualquer nó x tem pelo menos $2^{bh(x)} - 1$ nós internos

Intuição:

No mínimo precisamos de uma subárvore completa de nós pretos para satisfazer a invariante da *black property*.



1 = 2^1 -1

3 = 2^2 -1

7 = 2^3 -1

15 = 2^4 -1

Árvores Red-Black

Teorema - Altura de uma árvore Red-Black

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

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

Passo 1:

A subárvore de um qualquer nó x tem pelo menos $2^{bh(x)} - 1$ nós internos

Prova por Indução [Caso base]:

Se o nó x é uma folha, então $bh(x) = 0$ e realmente $2^0 - 1 = 1 - 1 = 0$.

(A folha é um nó nulo e não tem nós debaixo dela)

Árvores Red-Black

Teorema - Altura de uma árvore Red-Black

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

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

Passo 1:

A subárvore de um qualquer nó x tem pelo menos $2^{bh(x)} - 1$ nós internos

Prova por Indução [passo indutivo]:

Considermos um nó interno x da árvore com dois filhos.

Cada filho tem *black height* $bh(x)$ ou $bh(x) - 1$ caso seja vermelho ou preto respectivamente.

Pela hipótese indutiva cada filho tem pelo menos $2^{bh(x)-1} - 1$ nós internos

A subárvore de x tem então pelo menos $2 \times (2^{bh(x)-1} - 1) + 1$ nós internos.

$2 \times (2^{bh(x)-1} - 1) + 1 = 2^{bh(x)} - 1 \quad \square$

Árvores Red-Black

Teorema - Altura de uma árvore Red-Black

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

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

Passo 1:

A subárvore de um qualquer nó x tem pelo menos $2^{bh(x)} - 1$ nós internos

Passo 2:

Seja h a altura da árvore.

Pelo menos metade dos nós num caminho da raíz para uma folha (não incluindo a raíz) têm de ser pretos (pela *black property*)

Isto significa que $bh(raiz) \geq h/2$

Pelo passo 1, temos que então $n \geq 2^{h/2} - 1$

Movendo o 1 para o lado esquerdo e aplicando o logaritmo nos dois lados:

$\log_2(n+1) \geq h/2$, ou seja, $h \leq 2\log_2(n+1)$ $\square$

- Como fazer uma **inserção**?

Inserção de um nó numa árvore red-black não vazia

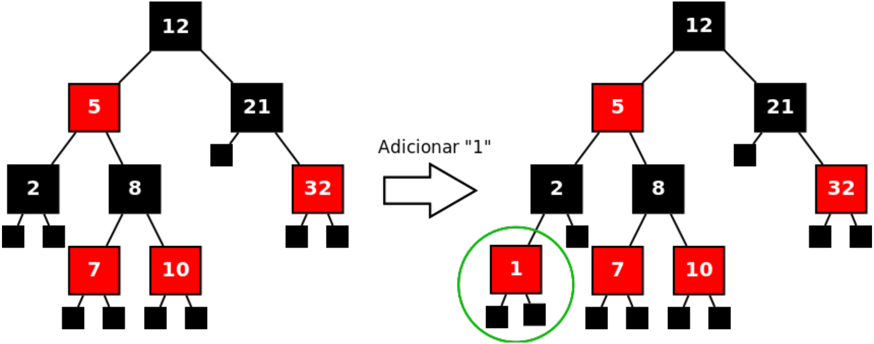
- Inserir como numa qualquer árvore binária de pesquisa
- Colorir o nó inserido de vermelho (acrescentando os nós folha "nulos")
- Recolorir e restruturar se necessário (restaurar invariantes)

- 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

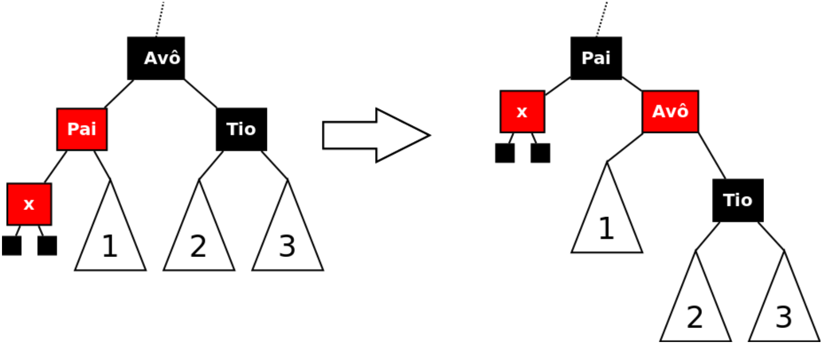
Quando o pai do nó inserido é um nó **preto** não é preciso fazer nada:

Exemplo:



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

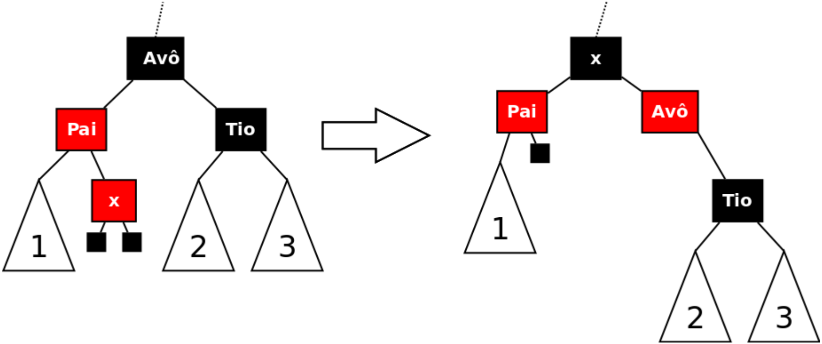


Descrição: rotação de avô à direita seguida de troca de cores entre pai e avô

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



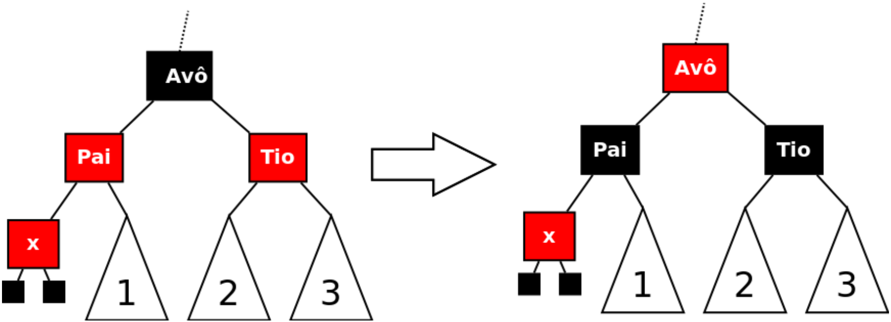
Descrição: rotação à esquerda de pai, seguida dos movimentos de 1.a

[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



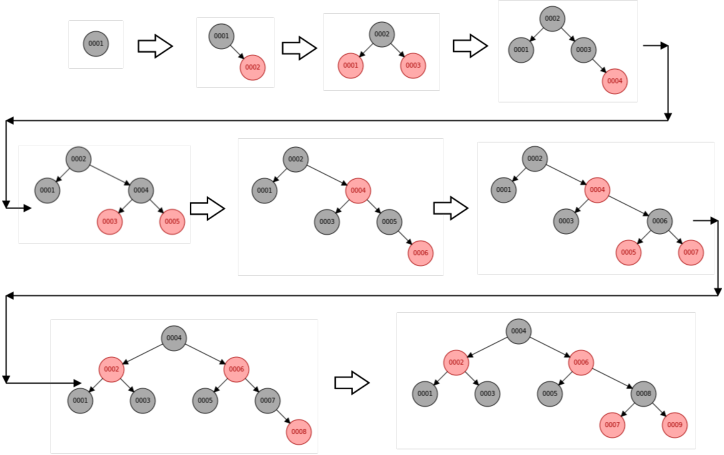
Descrição: trocar cores de pai, tio e avô

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.