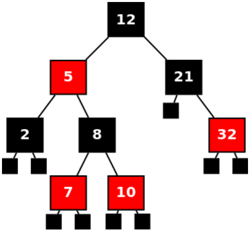
Árvores Binárias de Pesquisa Equilibradas

Pedro Ribeiro

DCC/FCUP

2017/2018



(inclui contribuições de Pedro Paredes)

Motivação

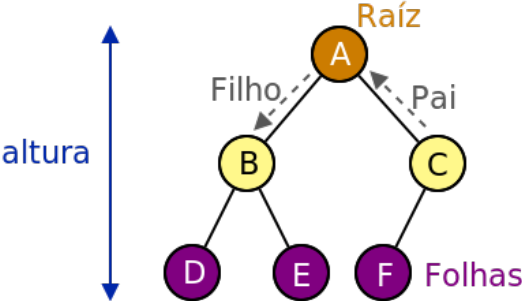
- Seja S um conjunto de objectos/itens **"comparáveis"**:
  - Sejam a e b dois objectos.
    São "comparáveis" se for possível dizer se $a < b$, $a = b$ ou $a > b$.
  - Um exemplo seriam números, mas poderiam ser outra coisa (alunos com um nome e nº mecanográfico; equipas com pontos, golos marcados e 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
  - **Ordenar** um conjunto S
  - ...

- **Árvores Binárias de Pesquisa!**

Árvores Binárias - Notação

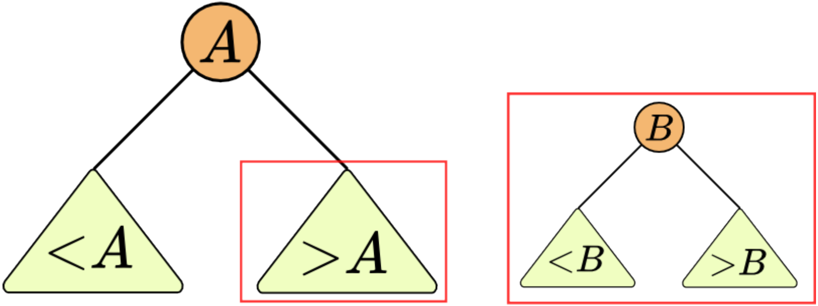
- Resumo da **notação** de árvores binárias:



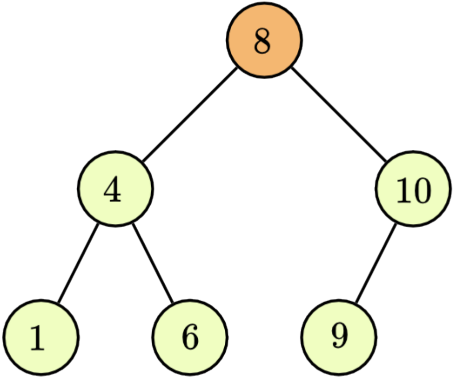
- O nó A é a **raíz** e os nós D, E e F são as **folhas**
- Os nós $\{B, D, E\}$ constituem uma **sub-árvore**
- O nó A é **pai** do nó C
- Os nós D e E são **filhos** do nó B
- O nó B é **irmão** do C
- ...

Árvores Binárias de Pesquisa - Resumo

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



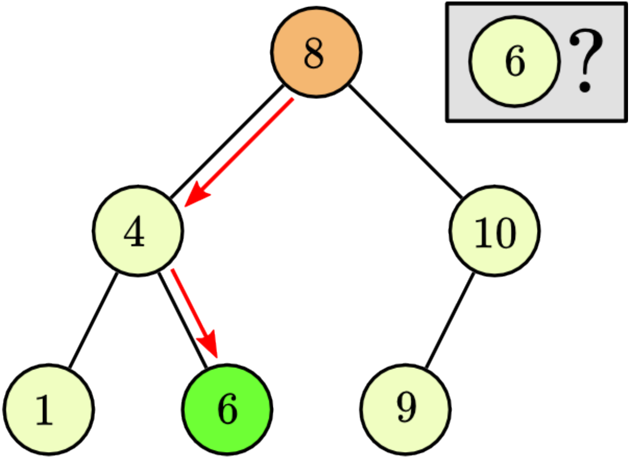
Árvores Binárias de Pesquisa - Exemplo



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

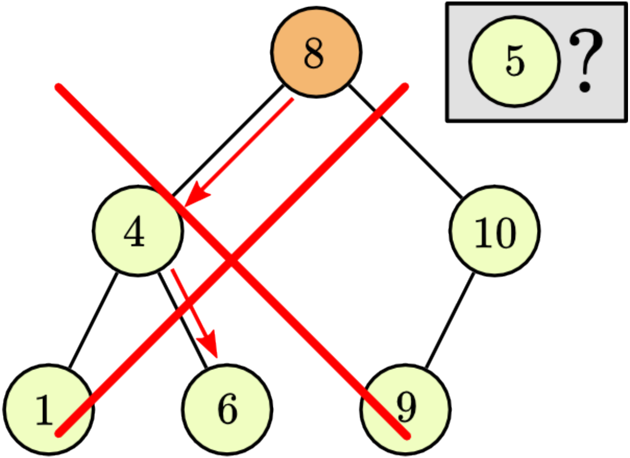
Árvores Binárias de Pesquisa - Procura

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



Árvores Binárias de Pesquisa - Procura

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



Árvores Binárias de Pesquisa - Procura

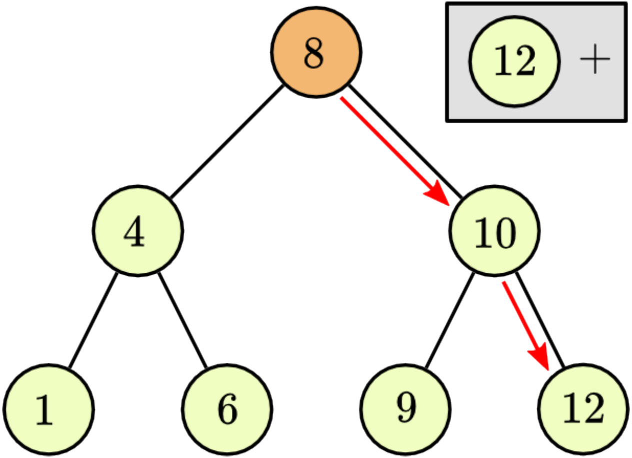
- **Pesquisar valores** em árvores binárias de pesquisa:

Pesquisa numa árvore binária de pesquisa

Pesquisa(T, v):
  Se Nulo(T) **então**
    retorna *NÃO*
  Se $v < T.valor$ **então**
    retorna Pesquisa($T.filho\_esquerdo, v$)
  Senão se $v > T.valor$ **então**
    retorna Pesquisa($T.filho\_direito, v$)
  Senão
    retorna *SIM*

Árvores Binárias de Pesquisa - Inserção

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



Árvores Binárias de Pesquisa - Inserção

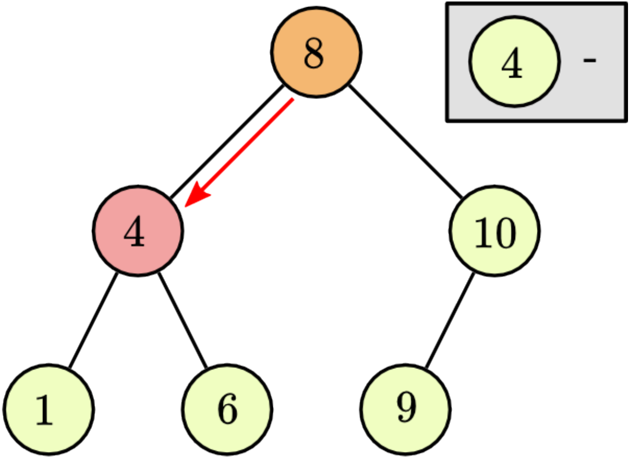
- **Inserir valores** em árvores binárias de pesquisa:

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**($T.filho\_esquerdo, v$)
  Senão se $v > T.valor$ **então**
    retorna $T.direito =$ **Insere**($T.filho\_direito, v$)
  Senão
    retorna T

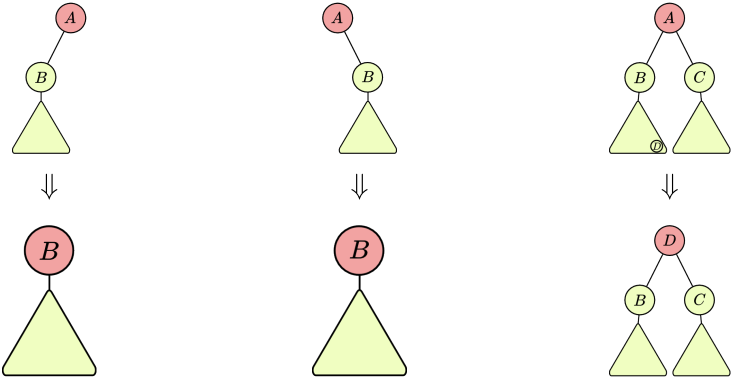
Árvores Binárias de Pesquisa - Remoção

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



Árvores Binárias de Pesquisa - Remoção

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



Árvores Binárias de Pesquisa - Tempo de Execução

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

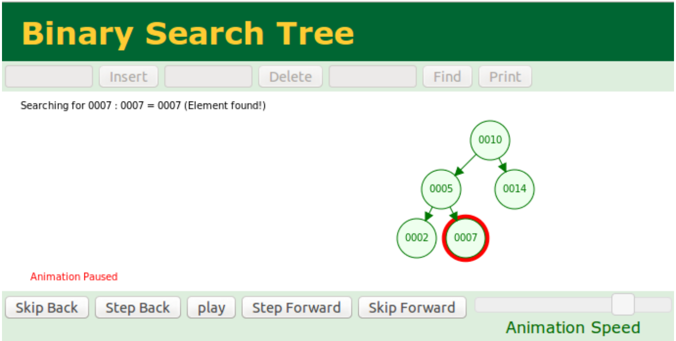
Complexidade de operações numa árvore binária de pesquisa

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 $\mathcal{O}(H)$.

Árvores Binárias de Pesquisa - Visualização

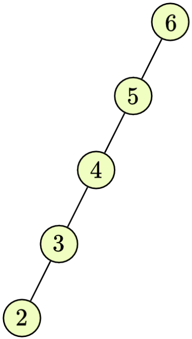
- Podem visualizar a pesquisa, inserção e remoção (experimentem o url indicado):

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



Desiquilíbrio numa Árvore Binária de Pesquisa

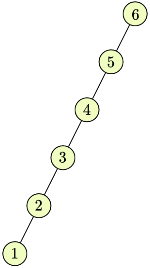
- O **problema** do método anterior:



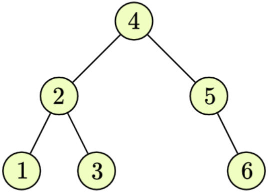
A altura da árvore pode ser da ordem de $\mathcal{O}(N)$ (N, número de elementos)

Árvores equilibradas

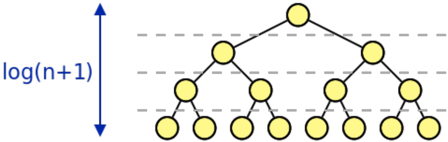
- Queremos árvores... **equilibradas**



vs



- Numa árvore equilibrada com n nós, a altura é... da ordem de $\log(n)$

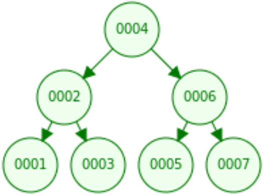


Nós no nível	Total de Nós
1 = 2^0	1 = 2^1 - 1
2 = 2^1	3 = 2^2 - 1
4 = 2^2	7 = 2^3 - 1
8 = 2^3	15 = 2^4 - 1

Árvores equilibradas

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

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

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

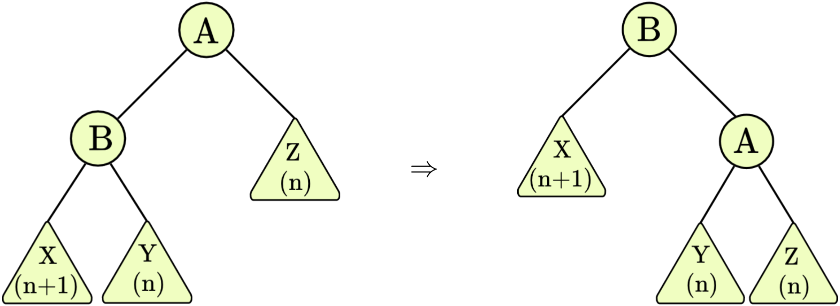
- **Red-Black Trees**
- AVL Trees
- Splay Trees
- Treap

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):

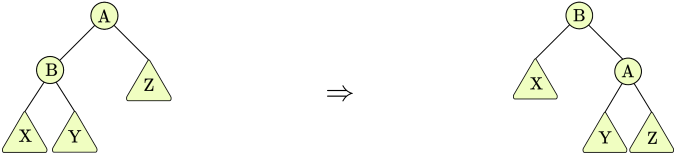


Esta operação base chama-se de **rotação à direita**

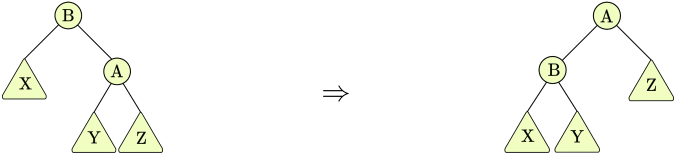
Estratégias de Balanceamento

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

Rotação à direita

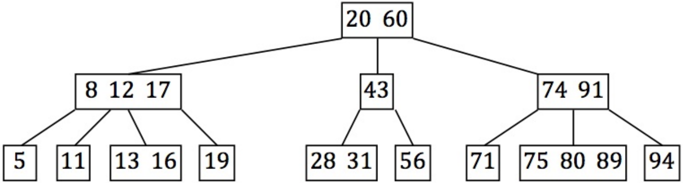


Rotação à esquerda



Árvores Red-Black

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



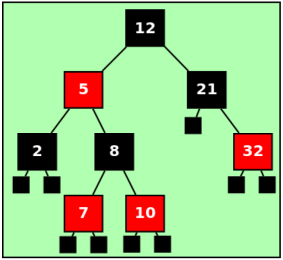
- O artigo original é de 1978 e foi escrito por L. Guibas e R. Sedgewick (*"A Dichromatic Framework for Balanced Trees"*)
- Os autores dizem que se usaram as cores preta e vermelha porque eram as que ficavam melhor quando impressas e eram as cores das canetas que tinham para desenhar as árvores :)

Árvores Red-Black

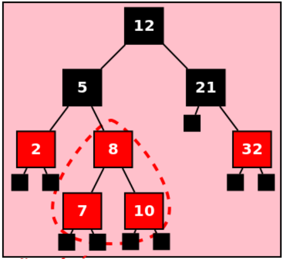
Árvore Red-Black

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

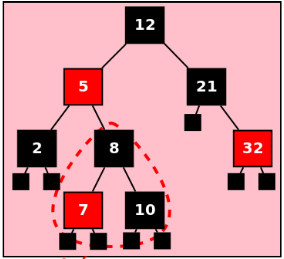
- **(root property)** A raíz da árvore é preta
- **(leaf property)** As folhas são nós (nulos/vazios) pretos
- **(red property)** Os filhos de um nó vermelho são pretos
- **(black property)** Para cada da nó, um caminho para qualquer uma das suas folhas descendentes tem o mesmo número de nós pretos



Árvore Red-Black



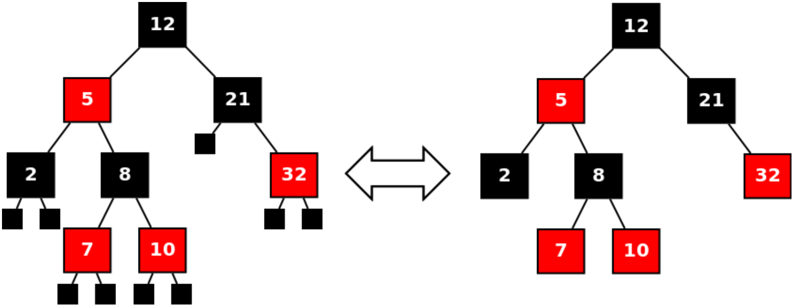
Não é Árvore Red-Black
("red property" não respeitada)



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

Árvores Red-Black

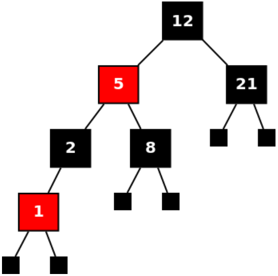
- Por uma questão visual, por vezes as imagens mostradas podem não conter os nós "nulos", mas podem assumir que eles existem
  Aos nós não nulos chamamos de **nós internos**.



- 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 "equilíbrio" 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 no máximo duas vezes maior que a de um ramo irmão



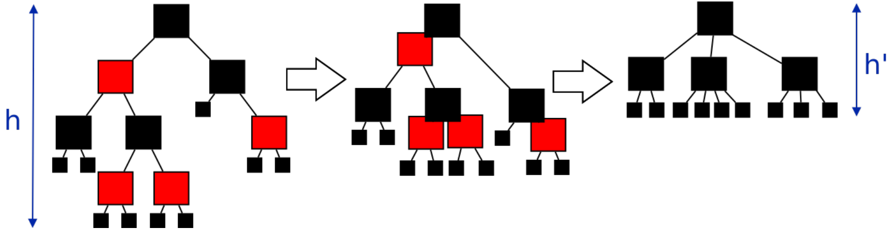
Árvores Red-Black

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 nos seus nós pais pretos:

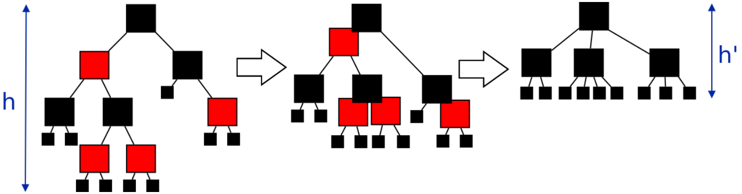


- Este processo produz uma árvore com 2, 3 ou 4 filhos
- Esta árvore 2-3-4 tem folhas a uma profundidade uniforme de **h'** (essa profundidade é a *black height*)

Árvores Red-Black

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 $\mathcal{O}(\log n)$*]*



- A altura desta árvore é no mínimo 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 internos da nova árvore é $\geq 2^{h'} - 1$ (é uma árvore 2-3-4)
- A árvore original tinha ainda mais nós internos que a nova: $n \geq 2^{h'} - 1$
- $n+1 \geq 2^{h'}$
- $\log_2(n+1) \geq h' \geq h/2$
- $h \leq 2\log_2(n+1) \quad \square$

Árvores Red-Black

- Como fazer uma **inserção**?

Inserção de um nó numa árvore red-black não vazia

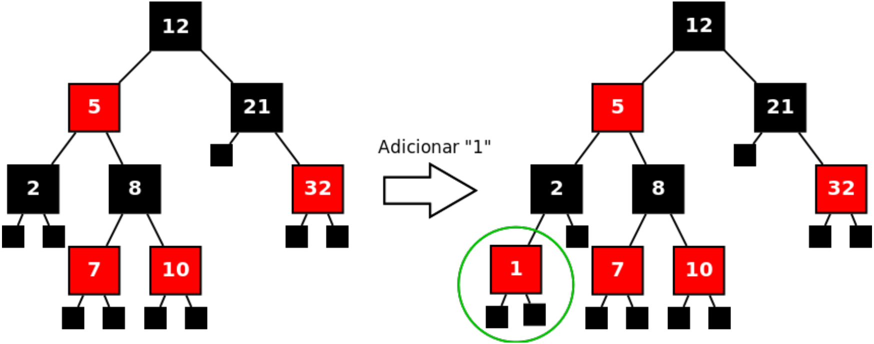
- Inserir como numa qualquer árvore binária de pesquisa
- Colorir o nó inserido de vermelho (acrescentando os nós folha "nulos")
- Recolorir e restruturar se necessário (restaurar invariantes)

- 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:

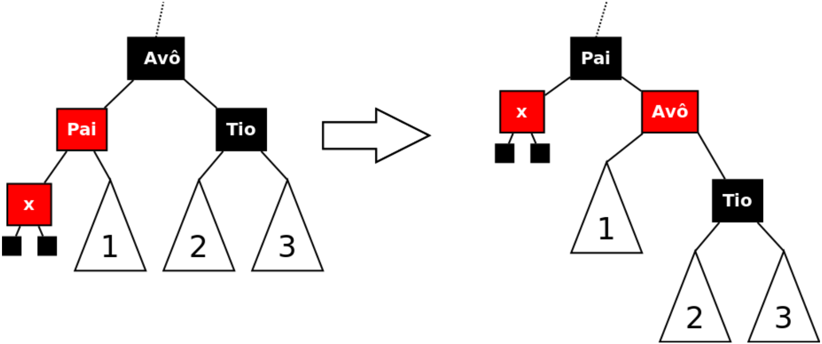
Exemplo:



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

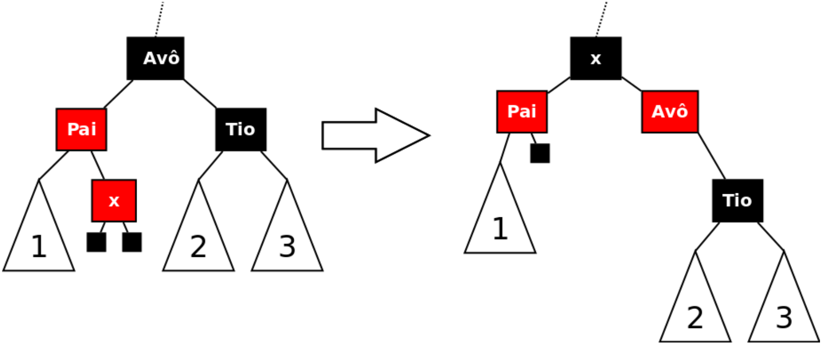


Descrição: rotação de avô à direita seguida de troca de cores entre pai e avô

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



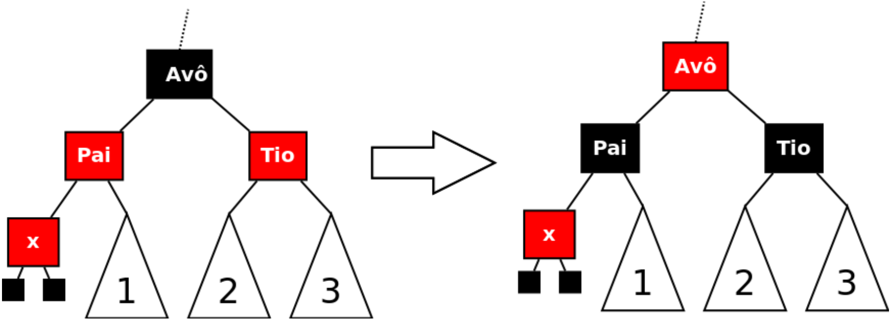
Descrição: rotação à esquerda de pai, seguida dos movimentos de 1.a

[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



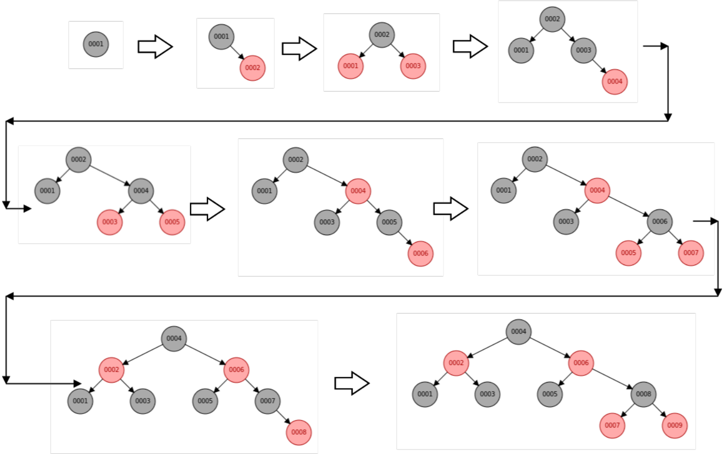
Descrição: trocar cores de pai, tio e avô

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)

- As árvores Red-Black são muito usadas nas linguagens usuais.
Exemplos de estruturas de dados que as usam:
  - 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 com o mesmo tipo de finalidade (*find*, *insert*, *remove*, *min*, *max*)
- Um exemplo são as **árvores AVL**:
  - Mantêm um equilíbrio mais "apertado": 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).
  - Isto garante uma altura $\sim 1.44 \log(n)$ (vs altura $\sim 2 \log(n)$ das RB) (pesquisa é portanto "ligeiramente" mais rápida)
  - Rotações para manter equilíbrio (um pouco mais pesadas que RB)
  - Ocupam um pouco mais de memória
    (RB só precisam da cor, AVL precisam do desnível)

  Espreitem uma visualização:

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

Outros tipos de árvores

- Existem muitos mais tipos de árvores de pesquisa com o mesmo tipo de finalidade (*find*, *insert*, *remove*, *min*, *max*)
- Um outro 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.